OSGi Service Platform Residential Specification

The OSGi Alliance

Release 4, Version 4.3
January 2012

OSGi™ Alliance

Feedback

This specification can be downloaded from the OSGi Alliance web site:

`http://www.osgi.org`

Comments about this specification can be raised at:

`http://www.osgi.org/bugzilla/`

Table Of Contents

1 Introduction

When the OSGi Alliance started in 1998 the focus was in residential gateways, it actually was an acronym that contained the word *gateway* before it became a name. Since that time, OSGi flourished in several different markets but did not gain a wide-spread adoption in the residential markets so far. However, some of the road blocks to this wide spread adoption have changed:

- Hardware cost of service gateways have been reduced.
- Capabilities of those small devices increased manyfold.
- The license fee for Java Virtual Machines has decreased.
- The number of devices in a house hold has increased.
- Always-on and broadband access to the Internet has become pervasive.

This specification, produced by the OSGi Residential Expert Group (REG), defines a set of new and refined service guidelines that focus on the residential market. The REG is chartered to define the requirements and specifications to tailor the OSGi Service Platform for fixed network connected devices. Examples of such devices include residential gateways, building automation controllers, white goods, consumer electronics, and many other devices.

Technical Areas addressed by the REG include the requirements, functional specifications, and APIs for gateway devices. The REG also created a functional model for local and remote management of gateway devices. This model resolves the requirements of inter-operation with existing management systems and protocols, the need to remotely manage the user applications life cycles, the need for large-scale deployments and adequate security.

The services of this Residential Specification have been designed with the residential market in mind. Requirements and management protocols for this environment are defined in the specifications by consortias like the [3] *Home Gateway Initiative*, (HGI) the [6] *Broadband Forum* (BBF) and the [5] *UPnP™ Forum*. These specifications provide requirements for execution environments in a Consumer Premises Equipment (CPE) and other consumer devices, as well as protocols for the management of residential environments. Here, the scope of management protocols span the remote management of thousands and millions of CPE devices by a telecommunications provider as well as local management of single consumer devices in a home or residence building.

The services guidelines of the Residential Specification have been designed to integrate with OSGi, fulfill the requirements of the mentioned consortias and cooperate with each other. None of the listed service specifications is mandatory; all service specifications are optional.

It is not suggested, or expected, that a solution will support all listed specifications. More likely, platform providers define their specific runtime environment. It is highly recommended to choose the mandatory and optional services defined by the [4] *HG Requirements for Software Execution Environment* as a basis. The same is true for the management protocols defined by BBF and the UPnP Forum. A solution can further include other core and compendium services that are not listed as part the Residential Specification.

1.1 Overview of the Residential Specifications

1.1.1 Remote Management

Support for remotely managing the service and their applications is essential to all systems that are installed on customer's premises. The specification therefore has special focus on large scale remote management of the OSGi Service Platform. The architecture provides a solution to allow management over different protocols although the primary focus is to allow the use Broadband Forum's suite of specifications on an OSGi residential gateway. This section introduces the related specifications.

- *Dmt Admin Service Specification* – The Dmt Admin Service Specification provides an API for a remote manager to manage the device and its diverse services running on it. The Dmt Admin provides a generic tree structure, the Device Management Tree (DMT), to a Protocol Adapter. The nodes of those trees are implemented by the devices and services. Different Protocol Adapters can leverage the same DMT for different protocols. The Dmt Admin service also provides guidelines for object models that can be made available over different protocols. For more details see the *Dmt Admin Service Specification* on page 285.
- *Residential Device Management* – This specification defines a Residential Management Tree, the RMT. This tree provides a general Dmt Admin object model that allows browsing and managing the OSGi Service Platform remotely over different Protocol Adapters. The RMT provides access to the Framework and the Log service. It also provides a filter function on top of Dmt Admin. See the *Residential Device Management Tree Specification* on page 7 for details.
- *TR-157a3 Software Module* – [6] *Broadband Forum* has defined a generic model for mapping software modules in [8] *TR-157 Amendment 3 Component Objects for CWMP*. This specification provides a recommended mapping for the generic concepts to the OSGi Framework concepts. See *TR-157 Amendment 3 Software Module Guidelines* on page 29.
- *TR-069 Connector Service Specification* – The Dmt Admin service and the TR-069 protocol have different semantics and primitives. This specification contains the *TR069 Connector Service Specification* on page 409. This specification provides an API based on the TR-069 Remote Procedure Calls concept that is implemented on top of Dmt Admin. This connector supports data conversion and the object modeling constructs defined in the Dmt Admin service, *OSGi Object Modeling* on page 327.

1.1.2 Management and Configuration services

The OSGi Service Platform is unique in that it does not hide anything, all aspects are manageable from the system itself. To locally manage the system, the following services are available:

- *Conditional Permission Admin Service Specification* – The Conditional Permission Admin service allows an operator to control the Java Permissions to be granted to the bundles running on the OSGi Service Platform using a condition based model. See chapter 50 in [2] *OSGi Service Platform Core Specification,Release 4, Version 4.3*.
- *Permission Admin Service Specification* – The Permission Admin service allows an operator to control the Java Permissions to be granted to the bundles running on the OSGi Service Platform based on the bundle location. The Permission Admin has been superseded by the Conditional Permission admin, but is included for backwards compatibility. See chapter 51 in [2] *OSGi Service Platform Core Specification,Release 4, Version 4.3*.
- *URL Handlers Service Specification* – This specification standardizes the mechanism to extend the Java run-time with new URL schemes and content handlers. Dynamically extending the URL schemes that are supported in an OSGi Service Platform is a very powerful concept to provide more functionality to existing applications. See chapter 51 in [2] *OSGi Service Platform Core Specification,Release 4, Version 4.3*.
- *User Admin Service Specification* – The User Admin Service Specification provides authorization for OSGi Service Platform actions based on authenticated users instead of using the Java code-based permission model. See the *User Admin Service Specification* on page 151.
- *Initial Provisioning Specification* – The Initial Provisioning specification defines how a Management Agent and other initial bundles can be deployed on an uninitialized OSGi Service Platform. It gives a structured view of the problems and their corresponding resolution methods. The purpose of this specification is to enable the management of a Service Platform by an operator, and (optionally) to hand over the management of the Service Platform later to another operator. See the *Initial Provisioning Specification* on page 171 for more details.
- *Configuration Admin Service Specification* – The Configuration Admin service allows an operator or an application bundle developer to set the configuration information of bundles. See *Configuration Admin Service Specification* on page 89.
- *Metatype Service Specification* – The Metatype specification defines interfaces that allow bundle developers to describe attribute types in a computer readable form using metadata. It is mostly

used in conjunction with the Configuration Admin Service. See *Metatype Service Specification* on page 129 for details.

1.1.3 Component Models

Component models allow the code in bundles to remain unaware of OSGi API by using Dependency Injection (DI) while still providing full support for the OSGi service model.

- *Declarative Services Specification* – The Declarative Services specification provides dependency injection for services. It handles the service life cycle dynamics by notifying the component or managing the components life cycle. See *Declarative Services Specification* on page 215.

1.1.4 HTTP and Servlets

HTTP Server and Servlets functions are often needed for the residential gateway. The Specification contains this specification:

- *Http Service Specification* – Developers typically need to develop communication and user interface solutions for standard technologies such as HTTP, HTML, XML, and servlets. See the *Http Service Specification* on page 47.

1.1.5 Event models

The OSGi service model is based on synchronous APIs. Support for asynchronous invocations and event driven interactions usually involves the definition of listeners. However, this model does not scale well for fine grained events that must be dispatched to many different handlers. The Specification therefore contains the Event Admin Service Specification:

- *Event Admin Service Specification* – The Event Admin service provides an inter-bundle communication mechanism. It is based on a event publish and subscribe model, popular in many message based systems. See *Event Admin Service Specification* on page 265.

1.1.6 Other Residential Services

A residential gateway can directly attach devices, for example via a USB adaptor, through a home network. There is therefore a need to have a unified device abstraction, discovery and control model. For this purpose, this Specification contains the following services:

- *Device Access Specification* – The Device Access specification supports the coordination of automatic detection and attachment of existing devices on an OSGi Service Platform, facilitates hot-plugging and -unplugging of new devices, and downloads and installs device drivers on demand. See *Device Access Specification* on page 63.
- *UPnP™ Device Service Specification* – The UPnP specification specifies how OSGi bundles can be developed that inter-operate with UPnP (Universal Plug and Play) devices and UPnP control points. The specification is based on[7] *UPnP Device Architecture 1.0*. See *UPnP™ Device Service Specification* on page 189.

1.1.7 Miscellaneous Supporting Services

Services providing solutions to common infrastructure requirements include:

- *Log Service Specification* – Provides a general purpose message logger for the OSGi Service Platform. See the *Log Service Specification* on page 37.
- *XML Parser Service Specification* – Addresses how the classes defined in JAXP can be used in an OSGi Service Platform. See *XML Parser Service Specification* on page 459.
- *Tracker Specification* – Simplifies tracking the life cycle of bundles and services. See *Tracker Specification* on page 439.

1.2 Version Information

This document is the Residential Specification for the OSGi Service Platform Release 4, Version 4.3. Components in this specification have their own specification version, independent of the OSGi Service Platform, Release 4, Version 4.3 specification. The following table summarizes the packages and specification versions for the different subjects.

Table 1.1 Packages and versions

Item	Package(s)	Version
[2] *OSGi Service Platform Core Specification,Release 4, Version 4.3*	org.osgi.framework	Version 1.6
	org.osgi.framework.hooks.bundle	Version 1.0
	org.osgi.framework.hooks.resolver	Version 1.0
	org.osgi.framework.hooks.service	Version 1.1
	org.osgi.framework.hooks.weaving	Version 1.0
	org.osgi.framework.launch	Version 1.0
	org.osgi.framework.startlevel	Version 1.0
	org.osgi.framework.wiring	Version 1.0
	org.osgi.service.condpermadmin	Version 1.1
	org.osgi.service.url	Version 1.0
	org.osgi.service.permissionadmin	Version 1.2
2 Residential Device Management Tree Specification	org.osgi.dmt.residential*	Version 1.0
101 Log Service Specification	org.osgi.service.log	Version 1.3
102 Http Service Specification	org.osgi.service.http	Version 1.2
103 Device Access Specification	org.osgi.service.device	Version 1.1
104 Configuration Admin Service Specification	org.osgi.service.cm	Version 1.4
105 Metatype Service Specification	org.osgi.service.metatype	Version 1.2
107 User Admin Service Specification	org.osgi.service.useradmin	Version 1.1
110 Initial Provisioning Specification	org.osgi.service.provisioning	Version 1.2
111 UPnP™ Device Service Specification	org.osgi.service.upnp	Version 1.2
112 Declarative Services Specification	org.osgi.service.component org.osgi.service.component.annotations	Version 1.2
113 Event Admin Service Specification	org.osgi.service.event	Version 1.3
117 Dmt Admin Service Specification	org.osgi.service.dmt org.osgi.service.dmt.spi org.osgi.service.dmt.notification org.osgi.service.dmt.notification.spi org.osgi.service.dmt.security	Version 2.0
131 TR069 Connector Service Specification	org.osgi.service.tr069todmt	Version 1.0
701 Tracker Specification	org.osgi.util.tracker	Version 1.5
702 XML Parser Service Specification	org.osgi.util.xml	Version 1.0

* – This is not a Java package but contains DMT Types.

When a component is represented in a bundle, a version attribute is needed in the declaration of the Import-Package or Export-Package manifest headers.

1.3 References

[1] *Bradner, S., Key words for use in RFCs to Indicate Requirement Levels*
http://www.ietf.org/rfc/rfc2119.txt, March 1997.

[2] *OSGi Service Platform Core Specification,Release 4, Version 4.3*
http://www.osgi.org/Specifications/HomePage

[3] *Home Gateway Initiative*
http://www.homegatewayinitiative.org

[4] *HG Requirements for Software Execution Environment*
http://www.homegatewayinitiative.org/publis/RD-008- R3.pdf

[5] *UPnP™ Forum*
http://upnp.org

[6] *Broadband Forum*
http://www.broadband-forum.org

[7] *UPnP Device Architecture 1.0*
http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0.pdf

[8] *TR-157 Amendment 3 Component Objects for CWMP*
http://www.broadband-forum.org/technical/download/TR-157_Amendment-3.pdf

2 Residential Device Management Tree Specification

Version 1.0

2.1 Introduction

The chapter defines the Device Management Tree (DMT) for residential applications called the *Residential Management Tree* (RMT). This RMT is based on the *Dmt Admin Service Specification* on page 285. The RMT allows remote managers to manage the residential device through an abstract tree. As this tree is an abstract representation, different management protocols can use the same underlying management components, the Dmt Admin Plugins, in the OSGi framework.

This chapter requires full understanding of the concepts in the *Dmt Admin Service Specification* on page 285 and uses its terminology.

2.1.1 Essentials

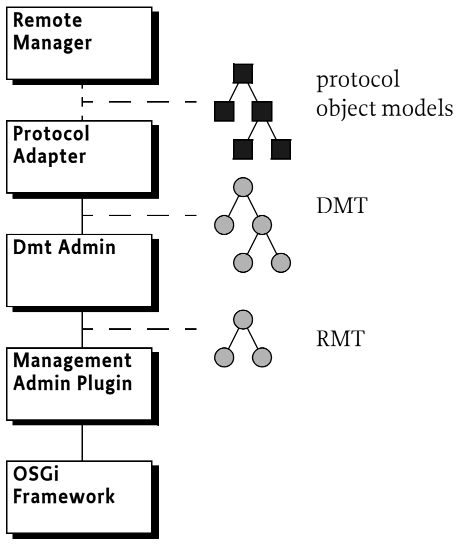
The following essentials are associated with the Residential Management Tree specification:

- *Complete* – The RMT must cover all functionality to completely manage an OSGi Framework as defined by the Release 4, Version 4.3 Core specification.
- *Performance* – The RMT runs on devices with limited resources.
- *Searchable* – Provide an efficient way to search the RMT remotely.
- *Services* – Provide efficient access to standardized services like the Log Service.

2.1.2 Entities

- *Remote Manager* – The entity that remotely controls an OSGi Framework.
- *Management Agent* – An entity running on the device that is responsible for the management of the local OSGi Framework. It usually acts as a proxy for a Remote Manager.
- *Protocol Adapter* – Communicates with a Remote Manager and translates the protocol instructions to instructions to a local Management Agent.
- *DMT* – The Device Management Tree. This is the general structure available through the Dmt Admin service.
- *RMT* – The Residential Management Tree. This is the part of the DMT that is involved with residential management.

Figure 2.1 Device Management Architecture



2.2 The Residential Management Tree

The *OSGi* node is the root node for OSGi specific information. This OSGi node can be placed anywhere in the Device Management Tree and acts as parent to all the top level nodes in this specification. Therefore, in this specification the parent node of, for example, the Framework node is referred to as $, which effectively represents the OSGi node. The description of the nodes are using the types defined in *OSGi Object Modeling* on page 327.

The value of $ for a specific system can be defined with the following Framework property:

```
org.osgi.dmt.residential
```

For this specifications, the RMT Consists of the following top level nodes:

- Framework – Managing the local Framework
- Filter – Searching nodes in the DMT
- Log – Access to the log

2.3 Managing Bundles

The Framework node provides a remote management model for managing the life cycle of bundles and inspecting the Framework's state.

To change the state, for example install a new bundle, requires an atomic session on at least the Framework node. The model is constructed to reflect the requested state. When the session is committed, the underlying Plugin must effectuate these requested states into the real state.

For example, to install a bundle it is first necessary to create a new Bundle child node. The Bundle node is a MAP node, the name of the child node is the location of the bundle as given in the installBundle(location,input stream) method and returned from the getLocation() method.

This location should not be treated as the actual URL of the bundle, the location is better intended to be used a management name for the bundle as the remote manager can choose it. It is normally best to make this name a reverse domain name, for example com.acme.admin. The name " System Bundle" is a reserved name for the system bundle. The Framework management plugin must therefore not treat the location as a URL.

Creating the child node has no effect as long as the session is not committed. This new Bundle node automatically gets the members defined in the Bundle type.

The URL node should be set to the download URL, the URL used to download the JAR file from. The URL node is used as the download URL for an install operation (after the node is created newly) or the update location when the node is changed after the bundle had been installed in a previous session. Creating a new Bundle node without setting the URL must generate an error when the session is committed.

To start this newly installed bundle, the manager can set the RequestedState to ACTIVE. If this bundle needs to be started when the framework is restarted, then the AutoStart node can be set to true. If there bundles to be uninstalled then their RequestedState node must be set to UNINSTALLED as it is not possible to delete a Bundle node. The RequestedState must be applied after the bundle has been installed or updated. An uninstalled bundle will be automatically removed from the RMT.

The RequestedState node is really the requested state, depending on start levels and other existing conditions the bundle can either follow the requested state or have another state if, for example, its start level is not met. The RequestedState must be stored persistently between invocations, its initial value is INSTALLED.

The manager can create any number of new Bundle nodes to install a number of bundles at the same time during commit. It can also change the life cycle of existing bundles. None of these changes must have any effect until the session is committed.

If the session is finally committed, the Plugin must compare the state in the Dmt Admin tree with the actual state and update the framework accordingly. The order in which the operations occur is up to the implementation except for framework operations, they must always occur last. After bundles have been installed, uninstalled, or updated, the Plugin must refresh all the packages to ensure that the remote management system sees a consistent state.

Downloading the bundles from a remote system can take substantial time. As the commit is used synchronously, it is sometimes advisable to download the bundles to the device before they are installed.

If any error occurs, any changes that were made since the beginning of the last transaction point must be rolled back. An error should be reported. The remote manager therefore gets an atomic behavior, either all changes succeed or all fail. A manager should also be aware that if its own bundle, or any of its dependencies, is updated it will be stopped and will not be able to properly report the outcome to the management system, either a failure or success.

2.3.1 Bundle Life Cycle Example

For example, the following code installs my_bundle, updates up_bundle, and uninstalls old_bundle:

```
String $ = ... // get the OSGi node
DmtSession session = admin.getSession($ + "/Framework",
           DmtSession.LOCK_TYPE_ATOMIC);
try {
  session.createInteriorNode("Bundle/my_bundle");
  session.setNodeValue("Bundle/my_bundle/URL", new DmtData(
    "http://www.example.com/bundles/my_bundle.jar"));
  session.setNodeValue("Bundle/my_bundle/AutoStart",
    DmtData.TRUE_VALUE);
  session.setNodeValue("Bundle/my_bundle/RequestedState",
    new DmtData("ACTIVE"));

  session.setNodeValue("Bundle/up_bundle/URL", new DmtData(
    "http://www.example.com/bundles/up_bundle-2.jar"));

  session.setNodeValue("Bundle/old_bundle/RequestedState",
    new DmtData("UNINSTALLED"));
  try {
    session.commit();
  } catch (Exception e) {
     // failure ...
     log....
  }
} catch (Exception e) {
  session.rollback();
  log...
}
```

2.3.2 Framework Restart

There are no special operations for managing the life cycle of the Framework, these operations are done on the System Bundle, or bundle 0. The framework can be stopped or restarted:

- *Restart* – Restarting is an update, requiring the URL to be set to a new URL. This must shutdown the framework after the commit has succeeded.
- *Stopping* – Stopping is setting the RequestedState to INSTALLED

If the URL node has changed, the RequestedState will be ignored and the framework must only be restarted.

Sessions that modify nodes inside the Framework sub-tree must always be atomic and opened on the Framework node. The Data Plugin managing the Framework node is only required to handle a single simultaneous atomic session for its whole sub-tree.

For example, the following code restarts the framework after the commit has succeeded.

```
DmtSession session = admin.getSession($ + "/Framework",
           DmtSession.LOCK_TYPE_ATOMIC);
session.setNodeValue("Bundle/System Bundle/URL",
  new DmtData(""));
session.commit();
```

2.3.3 Access to Wiring

During runtime a bundle is wired to several different entities, other bundles, fragments, packages, and services. The framework defines a general Requirement-Capability model and this model is reflected in the Wiring API in[1] *OSGi Service Platform Core Specification,Release 4, Version 4.3*. The Requirement-Capability model maps to a very generic way of describing wires between requirers and providers that is applicable to all of the OSGi constructs.

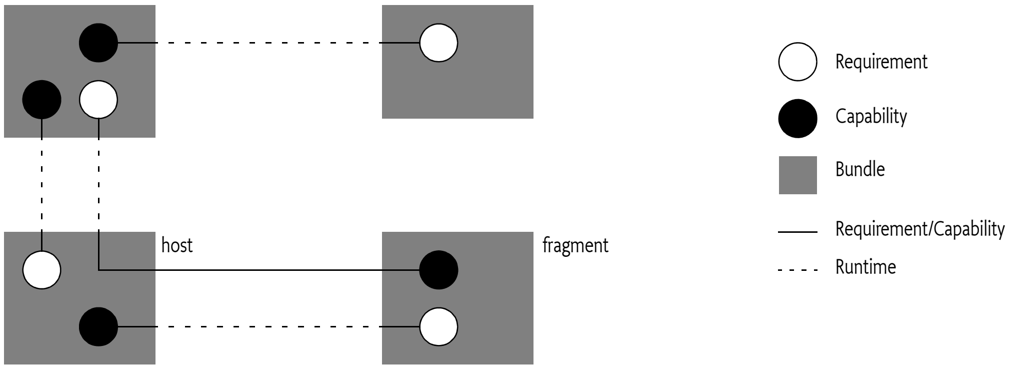
The Core defines namespaces for:

- osgi.wiring.bundle – The namespace for the Require-Bundle header. It wires the bundle with the Require-Bundle header to the bundle with the required Bundle-SymbolicName and Bundle-Version header.
- osgi.wiring.host – The namespace for the Fragment-Host header. It wires from bundle with the Fragment-Host header to the bundle with the required Bundle-SymbolicName and Bundle-Version header.

- osgi.wiring.package – The namespace for the Import/Export-Package header. It wires from bundle with the Import-Package header to the bundle with the Export-Package header.

In the Core API, the wiring is based on the Bundle revisions. However, this specification requires that all bundles are refreshed after a management operation to ensure a consistent wiring state. The management model therefore ignores the Bundle Revision and instead provides wiring only for bundles since the manager is unable to see different revision of a bundle anyway. The general Requirement-Capability model is depicted in Figure 2.2.

Figure 2.2 Requirements and Capabilities and their Wiring



The core does not specify a namespace for services. However, services can also be modeled with requirements capabilities. The registrar is the provider and the service properties are the capability. The getter is the requirer, its filter is the requirement. This specification therefore also defines a namespaces for services:

```
osgi.wiring.rmt.service
```

This namespace is defined in *osgi.wiring.rmt.service Namespace* on page 14.

To access the wiring, each Bundle node has a Wires node. This is a MAP of LIST of Wire. The key of the MAP node is the name of the namespace, that is, the wires are organized by namespace. This provides convenient access to all wires of a given namespace. The value of the MAP node is a LIST node, providing sequential access to the actual wires.

A Wire node provides the following information:

- Namespace – The namespace of the wire
- Requirement – The requirement that cause the wire
- Capability – The capability that satisfied the wire
- Requirer – The location of the bundle that required the wire
- Provider – The location of the bundle that satisfied the requirement

2.3.4 Wiring Example

The following example code demonstrates how the wires can be printed out:

```
String prefix ="Bundle/my_bundle/Wires/osgi.wiring.package";
String [] wires = session.getChildNodeNames(prefix);
for ( String wire : wires ) {
  String name = session.getNodeValue(prefix + "/"
     + wire + "/Capability/Attribute/osgi.wiring.package" ).getString();
  String provider = session.getNodeValue(prefix + "/"
     + wire + "/Provider" ).getString();
  String requirer = session.getNodeValue(prefix + "/"
     + wire + "/Requirer" ).getString();
```

```
    System.out.printf("%-20s %-30s %s\n", name, provider, requirer);
  }
```

2.4 Filtering

Frequently it is necessary to search through the tree of nodes for nodes matching specific criteria. Having to use Java to do this filtering can become cumbersome and impossible if the searching has to happen remotely. For that reason, the RMT contains a Filternode. This node allows a manager to specify a Target and a Filter. The Target is an absolute URI that defines a set of nodes that the Filter Plugin must search. This set is defined by allowing wildcards in the target. A single asterisk ('*' \u002A) matches a single level, the minus sign ('-' \u002C) specifies any number of levels and must not be used at the end of the URI. This implies that there is always a *final node*. The reason that a minus sign must not be last is that the final node's type would be undefined, any node on any sub-level would match.

The Target node must be specified as an absolute URI that must always end in a slash to signify that it represents a path to an interior node. The URI is absolute because the Filter is specified in a persistent node. It is possible to open a session, create the filter specification, close the session, and then open a new session, and use the earlier specified Target. As the two involved session do not have to have the same session, the base could differ, making it hard to use relative addressing. However, the result is always unique to a session. It is therefore possible to use relative URIs in the read out of the result.

For example, the tree in Figure 2.3 defines a sub-tree.

Figure 2.3 Example Sub-Tree

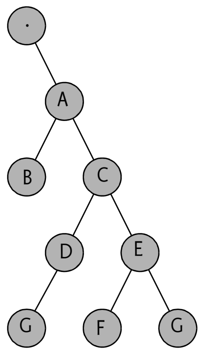


Table 2.1 shows a number of example targets on the previous sub-tree and their resulting final nodes, assuming the result is read in a session open on ./A.

Table 2.1 Example Target and results on a session opened on ./A

Target	Final nodes
./A/*/	B, C
./A/*/E/*/	C/E/F, C/E/G
./A/-/G/	C/D/G, C/E/G
./A/*/*/*/	C/D/G, C/E/F, C/E/G
./A/-/*/	This is an error, ./A/-/*/ is the same as ./A/-/, which is not allowed.
./A/*/*/	C/D, C/E

The Filter specifies a standard OSGi Filter expression that is applied to the final nodes. If no filter is specified then all final nodes match. However, when there is a filter specified it is applied against the final node and only the final nodes that are matching the filter as included in the result.

A node is matched against a filter by using some of its children as properties. The properties of a node are defined by:

- Primitive child nodes, or
- LIST nodes that have primitive as child nodes. Such nodes must be treated as multi-valued properties.

The matching rules in the filter must follow the standard OSGi Filter rules. If the filter matches such a node then it must be available as a session relative URI in the ResultUriList node. The relative URIs are listed in the ResultUriList.

The result nodes must only include nodes that satisfy the following conditions:

- The node must match the Target node's URI specification
- The node must be visible in the current session
- The node must not reside in the Filter sub-tree
- The node must be an interior node
- The caller must have access to the node
- It must be possible to get all the values of the child nodes that are necessary for filter matching
- The node must match the filter if a filter is specified

The result is also available as a sub-tree under the Resultnode and can be traversed as sub-tree in Result. This tree contains all the result nodes and their sub-tree. The results under the Result node are a snapshot and cannot be modified, they are read only. This result can be removed after the session is closed.

2.4.1 Example

For example, the following code prints out the location of active bundles:

```
session.createInteriorNode("Filter/mq-1");
session.setNodeValue("Filter/mq-1/Target",
  new DmtData($+"/Framework/Bundle/*/"));
session.setNodeValue("Filter/mq-1/Filter", new DmtData("(AutoStart=true)"));

String[] autostarted = session.getChildNodeNames(
   "Filter/mq-1/Result/Framework/Bundle");
System.out.println("Auto started bundles");
for ( String location : autostarted)
  System.out.println(location);

session.deleteNode("Filter/mq-1");
```

2.5 Log Access

The Log node provides access to the Log Service, the node contains a LIST of LogEntry nodes. The length of this list is implementation dependent. The list is sorted in most recent first order. This allows a manager to retrieve the latest logs. For example, the following code print out the latest 100 log entries:

```
DataSession session = admin.getSession($+"/Log/LogEntries");
try {
  for ( int i =0; i<100; i++ ) {
    Date date = session.getNodeValue( i+"/Time").getDateTime();
    String message = session.getNodeValue( i+"/Message").getString();
    System.out.println( date + " " + message );
  }
} finally {
  session.close();
}
```

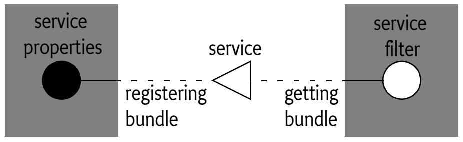
2.6 osgi.wiring.rmt.service Namespace

This section defines a namespace for the Requirement-Capability model to maintain services through the standard wiring API. A service is a capability, the Capability attributes are the service properties. The bundle that gets the service has a requirement on that service.

The filter of the service requirement is not the original filter since this is not possible to obtain reliably. Instead the filter must assert of the service.id, for example: (service.id=123).

The resulting filter is specified as the filter: directive on the Requirement. This is depicted in Figure 2.4.

Figure 2.4 Requirements and Capabilities and their Wiring



The osgi.wiring.rmt.service attributes are defined in Table 2.2:

Table 2.2 osgi.wiring.rmt.service namespace

Attribute Name	Type	Syntax	Description
osgi.wiring.rmt.service	String	service.id	The service id.
objectClass	String[]	fqn	Fully qualified name of the types under which this service is listed
*	*	*	Any service property

2.7 Tree Summary

2.7.1 Framework

```
$                                     _G__  NODE                  1   P

  Framework                           _G__  NODE                  1   P
    StartLevel                        _GR_  integer               1   A
    InitialBundleStartLevel           _GR_  integer               1   A

                                                       org.osgi/1.0/MAP
    Bundle                            _G__  MAP                   1   A
      [string]                        AG__  NODE               0..*   D
        State                         _G__  string              0,1   A
        StartLevel                    _GR_  integer               1   A
        InstanceId                    _G__  integer               1   A
        URL                           _GR_  string                1   A
        AutoStart                     _GR_  boolean               1   A
```

```
FaultType                    _G__  integer              1   A
FaultMessage                 _G__  string               1   A
BundleId                     _G__  long               0,1   A
SymbolicName                 _G__  string             0,1   A
Version                      _G__  string             0,1   A

                                         org.osgi/1.0/LIST
BundleType                   _G__  LIST               0,1   A
  [list]                     _G__  string            0..*   D

                                          org.osgi/1.0/MAP
Headers                      _G__  MAP                0,1   A
  [string]                   _G__  string            0..*   D
Location                     _G__  string               1   A
RequestedState               _GR_  string               1   A
LastModified                 _G__  dateTime           0,1   A

                                          org.osgi/1.0/MAP
Wires                        _G__  MAP                0,1   A

                                         org.osgi/1.0/LIST
  [string]                   _G__  LIST              0..*   D
    [list]                   _G__  NODE              0..*   D
      Provider               _G__  string               1   A
      InstanceId             _G__  integer              1   A
      Namespace              _G__  string               1   A
      Requirement            _G__  NODE                 1   A
        Filter               _G__  string               1   A

                                          org.osgi/1.0/MAP
        Directive            _G__  MAP                  1   A
          [string]           _G__  string            0..*   D

                                          org.osgi/1.0/MAP
        Attribute            _G__  MAP                  1   A
          [string]           _G__  string            0..*   D
      Capability             _G__  NODE                 1   A

                                          org.osgi/1.0/MAP
        Directive            _G__  MAP                  1   A
          [string]           _G__  string            0..*   D

                                          org.osgi/1.0/MAP
        Attribute            _G__  MAP                  1   A
          [string]           _G__  string            0..*   D
      Requirer               _G__  string               1   A

                                         org.osgi/1.0/LIST
Signers                      _G__  LIST               0,1   A
  [list]                     _G__  NODE              0..*   D
    InstanceId               _G__  integer              1   A
    IsTrusted                _G__  boolean              1   A

                                         org.osgi/1.0/LIST
    CertificateChain         _G__  LIST                 1   A
```

```
                  [list]                          _G__  string                 0..*   D

                                                                    org.osgi/1.0/LIST
            Entries                               _G__  LIST                   0,1    A
              [list]                              _G__  NODE                   0..*   D
                InstanceId                        _G__  integer                  1    A
                Path                              _G__  string                   1    A
                Content                           _G__  binary                   1    A

                                                                     org.osgi/1.0/MAP
        Property                                  _G__  MAP                      1    A
          [string]                                _G__  string                 0..*   D
```

2.7.2 Filters

```
                                                                     org.osgi/1.0/MAP
      Filter                                      _G__  MAP                    0,1    P
        [string]                                  AG_D  NODE                   0..*   D
          Filter                                  _GR_  string                   1    A
          Target                                  _GR_  string                   1    A
          Limit                                   _GR_  integer                  1    A
          Result                                  _G__  Node                     1    A

                                                                    org.osgi/1.0/LIST
          ResultUriList                           _G__  LIST                     1    A
            [list]                                _G__  node_uri               0..*   D
          InstanceId                              _G__  integer                  1    A
```

2.7.3 Log

```
      Log                                         _G__  NODE                   0,1    P

                                                                    org.osgi/1.0/LIST
        LogEntries                                _G__  LIST                     1    A
          [list]                                  _G__  NODE                   0..*   D
            Bundle                                _G__  string                   1    A
            Time                                  _G__  dateTime                 1    A
            Level                                 _G__  integer                  1    A
            Message                               _G__  string                   1    A
            Exception                             _G__  string                 0,1    A
```

2.8 org.osgi.dmt.residential

2.8.1 $

The $ describes the root node for OSGi Residential Management. The path to this node is defined in the system property: org.osgi.dmt.residential.

Table 2.3 Sub-tree Description for $

Name	Act	Type	Card	S	Description
Filter	Get	MAP	0,1	P	The Filters node searches the nodes in a tree that correspond to a target URI and an optional filter expression. A new Filter is created by adding a node to the Filters node. The name of the node is chosen by the remote manager. If multiple managers are active they must agree on a scheme to avoid conflicts or an atomic sessions must be used to claim exclusiveness. Filter nodes are persistent but an implementation can remove the node after a suitable timeout that should at least be 1 hour. If this functionality is not supported on this device then the node is not present.
[String]	Add Del Get	Filter	0..*	D	
Framework	Get	Framework	1	P	The Framework node used to manage the local framework.
Log	Get	Log	0,1	P	Access to the optional Log. If this functionality is not supported on this device then the node is not present.

2.8.2 Bundle

The management node for a Bundle. It provides access to the life cycle control of the bundle as well to its metadata, resources, and wiring.

To install a new bundle an instance of this node must be created. Since many of the sub-nodes are not yet valid as the information from the bundle is not yet available. These nodes are marked to be optional and will only exists after the bundle has been really installed.

2.8.2.1 FRAGMENT = "FRAGMENT"

The type returned for a fragment bundle.

2.8.2.2 INSTALLED = "INSTALLED"

The Bundle INSTALLED state.

2.8.2.3 RESOLVED = "RESOLVED"

The Bundle RESOLVED state.

2.8.2.4 STARTING = "STARTING"

The Bundle STARTING state.

2.8.2.5 ACTIVE = "ACTIVE"

The Bundle ACTIVE state.

2.8.2.6 STOPPING = "STOPPING"

The Bundle STOPPING state.

2.8.2.7 UNINSTALLED = "UNINSTALLED"

The Bundle UNINSTALLED state.

Table 2.4 Sub-tree Description for Bundle

Name	Act	Type	Card	S	Description
URL	Get Set	string	1	A	The URL to download the archive from for this bundle. By default this is the empty string. In an atomic session this URL can be replaced to a new URL, which will trigger an update of this bundle during commit. If this value is set it must point to a valid JAR from which a URL can be downloaded, unless it is the system bundle. If it is the empty string no action must be taken except when it is the system bundle. If the URL of Bundle 0 (The system bundle) is replaced to any value, including the empty string, then the framework will restart. If both a the URL node has been set the bundle must be updated before any of the other aspects are handled like RequestedState and StartLevel.
AutoStart	Get Set	boolean	1	A	Indicates if this Bundle must be started when the Framework is started. If the AutoStart node is true then this bundle is started when the framework is started and its StartLevel is met. If the AutoStart node is set to true and the bundle is not started then it will automatically be started if the start level permits it. If the AutoStart node is set to false then the bundle must not be stopped immediately. If the AutoStart value of the System Bundle is changed then the operation must be ignored. The default value for this node is true
FaultType	Get	integer	0,1	A	The BundleException type associated with a failure on this bundle, -1 if no fault is associated with this bundle. If there was no Bundle Exception associated with the failure the code must be 0 (UNSPECIFIED). The FaultMessage provides a human readable message. Only present after the bundle is installed.
FaultMessage	Get	string	0,1	A	A human readable message detailing an error situation or an empty string if no fault is associated with this bundle. Only present after the bundle is installed.
BundleId	Get	long	0,1	A	The Bundle Id as defined by the getBundleId() method. If there is no installed Bundle yet, then this node is not present.
SymbolicName	Get	string	0,1	A	The Bundle Symbolic Name as defined by the Bundle getSymbolicName() method. If this result is null then the value of this node must be the empty string. If there is no installed Bundle yet, then this node is not present.
Version	Get	string	0,1	A	The Bundle's version as defined by the Bundle getVersion() method. If there is no installed Bundle yet, then this node is not present.

Table 2.4 Sub-tree Description for Bundle

Name	Act	Type	Card	S	Description
BundleType	Get	LIST	0,1	A	A list of the types of the bundle. Currently only a single type is provided:
[list]	Get	string	0..*	D	FRAGMENT If there is no installed Bundle yet, then this node is not present.
Headers	Get	MAP	0,1	A	The Bundle getHeaders() method. If there is no installed Bundle yet, then this node is not
[String]	Get	string	0..*	D	present.
Location	Get	string	1	A	The Bundle's Location as defined by the Bundle getLocation() method. The location is specified by the management agent when the bundle is installed. This location should be a unique name for a bundle chosen by the management system. The Bundle Location is immutable for the Bundle's life (it is not changed when the Bundle is updated). The Bundle Location is also part of the URI to this node.
State	Get	string	0,1	A	Return the state of the current Bundle. The values can be: INSTALLED RESOLVED STARTING ACTIVE STOPPING If there is no installed Bundle yet, then this node is not present. The default value is UNINSTALLED after creation.

Table 2.4 Sub-tree Description for Bundle

Name	Act	Type	Card	S	Description
RequestedState	Get Set	string	1	A	Is the requested state the manager wants the bundle to be in. Can be: INSTALLED - Ensure the bundle is stopped and refreshed. RESOLVED - Ensure the bundle is resolved. ACTIVE - Ensure the bundle is started. UNINSTALLED - Uninstall the bundle. The Requested State is a request. The management agent must attempt to achieve the desired state but there is a no guarantee that this state is achievable. For example,a Framework can resolve a bundle at any time or the active start level can prevent a bundle from running. Any errors must be reported on FaultType and FaultMessage. If the AutoStart node is true then the bundle must be persistently started, otherwise it must be transiently started. If the StartLevel is not met then the commit must fail if AutoStart is false as a Bundle cannot be transiently started when the start level is not met. If both a the URL node has been set as well as the RequestedState node then this must result in an update after which the bundle should go to the RequestedState. The RequestedState must be stored persistently so that it contains the last requested state. The initial value of the RequestedState must be INSTALLED.
StartLevel	Get Set	integer	1	A	The Bundle's current Start Level as defined by the BundleStartLevel adapt interface getStartLevel() method. Changing the StartLevel can change the Bundle State as a bundle can become eligible for starting or stopping. If the URL node is set then a bundle must be updated before the start level is set,
LastModified	Get	datetime	0,1	A	The Last Modified time of this bundle as defined by the Bundle getlastModified() method. If there is no installed Bundle yet then this node is not present.

Table 2.4 Sub-tree Description for Bundle

Name	Act	Type	Card	S	Description
Wires	Get	MAP	0,1	A	A MAP of name space -> to Wire. A Wire is a relation between to bundles where the type of the relation is defined by the name space. For example, osgi.wiring.package name space defines the exporting and importing of packages. Standard osgi name spaces are: osgi.wiring.bundle osgi.wiring.package osgi.wiring.host As the Core specification allows custom name spaces this list can be more extensive. This specification adds one additional name space to reflect the services, this is the osgi.wiring.service name space. This name space will have a wire for each time a registered service by this Bundle was gotten for the first time by a bundle. A capability in the service name space holds all the registered service properties. The requirement has no attributes and a single filter directive that matches the service id property. If there is no installed Bundle yet then this node is not present.
[String]	Get	LIST	0..*	D	
[list]	Get	Wire	0..*	D	
Signers	Get	LIST	0,1	A	Return all signers of the bundle. See the Bundle getSignerCertificates() method with the SIGNERS_ALL parameter. If there is no installed Bundle yet then this node is not present.
[list]	Get	Certificate	0..*	D	
Entries	Get	LIST	0,1	A	An optional node providing access to the entries in the Bundle's JAR. This list must be created from the Bundle getEntryPaths() method called with an empty String. For each found entry, an Entry object must be made available. If there is no installed Bundle yet then this node is not present.
[list]	Get	Entry	0..*	D	
InstanceId	Get	integer	1	A	Instance Id used by foreign protocol adapters as a unique integer key not equal to 0. The instance id for a bundle must be (Bundle Id % 2^32) + 1.

2.8.3 Bundle.Certificate

Place holder for the Signers DN names.

Table 2.5 Sub-tree Description for Bundle.Certificate

Name	Act	Type	Card	S	Description
IsTrusted	Get	boolean	1	A	Return if this Certificate is trusted.

Table 2.5 Sub-tree Description for Bundle.Certificate

Name	Act	Type	Card	S	Description
CertificateChain	Get	LIST	1	A	A list of signer DNs of the certificates in the chain.
[list]	Get	string	0..*	D	
InstanceId	Get	integer	1	A	Instance Id to allow addressing by Instance Id.

2.8.4 Bundle.Entry

An Entry describes an entry in the Bundle, it combines the path of an entry with the content. Only entries that have content will be returned, that is, empty directories in the Bundle's archive are not returned.

Table 2.6 Sub-tree Description for Bundle.Entry

Name	Act	Type	Card	S	Description
Path	Get	string	1	A	The path in the Bundle archive to the entry.
Content	Get	binary	1	A	The binary content of the entry.
InstanceId	Get	integer	1	A	Instance Id to allow addressing by Instance Id.

2.8.5 Filter

A Filter node can find the nodes in a given sub-tree that correspond to a given filter expression. This Filter node is a generic mechanism to select a part of the sub-tree (*except* itself).

Searching is done by treating an interior node as a map where its leaf nodes are attributes for a filter expression. That is, an interior node matches when a filter matches on its children. The matching nodes' URIs are gathered under a `ResultUriList` node and as a virtual sub-tree under the `Result` node.

The Filter node can specify the `Target` node. The `Target` is an absolute URI ending in a slash, potentially with wild cards. Only nodes that match the target node are included in the result.

There are two different wild cards:

- *Asterisk* — (\\u002A '*') Specifies a wild card for one interior node name only. That is `A/*/` matches an interior nodes `A/B`, `A/C`, but not `A/X/Y`. The asterisk wild card can be used anywhere in the URI like `A/*/C`. Partial matches are not supported, that is a URI like `A/xyz*` is invalid.
- *Minus sign ('-' \\u002A)* — Specifies a wildcard for any number of descendant nodes. This is `A/-/X/` matches `A/B/X`, `A/C/X`, but also `A/X`. Partial matches are not supported, that is a URI like `A/xyz-` is not supported. The - wild card must not be used at the last segment of a URI

The `Target` node selects a set of nodes `N` that can be viewed as a list of URIs or as a virtual sub-tree. The `Target` node is the virtual sub-tree (beginning at the session base) and the `ResultUriList` is a LIST of session relative URIs. The actual selection of the nodes must be postponed until either of these nodes (or one of their sub-nodes) is accessed for the first time. Either nodes represent a read-only snapshot that is valid until the end of the session.

It is possible to further refine the selection by specifying the Filter node. The Filter node is an LDAP filter expression or a simple wild card ('*') which selects all the nodes. As the wild card is the default, all nodes selected by the `Target` are selected by default.

The Filter must be applied to each of the nodes selected by target in the set N. By definition, these nodes are *interior nodes only*. LDAP expressions assert values depending on their *key*. In this case, the child leaf nodes of a node in set N are treated as the property on their parent node.

The attribute name in the LDAP filter can only reference a direct leaf node of the node in the set N or an interior node with the DDF type DmtConstants.DDF_LIST with leaf nodes as children, i.e. a *LIST*. A LIST of primitives must be treated in the filter as a multi valued property, any of its values satisfy an assertion on that attribute.

Attribute names must not contains a slash, that is, it is only possible to assert values directly below the node selected by the target.

Each of these leaf nodes and LISTs can be used in the LDAP Filter as a key/value pair. The comparison must be done with the type used in the Dmt Data object of the compared node. That is, if the Dmt Admin data is a number, then the comparison rules of the number must be used. The attributes given to the filter must be converted to the Java object that represents their type.

The set N must therefore consists only of nodes where the Filter matches.

It is allowed to change the Target or the Filter node after the results are read. In that case, the Result and ResultUriList must be cleared instantaneously and the search redone once either result node is read.

The initial value of Target is the empty string, which indicates no target.

Table 2.7 *Sub-tree Description for Filter*

Name	**Act**	**Type**	**Card**	**S**	**Description**
Target	Get Set	string	1	A	A URI always ending in a slash ('/'), relative the current session, with optional wildcards, selecting a set of sub-nodes N. Wildcards can be an asterisk (∗ '∗') or a minus sign (- '-'). An asterisk can be used in place of a single node name in the URI, a minus sign stands for any number of consecutive node names. The default value of this node is the empty string, which indicates that no nodes must be selected. Changing this value must clear any existing results. If the Result() or ResultUriList is read to get N then a new search must be executed. A URI must always end in '/' to indicate that the target can only select interior nodes.
Filter	Get Set	string	1	A	An optional filter expression that filters nodes in the set N selected by Target. The filter expression is an LDAP filter or an asterisk ('∗'). An asterisk is the default value and matches any node in set N. If an LDAP expression is set in the Filter node then the set N must only contain nodes that match the given filter. The values the filter asserts are the immediate leafs and LIST nodes of the nodes in set N. The name of these child nodes is the name of the attribute matched in the filter. The nodes can be removed by the Filter implementation after a timeout defined by the implementation.

Table 2.7 Sub-tree Description for Filter

Name	Act	Type	Card	S	Description
Limit	Get Set	integer	1	A	Limits the number of results to the given number. If this node is not set there is no limit. The default value is not set, thus no limit.
Result	Get	NODE	1	A	The Result tree is a virtual read-only tree of all nodes that were selected by the Target and matched the Filter, that is, all nodes in set N. The Target contains a relative URI (with optional wildcards) from the parent of the Filters node. The Result node acts as the parent of this same relative path for each node in N. The Result node is a snapshot taken the first time it is accessed after a change in the Filter and/or the Target nodes.
ResultUriList	Get	LIST	1	A	A list of URIs of nodes in the Device Management Tree from the node selected by the Target
[list]	Get	node_uri	0..*	D	that match the Filter node. All URIs are relative to current session. The Result node is a snapshot taken the first time it is accessed after a change in the Filter and/or the Target nodes.
InstanceId	Get	integer	1	A	Instance Id to allow addressing by Instance Id.

2.8.6 Framework

The Framework node represents the information about the Framework itself. The Framework node allows manipulation of the OSGi framework, start level, framework life cycle, and bundle life cycle.

All modifications to a Framework object must occur in an atomic session. All changes to the framework must occur during the commit.

The Framework node allows the manager to install (create a new child node in Bundle), to uninstall change the state of the bundle (see Bundle.RequestedState()), update the bundle (see URL), start/stop bundles, and update the framework. The implementation must execute these actions in the following order during the commit of the session:

1 Create a snapshot of the current installed bundles and their state.
2 stop all bundles that will be uninstalled and updated
3 Uninstall all the to be uninstalled bundles (bundles whose RequestedState is Bundle.UNINSTALLED)
4 Update all bundles that have a modified URL with this URL using the Bundle update(InputStream) method in the order that the order that the URLs were last set.
5 Install any new bundles from their URL in the order that the order that the URLs were last set.
6 Refresh all bundles that were updated and installed
7 Ensure that all the bundles have their correct start level
8 If the RequestedState was set, follow this state. Otherwise ensure that any Bundles that have the AutoStart flag set to true are started persistently. Transiently started bundles that were stopped in this process are not restarted. The bundle id order must be used.
9 Wait until the desired start level has been reached
10 Return from the commit without error.

If any of the above steps runs in an error (except the restart) than the actions should be undone and the system state must be restored to the snapshot.

If the System Bundle was updated (its URL) node was modified, then after the commit has returned successfully, the OSGi Framework must be restarted.

Table 2.8 Sub-tree Description for Framework

Name	Act	Type	Card	S	Description
StartLevel	Get Set	integer	1	A	The StartLevel manages the Framework's current Start Level. Maps to the Bundle Start Level set/getStartLevel() methods. This node can set the requested Framework's StartLevel, however it doesn't store the value. This node returns the Framework's StartLevel at the moment of the call.
InitialBundleStart Level	Get Set	integer	1	A	Configures the initial bundle start level, maps to the the FrameworkStartLevel set/ getInitialBundleStartLevel() method.
Bundle	Get	MAP	1	A	The MAP of location -> Bundle. Each Bundle is uniquely identified by its location. The loca-
[String]	Add Get	Bundle	0..*	D	tion is a string that must be unique for each bundle and can be chosen by the management system. The Bundles node will be automatically filled from the installed bundles, representing the actual state. New bundles can be installed by creating a new node with a given location. At commit, this bundle will be installed from their Bundle.URL node. The location of the System Bundle must be "System Bundle" (see the Core's Constants.SYSTEM_BUNDLE_LOCATION), this node cannot be uninstalled and most operations on this node have special meaning. It is strongly recommended to use a logical name for the location of a bundle, for example reverse domain names or a UUID. To uninstall a bundle, set the Bundle.RequestedState to UNINSTALLED, the nodes in Bundle cannot be deleted.
Property	Get	MAP	1	A	The Framework Properties. The Framework properties come from the Bundle Context
[String]	Get	string	0..*	D	getProperty() method. However, this method does not provide the names of the available properties. If the handler of this node is aware of the framework properties then these should be used to provide the node names. If these properties are now known, the handler must synthesize the names from the following sources System Properties (as they are backing the Framework properties) Launching properties as defined in the OSGi Core specification Properties in the residential specification Other known properties

2.8.7 Wire

A Wire is a link between two bundles where the semantics of this link is defined by the used name space. This is closely modeled after the Wiring API in the Core Framework.

Table 2.9 Sub-tree Description for Wire

Name	Act	Type	Card	S	Description
Namespace	Get	string	1	A	The name space of this wire. Can be: osgi.wiring.bundle - Defined in the OSGi Core osgi.wiring.package - Defined in the OSGi Core osgi.wiring.host - Defined in the OSGi Core osgi.wiring.rmt.service - Defined in this specification * - Generic name spaces The osgi.wiring.rmt.service name space is not defined by the OSGi Core as it is not part of the module layer. The name space has the following layout: Requirement - A filter on the service.id service property. Capability - All service properties as attributes. No defined directives. Requirer - The bundle that has gotten the service Provider - The bundle that has registered the service There is a wire for each registration-get pair. That is, if a service is registered by A and gotten by B and C then there are two wires: B->A and C->A.
Requirement	Get	Requirement	1	A	The Requirement that caused this wire.
Capability	Get	Capability	1	A	The Capability that satisfied the requirement of this wire.
Requirer	Get	string	1	A	The location of the Bundle that contains the requirement for this wire.
Provider	Get	string	1	A	The location of the Bundle that provides the capability for this wire.
InstanceId	Get	integer	1	A	Instance Id to allow addressing by Instance Id.

2.8.8 Wire.Capability

Describes a Capability.

Table 2.10 Sub-tree Description for Wire.Capability

Name	Act	Type	Card	S	Description
Directive	Get	MAP	1	A	The Directives for this requirement.
[String]	Get	string	0..*	D	

Table 2.10 Sub-tree Description for Wire.Capability

Name	Act	Type	Card	S	Description
Attribute	Get	MAP	1	A	The Attributes for this capability.
[String]	Get	string	0..*	D	

2.8.9 Wire.Requirement

Describes a Requirement.

Table 2.11 Sub-tree Description for Wire.Requirement

Name	Act	Type	Card	S	Description
Filter	Get	string	1	A	The Filter string for this requirement.
Directive	Get	MAP	1	A	The Directives for this requirement. These directives must contain the filter: directive as described by the Core.
[String]	Get	string	0..*	D	
Attribute	Get	MAP	1	A	The Attributes for this requirement.
[String]	Get	string	0..*	D	

2.9 org.osgi.dmt.service.log

2.9.1 Log

Provides access to the Log Entries of the Log Service.

Table 2.12 Sub-tree Description for Log

Name	Act	Type	Card	S	Description
LogEntries	Get	LIST	1	A	A potentially long list of Log Entries. The length of this list is implementation dependent. The order of the list is most recent event at index 0 and later events with higher consecutive indexes. No new entries must be added to the log when there is an open exclusive or atomic session.
[list]	Get	LogEntry	0..*	D	

2.9.2 LogEntry

A Log Entry node is the representation of a LogEntry from the OSGi Log Service.

Table 2.13 Sub-tree Description for LogEntry

Name	Act	Type	Card	S	Description
Time	Get	datetime	1	A	Time of the Log Entry.
Level	Get	integer	1	A	The severity level of the log entry. The value is the same as the Log Service level values: LOG_ERROR 1 LOG_WARNING 2 LOG_INFO 3 LOG_DEBUG 4 Other values are possible because the Log Service allows custom levels.
Message	Get	string	1	A	Textual, human-readable description of the log entry.
Bundle	Get	string	1	A	The location of the bundle that originated this log or an empty string.
Exception	Get	string	0,1	A	Human readable information about an exception. Provides the exception information if any, optionally including the stack trace.

2.10 References

[1] *OSGi Service Platform Core Specification,Release 4, Version 4.3*
http://www.osgi.org/Specifications/HomePage

3 TR-157 Amendment 3 Software Module Guidelines

[1] *Broadband Forum* (BBF) has defined an object model for managing the software modules in a CPE. The BBF Software Modules object defines Execution Environments, Deployment Units, and Execution Units. These concepts are mapped in Table 3.1.

Table 3.1 Mapping of concepts

Software Modules Concept	OSGi Concept
Execution Environment	OSGi Framework
Deployment Unit	Bundle
Execution Unit	Bundle

There can be multiple Execution Environments of the same or different types. The parent Execution Environment is either the native environment, for example Linux, or it can be another Framework. A BBF Deployment Unit and Execution Unit both map to a bundle since there is no need to separate those concepts in OSGi. An implementation of this object model should have access to all the Execution Environments as the Deployment Units and Execution Units are represented in a single table.

This section is not a specification in the normal sense. The intention of this chapter is to provide guidelines for implementers of the [4] *TR-157a3 Internet Gateway Device Software Modules* on an OSGi Service Platform.

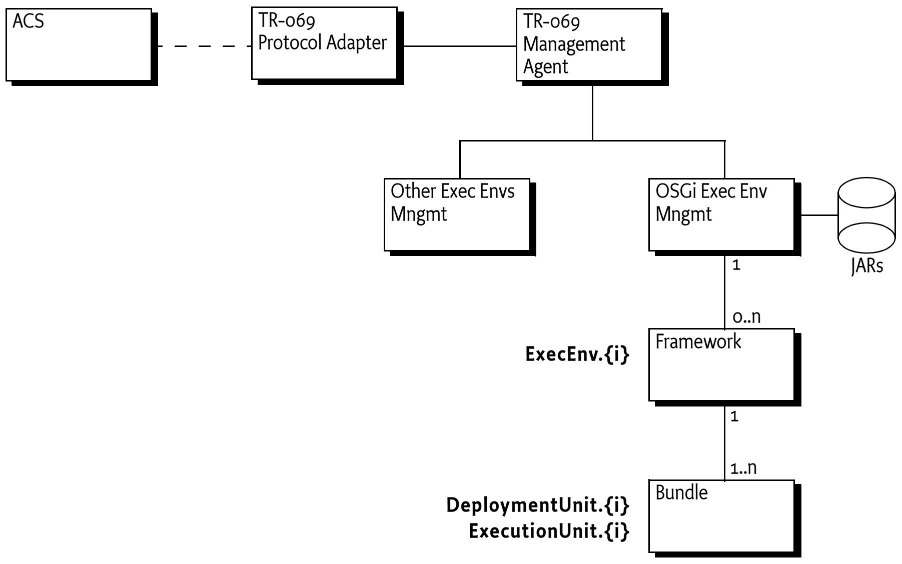
3.1 Management Agent

The Broadband Forum TR-157 Software Modules standard provides a uniform view of the different execution environments that are available in a device. Execution Environments can model the underlying operating system, an OSGi framework, or other environments that support managing the execution of code.

Most parameters in the Software Modules object model map very well to their OSGi counter parts. However, there are a number of issues that require support from a *management agent*. This management agent must maintain state to implement the contract implied by the Software Modules standard. For example, the OSGi Framework does not have an Initial Start Level, an OSGi Framework always starts at an environment property defined start level. However, the standard requires that a Framework must start at a given level after it is launched.

There are many other actions that require a management agent to provide the functionality required by TR-157 that is not build into the OSGi Framework since the standard requires a view that covers the whole device, not just the OSGi environment. The assumed architecture is depicted in Figure 3.1.

Figure 3.1 *Management Agent Architecture*



3.2 Parameter Mapping

Table 3.2 provides OSGi specific information for the different parameters in the Software Modules object model.

Table 3.2 *OSGi Specific Information for the BBF Software Modules object model*

TR-069 Software Module Object Parameter	**Mapping in case of OSGi**
Device.SoftwareModules.	
ExecEnvNumberOfEntries	
DeploymentUnitNumberOfEntries	
ExecutionUnitNumberOfEntries	
Device.SoftwareModules.ExecEnv.{i}.	
Enable	Indicates whether or not this OSGi Framework is enabled. Disabling an enabled OSGi Framework must stop it, while enabling a disabled OSGi Framework must launch it. When an Execution Environment is disabled, Bundles installed in that OSGi Framework will be unaffected, but any Bundles on that OSGi Framework are automatically made inactive. When an OSGi Framework is disabled it is impossible to make changes to the installed bundles, install new bundles, or query any information about the bundles. Disabling the OSGi Framework could place the device in a non-manageable state. For example, if the OSGi Framework runs the Protocol Adapter or has a management agent then it is possible that the device can no longer be restarted.
Status	Indicates the status of the OSGi Framework. Enumeration of: • Up – The OSGi Framework is up and running. • Error – The OSGi Framework could not be launched. • Disabled – The OSGi Framework is not enabled

Table 3.2 OSGi Specific Information for the BBF Software Modules object model

TR-069 Software Module Object Parameter	Mapping in case of OSGi
Reset	Setting this parameter to true causes this OSGi Framework to revert back to the state it was in when the device last issued a 0 BOOTSTRAP Inform event (bootstrap). The following requirements dictate what must happen for the reset to be complete: • The system must restore the set of bundles that were present at the last bootstrap event. That means that installed bundles since that moment must be uninstalled, updated bundles rolled back, and uninstalled bundles re-installed. • The OSGi Framework must roll back to the version it had during the previous rollback. • The OSGi Framework must be restarted after the previous requirements have been met. The value of this parameter is not part of the device configuration and is always false when read.
Alias	A non-volatile handle used to reference this instance for alias based addressing.
Name	A Name that adequately distinguishes this OSGi Framework from all other OSGi Frameworks. This must be the OSGi Framework UUID as stored in the org.osgi.framework.uuid property.
Type	Indicates the complete type and specification version of this ExecEnv. For an OSGi Framework it must be: `OSGi <version>` Where the <version> is the value of the framework property org.osgi.framework.version
InitialRunLevel	The run level that this ExecEnv will be in upon startup (whether that is caused by a CPE Boot or the Execution Environment starting). Run levels map to directly OSGi start levels. However, the OSGi Framework has no concept of an initial start level, it can use the org.osgi.framework.startlevel.beginning environment property but this requires a management to control it. A management agent must therefore handle this value and instruct the OSGi Framework to move to this start level after a reboot. If the value of CurrentRunLevel is set to -1, then the value of this parameter is irrelevant when read. Setting its value to -1 must have no impact on the start level of this OSGi Framework.
RequestedRunLevel	Sets the start level of this OSGi Framework, meaning that altering this parameter's value will change the value of the CurrentRunLevel asynchronously. Start levels dictate which Bundles will be started. Setting this value when CurrentRunLevel is -1 must have no impact on the start Level of this OSGi Framework. The value of this parameter is not part of the device configuration and must always be -1 when read.
CurrentRunLevel	The start level that this OSGi Framework is currently operating in. This value is altered by changing the RequestedRunLevel parameter. Upon startup (whether that is caused by a CPE Boot or the Execution Environment starting) CurrentRunLevel must be set equal to InitialRunLevel by some management agent. If Run Levels are not supported by this OSGi Framework then CurrentRunLevel must be -1.

Table 3.2 OSGi Specific Information for the BBF Software Modules object model

TR-069 Software Module Object Parameter	Mapping in case of OSGi
Version	The Version of this OSGi Framework as specified by its Vendor. This is not the version of its specification. Must be the value of the System Bundle's getVersion() method.
Vendor	The vendor that produced this OSGi Framework, the value of the org.osgi.framework.vendor Framework property.
ParentExecEnv	The value must be the path name of a row in the ExecEnv table, it can either be the operating system or another OSGi Framework if the framework is nested. If the referenced object is deleted, the parameter value must be set to an empty string. If this value is an empty string then this is the *Primary Execution Environment*.
AllocatedDiskSpace	Implementation specific.
AvailableDiskSpace	Implementation specific.
AllocatedMemory	Implementation specific.
AvailableMemory	Implementation specific.
ProcessorRefList	Comma-separated list of paths into the DeviceInfo.Processor table. If the referenced object is deleted, the corresponding item must be removed from the list. Represents the processors that this OSGi Framework has available to it.
ActiveExecutionUnits	Comma-separated list of paths into the ExecutionUnit table. If the referenced object is deleted, the corresponding item must be removed from the list. Represents the Bundles currently active on this OSGi Framework.
Device.SoftwareModules. DeploymentUnit.{i}.	This table serves as the Bundles inventory and contains status information about each Bundle. A new instance of this table gets created during the installation of a Bundle.
UUID	A Universally Unique Identifier either provided by the ACS, or generated by the CPE, at the time of Deployment Unit Installation. The format of this value is defined by [2] *RFC 4122 A Universally Unique IDentifier (UUID) URN Namespace* Version 3 (Name-Based) and [5] *TR-069a3 CPE WAN Management Protocol*. This value must not be altered when the Bundle is updated. A management agent should use the UUID as the bundle location since the location plays the same role.
DUID	The Bundle id from the getBundleId() method.
Alias	A non-volatile handle used to reference this instance.
Name	Indicates the Bundle Symbolic Name of this Bundle. The value of this parameter is used in the generation of the UUID based on the rules defined in [5] *TR-069a3 CPE WAN Management Protocol*.

Table 3.2 OSGi Specific Information for the BBF Software Modules object model

TR-069 Software Module Object Parameter	**Mapping in case of OSGi**
Status	Indicates the status of this Bundle. Enumeration of: • Installing – This bundle is in the process of being Installed and should transition to the Installed state. This state will never be visible in an OSGi Framework. • Installed – This bundle has been successfully installed.This maps to the Bundle INSTALLED or RESOLVED state. • Updating – This bundle is in the process of being updated and should transition to the Installed state. This state will never be visible in an OSGi Framework. • Uninstalling – This bundle is in the process of being uninstalled and should transition to the uninstalled state.This state will never be visible in an OSGi Framework. • Uninstalled – This bundle has been successfully uninstalled. This state will never be visible in an OSGi Framework.
Resolved	Indicates whether or not this DeploymentUnit has resolved all of its dependencies. Must be true if this Bundle's state is ACTIVE, STARTING, STOPPING, or RESOLVED. Otherwise it must be false.
URL	Contains the URL used by the most recent ChangeDUState RPC to either Install or Update this Bundle. This must be remembered by a management agent since this information is not available in a Bundle.
Description	Textual description of this Bundle, must be the value of the Bundle-Description manifest header or an empty string if not present.
Vendor	The author of this DeploymentUnit formatted as a domain name. The value of this parameter is used in the generation of the UUID based on the rules defined in [5] *TR-069a3 CPE WAN Management Protocol*. The recommended value is the value of the Bundle-Vendor header.
Version	Version of this Bundle, it mist be he value of the geVersion() method.
VendorLogList	Empty String
VendorConfigList	Empty String
ExecutionUnitList	A path into the ExecutionUnit table for the corresponding ExecutionUnit for this Bundle, which is also the bundle since the relation is 1:1.
ExecutionEnvRef	The value must be the path name of a row in the ExecEnv table of the corresponding OSGi Framework.
Device.SoftwareModules.ExecutionUnit.{i}.	This table serves as the Execution Unit inventory and contains both status information about each Execution Unit as well as configurable parameters for each Execution Unit. This list contains all the bundles since in an OSGi Framework Deployment Unit and Execution Unit are mapped to Bundles.
EUID	Table wide identifier for a bundle chosen by the OSGi Framework during installation of the associated DeploymentUnit. The value must be unique across ExecEnv instances. It is recommended that this be a combination of the ExecEnv.{i}.Name and an OSGi Framework local unique value. The unique value for an OSGi framework should be the Bundle Location.
Alias	A non-volatile handle used to reference this instance.
Name	The name should be unique across all Bundles instances contained within its associated DeploymentUnit. As the Deployment Unit and the Execution Unit are the same the value must be the Bundle Symbolic Name.

Table 3.2 OSGi Specific Information for the BBF Software Modules object model

TR-069 Software Module Object Parameter	Mapping in case of OSGi
ExecEnvLabel	The name must be unique across all Bundles contained within a specific OSGi Framework. This must therefore be the Bundle Id.
AutoStart	If true and the proper start level is met, then this Bundle will be automatically started by the device after its OSGi Framework's start level is met. If false this Bundle must not be started after launch until it is explicitly commanded to do so. An OSGi bundle is persistently started or transiently started. It is not possible to change this state without affecting the active state of the bundle. Therefore, if the AutoStart is set to true, the bundle must be started persistently, even if it is already started. This will record the persistent start state. If the AutoStart is set to false, the bundle must be stopped. Therefore, in an OSGi Framework setting the AutoStart flag to true has the side effect that the bundle is started if it was not active; setting it to false will stop the bundle.
RunLevel	Determines when this Bundle will be started. If AutoStart is true and the CurrentRunLevel is greater than or equal to this RunLevel, then this ExecutionUnit must be started, if run levels are enabled. This maps directly to the Bundles start level.
Status	Indicates the status of this ExecutionUnit. Enumeration of: • Idle – This Bundle is in an Idle state and not running. This maps to the Bundle INSTALLED or Bundle RESOLVED state. • Starting – This Bundle is in the process of starting and should transition to the Active state. This maps to the STARTING state in OSGi. In an OSGi Framework, lazily activated bundles can remain in the STARTING state for a long time. • Active – This instance is currently running. This maps to the Bundle ACTIVE state. • Stopping – This instance is in the process of stopping and should transition to the Idle state.
RequestedState	Indicates the state transition that the ACS is requesting for this Bundle. Enumeration of: • Idle – If this Bundle is currently in STARTING or ACTIVE state then the CPE must attempt to stop the Bundle; otherwise this requested state is ignored. • Active – If this Bundle is currently in the INSTALLED or RESOLVED state the management agent must attempt to start the Bundle. If this ExecutionUnit is in the STOPPING state the request is rejected and a fault raised. Otherwise this requested state is ignored. If this Bundle is disabled and an attempt is made to alter this value, then a CWMP Fault must be generated. The value of this parameter is not part of the device configuration and is always an empty string when read. Bundles must be started transiently when the AutoStart is false, otherwise persistently.

Table 3.2 OSGi Specific Information for the BBF Software Modules object model

TR-069 Software Module Object Parameter	Mapping in case of OSGi
ExecutionFaultCode	If while running or transitioning between states this Bundle raises an Exception then this parameter embodies the problem. Enumeration of: • NoFault – No fault, default value. • FailureOnStart – Threw an exception when started. • FailureOnAutoStart – Failed to be started by the framework, this must be intercepted by the management agent because this is a Framework Error event. • FailureOnStop – Raised an exception while stopping • FailureWhileActive – Raised when a bundle cannot be restarted after a background operation of the Framework, for example refreshing. • DependencyFailure – Failed to resolve • UnStartable – Cannot be raised in OSGi since this is the same error as FailureOnStart. For fault codes not included in this list, the vendor can include vendor-specific values, which must use the format defined in Section 3.3 of [6] *TR-106a4 Data Model Template for TR-069-Enabled Devices*.
ExecutionFaultMessage	If while running or transitioning between states this Bundle identifies a fault this parameter provides a more detailed explanation of the problem enumerated in the ExecutionFaultCode. If ExecutionFaultCode has the value of NoFault then the value of this parameter must be an empty string and ignored. This message must be the message value of the exception thrown by the Bundle.
Vendor	Vendor of this Bundle. The value of the Bundle-Vendor manifest header
Description	Textual description of this Bundle. The value of the Bundle-Description manifest header
Version	Version of the Bundle. The value of the getVersion() method.
VendorLogList	Empty string.
VendorConfigList	Empty string.
DiskSpaceInUse	Implementation defined
MemoryInUse	Implementation defined
References	Empty String
AssociatedProcessList	Empty String as an OSGi bundle reuses the process of the VM.
SupportedDataModelList	Comma-separated list of strings. Each list item must be the path name of a row in the DeviceInfo.SupportedDataModel table. If the referenced object is deleted, the corresponding item must be removed from the list. Represents the CWMP-DT schema instances that have been introduced to this device because of the existence of this ExecutionUnit. In OSGi this is implementation defined.
ExecutionEnvRef	The path to the OSGi Framework that hosts this bundle in the ExecEnv table.
Device.SoftwareModules. ExecutionUnit.{i}.Extensions.	This object proposes a general location for vendor extensions specific to this Execution Unit, which allows multiple Execution Units to expose parameters without the concern of conflicting parameter names. This part is not used in OSGi.

3.3 References

[1] *Broadband Forum*
http://www.broadband-forum.org

[2] *RFC 4122 A Universally Unique IDentifier (UUID) URN Namespace*
http://tools.ietf.org/html/rfc4122

[3] *TR-157a3 Component Objects for CWMP*
http://www.broadband-forum.org/technical/download/TR-157_Amendment-3.pdf

[4] *TR-157a3 Internet Gateway Device Software Modules*
http://www.broadband-forum.org/cwmp/
tr-157-1-3-0-igd.html#D.InternetGatewayDevice.SoftwareModules

[5] *TR-069a3 CPE WAN Management Protocol*
http://www.broadband-forum.org/technical/download/TR-069_Amendment-3.pdf

[6] *TR-106a4 Data Model Template for TR-069-Enabled Devices*
http://www.broadband-forum.org/technical/download/TR-106_Amendment-4.pdf

101 Log Service Specification

Version 1.3

101.1 Introduction

The Log Service provides a general purpose message logger for the OSGi Service Platform. It consists of two services, one for logging information and another for retrieving current or previously recorded log information.

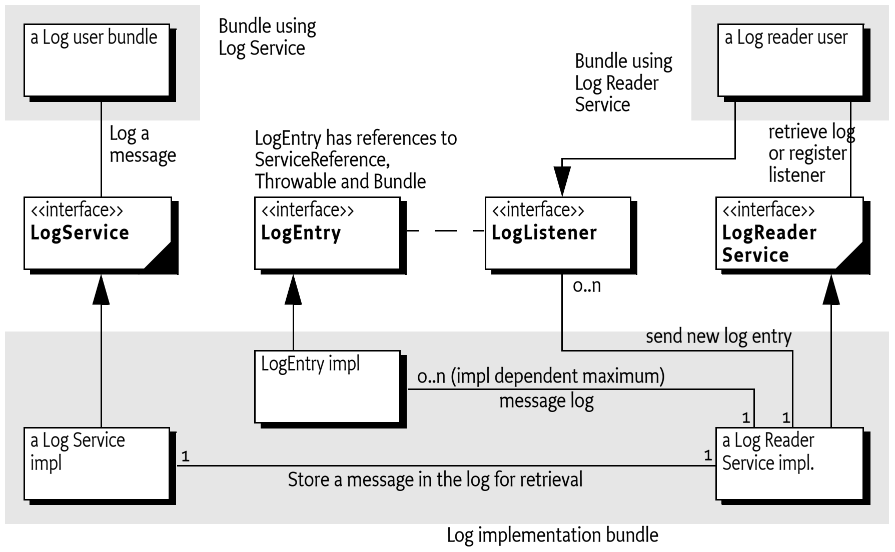
This specification defines the methods and semantics of interfaces which bundle developers can use to log entries and to retrieve log entries.

Bundles can use the Log Service to log information for the Operator. Other bundles, oriented toward management of the environment, can use the Log Reader Service to retrieve Log Entry objects that were recorded recently or to receive Log Entry objects as they are logged by other bundles.

101.1.1 Entities

- *LogService* – The service interface that allows a bundle to log information, including a message, a level, an exception, a `ServiceReference` object, and a `Bundle` object.
- *LogEntry* - An interface that allows access to a log entry in the log. It includes all the information that can be logged through the Log Service and a time stamp.
- *LogReaderService* - A service interface that allows access to a list of recent `LogEntry` objects, and allows the registration of a `LogListener` object that receives `LogEntry` objects as they are created.
- *LogListener* - The interface for the listener to `LogEntry` objects. Must be registered with the Log Reader Service.

Figure 101.1 Log Service Class Diagram org.osgi.service.log package


a Log user bundle
Bundle using Log Service
Bundle using Log Reader Service
a Log reader user
Log a message
LogEntry has references to ServiceReference, Throwable and Bundle
retrieve log or register listener
<<interface>> LogService
<<interface>> LogEntry
<<interface>> LogListener
<<interface>> LogReader Service
0..n
send new log entry
LogEntry impl
0..n (impl dependent maximum) message log
1
1
a Log Service impl
1
1
a Log Reader Service impl.
Store a message in the log for retrieval
Log implementation bundle

101.2 The Log Service Interface

The LogService interface allows bundle developers to log messages that can be distributed to other bundles, which in turn can forward the logged entries to a file system, remote system, or some other destination.

The LogService interface allows the bundle developer to:

- Specify a message and/or exception to be logged.
- Supply a log level representing the severity of the message being logged. This should be one of the levels defined in the LogService interface but it may be any integer that is interpreted in a user-defined way.
- Specify the Service associated with the log requests.

By obtaining a LogService object from the Framework service registry, a bundle can start logging messages to the LogService object by calling one of the LogService methods. A Log Service object can log any message, but it is primarily intended for reporting events and error conditions.

The LogService interface defines these methods for logging messages:

- log(int, String) – This method logs a simple message at a given log level.
- log(int, String, Throwable) – This method logs a message with an exception at a given log level.
- log(ServiceReference, int, String) – This method logs a message associated with a specific service.
- log(ServiceReference, int, String, Throwable) – This method logs a message with an exception associated with a specific service.

While it is possible for a bundle to call one of the log methods without providing a ServiceReference object, it is recommended that the caller supply the ServiceReference argument whenever appropriate, because it provides important context information to the operator in the event of problems.

The following example demonstrates the use of a log method to write a message into the log.

```
logService.log(
   myServiceReference,
   LogService.LOG_INFO,
   "myService is up and running"
);
```

In the example, the myServiceReference parameter identifies the service associated with the log request. The specified level, LogService.LOG_INFO, indicates that this message is informational.

The following example code records error conditions as log messages.

```
try {
   FileInputStream fis = new FileInputStream("myFile");
   int b;
   while ( (b = fis.read()) != -1 ) {
      ...
   }
   fis.close();
}
catch ( IOException exception ) {
   logService.log(
      myServiceReference,
      LogService.LOG_ERROR,
      "Cannot access file",
      exception );
}
```

Notice that in addition to the error message, the exception itself is also logged. Providing this information can significantly simplify problem determination by the Operator.

101.3 Log Level and Error Severity

The log methods expect a log level indicating error severity, which can be used to filter log messages when they are retrieved. The severity levels are defined in the LogService interface.

Callers must supply the log levels that they deem appropriate when making log requests. The follow-

Table 101.1 Log Levels

Level	Descriptions
LOG_DEBUG	Used for problem determination and may be irrelevant to anyone but the bundle developer.
LOG_ERROR	Indicates the bundle or service may not be functional. Action should be taken to correct this situation.
LOG_INFO	May be the result of any change in the bundle or service and does not indicate a problem.
LOG_WARNING	Indicates a bundle or service is still functioning but may experience problems in the future because of the warning condition.

ing table lists the log levels.

101.4 Log Reader Service

The Log Reader Service maintains a list of LogEntry objects called the *log*. The Log Reader Service is a service that bundle developers can use to retrieve information contained in this log, and receive notifications about LogEntry objects when they are created through the Log Service.

The size of the log is implementation-specific, and it determines how far into the past the log entries go. Additionally, some log entries may not be recorded in the log in order to save space. In particular, LOG_DEBUG log entries may not be recorded. Note that this rule is implementation-dependent. Some implementations may allow a configurable policy to ignore certain LogEntry object types.

The LogReaderService interface defines these methods for retrieving log entries.

- getLog() – This method retrieves past log entries as an enumeration with the most recent entry first.
- addLogListener(LogListener) – This method is used to subscribe to the Log Reader Service in order to receive log messages as they occur. Unlike the previously recorded log entries, all log messages must be sent to subscribers of the Log Reader Service as they are recorded.
  A subscriber to the Log Reader Service must implement the LogListener interface.
  After a subscription to the Log Reader Service has been started, the subscriber's LogListener.logged method must be called with a LogEntry object for the message each time a message is logged.

The LogListener interface defines the following method:

- logged(LogEntry) – This method is called for each LogEntry object created. A Log Reader Service implementation must not filter entries to the LogListener interface as it is allowed to do for its log. A LogListener object should see all LogEntry objects that are created.

The delivery of LogEntry objects to the LogListener object should be done asynchronously.

101.5 Log Entry Interface

The LogEntry interface abstracts a log entry. It is a record of the information that was passed when an event was logged, and consists of a superset of information which can be passed through the LogService methods. The LogEntry interface defines these methods to retrieve information related to LogEntry objects:

- getBundle() – This method returns the Bundle object related to a LogEntry object.
- getException() – This method returns the exception related to a LogEntry object. In some implementations, the returned exception may not be the original exception. To avoid references to a bundle defined exception class, thus preventing an uninstalled bundle from being garbage collected, the Log Service may return an exception object of an implementation defined Throwable subclass. This object will attempt to return as much information as possible, such as the message and stack trace, from the original exception object .
- getLevel() – This method returns the severity level related to a LogEntry object.
- getMessage() – This method returns the message related to a LogEntry object.
- getServiceReference() –This method returns the ServiceReference object of the service related to a LogEntry object.
- getTime() – This method returns the time that the log entry was created.

101.6 Mapping of Events

Implementations of a Log Service must log Framework-generated events and map the information to LogEntry objects in a consistent way. Framework events must be treated exactly the same as other logged events and distributed to all LogListener objects that are associated with the Log Reader Service. The following sections define the mapping for the three different event types: Bundle, Service, and Framework.

101.6.1 Bundle Events Mapping

A Bundle Event is mapped to a LogEntry object according to Table 101.2, “Mapping of Bundle Events to Log Entries,” on page 40.

Table 101.2 Mapping of Bundle Events to Log Entries

Log Entry method	Information about Bundle Event
getLevel()	LOG_INFO
getBundle()	Identifies the bundle to which the event happened. In other words, it identifies the bundle that was installed, started, stopped, updated, or uninstalled. This identification is obtained by calling getBundle() on the BundleEvent object.
getException()	null
getServiceReference()	null
getMessage()	The message depends on the event type: • INSTALLED – "BundleEvent INSTALLED" • STARTED – "BundleEvent STARTED" • STOPPED – "BundleEvent STOPPED" • UPDATED – "BundleEvent UPDATED" • UNINSTALLED – "BundleEvent UNINSTALLED" • RESOLVED – "BundleEvent RESOLVED" • UNRESOLVED – "BundleEvent UNRESOLVED"

101.6.2 Service Events Mapping

A Service Event is mapped to a LogEntry object according to Table 101.3, “Mapping of Service Events to Log Entries,” on page 41.

Table 101.3 Mapping of Service Events to Log Entries

Log Entry method	Information about Service Event
getLevel()	LOG_INFO, except for the ServiceEvent.MODIFIED event. This event can happen frequently and contains relatively little information. It must be logged with a level of LOG_DEBUG.
getBundle()	Identifies the bundle that registered the service associated with this event. It is obtained by calling getServiceReference().getBundle() on the ServiceEvent object.
getException()	null
getServiceReference()	Identifies a reference to the service associated with the event. It is obtained by calling getServiceReference() on the ServiceEvent object.
getMessage()	This message depends on the actual event type. The messages are mapped as follows: • REGISTERED – "ServiceEvent REGISTERED" • MODIFIED – "ServiceEvent MODIFIED" • UNREGISTERING – "ServiceEvent UNREGISTERING"

101.6.3 Framework Events Mapping

A Framework Event is mapped to a LogEntry object according to Table 101.4, “Mapping of Framework Event to Log Entries,” on page 41.

Table 101.4 Mapping of Framework Event to Log Entries

Log Entry method	Information about Framework Event
getLevel()	LOG_INFO, except for the FrameworkEvent.ERROR event. This event represents an error and is logged with a level of LOG_ERROR.
getBundle()	Identifies the bundle associated with the event. This may be the system bundle. It is obtained by calling getBundle() on the FrameworkEvent object.
getException()	Identifies the exception associated with the error. This will be null for event types other than ERROR. It is obtained by calling getThrowable() on the FrameworkEvent object.
getServiceReference()	null
getMessage()	This message depends on the actual event type. The messages are mapped as follows: • STARTED – "FrameworkEvent STARTED" • ERROR – "FrameworkEvent ERROR" • PACKAGES_REFRESHED – "FrameworkEvent PACKAGES REFRESHED" • STARTLEVEL_CHANGED – "FrameworkEvent STARTLEVEL CHANGED" • WARNING – "FrameworkEvent WARNING" • INFO – "FrameworkEvent INFO"

101.6.4 Log Events

Log events must be delivered by the Log Service implementation to the Event Admin service (if present) asynchronously under the topic:

```
org/osgi/service/log/LogEntry/<event type>
```

The logging level is used as event type:

```
LOG_ERROR
LOG_WARNING
LOG_INFO
LOG_DEBUG
LOG_OTHER (when event is not recognized)
```

The properties of a log event are:

- bundle.id – (Long) The source bundle's id.
- bundle.symbolicName – (String) The source bundle's symbolic name. Only set if not null.
- bundle – (Bundle) The source bundle.
- log.level – (Integer) The log level.
- message – (String) The log message.
- timestamp – (Long) The log entry's timestamp.
- log.entry – (LogEntry) The LogEntry object.

If the log entry has an associated Exception:

- exception.class – (String) The fully-qualified class name of the attached exception. Only set if the getExceptionmethod returns a non-null value.
- exception.message – (String) The message of the attached Exception. Only set if the Exception message is not null.
- exception – (Throwable) The Exception returned by the getException method.

If the getServiceReference method returns a non-null value:

- service – (ServiceReference) The result of the getServiceReference method.
- service.id – (Long) The id of the service.
- service.pid – (String) The service's persistent identity. Only set if the service.pid service property is not null.
- service.objectClass – (String[]) The object class of the service object.

101.7 Security

The Log Service should only be implemented by trusted bundles. This bundle requires ServicePermission[LogService|LogReaderService, REGISTER]. Virtually all bundles should get ServicePermission[LogService, GET]. The ServicePermission[LogReaderService, GET] should only be assigned to trusted bundles.

101.8 org.osgi.service.log

Log Service Package Version 1.3.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.log; version="[1.3,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.log; version="[1.3,1.4)"

101.8.1 Summary

- LogEntry – Provides methods to access the information contained in an individual Log Service log entry.
- LogListener – Subscribes to LogEntry objects from the LogReaderService.
- LogReaderService – Provides methods to retrieve LogEntry objects from the log.
- LogService – Provides methods for bundles to write messages to the log.

101.8.2 Permissions

101.8.3 public interface LogEntry

Provides methods to access the information contained in an individual Log Service log entry.

A LogEntry object may be acquired from the LogReaderService.getLog method or by registering a LogListener object.

See Also LogReaderService.getLog, LogListener

Concurrency Thread-safe

No Implement Consumers of this API must not implement this interface

101.8.3.1 public Bundle getBundle ( )

□ Returns the bundle that created this LogEntry object.

Returns The bundle that created this LogEntry object; null if no bundle is associated with this LogEntry object.

101.8.3.2 public Throwable getException ( )

□ Returns the exception object associated with this LogEntry object.

In some implementations, the returned exception may not be the original exception. To avoid references to a bundle defined exception class, thus preventing an uninstalled bundle from being garbage collected, the Log Service may return an exception object of an implementation defined Throwable subclass. The returned object will attempt to provide as much information as possible from the original exception object such as the message and stack trace.

Returns Throwable object of the exception associated with this LogEntry;null if no exception is associated with this LogEntry object.

101.8.3.3 public int getLevel ( )

□ Returns the severity level of this LogEntry object.

This is one of the severity levels defined by the LogService interface.

Returns Severity level of this LogEntry object.

See Also LogService.LOG_ERROR, LogService.LOG_WARNING, LogService.LOG_INFO, LogService.LOG_DEBUG

101.8.3.4 public String getMessage ( )

□ Returns the human readable message associated with this LogEntry object.

Returns String containing the message associated with this LogEntry object.

101.8.3.5 public ServiceReference getServiceReference ( )

□ Returns the ServiceReference object for the service associated with this LogEntry object.

Returns ServiceReference object for the service associated with this LogEntry object; null if no ServiceReference object was provided.

101.8.3.6 public long getTime ( )

□ Returns the value of currentTimeMillis() at the time this LogEntry object was created.

Returns The system time in milliseconds when this `LogEntry` object was created.

See Also `System.currentTimeMillis()`

101.8.4 public interface LogListener extends EventListener

Subscribes to `LogEntry` objects from the `LogReaderService`.

A `LogListener` object may be registered with the Log Reader Service using the `LogReaderService.addLogListener` method. After the listener is registered, the `logged` method will be called for each `LogEntry` object created. The `LogListener` object may be unregistered by calling the `LogReaderService.removeLogListener` method.

See Also `LogReaderService`, `LogEntry`, `LogReaderService.addLogListener(LogListener)`, `LogReaderService.removeLogListener(LogListener)`

Concurrency Thread-safe

101.8.4.1 public void logged (LogEntry entry)

entry A `LogEntry` object containing log information.

□ Listener method called for each LogEntry object created.

As with all event listeners, this method should return to its caller as soon as possible.

See Also `LogEntry`

101.8.5 public interface LogReaderService

Provides methods to retrieve `LogEntry` objects from the log.

There are two ways to retrieve `LogEntry` objects:

- The primary way to retrieve `LogEntry` objects is to register a `LogListener` object whose `LogListener.logged` method will be called for each entry added to the log.
- To retrieve past `LogEntry` objects, the `getLog` method can be called which will return an `Enumeration` of all `LogEntry` objects in the log.

See Also `LogEntry`, `LogListener`, `LogListener.logged(LogEntry)`

Concurrency Thread-safe

101.8.5.1 public void addLogListener (LogListener listener)

listener A `LogListener` object to register; the `LogListener` object is used to receive `LogEntry` objects.

□ Subscribes to `LogEntry` objects.

This method registers a `LogListener` object with the Log Reader Service. The `LogListener.logged(LogEntry)` method will be called for each `LogEntry` object placed into the log.

When a bundle which registers a `LogListener` object is stopped or otherwise releases the Log Reader Service, the Log Reader Service must remove all of the bundle's listeners.

If this Log Reader Service's list of listeners already contains a listener `l` such that `(l==listener)`, this method does nothing.

See Also `LogListener`, `LogEntry`, `LogListener.logged(LogEntry)`

101.8.5.2 public Enumeration getLog ( )

□ Returns an `Enumeration` of all `LogEntry` objects in the log.

Each element of the enumeration is a LogEntry object, ordered with the most recent entry first. Whether the enumeration is of all LogEntry objects since the Log Service was started or some recent past is implementation-specific. Also implementation-specific is whether informational and debug LogEntry objects are included in the enumeration.

Returns An Enumeration of all LogEntry objects in the log.

101.8.5.3 public void removeLogListener (LogListener listener)

listener A LogListener object to unregister.

□ Unsubscribes to LogEntry objects.

This method unregisters a LogListener object from the Log Reader Service.

If listener is not contained in this Log Reader Service's list of listeners, this method does nothing.

See Also LogListener

101.8.6 public interface LogService

Provides methods for bundles to write messages to the log.

LogService methods are provided to log messages; optionally with a ServiceReference object or an exception.

Bundles must log messages in the OSGi environment with a severity level according to the following hierarchy:

1. LOG_ERROR
2. LOG_WARNING
3. LOG_INFO
4. LOG_DEBUG

Concurrency Thread-safe

No Implement Consumers of this API must not implement this interface

101.8.6.1 public static final int LOG_DEBUG = 4

A debugging message (Value 4).

This log entry is used for problem determination and may be irrelevant to anyone but the bundle developer.

101.8.6.2 public static final int LOG_ERROR = 1

An error message (Value 1).

This log entry indicates the bundle or service may not be functional.

101.8.6.3 public static final int LOG_INFO = 3

An informational message (Value 3).

This log entry may be the result of any change in the bundle or service and does not indicate a problem.

101.8.6.4 public static final int LOG_WARNING = 2

A warning message (Value 2).

This log entry indicates a bundle or service is still functioning but may experience problems in the future because of the warning condition.

101.8.6.5 public void log (int level , String message)

level The severity of the message. This should be one of the defined log levels but may be any integer that is interpreted in a user defined way.

message Human readable string describing the condition or null.

□ Logs a message.

The ServiceReference field and the Throwable field of the LogEntry object will be set to null.

See Also LOG_ERROR , LOG_WARNING , LOG_INFO , LOG_DEBUG

101.8.6.6 public void log (int level , String message , Throwable exception)

level The severity of the message. This should be one of the defined log levels but may be any integer that is interpreted in a user defined way.

message The human readable string describing the condition or null.

exception The exception that reflects the condition or null.

□ Logs a message with an exception.

The ServiceReference field of the LogEntry object will be set to null.

See Also LOG_ERROR , LOG_WARNING , LOG_INFO , LOG_DEBUG

101.8.6.7 public void log (ServiceReference sr , int level , String message)

sr The ServiceReference object of the service that this message is associated with or null.

level The severity of the message. This should be one of the defined log levels but may be any integer that is interpreted in a user defined way.

message Human readable string describing the condition or null.

□ Logs a message associated with a specific ServiceReference object.

The Throwable field of the LogEntry will be set to null.

See Also LOG_ERROR , LOG_WARNING , LOG_INFO , LOG_DEBUG

101.8.6.8 public void log (ServiceReference sr , int level , String message , Throwable exception)

sr The ServiceReference object of the service that this message is associated with.

level The severity of the message. This should be one of the defined log levels but may be any integer that is interpreted in a user defined way.

message Human readable string describing the condition or null.

exception The exception that reflects the condition or null.

□ Logs a message with an exception associated and a ServiceReference object.

See Also LOG_ERROR , LOG_WARNING , LOG_INFO , LOG_DEBUG

102 Http Service Specification

Version 1.2

102.1 Introduction

An OSGi Service Platform normally provides users with access to services on the Internet and other networks. This access allows users to remotely retrieve information from, and send control to, services in an OSGi Service Platform using a standard web browser.

Bundle developers typically need to develop communication and user interface solutions for standard technologies such as HTTP, HTML, XML, and servlets.

The Http Service supports two standard techniques for this purpose:

- *Registering servlets* – A servlet is a Java object which implements the Java Servlet API. Registering a servlet in the Framework gives it control over some part of the Http Service URI name-space.
- *Registering resources* – Registering a resource allows HTML files, image files, and other static resources to be made visible in the Http Service URI name-space by the requesting bundle.

Implementations of the Http Service can be based on:

- [1] *HTTP 1.0 Specification RFC-1945*
- [2] *HTTP 1.1 Specification RFC-2616*

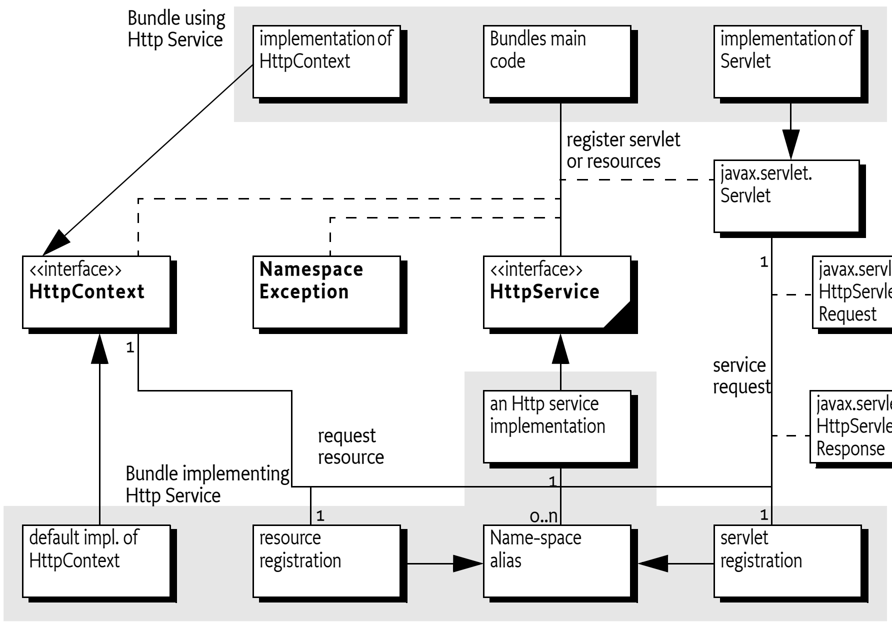
Alternatively, implementations of this service can support other protocols if these protocols can conform to the semantics of the `javax.servlet` API. This additional support is necessary because the Http Service is closely related to [3] *Java Servlet Technology*. Http Service implementations must support at least version 2.1 of the Java Servlet API.

102.1.1 Entities

This specification defines the following interfaces which a bundle developer can implement collectively as an Http Service or use individually:

- `HttpContext` – Allows bundles to provide information for a servlet or resource registration.
- `HttpService` – Allows other bundles in the Framework to dynamically register and unregister resources and servlets into the Http Service URI name-space.
- `NamespaceException` – Is thrown to indicate an error with the caller's request to register a servlet or resource into the Http Service URI name-space.

Figure 102.1 Http Service Overview Diagram


Bundle using Http Service
implementation of HttpContext
Bundles main code
implementation of Servlet
register servlet or resources
javax.servlet. Servlet
<<interface>> HttpContext
Namespace Exception
<<interface>> HttpService
javax.servlet.http HttpServlet Request
service request
javax.servlet.http HttpServlet Response
an Http service implementation
request resource
Bundle implementing Http Service
default impl. of HttpContext
resource registration
Name-space alias
servlet registration
1
0..n

102.2 Registering Servlets

javax.servlet.Servlet objects can be registered with the Http Service by using the HttpService interface. For this purpose, the HttpService interface defines the method registerServlet(String, javax.servlet.Servlet,Dictionary,HttpContext).

For example, if the Http Service implementation is listening to port 80 on the machine www.acme.com and the Servlet object is registered with the name "/servlet", then the Servlet object's service method is called when the following URL is used from a web browser:

```
http://www.acme.com/servletname=bugs
```

All Servlet objects and resource registrations share the same name-space. If an attempt is made to register a resource or Servlet object under the same name as a currently registered resource or Servlet object, a NamespaceException is thrown. See *Mapping HTTP Requests to Servlet and Resource Registrations* on page 51 for more information about the handling of the Http Service name-space.

Each Servlet registration must be accompanied with an HttpContext object. This object provides the handling of resources, media typing, and a method to handle authentication of remote requests. See *Authentication* on page 54.

For convenience, a default HttpContext object is provided by the Http Service and can be obtained with createDefaultHttpContext(). Passing a null parameter to the registration method achieves the same effect.

Servlet objects require a ServletContext object. This object provides a number of functions to access the Http Service Java Servlet environment. It is created by the implementation of the Http Service for each unique HttpContext object with which a Servlet object is registered. Thus, Servlet objects registered with the same HttpContext object must also share the same ServletContext object.

Servlet objects are initialized by the Http Service when they are registered and bound to that specific Http Service. The initialization is done by calling the Servlet object's Servlet.init(ServletConfig) method. The ServletConfig parameter provides access to the initialization parameters specified when the Servlet object was registered.

Therefore, the same Servlet instance must not be reused for registration with another Http Service, nor can it be registered under multiple names. Unique instances are required for each registration.

The following example code demonstrates the use of the registerServlet method:

```
Hashtable initparams = new Hashtable();
initparams.put( "name", "value" );

Servlet myServlet = new HttpServlet() {
   String    name = "<not set>";

   public void init( ServletConfig config ) {
      this.name = (String)
         config.getInitParameter( "name" );
   }

   public void doGet(
      HttpServletRequest req,
      HttpServletResponse rsp
   ) throws IOException {
      rsp.setContentType( "text/plain" );
      req.getWriter().println( this.name );
   }
};

getHttpService().registerServlet(
   "/servletAlias",
   myServlet,
   initparams,
   null // use default context
);
// myServlet has been registered
// and its init method has been called. Remote
// requests are now handled and forwarded to
// the servlet.
...
getHttpService().unregister("/servletAlias");
// myServlet has been unregistered and its
// destroy method has been called
```

This example registers the servlet, myServlet, at alias: /servletAlias. Future requests for http://www.acme.com/servletAlias maps to the servlet, myServlet, whose service method is called to process the request. (The service method is called in the HttpServlet base class and dispatched to a doGet, doPut, doPost, doOptions, doTrace, or doDelete call depending on the HTTP request method used.)

102.3 Registering Resources

A resource is a file containing images, static HTML pages, sounds, movies, applets, etc. Resources do not require any handling from the bundle. They are transferred directly from their source--usually the JAR file that contains the code for the bundle--to the requestor using HTTP.

Resources could be handled by Servlet objects as explained in *Registering Servlets* on page 48. Transferring a resource over HTTP, however, would require very similar Servlet objects for each bundle. To prevent this redundancy, resources can be registered directly with the Http Service via the HttpService interface. This HttpService interface defines the registerResources(String,String, HttpContext) method for registering a resource into the Http Service URI name-space.

The first parameter is the external alias under which the resource is registered with the Http Service. The second parameter is an internal prefix to map this resource to the bundle's name-space. When a request is received, the HttpService object must remove the external alias from the URI, replace it with the internal prefix, and call the getResource(String) method with this new name on the associated HttpContext object. The HttpContext object is further used to get the MIME type of the resource and to authenticate the request.

Resources are returned as a java.net.URL object. The Http Service must read from this URL object and transfer the content to the initiator of the HTTP request.

This return type was chosen because it matches the return type of the java.lang.Class.getResource(String resource) method. This method can retrieve resources directly from the same place as the one from which the class was loaded – often a package directory in the JAR file of the bundle. This method makes it very convenient to retrieve resources from the bundle that are contained in the package.

The following example code demonstrates the use of the register
Resources method:

```
package com.acme;
...
HttpContext context = new HttpContext() {
   public boolean handleSecurity(
      HttpServletRequest request,
      HttpServletResponse response
   ) throws IOException {
      return true;
   }

   public URL getResource(String name) {
      return getClass().getResource(name);
   }

   public String getMimeType(String name) {
      return null;
   }
};

getHttpService().registerResources (
   "/files",
   "www",
   context
);
...
getHttpService().unregister("/files");
```

This example registers the alias /files on the Http Service. Requests for resources below this name-space are transferred to the HttpContext object with an internal name of www/<name>. This example uses the Class.get
Resource(String) method. Because the internal name does not start with a

"/", it must map to a resource in the "com/acme/www" directory of the JAR file. If the internal name did start with a "/", the package name would not have to be prefixed and the JAR file would be searched from the root. Consult the java.lang.Class.getResource(String) method for more information.

In the example, a request for http://www.acme.com/files/myfile.html must map to the name "com/acme/www/myfile.html" which is in the bundle's JAR file.

More sophisticated implementations of the getResource(String) method could filter the input name, restricting the resources that may be returned or map the input name onto the file system (if the security implications of this action are acceptable).

Alternatively, the resource registration could have used a default HttpContext object, as demonstrated in the following call to registerResources:

```
getHttpService().registerResources(
   "/files",
   "/com/acme/www",
   null
);
```

In this case, the Http Service implementation would call the createDefaultHttpContext() method and use its return value as the HttpContext argument for the registerResources method. The default implementation must map the resource request to the bundle's resource, using
Bundle.getResource(String). In the case of the previous example, however, the internal name must now specify the full path to the directory containing the resource files in the JAR file. No automatic prefixing of the package name is done.

The getMimeType(String) implementation of the default HttpContext object should rely on the default mapping provided by the Http Service by returning null. Its handleSecurity(HttpServletRequest,HttpServletResponse) may implement an authentication mechanism that is implementation-dependent.

102.4 Mapping HTTP Requests to Servlet and Resource Registrations

When an HTTP request comes in from a client, the Http Service checks to see if the requested URI matches any registered aliases. A URI matches only if the path part of the URI is exactly the same string. Matching is case sensitive.

If it does match, a matching registration takes place, which is processed as follows:

1. If the registration corresponds to a servlet, the authorization is verified by calling the handleSecurity method of the associated HttpContext object. See *Authentication* on page 54. If the request is authorized, the servlet must be called by its service method to complete the HTTP request.
2. If the registration corresponds to a resource, the authorization is verified by calling the handleSecurity method of the associated HttpContext object. See *Authentication* on page 54. If the request is authorized, a target resource name is constructed from the requested URI by substituting the alias from the registration with the internal name from the registration if the alias is not "/". If the alias is "/", then the target resource name is constructed by prefixing the requested URI with the internal name. An internal name of "/" is considered to have the value of the empty string ("") during this process.
3. The target resource name must be passed to the getResource method of the associated HttpContext object.

4. If the returned URL object is not null, the Http Service must return the contents of the URL to the client completing the HTTP request. The translated target name, as opposed to the original requested URI, must also be used as the argument to HttpContext.getMimeType.
5. If the returned URL object is null, the Http Service continues as if there was no match.
6. If there is no match, the Http Service must attempt to match sub-strings of the requested URI to registered aliases. The sub-strings of the requested URI are selected by removing the last "/" and everything to the right of the last "/".

The Http Service must repeat this process until either a match is found or the sub-string is an empty string. If the sub-string is empty and the alias "/" is registered, the request is considered to match the alias "/". Otherwise, the Http Service must return HttpServletResponse.SC_NOT_FOUND(404) to the client.

For example, an HTTP request comes in with a request URI of "/fudd/bugs/foo.txt", and the only registered alias is "/fudd". A search for "/fudd/bugs/foo.txt" will not match an alias. Therefore, the Http Service will search for the alias "/fudd/bugs" and the alias "/fudd". The latter search will result in a match and the matched alias registration must be used.

Registrations for identical aliases are not allowed. If a bundle registers the alias "/fudd", and another bundle tries to register the exactly the same alias, the second caller must receive a NamespaceException and its resource or servlet must *not* be registered. It could, however, register a similar alias – for example, "/fudd/bugs", as long as no other registration for this alias already exists.

The following table shows some examples of the usage of the name-space.

Table 102.1 Examples of Name-space Mapping

Alias	Internal Name	URI	getResource Parameter
/	(empty string)	/fudd/bugs	/fudd/bugs
/	/	/fudd/bugs	/fudd/bugs
/	/tmp	/fudd/bugs	/tmp/fudd/bugs
/fudd	(empty string)	/fudd/bugs	/bugs
/fudd	/	/fudd/bugs	/bugs
/fudd	/tmp	/fudd/bugs	/tmp/bugs
/fudd	tmp	/fudd/bugs/x.gif	tmp/bugs/x.gif
/fudd/bugs/x.gif	tmp/y.gif	/fudd/bugs/x.gif	tmp/y.gif

102.5 The Default Http Context Object

The HttpContext object in the first example demonstrates simple implementations of the HttpContext interface methods. Alternatively, the example could have used a default HttpContext object, as demonstrated in the following call to registerServlet:

```
getHttpService().registerServlet(
   "/servletAlias",
   myServlet,
   initparams,
   null
);
```

In this case, the Http Service implementation must call createDefault HttpContext and use the return value as the HttpContext argument.

If the default HttpContext object, and thus the ServletContext object, is to be shared by multiple servlet registrations, the previous servlet registration example code needs to be changed to use the same default HttpContext object. This change is demonstrated in the next example:

```
HttpContext defaultContext =
   getHttpService().createDefaultHttpContext();

getHttpService().registerServlet(
   "/servletAlias",
   myServlet,
   initparams,
   defaultContext
);

// defaultContext can be reused
// for further servlet registrations
```

102.6 Multipurpose Internet Mail Extension (MIME) Types

MIME defines an extensive set of headers and procedures to encode binary messages in US-ASCII mails. For an overview of all the related RFCs, consult [4] *MIME Multipurpose Internet Mail Extension*.

An important aspect of this extension is the type (file format) mechanism of the binary messages. The type is defined by a string containing a general category (text, application, image, audio and video, multipart, and message) followed by a "/" and a specific media type, as in the example, "text/html" for HTML formatted text files. A MIME type string can be followed by additional specifiers by separating key=value pairs with a ';'. These specifiers can be used, for example, to define character sets as follows:

```
text/plain ; charset=iso-8859-1
```

The Internet Assigned Number Authority (IANA) maintains a set of defined MIME media types. This list can be found at [5] *Assigned MIME Media Types*. MIME media types are extendable, and when any part of the type starts with the prefix "x-", it is assumed to be vendor-specific and can be used for testing. New types can be registered as described in [6] *Registration Procedures for new MIME media types*.

HTTP bases its media typing on the MIME RFCs. The "Content-Type" header should contain a MIME media type so that the browser can recognize the type and format the content correctly.

The source of the data must define the MIME media type for each transfer. Most operating systems do not support types for files, but use conventions based on file names, such as the last part of the file name after the last ".". This extension is then mapped to a media type.

Implementations of the Http Service should have a reasonable default of mapping common extensions to media types based on file extensions.

Table 102.2 Sample Extension to MIME Media Mapping

Extension	MIME media type	Description
.jpg .jpeg	image/jpeg	JPEG Files
.gif	image/gif	GIF Files
.css	text/css	Cascading Style Sheet Files
.txt	text/plain	Text Files
.wml	text/vnd.wap.wml	Wireless Access Protocol (WAP) Mark Language

Table 102.2 Sample Extension to MIME Media Mapping

Extension	MIME media type	Description
.htm .html	text/html	Hyper Text Markup Language
.wbmp	image/vnd.wap.wbmp	Bitmaps for WAP

Only the bundle developer, however, knows exactly which files have what media type. The HttpContext interface can therefore be used to map this knowledge to the media type. The HttpContext class has the following method for this: getMimeType(String).

The implementation of this method should inspect the file name and use its internal knowledge to map this name to a MIME media type.

Simple implementations can extract the extension and look up this extension in a table.

Returning null from this method allows the Http Service implementation to use its default mapping mechanism.

102.7 Authentication

The Http Service has separated the authentication and authorization of a request from the execution of the request. This separation allows bundles to use available Servlet sub-classes while still providing bundle specific authentication and authorization of the requests.

Prior to servicing each incoming request, the Http Service calls the handleSecurity(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse) method on the HttpContext object that is associated with the request URI. This method controls whether the request is processed in the normal manner or an authentication error is returned.

If an implementation wants to authenticate the request, it can use the authentication mechanisms of HTTP. See [7] *RFC 2617: HTTP Authentication: Basic and Digest Access Authentication*. These mechanisms normally interpret the headers and decide if the user identity is available, and if it is, whether that user has authenticated itself correctly.

There are many different ways of authenticating users, and the handleSecurity method on the HttpContext object can use whatever method it requires. If the method returns true, the request must continue to be processed using the potentially modified HttpServletRequest and HttpServletResponse objects. If the method returns false, the request must *not* be processed.

A common standard for HTTP is the basic authentication scheme that is not secure when used with HTTP. Basic authentication passes the password in base 64 encoded strings that are trivial to decode into clear text. Secure transport protocols like HTTPS use SSL to hide this information. With these protocols basic authentication is secure.

Using basic authentication requires the following steps:

1. If no Authorization header is set in the request, the method should set the WWW-Authenticate header in the response. This header indicates the desired authentication mechanism and the realm. For example, WWW-Authenticate: Basic realm="ACME".
   The header should be set with the response object that is given as a parameter to the handleSecurity method. The handleSecurity method should set the status to HttpServletResponse.SC_UNAUTHORIZED (401) and return false.
2. Secure connections can be verified with the ServletRequest.getScheme() method. This method returns, for example, "https" for an SSL connection; the handleSecurity method can use this and other information to decide if the connection's security level is acceptable. If not, the handleSecurity method should set the status to HttpServletResponse.SC_FORBIDDEN (403) and return false.

3. Next, the request must be authenticated. When basic authentication is used, the Authorization header is available in the request and should be parsed to find the user and password. See [7] *RFC 2617: HTTP Authentication: Basic and Digest Access Authentication* for more information.
   If the user cannot be authenticated, the status of the response object should be set to HttpServletResponse.SC_UNAUTHORIZED (401) and return false.
4. The authentication mechanism that is actually used and the identity of the authenticated user can be of interest to the Servlet object. Therefore, the implementation of the handleSecurity method should set this information in the request object using the ServletRequest.setAttribute method. This specification has defined a number of OSGi-specific attribute names for this purpose:
   - AUTHENTICATION_TYPE - Specifies the scheme used in authentication. A Servlet may retrieve the value of this attribute by calling the HttpServletRequest.getAuthType method. This attribute name is org.osgi.service.http.authentication.type.
   - REMOTE_USER - Specifies the name of the authenticated user. A Servlet may retrieve the value of this attribute by calling the HttpServletRequest.getRemoteUser method. This attribute name is org.osgi.service.http.authentication.remote.user.
   - AUTHORIZATION - If a User Admin service is available in the environment, then the handleSecurity method should set this attribute with the Authorization object obtained from the User Admin service. Such an object encapsulates the authentication of its remote user. A Servlet may retrieve the value of this attribute by calling ServletRequest.getAttribute(HttpContext.AUTHORIZATION). This header name is org.osgi.service.useradmin.authorization.
5. Once the request is authenticated and any attributes are set, the handleSecurity method should return true. This return indicates to the Http Service that the request is authorized and processing may continue. If the request is for a Servlet, the Http Service must then call the service method on the Servlet object.

102.8 Security

This section only applies when executing in an OSGi environment which is enforcing Java permissions.

102.8.1 Accessing Resources with the Default Http Context

The Http Service must be granted AdminPermission[*,RESOURCE] so that bundles may use a default HttpContext object. This is necessary because the implementation of the default HttpContext object must call Bundle.getResource to access the resources of a bundle and this method requires the caller to have AdminPermission[bundle,RESOURCE].

Any bundle may access resources in its own bundle by calling Class.getResource. This operation is privileged. The resulting URL object may then be passed to the Http Service as the result of a HttpContext.getResource call. No further permission checks are performed when accessing bundle resource URL objects, so the Http Service does not need to be granted any additional permissions.

102.8.2 Accessing Other Types of Resources

In order to access resources that were not registered using the default HttpContext object, the Http Service must be granted sufficient privileges to access these resources. For example, if the getResource method of the registered HttpContext object returns a file URL, the Http Service requires the corresponding FilePermission to read the file. Similarly, if the getResource method of the registered HttpContext object returns an HTTP URL, the Http Service requires the corresponding SocketPermission to connect to the resource.

Therefore, in most cases, the Http Service should be a privileged service that is granted sufficient permission to serve any bundle's resources, no matter where these resources are located. Therefore, the Http Service must capture the AccessControlContext object of the bundle registering resources or a servlet, and then use the captured AccessControlContext object when accessing resources returned by the registered HttpContext object. This situation prevents a bundle from registering resources that it does not have permission to access.

Therefore, the Http Service should follow a scheme like the following example. When a resource or servlet is registered, it should capture the context.

```
AccessControlContext acc =
    AccessController.getContext();
```

When a URL returned by the getResource method of the associated HttpContext object is called, the Http Service must call the getResource method in a doPrivileged construct using the AccessControlContext object of the registering bundle:

```
AccessController.doPrivileged(
  new PrivilegedExceptionAction() {
    public Object run() throws Exception {
    ...
    }
  }, acc);
```

The Http Service must only use the captured AccessControlContext when accessing resource URL objects.

102.8.3 Servlet and HttpContext objects

This specification does not require that the Http Service is granted All Permission or wraps calls to the Servlet and Http Context objects in a doPrivileged block. Therefore, it is the responsibility of the Servlet and Http Context implementations to use a doPrivileged block when performing privileged operations.

102.9 Configuration Properties

If the Http Service does not have its port values configured through some other means, the Http Service implementation should use the following properties to determine the port values upon which to listen.

The following OSGi environment properties are used to specify default HTTP ports:

- org.osgi.service.http.port – This property specifies the port used for servlets and resources accessible via HTTP. The default value for this property is 80.
- org.osgi.service.http.port.secure – This property specifies the port used for servlets and resources accessible via HTTPS. The default value for this property is 443.

102.10 org.osgi.service.http

Http Service Package Version 1.2.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle’s manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.http; version=”[1.2,2.0)”

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.http; version="[1.2,1.3)"

102.10.1 Summary

- HttpContext – This interface defines methods that the Http Service may call to get information about a registration.
- HttpService – The Http Service allows other bundles in the OSGi environment to dynamically register resources and servlets into the URI namespace of Http Service.
- NamespaceException – A NamespaceException is thrown to indicate an error with the caller's request to register a servlet or resources into the URI namespace of the Http Service.

102.10.2 Permissions

102.10.3 public interface HttpContext

This interface defines methods that the Http Service may call to get information about a registration.

Servlets and resources may be registered with an HttpContext object; if no HttpContext object is specified, a default HttpContext object is used. Servlets that are registered using the same HttpContext object will share the same ServletContext object.

This interface is implemented by users of the HttpService.

102.10.3.1 public static final String AUTHENTICATION_TYPE = "org.osgi.service.http.authentication.type"

HttpServletRequest attribute specifying the scheme used in authentication. The value of the attribute can be retrieved by HttpServletRequest.getAuthType. This attribute name is org.osgi.service.http.authentication.type.

Since 1.1

102.10.3.2 public static final String AUTHORIZATION = "org.osgi.service.useradmin.authorization"

HttpServletRequest attribute specifying the Authorization object obtained from the org.osgi.service.useradmin.UserAdmin service. The value of the attribute can be retrieved by HttpServletRequest.getAttribute(HttpContext.AUTHORIZATION). This attribute name is org.osgi.service.useradmin.authorization.

Since 1.1

102.10.3.3 public static final String REMOTE_USER = "org.osgi.service.http.authentication.remote.user"

HttpServletRequest attribute specifying the name of the authenticated user. The value of the attribute can be retrieved by HttpServletRequest.getRemoteUser. This attribute name is org.osgi.service.http.authentication.remote.user.

Since 1.1

102.10.3.4 public String getMimeType (String name)

name determine the MIME type for this name.

□ Maps a name to a MIME type. Called by the Http Service to determine the MIME type for the name. For servlet registrations, the Http Service will call this method to support the ServletContext method getMimeType. For resource registrations, the Http Service will call this method to determine the MIME type for the Content-Type header in the response.

Returns MIME type (e.g. text/html) of the name or null to indicate that the Http Service should determine the MIME type itself.

102.10.3.5 public URL getResource (String name)

name the name of the requested resource

□ Maps a resource name to a URL.

Called by the Http Service to map a resource name to a URL. For servlet registrations, Http Service will call this method to support the `ServletContext` methods `getResource` and `getResourceAsStream`. For resource registrations, Http Service will call this method to locate the named resource. The context can control from where resources come. For example, the resource can be mapped to a file in the bundle's persistent storage area via `bundleContext.getDataFile(name).toURL()` or to a resource in the context's bundle via `getClass().getResource(name)`

Returns URL that Http Service can use to read the resource or `null` if the resource does not exist.

102.10.3.6 public boolean handleSecurity (HttpServletRequest request , HttpServletResponse response) throws IOException

request the HTTP request

response the HTTP response

□ Handles security for the specified request.

The Http Service calls this method prior to servicing the specified request. This method controls whether the request is processed in the normal manner or an error is returned.

If the request requires authentication and the Authorization header in the request is missing or not acceptable, then this method should set the WWW-Authenticate header in the response object, set the status in the response object to Unauthorized(401) and return `false`. See also RFC 2617: *HTTP Authentication: Basic and Digest Access Authentication* (available at http://www.ietf.org/rfc/rfc2617.txt).

If the request requires a secure connection and the `getScheme` method in the request does not return 'https' or some other acceptable secure protocol, then this method should set the status in the response object to Forbidden(403) and return `false`.

When this method returns `false`, the Http Service will send the response back to the client, thereby completing the request. When this method returns `true`, the Http Service will proceed with servicing the request.

If the specified request has been authenticated, this method must set the `AUTHENTICATION_TYPE` request attribute to the type of authentication used, and the `REMOTE_USER` request attribute to the remote user (request attributes are set using the `setAttribute` method on the request). If this method does not perform any authentication, it must not set these attributes.

If the authenticated user is also authorized to access certain resources, this method must set the `AUTHORIZATION` request attribute to the `Authorization` object obtained from the `org.osgi.service.useradmin.UserAdmin` service.

The servlet responsible for servicing the specified request determines the authentication type and remote user by calling the `getAuthType` and `getRemoteUser` methods, respectively, on the request.

Returns `true` if the request should be serviced, `false` if the request should not be serviced and Http Service will send the response back to the client.

Throws `IOException` – may be thrown by this method. If this occurs, the Http Service will terminate the request and close the socket.

102.10.4 public interface HttpService

The Http Service allows other bundles in the OSGi environment to dynamically register resources and servlets into the URI namespace of Http Service. A bundle may later unregister its resources or servlets.

See Also `HttpContext`

No Implement Consumers of this API must not implement this interface

102.10.4.1 public HttpContext createDefaultHttpContext ( )

☐ Creates a default HttpContext for registering servlets or resources with the HttpService, a new HttpContext object is created each time this method is called.

The behavior of the methods on the default HttpContext is defined as follows:

- getMimeType- Does not define any customized MIME types for the Content-Type header in the response, and always returns null.
- handleSecurity- Performs implementation-defined authentication on the request.
- getResource- Assumes the named resource is in the context bundle; this method calls the context bundle's Bundle.getResource method, and returns the appropriate URL to access the resource. On a Java runtime environment that supports permissions, the Http Service needs to be granted org.osgi.framework.AdminPermission[*,RESOURCE].

Returns a default HttpContext object.

Since 1.1

102.10.4.2 public void registerResources (String alias , String name , HttpContext context) throws NamespaceException

alias name in the URI namespace at which the resources are registered

name the base name of the resources that will be registered

context the HttpContext object for the registered resources, or null if a default HttpContext is to be created and used.

☐ Registers resources into the URI namespace.

The alias is the name in the URI namespace of the Http Service at which the registration will be mapped. An alias must begin with slash ('/') and must not end with slash ('/'), with the exception that an alias of the form "/" is used to denote the root alias. The name parameter must also not end with slash ('/') with the exception that a name of the form "/" is used to denote the root of the bundle. See the specification text for details on how HTTP requests are mapped to servlet and resource registrations.

For example, suppose the resource name /tmp is registered to the alias /files. A request for /files/foo.txt will map to the resource name /tmp/foo.txt.

```
httpservice.registerResources("/files", "/tmp", context);
```

The Http Service will call the HttpContext argument to map resource names to URLs and MIME types and to handle security for requests. If the HttpContext argument is null, a default HttpContext is used (see createDefaultHttpContext()).

Throws NamespaceException – if the registration fails because the alias is already in use.

IllegalArgumentException – if any of the parameters are invalid

102.10.4.3 public void registerServlet (String alias , Servlet servlet , Dictionary initparams , HttpContext context) throws ServletException , NamespaceException

alias name in the URI namespace at which the servlet is registered

servlet the servlet object to register

initparams initialization arguments for the servlet or null if there are none. This argument is used by the servlet's ServletConfig object.

context the HttpContext object for the registered servlet, or null if a default HttpContext is to be created and used.

☐ Registers a servlet into the URI namespace.

The alias is the name in the URI namespace of the Http Service at which the registration will be mapped.

An alias must begin with slash ('/') and must not end with slash ('/'), with the exception that an alias of the form "/" is used to denote the root alias. See the specification text for details on how HTTP requests are mapped to servlet and resource registrations.

The Http Service will call the servlet's init method before returning.

```
httpService.registerServlet("/myservlet", servlet, initparams, context);
```

Servlets registered with the same HttpContext object will share the same ServletContext. The Http Service will call the context argument to support the ServletContext methods getResource, getResourceAsStream and getMimeType, and to handle security for requests. If the context argument is null, a default HttpContext object is used (see createDefaultHttpContext()).

Throws NamespaceException – if the registration fails because the alias is already in use.

javax.servlet.ServletException – if the servlet's init method throws an exception, or the given servlet object has already been registered at a different alias.

IllegalArgumentException – if any of the arguments are invalid

102.10.4.4 public void unregister (String alias)

alias name in the URI name-space of the registration to unregister

□ Unregisters a previous registration done by registerServlet or registerResources methods.

After this call, the registered alias in the URI name-space will no longer be available. If the registration was for a servlet, the Http Service must call the destroy method of the servlet before returning.

If the bundle which performed the registration is stopped or otherwise "unget"s the Http Service without calling unregister(String) then Http Service must automatically unregister the registration. However, if the registration was for a servlet, the destroy method of the servlet will not be called in this case since the bundle may be stopped. unregister(String) must be explicitly called to cause the destroy method of the servlet to be called. This can be done in the BundleActivator.stop method of the bundle registering the servlet.

Throws IllegalArgumentException – if there is no registration for the alias or the calling bundle was not the bundle which registered the alias.

102.10.5 public class NamespaceException extends Exception

A NamespaceException is thrown to indicate an error with the caller's request to register a servlet or resources into the URI namespace of the Http Service. This exception indicates that the requested alias already is in use.

102.10.5.1 public NamespaceException (String message)

message the detail message

□ Construct a NamespaceException object with a detail message.

102.10.5.2 public NamespaceException (String message , Throwable cause)

message The detail message.

cause The nested exception.

□ Construct a NamespaceException object with a detail message and a nested exception.

102.10.5.3 public Throwable getCause ( )

□ Returns the cause of this exception or `null` if no cause was set.

Returns The cause of this exception or `null` if no cause was set.

Since 1.2

102.10.5.4 public Throwable getException ( )

□ Returns the nested exception.

This method predates the general purpose exception chaining mechanism. The `getCause()` method is now the preferred means of obtaining this information.

Returns The result of calling `getCause()`.

102.10.5.5 public Throwable initCause (Throwable cause)

cause The cause of this exception.

□ Initializes the cause of this exception to the specified value.

Returns This exception.

Throws `IllegalArgumentException` – If the specified cause is this exception.

`IllegalStateException` – If the cause of this exception has already been set.

Since 1.2

102.11 References

[1] *HTTP 1.0 Specification RFC-1945*
http://www.ietf.org/rfc/rfc1945.txt, May 1996

[2] *HTTP 1.1 Specification RFC-2616*
http://www.ietf.org/rfc/rfc2616.txt, June 1999

[3] *Java Servlet Technology*
http://www.oracle.com/technetwork/java/javaee/servlet/index.html

[4] *MIME Multipurpose Internet Mail Extension*
http://www.mhonarc.org/~ehood/MIME/MIME.html

[5] *Assigned MIME Media Types*
http://www.iana.org/assignments/media-types

[6] *Registration Procedures for new MIME media types*
http://www.ietf.org/rfc/rfc2048.txt

[7] *RFC 2617: HTTP Authentication: Basic and Digest Access Authentication*
http://www.ietf.org/rfc/rfc2617.txt

103 Device Access Specification

Version 1.1

103.1 Introduction

A Service Platform is a meeting point for services and devices from many different vendors: a meeting point where users add and cancel service subscriptions, newly installed services find their corresponding input and output devices, and device drivers connect to their hardware.

In an OSGi Service Platform, these activities will dynamically take place while the Framework is running. Technologies such as USB and IEEE 1394 explicitly support plugging and unplugging devices at any time, and wireless technologies are even more dynamic.

This flexibility makes it hard to configure all aspects of an OSGi Service Platform, particularly those relating to devices. When all of the possible services and device requirements are factored in, each OSGi Service Platform will be unique. Therefore, automated mechanisms are needed that can be extended and customized, in order to minimize the configuration needs of the OSGi environment.

The Device Access specification supports the coordination of automatic detection and attachment of existing devices on an OSGi Service Platform, facilitates hot-plugging and -unplugging of new devices, and downloads and installs device drivers on demand.

This specification, however, deliberately does not prescribe any particular device or network technology, and mentioned technologies are used as examples only. Nor does it specify a particular device discovery method. Rather, this specification focuses on the attachment of devices supplied by different vendors. It emphasizes the development of standardized device interfaces to be defined in device categories, although no such device categories are defined in this specification.

103.1.1 Essentials

- *Embedded Devices* – OSGi bundles will likely run in embedded devices. This environment implies limited possibility for user interaction, and low-end devices will probably have resource limitations.
- *Remote Administration* – OSGi environments must support administration by a remote service provider.
- *Vendor Neutrality* – OSGi-compliant driver bundles will be supplied by different vendors; each driver bundle must be well-defined, documented, and replaceable.
- *Continuous Operation* – OSGi environments will be running for extended periods without being restarted, possibly continuously, requiring stable operation and stable resource consumption.
- *Dynamic Updates* – As much as possible, driver bundles must be individually replaceable without affecting unrelated bundles. In particular, the process of updating a bundle should not require a restart of the whole OSGi Service Platform or disrupt operation of connected devices.

A number of requirements must be satisfied by Device Access implementations in order for them to be OSGi-compliant. Implementations must support the following capabilities:

- *Hot-Plugging* – Plugging and unplugging of devices at any time if the underlying hardware and drivers allow it.
- *Legacy Systems* – Device technologies which do not implement the automatic detection of plugged and unplugged devices.
- *Dynamic Device Driver Loading* – Loading new driver bundles on demand with no prior device-specific knowledge of the Device service.

- *Multiple Device Representations* – Devices to be accessed from multiple levels of abstraction.
- *Deep Trees* – Connections of devices in a tree of mixed network technologies of arbitrary depth.
- *Topology Independence* – Separation of the interfaces of a device from where and how it is attached.
- *Complex Devices* – Multifunction devices and devices that have multiple configurations.

103.1.2 Operation

This specification defines the behavior of a device manager (which is *not* a service as might be expected). This device manager detects registration of Device services and is responsible for associating these devices with an appropriate Driver service. These tasks are done with the help of Driver Locator services and the Driver Selector service that allow a device manager to find a Driver bundle and install it.

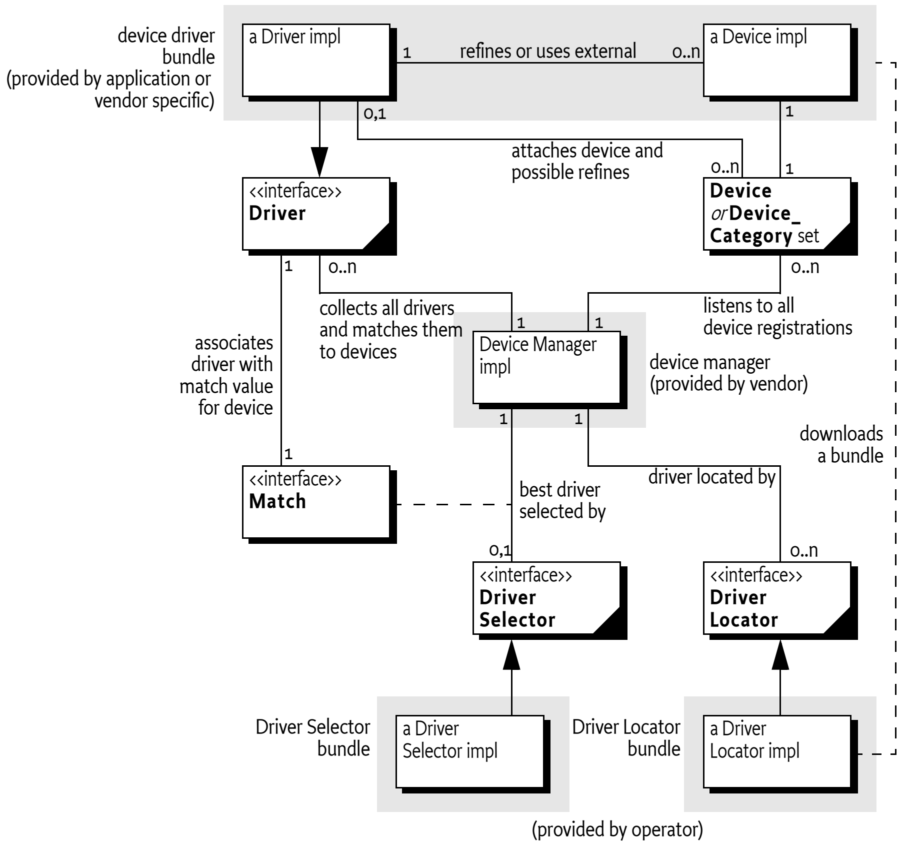
103.1.3 Entities

The main entities of the Device Access specification are:

- *Device Manager* – The bundle that controls the initiation of the attachment process behind the scenes.
- *Device Category* – Defines how a Driver service and a Device service can cooperate.
- *Driver* – Competes for attaching Device services of its recognized device category. See *Driver Services* on page 69.
- *Device* – A representation of a physical device or other entity that can be attached by a Driver service. See *Device Services* on page 65.
- *DriverLocator* – Assists in locating bundles that provide a Driver service. See *Driver Locator Service* on page 75.
- *DriverSelector* – Assists in selecting which Driver service is best suited to a Device service. See *The Driver Selector Service* on page 77.

Figure 103.1 show the classes and their relationships.

Figure 103.1 Device Access Class Overview


device driver bundle (provided by application or vendor specific)
a Driver impl
1
refines or uses external
0..n
a Device impl
0,1
1
attaches device and possible refines
0..n
1
<<interface>> Driver
Device or Device_ Category set
1
0..n
0..n
collects all drivers and matches them to devices
1
1
listens to all device registrations
associates driver with match value for device
Device Manager impl
device manager (provided by vendor)
1
1
downloads a bundle
1
<<interface>> Match
best driver selected by
driver located by
0,1
0..n
<<interface>> Driver Selector
<<interface>> Driver Locator
Driver Selector bundle
a Driver Selector impl
Driver Locator bundle
a Driver Locator impl
(provided by operator)

103.2 Device Services

A Device service represents some form of a device. It can represent a hardware device, but that is not a requirement. Device services differ widely: some represent individual physical devices and others represent complete networks. Several Device services can even simultaneously represent the same physical device at different levels of abstraction. For example:

- A USB network.
- A device attached on the USB network.
- The same device recognized as a USB to Ethernet bridge.
- A device discovered on the Ethernet using Salutation.
- The same device recognized as a simple printer.
- The same printer refined to a PostScript printer.

A device can also be represented in different ways. For example, a USB mouse can be considered as:

- A USB device which delivers information over the USB bus.
- A mouse device which delivers x and y coordinates and information about the state of its buttons.

Each representation has specific implications:

- That a particular device is a mouse is irrelevant to an application which provides management of USB devices.
- That a mouse is attached to a USB bus or a serial port would be inconsequential to applications that respond to mouse-like input.

Device services must belong to a defined *device category*, or else they can implement a generic service which models a particular device, independent of its underlying technology. Examples of this type of implementation could be Sensor or Actuator services.

A device category specifies the methods for communicating with a Device service, and enables inter-operability between bundles that are based on the same underlying technology. Generic Device services will allow inter-operability between bundles that are not coupled to specific device technologies.

For example, a device category is required for the USB, so that Driver bundles can be written that communicate to the devices that are attached to the USB. If a printer is attached, it should also be available as a generic Printer service defined in a Printer service specification, indistinguishable from a Printer service attached to a parallel port. Generic categories, such as a Printer service, should also be described in a Device Category.

It is expected that most Device service objects will actually represent a physical device in some form, but that is not a requirement of this specification. A Device service is represented as a normal service in the OSGi Framework and all coordination and activities are performed upon Framework services. This specification does not limit a bundle developer from using Framework mechanisms for services that are not related to physical devices.

103.2.1 Device Service Registration

A Device service is defined as a normal service registered with the Framework that either:

- Registers a service object under the interface org.osgi.service.Device with the Framework, or
- Sets the DEVICE_CATEGORY property in the registration. The value of DEVICE_CATEGORY is an array of String objects of all the device categories that the device belongs to. These strings are defined in the associated device category.

If this document mentions a Device service, it is meant to refer to services registered with the name org.osgi.service.device.Device *or* services registered with the DEVICE_CATEGORY property set.

When a Device service is registered, additional properties may be set that describe the device to the device manager and potentially to the end users. The following properties have their semantics defined in this specification:

- DEVICE_CATEGORY – A marker property indicating that this service must be regarded as a Device service by the device manager. Its value is of type String[], and its meaning is defined in the associated device category specification.
- DEVICE_DESCRIPTION – Describes the device to an end user. Its value is of type String.
- DEVICE_SERIAL – A unique serial number for this device. If the device hardware contains a serial number, the driver bundle is encouraged to specify it as this property. Different Device services representing the same physical hardware at different abstraction levels should set the same DEVICE_SERIAL, thus simplifying identification. Its value is of type String.
- service.pid – Service Persistent ID (PID), defined in org.osgi.framework.Constants. Device services should set this property. It must be unique among all registered services. Even different abstraction levels of the same device must use different PIDs. The service PIDs must be reproducible, so that every time the same hardware is plugged in, the same PIDs are used.

103.2.2 Device Service Attachment

When a Device service is registered with the Framework, the device manager is responsible for finding a suitable Driver service and instructing it to attach to the newly registered Device service. The Device service itself is passive: it only registers a Device service with the Framework and then waits until it is called.

The actual communication with the underlying physical device is not defined in the Device interface because it differs significantly between different types of devices. The Driver service is responsible for attaching the device in a device type-specific manner. The rules and interfaces for this process must be defined in the appropriate device category.

If the device manager is unable to find a suitable Driver service, the Device service remains unattached. In that case, if the service object implements the Device interface, it must receive a call to the noDriverFound() method. The Device service can wait until a new driver is installed, or it can unregister and attempt to register again with different properties that describe a more generic device or try a different configuration.

103.2.2.1 Idle Device Service

The main purpose of the device manager is to try to attach drivers to idle devices. For this purpose, a Device service is considered *idle* if no bundle that itself has registered a Driver service is using the Device service.

103.2.2.2 Device Service Unregistration

When a Device service is unregistered, no immediate action is required by the device manager. The normal service of unregistering events, provided by the Framework, takes care of propagating the unregistration information to affected drivers. Drivers must take the appropriate action to release this Device service and perform any necessary cleanup, as described in their device category specification.

The device manager may, however, take a device unregistration as an indication that driver bundles may have become idle and are thus eligible for removal. It is therefore important for Device services to unregister their service object when the underlying entity becomes unavailable.

103.3 Device Category Specifications

A device category specifies the rules and interfaces needed for the communication between a Device service and a Driver service. Only Device services and Driver services of the same device category can communicate and cooperate.

The Device Access service specification is limited to the attachment of Device services by Driver services, and does *not* enumerate different device categories.

Other specifications must specify a number of device categories before this specification can be made operational. Without a set of defined device categories, no inter-operability can be achieved.

Device categories are related to a specific device technology, such as USB, IEEE 1394, JINI, UPnP, Salutation, CEBus, Lonworks, and others. The purpose of a device category specification is to make all Device services of that category conform to an agreed interface, so that, for example, a USB Driver service of vendor A can control Device services from vendor B attached to a USB bus.

This specification is limited to defining the guidelines for device category definitions only. Device categories may be defined by the OSGi organization or by external specification bodies – for example, when these bodies are associated with a specific device technology.

103.3.1 Device Category Guidelines

A device category definition comprises the following elements:

- An interface that all devices belonging to this category must implement. This interface should lay out the rules of how to communicate with the underlying device. The specification body may define its own device interfaces (or classes) or leverage existing ones. For example, a serial port device category could use the javax.comm.SerialPort interface which is defined in [1] *Java Communications API*.

When registering a device belonging to this category with the Framework, the interface or class name for this category must be included in the registration.
- A set of service registration properties, their data types, and semantics, each of which must be declared as either MANDATORY or OPTIONAL for this device category.
- A range of match values specific to this device category. Matching is explained later in *The Device Attachment Algorithm* on page 78.

103.3.2 Sample Device Category Specification

The following is a partial example of a fictitious device category:

```
public interface /* com.acme.widget.*/ WidgetDevice {
   int MATCH_SERIAL                    = 10;
   int MATCH_VERSION                   =  8;
   int MATCH_MODEL                     =  6;
   int MATCH_MAKE                      =  4;
   int MATCH_CLASS                     =  2;
   void sendPacket( byte [] data );
   byte [] receivePacket( long timeout );
}
```

Devices in this category must implement the interface com.acme.widget.WidgetDevice to receive attachments from Driver services in this category.

Device properties for this fictitious category are defined in table Table 103.1.

Table 103.1 Example Device Category Properties, M=Mandatory, O=Optional

Property name	M/O	Type	Value
DEVICE_CATEGORY	M	String[]	{"Widget"}
com.acme.class	M	String	A class description of this device. For example "audio", "video", "serial", etc. An actual device category specification should contain an exhaustive list and define a process to add new classes.
com.acme.model	M	String	A definition of the model. This is usually vendor specific. For example "Mouse".
com.acme.manufacturer	M	String	Manufacturer of this device, for example "ACME Widget Division".
com.acme.revision	O	String	Revision number. For example, "42".
com.acme.serial	O	String	A serial number. For example "SN6751293-12-2112/A".

103.3.3 Match Example

Driver services and Device services are connected via a matching process that is explained in *The Device Attachment Algorithm* on page 78. The Driver service plays a pivotal role in this matching process. It must inspect the Device service (from its ServiceReference object) that has just been registered and decide if it potentially could cooperate with this Device service.

It must be able to answer a value indicating the quality of the match. The scale of this match value must be defined in the device category so as to allow Driver services to match on a fair basis. The scale must start at least at 1 and go upwards.

Driver services for this sample device category must return one of the match codes defined in the com.acme.widget.WidgetDevice interface or Device.MATCH_NONE if the Device service is not recognized. The device category must define the exact rules for the match codes in the device category specification. In this example, a small range from 2 to 10 (MATCH_NONE is 0) is defined for WidgetDevice devices. They are named in the WidgetDevice interface for convenience and have the following semantics.

Table 103.2 Sample Device Category Match Scale

Match name	Value	Description
MATCH_SERIAL	10	An exact match, including the serial number.
MATCH_VERSION	8	Matches the right class, make model, and version.
MATCH_MODEL	6	Matches the right class and make model.
MATCH_MAKE	4	Matches the make.
MATCH_CLASS	2	Only matches the class.

A Driver service should use the constants to return when it decides how closely the Device service matches its suitability. For example, if it matches the exact serial number, it should return MATCH_SERIAL.

103.4 Driver Services

A Driver service is responsible for attaching to suitable Device services under control of the device manager. Before it can attach a Device service, however, it must compete with other Driver services for control.

If a Driver service wins the competition, it must attach the device in a device category-specific way. After that, it can perform its intended functionality. This functionality is not defined here nor in the device category; this specification only describes the behavior of the Device service, not how the Driver service uses it to implement its intended functionality. A Driver service may register one or more new Device services of another device category or a generic service which models a more refined form of the device.

Both refined Device services as well as generic services should be defined in a Device Category. See *Device Category Specifications* on page 67.

103.4.1 Driver Bundles

A Driver service is, like *all* services, implemented in a bundle, and is recognized by the device manager by registering one or more Driver service objects with the Framework.

Such bundles containing one or more Driver services are called *driver bundles*. The device manager must be aware of the fact that the cardinality of the relationship between bundles and Driver services is 1:1...*.

A driver bundle must register *at least* one Driver service in its BundleActivator.start implementation.

103.4.2 Driver Taxonomy

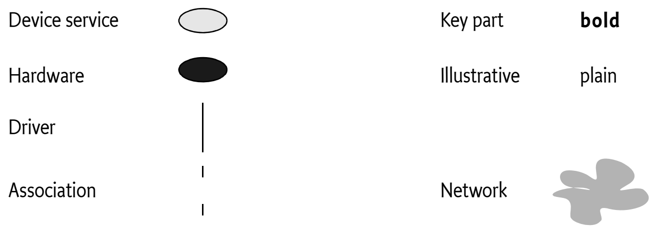
Device Drivers may belong to one of the following categories:

- Base Drivers (Discovery, Pure Discovery and Normal)
- Refining Drivers
- Network Drivers
- Composite Drivers
- Referring Drivers
- Bridging Drivers

- Multiplexing Drivers
- Pure Consuming Drivers

This list is not definitive, and a Driver service is not required to fit into one of these categories. The purpose of this taxonomy is to show the different topologies that have been considered for the Device Access service specification.

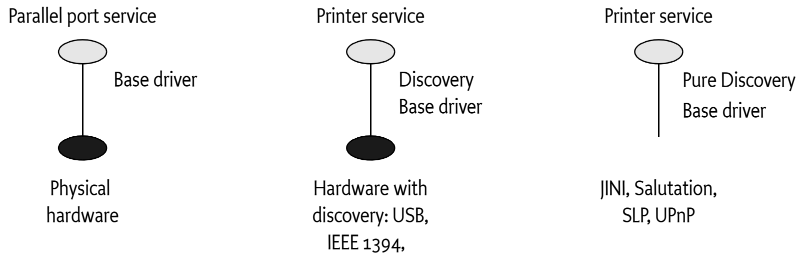
Figure 103.2 Legend for Device Driver Services Taxonomy



103.4.2.1 Base Drivers

The first category of device drivers are called *base drivers* because they provide the lowest-level representation of a physical device. The distinguishing factor is that they are not registered as Driver services because they do not have to compete for access to their underlying technology.

Figure 103.3 Base Driver Types



Base drivers discover physical devices using code not specified here (for example, through notifications from a device driver in native code) and then register corresponding Device services.

When the hardware supports a discovery mechanism and reports a physical device, a Device service is then registered. Drivers supporting a discovery mechanism are called *discovery base drivers*.

An example of a discovery base driver is a USB driver. Discovered USB devices are registered with the Framework as a generic USB Device service. The USB specification (see [2] *USB Specification*) defines a tightly integrated discovery method. Further, devices are individually addressed; no provision exists for broadcasting a message to all devices attached to the USB bus. Therefore, there is no reason to expose the USB network itself; instead, a discovery base driver can register the individual devices as they are discovered.

Not all technologies support a discovery mechanism. For example, most serial ports do not support detection, and it is often not even possible to detect whether a device is attached to a serial port.

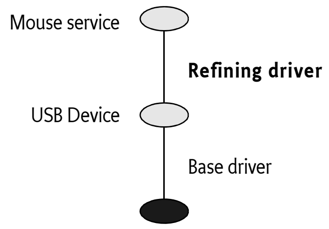
Although each driver bundle should perform discovery on its own, a driver for a non-discoverable serial port requires external help – either through a user interface or by allowing the Configuration Admin service to configure it.

It is possible for the driver bundle to combine automatic discovery of Plug and Play-compliant devices with manual configuration when non-compliant devices are plugged in.

103.4.2.2 Refining Drivers

The second category of device drivers are called *refining drivers*. Refining drivers provide a refined view of a physical device that is already represented by another Device service registered with the Framework. Refining drivers register a Driver service with the Framework. This Driver service is used by the device manager to attach the refining driver to a less refined Device service that is registered as a result of events within the Framework itself.

Figure 103.4 Refining Driver Diagram



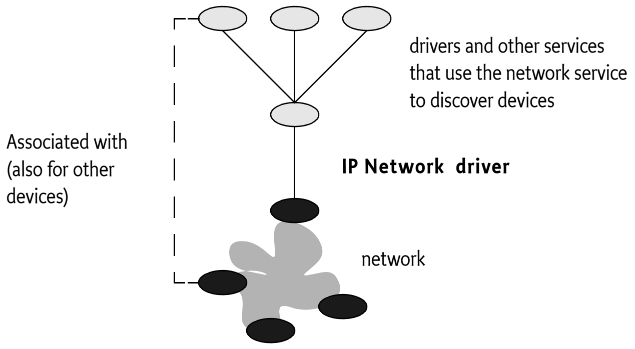
An example of a refining driver is a mouse driver, which is attached to the generic USB Device service representing a physical mouse. It then registers a new Device service which represents it as a Mouse service, defined elsewhere.

The majority of drivers fall into the refining driver type.

103.4.2.3 Network Drivers

An Internet Protocol (IP) capable network such as Ethernet supports individually addressable devices and allows broadcasts, but does not define an intrinsic discovery protocol. In this case, the entire network should be exposed as a single Device service.

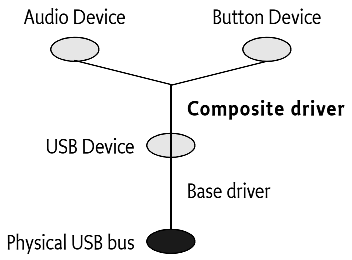
Figure 103.5 Network Driver diagram


drivers and other services that use the network service to discover devices
Associated with (also for other devices)
IP Network driver
network

103.4.2.4 Composite Drivers

Complex devices can often be broken down into several parts. Drivers that attach to a single service and then register multiple Device services are called *composite drivers*. For example, a USB speaker containing software-accessible buttons can be registered by its driver as two separate Device services: an Audio Device service and a Button Device service.

Figure 103.6 Composite Driver structure



This approach can greatly reduce the number of interfaces needed, as well as enhance reusability.

103.4.2.5 Referring Drivers

A referring driver is actually not a driver in the sense that it controls Device services. Instead, it acts as an intermediary to help locate the correct driver bundle. This process is explained in detail in *The Device Attachment Algorithm* on page 78.

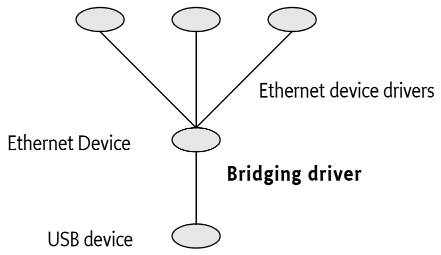
A referring driver implements the call to the `attach` method to inspect the Device service, and decides which Driver bundle would be able to attach to the device. This process can actually involve connecting to the physical device and communicating with it. The `attach` method then returns a `String` object that indicates the `DRIVER_ID` of another driver bundle. This process is called a referral.

For example, a vendor ACME can implement one driver bundle that specializes in recognizing all of the devices the vendor produces. The referring driver bundle does not contain code to control the device – it contains only sufficient logic to recognize the assortment of devices. This referring driver can be small, yet can still identify a large product line. This approach can drastically reduce the amount of downloading and matching needed to find the correct driver bundle.

103.4.2.6 Bridging Drivers

A bridging driver registers a Device service from one device category but attaches it to a Device service from another device category.

Figure 103.7 Bridging Driver Structure



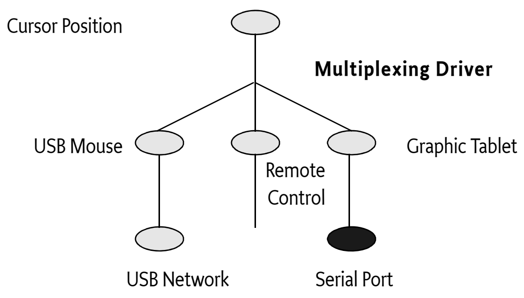
For example, USB to Ethernet bridges exist that allow connection to an Ethernet network through a USB device. In this case, the top level of the USB part of the Device service stack would be an Ethernet Device service. But the same Ethernet Device service can also be the bottom layer of an Ethernet layer of the Device service stack. A few layers up, a bridge could connect into yet another network.

The stacking depth of Device services has no limit, and the same drivers could in fact appear at different levels in the same Device service stack. The graph of drivers-to-Device services roughly mirrors the hardware connections.

103.4.2.7 Multiplexing Drivers

A *multiplexing driver* attaches a number of Device services and aggregates them in a new Device service.

Figure 103.8 Multiplexing Driver Structure

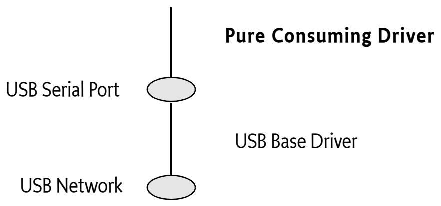


For example, assume that a system has a mouse on USB, a graphic tablet on a serial port, and a remote control facility. Each of these would be registered as a service with the Framework. A multiplexing driver can attach all three, and can merge the different positions in a central Cursor Position service.

103.4.2.8 Pure Consuming Drivers

A *pure consuming driver* bundle will attach to devices without registering a refined version.

Figure 103.9 Pure Consuming Driver Structure



For example, one driver bundle could decide to handle all serial ports through javax.comm instead of registering them as services. When a USB serial port is plugged in, one or more Driver services are attached, resulting in a Device service stack with a Serial Port Device service. A pure consuming driver may then attach to the Serial Port Device service and register a new serial port with the javax.comm.* registry instead of the Framework service registry. This registration effectively transfers the device from the OSGi environment into another environment.

103.4.2.9 Other Driver Types

It should be noted that any bundle installed in the OSGi environment may get and use a Device service without having to register a Driver service.

The following functionality is offered to those bundles that do register a Driver service and conform to the this specification:

- The bundles can be installed and uninstalled on demand.
- Attachment to the Device service is only initiated after the winning the competition with other drivers.

103.4.3 Driver Service Registration

Drivers are recognized by registering a Driver service with the Framework. This event makes the device manager aware of the existence of the Driver service. A Driver service registration must have a DRIVER_ID property whose value is a String object, uniquely identifying the driver to the device manager. The device manager must use the DRIVER_ID to prevent the installation of duplicate copies of the same driver bundle.

Therefore, this DRIVER_ID must:

- Depend only on the specific behavior of the driver, and thus be independent of unrelated aspects like its location or mechanism of downloading.
- Start with the reversed form of the domain name of the company that implements it: for example, com.acme.widget.1.1.
- Differ from the DRIVER_ID of drivers with different behavior. Thus, it must *also* be different for each revision of the same driver bundle so they may be distinguished.

When a new Driver service is registered, the Device Attachment Algorithm must be applied to each idle Device service. This requirement gives the new Driver service a chance to compete with other Driver services for attaching to idle devices. The techniques outlined in *Optimizations* on page 81 can provide significant shortcuts for this situation.

As a result, the Driver service object can receive match and attach requests before the method which registered the service has returned.

This specification does not define any method for new Driver services to *steal* already attached devices. Once a Device service has been attached by a Driver service, it can only be released by the Driver service itself.

103.4.4 Driver Service Unregistration

When a Driver service is unregistered, it must release all Device services to which it is attached. Thus, *all* its attached Device services become idle. The device manager must gather all of these idle Device services and try to re-attach them. This condition gives other Driver services a chance to take over the refinement of devices after the unregistering driver. The techniques outlined in *Optimizations* on page 81 can provide significant shortcuts for this situation.

A Driver service that is installed by the device manager must remain registered as long as the driver bundle is active. Therefore, a Driver service should only be unregistered if the driver bundle is stopping, an occurrence which may precede its being uninstalled or updated. Driver services should thus not unregister in an attempt to minimize resource consumption. Such optimizations can easily introduce race conditions with the device manager.

103.4.5 Driver Service Methods

The Driver interface consists of the following methods:

- match(ServiceReference) – This method is called by the device manager to find out how well this Driver service matches the Device service as indicated by the ServiceReference argument. The value returned here is specific for a device category. If this Device service is of another device category, the value Device.MATCH_NONE must be returned. Higher values indicate a better match. For the exact matching algorithm, see *The Device Attachment Algorithm* on page 78.
  Driver match values and referrals must be deterministic, in that repeated calls for the same Device service must return the same results so that results can be cached by the device manager.
- attach(ServiceReference) – If the device manager decides that a Driver service should be attached to a Device service, it must call this method on the Driver service object. Once this method is called, the Device service is regarded as attached to that Driver service, and no other Driver service must be called to attach to the Device service. The Device service must remain *owned* by the Driver service until the Driver bundle is stopped. No unattach method exists.
  The attach method should return null when the Device service is correctly attached. A referring driver (see *Referring Drivers* on page 72) can return a String object that specifies the DRIVER_ID of a driver that can handle this Device service. In this case, the Device service is not attached and the device manager must attempt to install a Driver service with the same DRIVER_ID via a Driver Locator service. The attach method must be deterministic as described in the previous method.

103.4.6 Idle Driver Bundles

An idle Driver bundle is a bundle with a registered Driver service, and is not attached to any Device service. Idle Driver bundles are consuming resources in the OSGi Service Platform. The device manager should uninstall bundles that it has installed and which are idle.

103.5 Driver Locator Service

The device manager must automatically install Driver bundles, which are obtained from Driver Locator services, when new Device services are registered.

A Driver Locator service encapsulates the knowledge of how to fetch the Driver bundles needed for a specific Device service. This selection is made on the properties that are registered with a device: for example, DEVICE_CATEGORY and any other properties registered with the Device service registration.

The purpose of the Driver Locator service is to separate the mechanism from the policy. The decision to install a new bundle is made by the device manager (the mechanism), but a Driver Locator service decides which bundle to install and from where the bundle is downloaded (the policy).

Installing bundles has many consequences for the security of the system, and this process is also sensitive to network setup and other configuration details. Using Driver Locator services allows the Operator to choose a strategy that best fits its needs.

Driver services are identified by the DRIVER_ID property. Driver Locator services use this particular ID to identify the bundles that can be installed. Driver ID properties have uniqueness requirements as specified in *Device Service Registration* on page 66. This uniqueness allows the device manager to maintain a list of Driver services and prevent unnecessary installs.

An OSGi Service Platform can have several different Driver Locator services installed. The device manager must consult all of them and use the combined result set, after pruning duplicates based on the DRIVER_ID values.

103.5.1 The DriverLocator Interface

The DriverLocator interface allows suitable driver bundles to be located, downloaded, and installed on demand, even when completely unknown devices are detected.

It has the following methods:

- findDrivers(Dictionary) – This method returns an array of driver IDs that potentially match a service described by the properties in the Dictionary object. A driver ID is the String object that is registered by a Driver service under the DRIVER_ID property.
- loadDriver(String) – This method returns an InputStream object that can be used to download the bundle containing the Driver service as specified by the driver ID argument. If the Driver Locator service cannot download such a bundle, it should return null. Once this bundle is downloaded and installed in the Framework, it must register a Driver service with the DRIVER_ID property set to the value of the String argument.

103.5.2 A Driver Example

The following example shows a very minimal Driver service implementation. It consists of two classes. The first class is SerialWidget. This class tracks a single WidgetDevice from *Sample Device Category Specification* on page 68. It registers a javax.comm.SerialPort service, which is a general serial port specification that could also be implemented from other device categories like USB, a COM port, etc. It is created when the SerialWidgetDriver object is requested to attach a WidgetDevice by the device manager. It registers a new javax.comm.SerialPort service in its constructor.

The org.osgi.util.tracker.ServiceTracker is extended to handle the Framework events that are needed to simplify tracking this service. The removedService method of this class is overridden to unregister the SerialPort when the underlying WidgetDevice is unregistered.

```
package com.acme.widget;
import org.osgi.service.device.*;
import org.osgi.framework.*;
import org.osgi.util.tracker.*;

class SerialWidget extends ServiceTracker
  implements javax.comm.SerialPort,
    org.osgi.service.device.Constants {
  ServiceRegistration       registration;

  SerialWidget( BundleContext c, ServiceReference r ) {
    super( c, r, null );
    open();
  }

  public Object addingService( ServiceReference ref ) {
    WidgetDevice dev = (WidgetDevice)
      context.getService( ref );
    registration = context.registerService(
        javax.comm.SerialPort.class.getName(),
        this,
        null );
      return dev;
  }

  public void removedService( ServiceReference ref,
    Object service ) {
    registration.unregister();
    context.ungetService(ref);
  }
  ... methods for javax.comm.SerialPort that are
  ... converted to underlying WidgetDevice
}
```

A SerialWidgetDriverobject is registered with the Framework in the Bundle Activator start method under the Driver interface. The device manager must call the match method for each idle Device service that is registered. If it is chosen by the device manager to control this Device service, a new SerialWidget is created that offers serial port functionality to other bundles.

```
public class SerialWidgetDriver implements Driver {
  BundleContext          context;

  String    spec =
      "(&"
    +" (objectclass=com.acme.widget.WidgetDevice)"
    +"  (DEVICE_CATEGORY=WidgetDevice)"
    +"  (com.acme.class=Serial)"
    + ")";

  Filter    filter;
```

```
        SerialWidgetDriver( BundleContext context )
            throws Exception {
            this.context = context;
            filter = context.createFilter(spec);
        }
        public int match( ServiceReference d ) {
            if ( filter.match( d ) )
                return WidgetDevice.MATCH_CLASS;
            else
                return Device.MATCH_NONE;
        }
        public synchronized String attach(ServiceReference r){
            new SerialWidget( context, r );
        }
    }
```

103.6 The Driver Selector Service

The purpose of the Driver Selector service is to customize the selection of the best Driver service from a set of suitable Driver bundles. The device manager has a default algorithm as described in *The Device Attachment Algorithm* on page 78. When this algorithm is not sufficient and requires customizing by the operator, a bundle providing a Driver Selector service can be installed in the Framework. This service must be used by the device manager as the final arbiter when selecting the best match for a Device service.

The Driver Selector service is a singleton; only one such service is recognized by the device manager. The Framework method BundleContext.getServiceReference must be used to obtain a Driver Selector service. In the erroneous case that multiple Driver Selector services are registered, the service.ranking property will thus define which service is actually used.

A device manager implementation must invoke the method select(ServiceReference,Match[]). This method receives a Service Reference to the Device service and an array of Match objects. Each Match object contains a link to the ServiceReference object of a Driver service and the result of the match value returned from a previous call to Driver.match. The Driver Selector service should inspect the array of Match objects and use some means to decide which Driver service is best suited. The index of the best match should be returned. If none of the Match objects describe a possible Driver service, the implementation must return DriverSelector.SELECT_NONE (-1).

103.7 Device Manager

Device Access is controlled by the device manager in the background. The device manager is responsible for initiating all actions in response to the registration, modification, and unregistration of Device services and Driver services, using Driver Locator services and a Driver Selector service as helpers.

The device manager detects the registration of Device services and coordinates their attachment with a suitable Driver service. Potential Driver services do not have to be active in the Framework to be eligible. The device manager must use Driver Locator services to find bundles that might be suitable for the detected Device service and that are not currently installed. This selection is done via a DRIVER_ID property that is unique for each Driver service.

The device manager must install and start these bundles with the help of a Driver Locator service. This activity must result in the registration of one or more Driver services. All available Driver services, installed by the device manager and also others, then participate in a bidding process. The Driver service can inspect the Device service through its `ServiceReference` object to find out how well this Driver service matches the Device service.

If a Driver Selector service is available in the Framework service registry, it is used to decide which of the eligible Driver services is the best match.

If no Driver Selector service is available, the highest bidder must win, with tie breaks defined on the `service.ranking` and `service.id` properties. The selected Driver service is then asked to `attach` the Device service.

If no Driver service is suitable, the Device service remains idle. When new Driver bundles are installed, these idle Device services must be reattached.

The device manager must reattach a Device service if, at a later time, a Driver service is unregistered due to an uninstallation or update. At the same time, however, it should prevent superfluous and non-optimal reattachments. The device manager should also garbage-collect driver bundles it installed which are no longer used.

The device manager is a singleton. Only one device manager may exist, and it must have no public interface.

103.7.1 Device Manager Startup

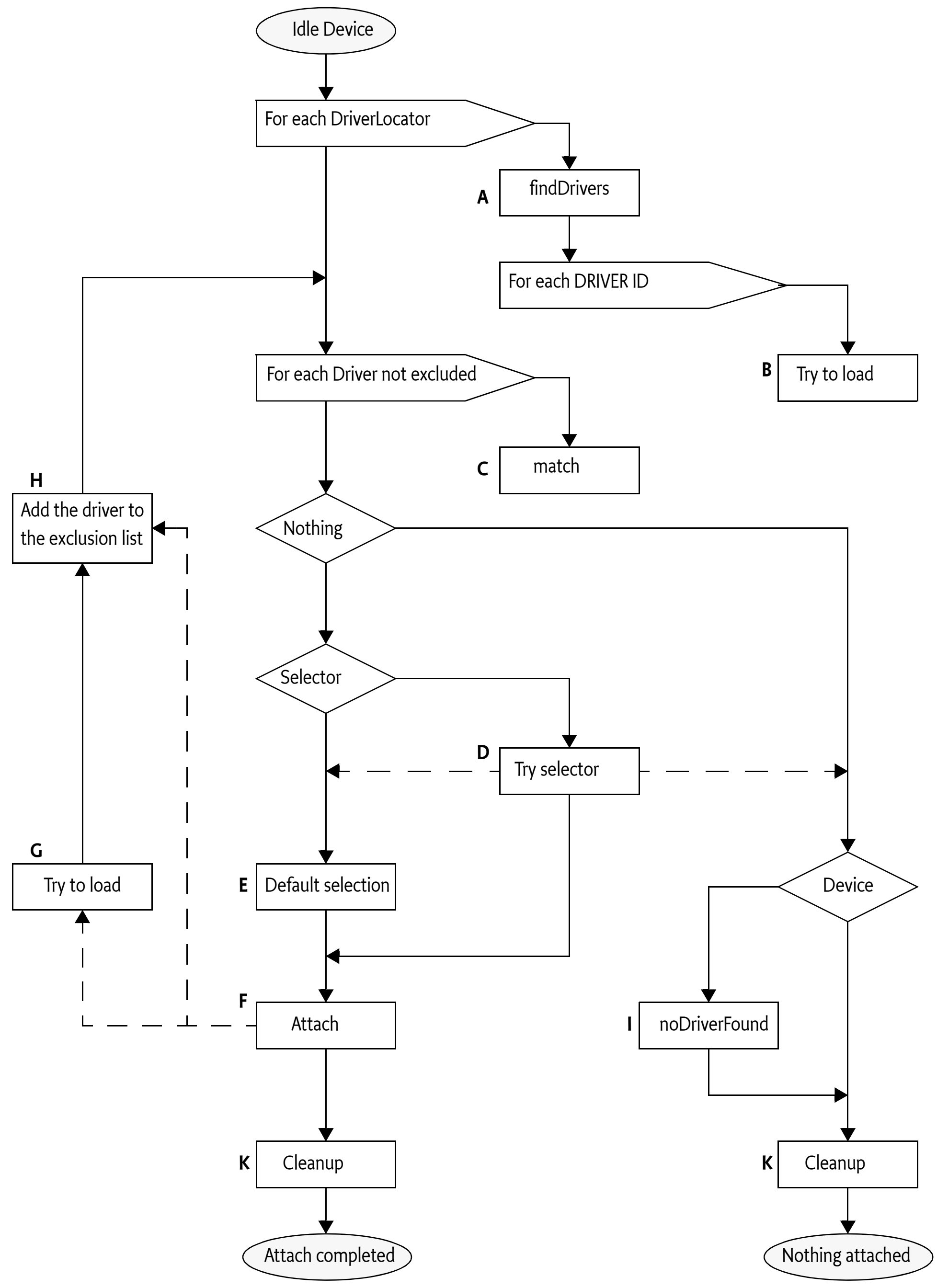
To prevent race conditions during Framework startup, the device manager must monitor the state of Device services and Driver services immediately when it is started. The device manager must not, however, begin attaching Device services until the Framework has been fully started, to prevent superfluous or non-optimal attachments.

The Framework has completed starting when the `FrameworkEvent.STARTED` event has been published. Publication of that event indicates that Framework has finished all its initialization and all bundles are started. If the device manager is started after the Framework has been initialized, it should detect the state of the Framework by examining the state of the system bundle.

103.7.2 The Device Attachment Algorithm

A key responsibility of the device manager is to attach refining drivers to idle devices. The following diagram illustrates the device attachment algorithm.

Figure 103.10 Device Attachment Algorithm



103.7.3 Legend

Table 103.3 *Driver attachment algorithm*

Step	Description
A	DriverLocator.findDrivers is called for each registered Driver Locator service, passing the properties of the newly detected Device service. Each method call returns zero or more DRIVER_ID values (identifiers of particular driver bundles). If the findDrivers method throws an exception, it is ignored, and processing continues with the next Driver Locator service. See *Optimizations* on page 81 for further guidance on handling exceptions.
B	For each found DRIVER_ID that does not correspond to an already registered Driver service, the device manager calls DriverLocator.loadDriver to return an InputStream containing the driver bundle. Each call to loadDriver is directed to one of the Driver Locator services that mentioned the DRIVER_ID in step A. If the loadDriver method fails, the other Driver Locator objects are tried. If they all fail, the driver bundle is ignored. If this method succeeds, the device manager installs and starts the driver bundle. Driver bundles must register their Driver services synchronously during bundle activation.
C	For each Driver service, except those on the exclusion list, call its Driver.match method, passing the ServiceReference object to the Device service. Collect all successful matches – that is, those whose return values are greater than Device.MATCH_NONE – in a list of active matches. A match call that throws an exception is considered unsuccessful and is not added to the list.
D	If there is a Driver Selector service, the device manager calls the DriverSelector.select method, passing the array of active Match objects. If the Driver Selector service returns the index of one of the Match objects from the array, its associated Driver service is selected for attaching the Device service. If the Driver Selector service returns DriverSelector.SELECT_NONE, no Driver service must be considered for attaching the Device service. If the Driver Selector service throws an exception or returns an invalid result, the default selection algorithm is used. Only one Driver Selector service is used, even if there is more than one registered in the Framework. See *The Driver Selector Service* on page 77.
E	The winner is the one with the highest match value. Tie breakers are respectively: • Highest service.ranking property. • Lowest service.id property.
F	The selected Driver service's attach method is called. If the attach method returns null, the Device service has been successfully attached. If the attach method returns a String object, it is interpreted as a referral to another Driver service and processing continues at G. See *Referring Drivers* on page 72. If an exception is thrown, the Driver service has failed, and the algorithm proceeds to try another Driver service after excluding this one from further consideration at Step H.

Table 103.3 Driver attachment algorithm

Step	Description
G	The device manager attempts to load the referred driver bundle in a manner similar to Step B, except that it is unknown which Driver Locator service to use. Therefore, the loadDriver method must be called on each Driver Locator service until one succeeds (or they all fail). If one succeeds, the device manager installs and starts the driver bundle. The driver bundle must register a Driver service during its activation which must be added to the list of Driver services in this algorithm.
H	The referring driver bundle is added to the exclusion list. Because each new referral adds an entry to the exclusion list, which in turn disqualifies another driver from further matching, the algorithm cannot loop indefinitely. This list is maintained for the duration of this algorithm. The next time a new Device service is processed, the exclusion list starts out empty.
I	If no Driver service attached the Device service, the Device service is checked to see whether it implements the Device interface. If so, the noDriverFound method is called. Note that this action may cause the Device service to unregister and possibly a new Device service (or services) to be registered in its place. Each new Device service registration must restart the algorithm from the beginning.
K	Whether an attachment was successful or not, the algorithm may have installed a number of driver bundles. The device manager should remove any idle driver bundles that it installed.

103.7.4 Optimizations

Optimizations are explicitly allowed and even recommended for an implementation of a device manager. Implementations may use the following assumptions:

- Driver match values and referrals must be deterministic, in that repeated calls for the same Device service must return the same results.
- The device manager may cache match values and referrals. Therefore, optimizations in the device attachment algorithm based on this assumption are allowed.
- The device manager may delay loading a driver bundle until it is needed. For example, a delay could occur when that DRIVER_ID's match values are cached.
- The results of calls to DriverLocator and DriverSelector methods are not required to be deterministic, and must not be cached by the device manager.
- Thrown exceptions must not be cached. Exceptions are considered transient failures, and the device manager must always retry a method call even if it has thrown an exception on a previous invocation with the same arguments.

103.7.5 Driver Bundle Reclamation

The device manager may remove driver bundles it has installed at any time, provided that all the Driver services in that bundle are idle. This recommended practice prevents unused driver bundles from accumulating over time. Removing driver bundles too soon, however, may cause unnecessary installs and associated delays when driver bundles are needed again.

If a device manager implements driver bundle reclamation, the specified matching algorithm is not guaranteed to terminate unless the device manager takes reclamation into account.

For example, assume that a new Device service triggers the attachment algorithm. A driver bundle recommended by a Driver Locator service is loaded. It does not match, so the Device service remains idle. The device manager is eager to reclaim space, and unloads the driver bundle. The disappearance of the Driver service causes the device manager to reattach idle devices. Because it has not kept a record of its previous activities, it tries to reattach the same device, which closes the loop.

On systems where the device manager implements driver bundle reclamation, all refining drivers should be loaded through Driver Locator services. This recommendation is intended to prevent the device manager from erroneously uninstalling pre-installed driver bundles that cannot later be reinstalled when needed.

The device manager can be updated or restarted. It cannot, however, rely on previously stored information to determine which driver bundles were pre-installed and which were dynamically installed and thus are eligible for removal. The device manager may persistently store cachable information for optimization, but must be able to cold start without any persistent information and still be able to manage an existing connection state, satisfying all of the requirements in this specification.

103.7.6 Handling Driver Bundle Updates

It is not straightforward to determine whether a driver bundle is being updated when the UNREGISTER event for a Driver service is received. In order to facilitate this distinction, the device manager should wait for a period of time after the unregistration for one of the following events to occur:

- A BundleEvent.UNINSTALLED event for the driver bundle.
- A ServiceEvent.REGISTERED event for another Driver service registered by the driver bundle.

If the driver bundle is uninstalled, or if neither of the above events are received within the allotted time period, the driver is assumed to be inactive. The appropriate waiting period is implementation-dependent and will vary for different installations. As a general rule, this period should be long enough to allow a driver to be stopped, updated, and restarted under normal conditions, and short enough not to cause unnecessary delays in reattaching devices. The actual time should be configurable.

103.7.7 Simultaneous Device Service and Driver Service Registration

The device attachment algorithm may discover new driver bundles that were installed outside its direct control, which requires executing the device attachment algorithm recursively. However, in this case, the appearance of the new driver bundles should be queued until completion of the current device attachment algorithm.

Only one device attachment algorithm may be in progress at any moment in time.

The following example sequence illustrates this process when a Driver service is registered:

- Collect the set of all idle devices.
- Apply the device attachment algorithm to each device in the set.
- If no Driver services were registered during the execution of the device attachment algorithm, processing terminates.
- Otherwise, restart this process.

103.8 Security

The device manager is the only privileged bundle in the Device Access specification and requires the org.osgi.framework.AdminPermission with the LIFECYCLE action to install and uninstall driver bundles.

The device manager itself should be free from any knowledge of policies and should not actively set bundle permissions. Rather, if permissions must be set, it is up to the Management Agent to listen to synchronous bundle events and set the appropriate permissions.

Driver Locator services can trigger the download of any bundle, because they deliver the content of a bundle to the privileged device manager and could potentially insert a Trojan horse into the environment. Therefore, Driver Locator bundles need the `ServicePermission[DriverLocator, REGISTER]` to register Driver Locator services, and the operator should exercise prudence in assigning this `ServicePermission`.

Bundles with Driver Selector services only require `ServicePermission[DriverSelector, REGISTER]` to register the `DriverSelector` service. The Driver Selector service can play a crucial role in the selection of a suitable Driver service, but it has no means to define a specific bundle itself.

103.9 org.osgi.service.device

Device Access Package Version 1.1.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.device; version="[1.1,2.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.device; version="[1.1,1.2)"`

103.9.1 Summary

- `Constants` – This interface defines standard names for property keys associated with `Device` and `Driver` services.
- `Device` – Interface for identifying device services.
- `Driver` – A `Driver` service object must be registered by each Driver bundle wishing to attach to Device services provided by other drivers.
- `DriverLocator` – A Driver Locator service can find and load device driver bundles given a property set.
- `DriverSelector` – When the device manager detects a new Device service, it calls all registered Driver services to determine if anyone matches the Device service.
- `Match` – Instances of `Match` are used in the `DriverSelector.select(ServiceReference, Match[])` method to identify Driver services matching a Device service.

103.9.2 Permissions

103.9.3 public interface Constants

This interface defines standard names for property keys associated with `Device` and `Driver` services.

The values associated with these keys are of type `java.lang.String`, unless otherwise stated.

See Also `Device` , `Driver`

Since 1.1

No Implement Consumers of this API must not implement this interface

103.9.3.1 public static final String DEVICE_CATEGORY = "DEVICE_CATEGORY"

Property (named "DEVICE_CATEGORY") containing a human readable description of the device categories implemented by a device. This property is of type `String[]`

Services registered with this property will be treated as devices and discovered by the device manager

103.9.3.2 public static final String DEVICE_DESCRIPTION = “DEVICE_DESCRIPTION”

Property (named “DEVICE_DESCRIPTION”) containing a human readable string describing the actual hardware device.

103.9.3.3 public static final String DEVICE_SERIAL = “DEVICE_SERIAL”

Property (named “DEVICE_SERIAL”) specifying a device’s serial number.

103.9.3.4 public static final String DRIVER_ID = “DRIVER_ID”

Property (named “DRIVER_ID”) identifying a driver.

A `DRIVER_ID` should start with the reversed domain name of the company that implemented the driver (e.g., `com.acme`), and must meet the following requirements:

- It must be independent of the location from where it is obtained.
- It must be independent of the `DriverLocator` service that downloaded it.
- It must be unique.
- It must be different for different revisions of the same driver.

This property is mandatory, i.e., every `Driver` service must be registered with it.

103.9.4 public interface Device

Interface for identifying device services.

A service must implement this interface or use the `Constants.DEVICE_CATEGORY` registration property to indicate that it is a device. Any services implementing this interface or registered with the `DEVICE_CATEGORY` property will be discovered by the device manager.

Device services implementing this interface give the device manager the opportunity to indicate to the device that no drivers were found that could (further) refine it. In this case, the device manager calls the `noDriverFound()` method on the `Device` object.

Specialized device implementations will extend this interface by adding methods appropriate to their device category to it.

See Also `Driver`

Concurrency Thread-safe

103.9.4.1 public static final int MATCH_NONE = 0

Return value from `Driver.match(ServiceReference)` indicating that the driver cannot refine the device presented to it by the device manager. The value is zero.

103.9.4.2 public void noDriverFound ( )

□ Indicates to this `Device` object that the device manager has failed to attach any drivers to it.

If this `Device` object can be configured differently, the driver that registered this `Device` object may unregister it and register a different Device service instead.

103.9.5 public interface Driver

A `Driver` service object must be registered by each Driver bundle wishing to attach to Device services provided by other drivers. For each newly discovered `Device` object, the device manager enters a bidding phase. The `Driver` object whose `match(ServiceReference)` method bids the highest for a particular `Device` object will be instructed by the device manager to attach to the `Device` object.

See Also `Device` , `DriverLocator`

Concurrency Thread-safe

103.9.5.1 public String attach (ServiceReference reference) throws Exception

reference the `ServiceReference` object of the device to attach to

- Attaches this Driver service to the Device service represented by the given `ServiceReference` object.

A return value of `null` indicates that this Driver service has successfully attached to the given Device service. If this Driver service is unable to attach to the given Device service, but knows of a more suitable Driver service, it must return the `DRIVER_ID` of that Driver service. This allows for the implementation of referring drivers whose only purpose is to refer to other drivers capable of handling a given Device service.

After having attached to the Device service, this driver may register the underlying device as a new service exposing driver-specific functionality.

This method is called by the device manager.

Returns `null` if this Driver service has successfully attached to the given Device service, or the `DRIVER_ID` of a more suitable driver

Throws `Exception` – if the driver cannot attach to the given device and does not know of a more suitable driver

103.9.5.2 public int match (ServiceReference reference) throws Exception

reference the `ServiceReference` object of the device to match

- Checks whether this Driver service can be attached to the Device service. The Device service is represented by the given `ServiceReference` and returns a value indicating how well this driver can support the given Device service, or `Device.MATCH_NONE` if it cannot support the given Device service at all.

The return value must be one of the possible match values defined in the device category definition for the given Device service, or `Device.MATCH_NONE` if the category of the Device service is not recognized.

In order to make its decision, this Driver service may examine the properties associated with the given Device service, or may get the referenced service object (representing the actual physical device) to talk to it, as long as it ungets the service and returns the physical device to a normal state before this method returns.

A Driver service must always return the same match code whenever it is presented with the same Device service.

The match function is called by the device manager during the matching process.

Returns value indicating how well this driver can support the given Device service, or `Device.MATCH_NONE` if it cannot support the Device service at all

Throws `Exception` – if this Driver service cannot examine the Device service

103.9.6 public interface DriverLocator

A Driver Locator service can find and load device driver bundles given a property set. Each driver is represented by a unique `DRIVER_ID`.

Driver Locator services provide the mechanism for dynamically downloading new device driver bundles into an OSGi environment. They are supplied by providers and encapsulate all provider-specific details related to the location and acquisition of driver bundles.

See Also `Driver`

Concurrency Thread-safe

103.9.6.1 public String[] findDrivers (Dictionary props)

props the properties of the device for which a driver is sought

☐ Returns an array of DRIVER_ID strings of drivers capable of attaching to a device with the given properties.

The property keys in the specified Dictionary objects are case-insensitive.

Returns array of driver DRIVER_ID strings of drivers capable of attaching to a Device service with the given properties, or null if this Driver Locator service does not know of any such drivers

103.9.6.2 public InputStream loadDriver (String id) throws IOException

id the DRIVER_ID of the driver that needs to be installed.

☐ Get an InputStream from which the driver bundle providing a driver with the giving DRIVER_ID can be installed.

Returns An InputStream object from which the driver bundle can be installed or null if the driver with the given ID cannot be located

Throws IOException – the input stream for the bundle cannot be created

103.9.7 public interface DriverSelector

When the device manager detects a new Device service, it calls all registered Driver services to determine if anyone matches the Device service. If at least one Driver service matches, the device manager must choose one. If there is a Driver Selector service registered with the Framework, the device manager will ask it to make the selection. If there is no Driver Selector service, or if it returns an invalid result, or throws an Exception, the device manager uses the default selection strategy.

Since 1.1

Concurrency Thread-safe

103.9.7.1 public static final int SELECT_NONE = -1

Return value from DriverSelector.select, if no Driver service should be attached to the Device service. The value is -1.

103.9.7.2 public int select (ServiceReference reference , Match[] matches)

reference the ServiceReference object of the Device service.

matches the array of all non-zero matches.

☐ Select one of the matching Driver services. The device manager calls this method if there is at least one driver bidding for a device. Only Driver services that have responded with nonzero (not Device.MATCH_NONE) match values will be included in the list.

Returns index into the array of Match objects, or SELECT_NONE if no Driver service should be attached

103.9.8 public interface Match

Instances of Match are used in the DriverSelector.select(ServiceReference, Match[]) method to identify Driver services matching a Device service.

See Also DriverSelector

Since 1.1

Concurrency Thread-safe

No Implement Consumers of this API must not implement this interface

103.9.8.1 public ServiceReference getDriver ( )

- Return the reference to a Driver service.

Returns ServiceReference object to a Driver service.

103.9.8.2 public int getMatchValue ( )

- Return the match value of this object.

Returns the match value returned by this Driver service.

103.10 References

[1] *Java Communications API*
http://www.oracle.com/technetwork/java/index-jsp-141752.html

[2] *USB Specification*
http://www.usb.org

[3] *Universal Plug and Play*
http://www.upnp.org

[4] *Jini, Service Discovery and Usage*
http://en.wikipedia.org/wiki/Jini

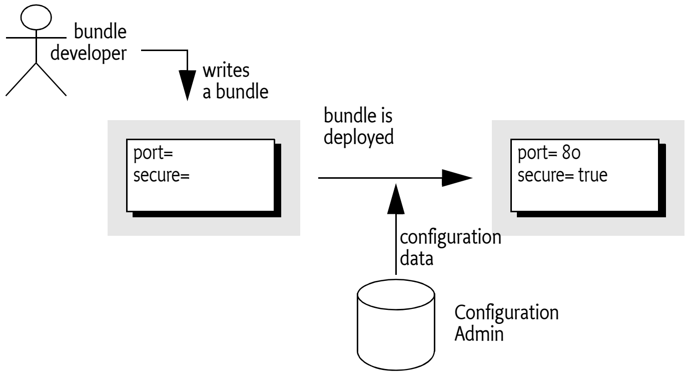
104 Configuration Admin Service Specification

Version 1.4

104.1 Introduction

The Configuration Admin service is an important aspect of the deployment of an OSGi Service Platform. It allows an Operator to configure deployed bundles. Configuring is the process of defining the configuration data for bundles and assuring that those bundles receive that data when they are active in the OSGi Service Platform.

Figure 104.1 Configuration Admin Service Overview



104.1.1 Essentials

The following requirements and patterns are associated with the Configuration Admin service specification:

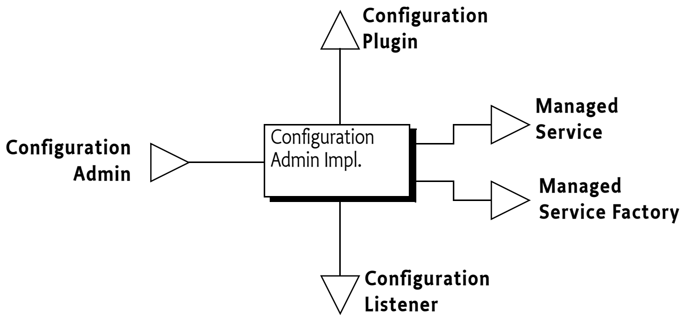
- *Local Configuration* – The Configuration Admin service must support bundles that have their own user interface to change their configurations.
- *Reflection* – The Configuration Admin service must be able to deduce the names and types of the needed configuration data.
- *Legacy* – The Configuration Admin service must support configuration data of existing entities (such as devices).
- *Object Oriented* – The Configuration Admin service must support the creation and deletion of instances of configuration information so that a bundle can create the appropriate number of services under the control of the Configuration Admin service.
- *Embedded Devices* – The Configuration Admin service must be deployable on a wide range of platforms. This requirement means that the interface should not assume file storage on the platform. The choice to use file storage should be left to the implementation of the Configuration Admin service.
- *Remote versus Local Management* – The Configuration Admin service must allow for a remotely managed OSGi Service Platform, and must not assume that configuration information is stored locally. Nor should it assume that the Configuration Admin service is always done remotely. Both implementation approaches should be viable.

- *Availability* – The OSGi environment is a dynamic environment that must run continuously (24/7/365). Configuration updates must happen dynamically and should not require restarting of the system or bundles.
- *Immediate Response* – Changes in configuration should be reflected immediately.
- *Execution Environment* – The Configuration Admin service will not require more than an environment that fulfills the minimal execution requirements.
- *Communications* – The Configuration Admin service should not assume "always-on" connectivity, so the API is also applicable for mobile applications in cars, phones, or boats.
- *Extendability* – The Configuration Admin service should expose the process of configuration to other bundles. This exposure should at a minimum encompass initiating an update, removing certain configuration properties, adding properties, and modifying the value of properties potentially based on existing property or service values.
- *Complexity Trade-offs* – Bundles in need of configuration data should have a simple way of obtaining it. Most bundles have this need and the code to accept this data. Additionally, updates should be simple from the perspective of the receiver.
  Trade-offs in simplicity should be made at the expense of the bundle implementing the Configuration Admin service and in favor of bundles that need configuration information. The reason for this choice is that normal bundles will outnumber Configuration Admin bundles.
- *Regions* – It should be possible to create groups of bundles and a manager in a single system that share configuration data that is not accessible outside the region.
- *Shared Information* – It should be possible to share configuration data between bundles.

104.1.2 Entities

- *Configuration information* – The information needed by a bundle before it can provide its intended functionality.
- *Configuration dictionary* – The configuration information when it is passed to the target service. It consists of a `Dictionary` object with a number of properties and identifiers.
- *Configuring Bundle* – A bundle that modifies the configuration information through the Configuration Admin service. This bundle is either a management bundle or the bundle for which the configuration information is intended.
- *Configuration Target* – The target service that will receive the configuration information. For services, there are two types of targets: `ManagedServiceFactory` or `ManagedService` objects.
- *Configuration Admin Service* – This service is responsible for supplying configuration target bundles with their configuration information. It maintains a database with configuration information, keyed on the `service.pid` of configuration target services. These services receive their configuration dictionary/dictionaries when they are registered with the Framework. Configurations can be modified or extended using Configuration Plugin services before they reach the target bundle.
- *Managed Service* – A Managed Service represents a client of the Configuration Admin service, and is thus a configuration target. Bundles should register a Managed Service to receive the configuration data from the Configuration Admin service. A Managed Service adds one or more unique `service.pid` service properties as a primary key for the configuration information.
- *Managed Service Factory* – A Managed Service Factory can receive a number of configuration dictionaries from the Configuration Admin service, and is thus also a configuration target service. It should register with one or more `service.pid` strings and receives zero or more configuration dictionaries. Each dictionary has its own PID that is distinct from the factory PID.
- *Configuration Object* – Implements the `Configuration` interface and contains the configuration dictionary for a Managed Service or one of the configuration dictionaries for a Managed Service Factory. These objects are manipulated by configuring bundles.
- *Configuration Plugin* Services – Configuration Plugin services are called before the configuration dictionary is given to the configuration targets. The plug-in can modify the configuration dictionary, which is passed to the Configuration Target.

Figure 104.2 Overall Service Diagram



104.1.3 Synopsis

This specification is based on the concept of a Configuration Admin service that manages the configuration of an OSGi Service Platform. It maintains a database of Configuration objects, locally or remotely. This service monitors the service registry and provides configuration information to services that are registered with a service.pid property, the Persistent IDentity (PID), and implement one of the following interfaces:

- *Managed Service* – A service registered with this interface receives its *configuration dictionary* from the database or receives null when no such configuration exists.
- *Managed Service Factory* – Services registered with this interface can receive several configuration dictionaries when registered. The database contains zero or more configuration dictionaries for this service. Each configuration dictionary is given sequentially to the service.

The database can be manipulated either by the Management Agent or bundles that configure themselves. Other parties can provide Configuration Plugin services. Such services participate in the configuration process. They can inspect the configuration dictionary and modify it before it reaches the target service.

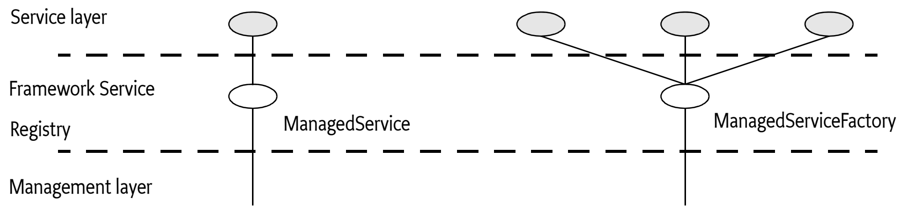
104.2 Configuration Targets

One of the more complicated aspects of this specification is the subtle distinction between the ManagedService and ManagedServiceFactory classes. Both receive configuration information from the Configuration Admin service and are treated similarly in most respects. Therefore, this specification refers to *configuration targets* or simply *targets* when the distinction is irrelevant.

The difference between these types is related to the cardinality of the configuration dictionary. A Managed Service is used when an existing entity needs a configuration dictionary. Thus, a one-to-one relationship always exists between the configuration dictionary and the configurable entity in the Managed Service. There can be multiple Managed Service targets registered with the same PID but a Managed Service can only configure a single entity in each given Managed Service.

A Managed Service Factory is used when part of the configuration is to define *how many instances are required* for a given Managed Service Factory. A management bundle can create, modify, and delete any number of instances for a Managed Service Factory through the Configuration Admin service. Each instance is configured by a single Configuration object. Therefore, a Managed Service Factory can have multiple associated Configuration objects.

Figure 104.3 Differentiation of ManagedService and ManagedServiceFactory Classes



A Configuration target updates the target when the underlying Configuration object is created, updated, or deleted. However, it is not called back when the Configuration Admin service is shutdown or the service is ungotten.

To summarize:

- A *Managed Service* must receive a single configuration dictionary when it is registered or when its configuration is modified.
- A *Managed Service Factory* must receive from zero to *n* configuration dictionaries when it registers, depending on the current configuration. The Managed Service Factory is informed of configuration dictionary changes: modifications, creations, and deletions.

104.3 The Persistent Identity

A crucial concept in the Configuration Admin service specification is the Persistent IDentity (PID) as defined in the Framework's service layer. Its purpose is to act as a primary key for objects that need a configuration dictionary. The name of the service property for PID is defined in the Framework in `org.osgi.framework.Constants.SERVICE_PID`.

The Configuration Admin service requires the use of one or more PIDs with Managed Service and Managed Service Factory registrations because it associates its configuration data with PIDs.

A service can register with multiple PIDs and PIDs can be shared between multiple targets (both Managed Service and Managed Service Factory targets) to receive the same information. If PIDs are to be shared between Bundles then the location of the Configuration must be a multi-location, see *Location Binding* on page 93.

The Configuration Admin must track the configuration targets on their actual PID. That is, if the `service.pid` service property is modified then the Configuration Admin must treat it as if the service was unregistered and then re-registered with the new PID.

104.3.1 PID Syntax

PIDs are intended for use by other bundles, not by people, but sometimes the user is confronted with a PID. For example, when installing an alarm system, the user needs to identify the different components to a wiring application. This type of application exposes the PID to end users.

PIDs should follow the symbolic-name syntax, which uses a very restricted character set. The following sections define some schemes for common cases. These schemes are not required, but bundle developers are urged to use them to achieve consistency.

104.3.1.1 Local Bundle PIDs

As a convention, descriptions starting with the bundle identity and a full stop ('.' `\u002D`) are reserved for a bundle. As an example, a PID of "`65.536`" would belong to the bundle with a bundle identity of 65.

104.3.1.2 Software PIDs

Configuration target services that are singletons can use a Java package name they own as the PID (the reverse domain name scheme) as long as they do not use characters outside the basic ASCII set. As an example, the PID named com.acme.watchdog would represent a Watchdog service from the ACME company.

104.3.1.3 Devices

Devices are usually organized on buses or networks. The identity of a device, such as a unique serial number or an address, is a good component of a PID. The format of the serial number should be the same as that printed on the housing or box, to aid in recognition.

Table 104.1 Schemes for Device-Oriented PID Names

Bus	Example	Format	Description
USB	USB.0123-0002-9909873	idVendor (hex 4) idProduct (hex 4) iSerialNumber (decimal)	Universal Serial Bus. Use the standard device descriptor.
IP	IP.172.16.28.21	IP nr (dotted decimal)	Internet Protocol
802	802-00:60:97:00:9A:56	MAC address with: separators	IEEE 802 MAC address (Token Ring, Ethernet,...)
ONE	ONE.06-00000021E461	Family (hex 2) and serial number including CRC (hex 6)	1-wire bus of Dallas Semiconductor
COM	COM.krups-brewer-12323	serial number or type name of device	Serial ports

104.4 The Configuration Object

A Configuration object contains the configuration dictionary, which is a set of properties that configure an aspect of a bundle. A bundle can receive Configuration objects by registering a configuration target service with a PID service property. See *The Persistent Identity* on page 92 for more information about PIDs.

During registration, the Configuration Admin service must detect these configuration target services and hand over their configuration dictionary via a callback. If this configuration dictionary is subsequently modified, the modified dictionary is handed over to the configuration target with the same callback.

The Configuration object is primarily a set of properties that can be updated by a Management Agent, user interfaces on the OSGi Service Platform, or other applications. Configuration changes are first made persistent, and then passed to the target service via a call to the updated method in the ManagedServiceFactory or ManagedService class.

A Configuration object must be uniquely bound to a Managed Service or Managed Service Factory. This implies that a bundle must not register a Managed Service Factory with a PID that is the same as the PID given to a Managed Service.

104.4.1 Location Binding

When a Configuration object is created with either getConfiguration(String) or createFactoryConfiguration(String), it becomes *bound* to the location of the calling bundle. This location is obtained with the getBundleLocation() method.

Location binding is a security feature that assures that only management bundles can modify configuration data, and other bundles can only modify their own configuration data. A Security Exception is thrown if a bundle does not have ConfigurationPermission[location, CONFIGURE].

The two argument versions of getConfiguration(String,String) and createFactoryConfiguration(String,String) take a location String as their second argument. These methods require the correct permission, and they create Configuration objects bound to the specified location.

Locations can be specified for a specific Bundle or use *multi-locations*. For a specific location the Configuration location must exactly match the location of the target's Bundle. A multi-location is any location that has the following syntax:

```
multi-location ::= '?' symbolic-name?
```

For example

```
?com.acme
```

The path after the question mark is the *multi-location name*, the multi-location name can be empty if only a question mark is specified. Configurations with a multi-location are dispatched to any target that has *visibility* to the Configuration. The visibility for a given Configuration c depends on the following rules:

- *Single-Location* – If c.location is not a multi-location then a Bundle only has visibility if the Bundle's location exactly matches c.location. In this case there is never a security check.
- *Multi-Location* – If c.location is a multi-location (that is, starts with a question mark):
  - *Security Off* – The Bundle always has visibility
  - *Security On* – The target's Bundle must have ConfigurationPermission[c.location,TARGET] as defined by the Bundle's hasPermission method. The resource name of the permission must include the question mark.

The permission matches on the whole name, including any leading ?. The TARGET action is only applicable in the multi-location scenario since the security is not checked for a single-location. There is therefore no point in granting a Bundle a permission with TARGET action for anything but a multi-location (starting with a ?).

It is therefore possible to register services with the same PID from different bundles. If a multi-location is used then each bundle will be evaluated for a corresponding configuration update. If the bundle has visibility then it is updated, otherwise it is not.

If multiple targets must be updated then the order of updating is the ranking order of their services.

If a target loses visibility because the Configuration's location changes then it must immediately be deleted from the perspective of that target. That is, the target must see a deletion (Managed Service Factory) or an update with null (Managed Service). If a configuration target gains visibility then the target must see a new update with the proper configuration dictionary. However, the associated events must not be sent as the underlying Configuration is not actually deleted nor modified.

Changes in the permissions must not initiate a recalculation of the visibility. If the permissions are changed this will not become visible until one of the other events happen that cause a recalculation of the visibility.

If the location is changed then the Configuration Admin must send a CM_LOCATION_CHANGED event to signal that the location has changed. It is up to the Configuration Listeners to update their state appropriately.

104.4.2 Dynamic Binding

Dynamic binding is available for backward compatibility with earlier versions. It is recommended that management agents explicitly set the location to a ? (a multi-location) to allow multiple bundles to share PIDs and not use the dynamic binding facility. If a management agent uses ?, it must at least have ConfigurationPermission[?,CONFIGURE] when security is on, it is also possible to use ConfigurationPermission[?*,CONFIGURE] to not limit the management agent. See *Regions* on page 106 for some examples of using the locations in isolation scenarios.

A null location parameter can be used to create Configuration objects that are not yet bound. In this case, the Configuration becomes bound to a specific location the first time that it is compared to a Bundle's location. If a bundle becomes dynamically bound to a Configuration then a CM_LOCATION_CHANGED event must be dispatched.

When this *dynamically bound* Bundle is subsequently uninstalled, configurations that are bound to this bundle must be released. That means that for such Configuration object's the bundle location must be set to null again so it can be bound again to another bundle.

104.4.3 Configuration Properties

A configuration dictionary contains a set of properties in a Dictionary object. The value of the property must be the same type as the set of types specified in the OSGi Core Specification in *Figure 3.1 Primary property types.*

The name or key of a property must always be a String object, and is not case-sensitive during look up, but must preserve the original case. The format of a property name should be:

```
property-name ::= public | private
public        ::= symbolic-name // See 1.3.2
private       ::= '.' symbolic-name
```

Properties can be used in other subsystems that have restrictions on the character set that can be used. The symbolic-name production uses a very minimal character set.

Bundles must not use nested vectors or arrays, nor must they use mixed types. Using mixed types or nesting makes it impossible to use the meta typing specification. See *Metatype Service Specification* on page 129.

104.4.4 Property Propagation

A configuration target should copy the public configuration properties (properties whose name does not start with a '.' or \u002E) of the Dictionary object argument in updated(Dictionary) into the service properties on any resulting service registration.

This propagation allows the development of applications that leverage the Framework service registry more extensively, so compliance with this mechanism is advised.

A configuration target may ignore any configuration properties it does not recognize, or it may change the values of the configuration properties before these properties are registered as service properties. Configuration properties in the Framework service registry are not strictly related to the configuration information.

Bundles that follow this recommendation to propagate public configuration properties can participate in horizontal applications. For example, an application that maintains physical location information in the Framework service registry could find out where a particular device is located in the house or car. This service could use a property dedicated to the physical location and provide functions that leverage this property, such as a graphic user interface that displays these locations.

Bundles performing service registrations on behalf of other bundles (e.g. OSGi Declarative Services) should propagate all public configuration properties and not propagate private configuration properties.

104.4.5 Automatic Properties

The Configuration Admin service must automatically add a number of properties to the configuration dictionary. If these properties are also set by a configuring bundle or a plug-in, they must always be overridden before they are given to the target service, see *Configuration Plugin* on page 108. Therefore, the receiving bundle or plug-in can assume that the following properties are defined by the Configuration Admin service and not by the configuring bundle:

- service.pid – Set to the PID of the associated Configuration object.

- service.factoryPid – Only set for a Managed Service Factory. It is then set to the PID of the associated Managed Service Factory.
- service.bundleLocation – Set to the location of the Configuration object. This property can only be used for searching, it may not appear in the configuration dictionary returned from the getProperties method due to security reasons, nor may it be used when the target is updated.

Constants for some of these properties can be found in org.osgi.framework.Constants and the ConfigurationAdmin interface. These service properties are all of type String.

104.4.6 Equality

Two different Configuration objects can actually represent the same underlying configuration. This means that a Configuration object must implement the equals and hashCode methods in such a way that two Configuration objects are equal when their PID is equal.

104.5 Managed Service

A Managed Service is used by a bundle that needs one or more configuration dictionaries. It therefore registers the Managed Service with one or more PIDs and is thus associated with one Configuration object in the Configuration Admin service for each registered PID. A bundle can register any number of ManagedService objects, but each must be identified with its own PID or PIDs.

A bundle should use a Managed Service when it needs configuration information for the following:

- *A Singleton* – A single entity in the bundle that needs to be configured.
- *Externally Detected Devices* – Each device that is detected causes a registration of an associated ManagedService object. The PID of this object is related to the identity of the device, such as the address or serial number.

104.5.1 Singletons

When an object must be instantiated only once, it is called a *singleton.* A singleton requires a single configuration dictionary. Bundles may implement several different types of singletons if necessary.

For example, a Watchdog service could watch the registry for the status and presence of services in the Framework service registry. Only one instance of a Watchdog service is needed, so only a single configuration dictionary is required that contains the polling time and the list of services to watch.

104.5.2 Networks

When a device in the external world needs to be represented in the OSGi Environment, it must be detected in some manner. The Configuration Admin service cannot know the identity and the number of instances of the device without assistance. When a device is detected, it still needs configuration information in order to play a useful role.

For example, a 1-Wire network can automatically detect devices that are attached and removed. When it detects a temperature sensor, it could register a Sensor service with the Framework service registry. This Sensor service needs configuration information specifically for that sensor, such as which lamps should be turned on, at what temperature the sensor is triggered, what timer should be started, in what zone it resides, and so on. One bundle could potentially have hundreds of these sensors and actuators, and each needs its own configuration information.

Each of these Sensor services should be registered as a Managed Service with a PID related to the physical sensor (such as the address) to receive configuration information.

Other examples are services discovered on networks with protocols like Jini, UPnP, and Salutation. They can usually be represented in the Framework service registry. A network printer, for example, could be detected via UPnP. Once in the service registry, these services usually require local configuration information. A Printer service needs to be configured for its local role: location, access list, and so on.

This information needs to be available in the Framework service registry whenever that particular Printer service is registered. Therefore, the Configuration Admin service must remember the configuration information for this Printer service.

This type of service should register with the Framework as a Managed Service in order to receive appropriate configuration information.

104.5.3 Configuring Managed Services

A bundle that needs configuration information should register one or more ManagedService objects with a PID service property. If it has a default set of properties for its configuration, it may include them as service properties of the Managed Service. These properties may be used as a configuration template when a Configuration object is created for the first time. A Managed Service optionally implements the MetaTypeProvider interface to provide information about the property types. See *Meta Typing* on page 110.

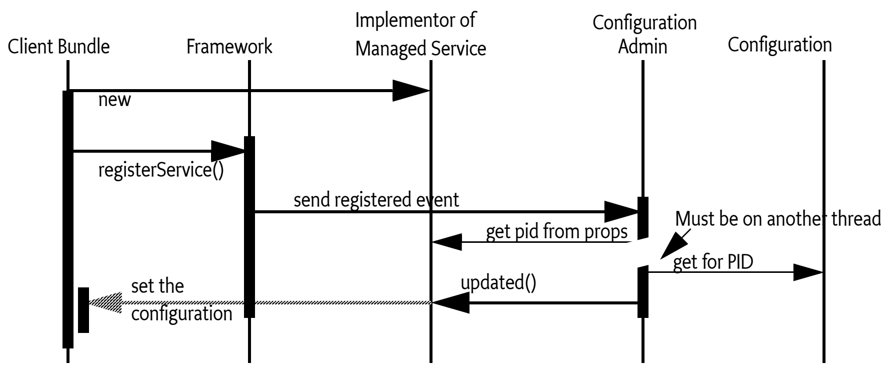
When this registration is detected by the Configuration Admin service, the following steps must occur:

- The configuration stored for the registered PID must be retrieved. If there is a Configuration object for this PID and the configuration is visible for the associated bundle then it is sent to the Managed Service with updated(Dictionary).
- If a Managed Service is registered and no configuration information is available or the configuration is not visible then the Configuration Admin service must call updated(Dictionary) with a null parameter.
- If the Configuration Admin service starts *after* a Managed Service is registered, it must call updated(Dictionary) on this service as soon as possible according to the prior rules. For this reason, a Managed Service must always get a callback when it registers *and* the Configuration Admin service is started.

Multiple Managed Services can register with the same PID, they are all updated as long as they have visibility to the configuration as defined by the location, see *Location Binding* on page 93.

The updated(Dictionary) callback from the Configuration Admin service to the Managed Service must take place asynchronously. This requirement allows the Managed Service to finish its initialization in a synchronized method without interference from the Configuration Admin service callback. Care should be taken not to cause deadlocks by calling the Framework within a synchronized method.

Figure 104.4 Managed Service Configuration Action Diagram



The updated method may throw a ConfigurationException. This object must describe the problem and what property caused the exception.

104.5.4 Race Conditions

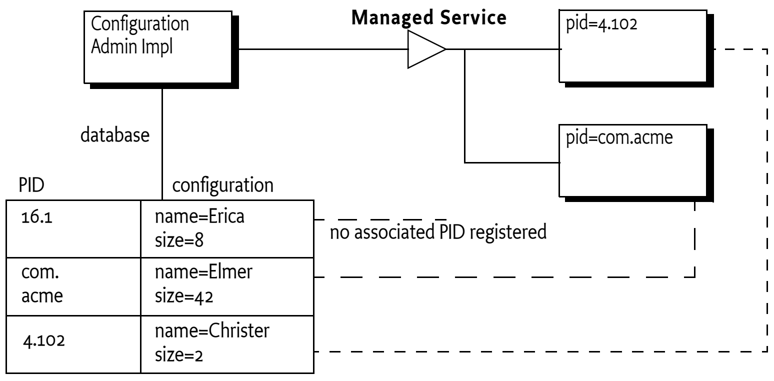
When a Managed Service is registered, the default properties may be visible in the service registry for a short period before they are replaced by the properties of the actual configuration dictionary. Care should be taken that this visibility does not cause race conditions for other bundles.

In cases where race conditions could be harmful, the Managed Service must be split into two pieces: an object performing the actual service and a Managed Service. First, the Managed Service is registered, the configuration is received, and the actual service object is registered. In such cases, the use of a Managed Service Factory that performs this function should be considered.

104.5.5 Examples of Managed Service

Figure 104.5 shows a Managed Service configuration example. Two services are registered under the ManagedService interface, each with a different PID.

Figure 104.5 PIDs and External Associations



The Configuration Admin service has a database containing a configuration record for each PID. When the Managed Service with service.pid = com.acme is registered, the Configuration Admin service will retrieve the properties name=Elmer and size=42 from its database. The properties are stored in a Dictionary object and then given to the Managed Service with the updated(Dictionary) method.

104.5.5.1 Configuring A Console Bundle

In this example, a bundle can run a single debugging console over a Telnet connection. It is a singleton, so it uses a ManagedService object to get its configuration information: the port and the network name on which it should register.

```
class SampleManagedService implements ManagedService {
  Dictionary             properties;
  ServiceRegistration    registration;
  Console                console;

  public void start(
    BundleContext context ) throws Exception {
    properties = new Hashtable();
    properties.put( Constants.SERVICE_PID,
      "com.acme.console" );

    registration = context.registerService(
      ManagedService.class.getName(),
      this,
      properties
    );
  }

  public synchronized void updated( Dictionary np ) {
    if ( np != null ) {
      properties = np;
      properties.put(
        Constants.SERVICE_PID, "com.acme.console" );
    }

    if (console == null)
      console = new Console();

    int port = ((Integer)properties.get("port"))
      .intValue();

    String network = (String) properties.get("network");
    console.setPort(port, network);
    registration.setProperties(properties);
  }
  ... further methods
}
```

104.5.6 Deletion

When a Configuration object for a Managed Service is deleted, the Configuration Admin service must call updated(Dictionary) with a null argument on a thread that is different from that on which the Configuration.delete was executed. This deletion must send out a Configuration Event CM_DELETED asynchronously to any registered Configuration Listener services after the updated method is called with a null.

104.6 Managed Service Factory

A Managed Service Factory is used when configuration information is needed for a service that can be instantiated multiple times. When a Managed Service Factory is registered with the Framework, the Configuration Admin service consults its database and calls updated(String,Dictionary) for each associated and visible Configuration object that matches the PIDs on the registration. It passes the identifier of the Configuration instance, which can be used as a PID, as well as a Dictionary object with the configuration properties.

A Managed Service Factory is useful when the bundle can provide functionality a number of times, each time with different configuration dictionaries. In this situation, the Managed Service Factory acts like a *class* and the Configuration Admin service can use this Managed Service Factory to *instantiate instances* for that *class*.

In the next section, the word *factory* refers to this concept of creating *instances* of a function defined by a bundle that registers a Managed Service Factory.

104.6.1 When to Use a Managed Service Factory

A Managed Service Factory should be used when a bundle does not have an internal or external entity associated with the configuration information but can potentially be instantiated multiple times.

104.6.1.1 Example Email Fetcher

An email fetcher program displays the number of emails that a user has – a function likely to be required for different users. This function could be viewed as a *class* that needs to be *instantiated* for each user. Each instance requires different parameters, including password, host, protocol, user id, and so on.

An implementation of the Email Fetcher service should register a ManagedServiceFactory object. In this way, the Configuration Admin service can define the configuration information for each user separately. The Email Fetcher service will only receive a configuration dictionary for each required instance (user).

104.6.1.2 Example Temperature Conversion Service

Assume a bundle has the code to implement a conversion service that receives a temperature and, depending on settings, can turn an actuator on and off. This service would need to be instantiated many times depending on where it is needed. Each instance would require its own configuration information for the following:

- Upper value
- Lower value
- Switch Identification
- ...

Such a conversion service should register a service object under a ManagedServiceFactory interface. A configuration program can then use this Managed Service Factory to create instances as needed. For example, this program could use a Graphic User Interface (GUI) to create such a component and configure it.

104.6.1.3 Serial Ports

Serial ports cannot always be used by the OSGi Device Access specification implementations. Some environments have no means to identify available serial ports, and a device on a serial port cannot always provide information about its type.

Therefore, each serial port requires a description of the device that is connected. The bundle managing the serial ports would need to instantiate a number of serial ports under the control of the Configuration Admin service, with the appropriate DEVICE_CATEGORY property to allow it to participate in the Device Access implementation.

If the bundle cannot detect the available serial ports automatically, it should register a Managed Service Factory. The Configuration Admin service can then, with the help of a configuration program, define configuration information for each available serial port.

104.6.2 Registration

Similar to the Managed Service configuration dictionary, the configuration dictionary for a Managed Service Factory is identified by a PID. The Managed Service Factory, however, also has a *factory* PID, which is the PID of the associated Managed Service Factory. It is used to group all Managed Service Factory configuration dictionaries together.

When a Configuration object for a Managed Service Factory is created (ConfigurationAdmin.createFactoryConfiguration(String,String) or createFactoryConfiguration(String)), a new unique PID is created for this object by the Configuration Admin service. The scheme used for this PID is defined by the Configuration Admin service and is unrelated to the factory PID, which is chosen by the registering bundle.

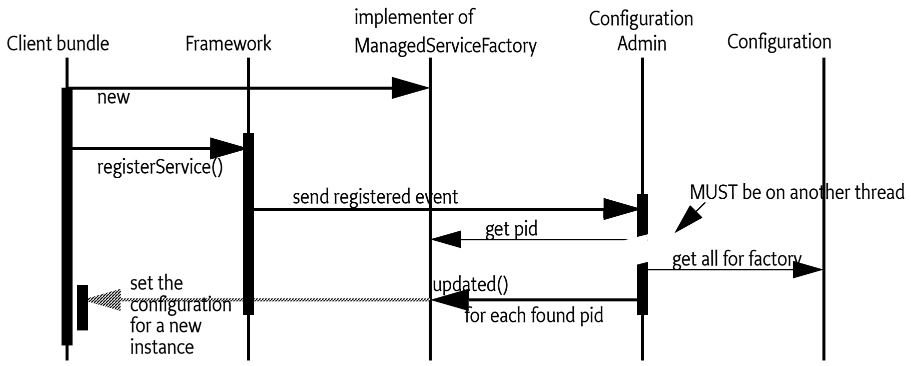
When the Configuration Admin service detects the registration of a Managed Service Factory, it must find all visible configuration dictionaries for this factory and must then sequentially call ManagedServiceFactory.updated(String,Dictionary) for each configuration dictionary. The first argument is the PID of the Configuration object (the one created by the Configuration Admin service) and the second argument contains the configuration properties.

The Managed Service Factory should then create any artifacts associated with that factory. Using the PID given in the Configuration object, the bundle may register new services (other than a Managed Service) with the Framework, but this is not required. This may be necessary when the PID is useful in contexts other than the Configuration Admin service.

The receiver must *not* register a Managed Service with this PID because this would force two Configuration objects to have the same PID. If a bundle attempts to do this, the Configuration Admin service should log an error and must ignore the registration of the Managed Service.

The Configuration Admin service must guarantee that no race conditions exist between initialization, updates, and deletions.

Figure 104.6 Managed Service Factory Action Diagram



A Managed Service Factory has only one update method: updated(String,Dictionary). This method can be called any number of times as Configuration objects are created or updated.

The Managed Service Factory must detect whether a PID is being used for the first time, in which case it should create a new *instance*, or a subsequent time, in which case it should update an existing instance.

The Configuration Admin service must call updated(String,Dictionary) on a thread that is different from the one that executed the registration. This requirement allows an implementation of a Managed Service Factory to use a synchronized method to assure that the callbacks do not interfere with the Managed Service Factory registration.

The updated(String,Dictionary) method may throw a ConfigurationException object. This object describes the problem and what property caused the problem. These exceptions should be logged by a Configuration Admin service.

Multiple Managed Service Factory services can be registered with the same PID. Each of those services that have visibility to the corresponding configuration will be updated in service ranking order.

104.6.3 Deletion

If a configuring bundle deletes an instance of a Managed Service Factory, the deleted(String) method is called. The argument is the PID for this instance. The implementation of the Managed Service Factory must remove all information and stop any behavior associated with that PID. If a service was registered for this PID, it should be unregistered.

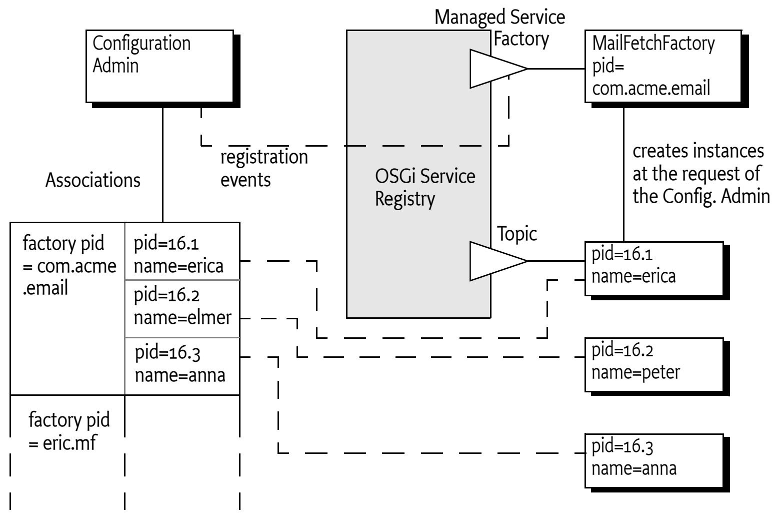
Deletion will asynchronously send out a Configuration Event CM_DELETED to all registered Configuration Listener services.

104.6.4 Managed Service Factory Example

Figure 104.7 highlights the differences between a Managed Service and a Managed Service Factory. It shows how a Managed Service Factory implementation receives configuration information that was created before it was registered.

- A bundle implements an EMail Fetcher service. It registers a ManagedServiceFactory object with PID=com.acme.email.
- The Configuration Admin service notices the registration and consults its database. It finds three Configuration objects for which the factory PID is equal to com.acme.email. It must call updated(String,Dictionary) for each of these Configuration objects on the newly registered ManagedServiceFactory object.
- For each configuration dictionary received, the factory should create a new instance of a EMailFetcher object, one for erica (PID=16.1), one for anna (PID=16.3), and one for elmer (PID=16.2).
- The EMailFetcher objects are registered under the Topic interface so their results can be viewed by an online display.
  If the EMailFetcher object is registered, it may safely use the PID of the Configuration object because the Configuration Admin service must guarantee its suitability for this purpose.

Figure 104.7 Managed Service Factory Example



104.6.5 Multiple Consoles Example

This example illustrates how multiple consoles, each of which has its own port and interface can run simultaneously. This approach is very similar to the example for the Managed Service, but highlights the difference by allowing multiple consoles to be created.

```
class ExampleFactory implements ManagedServiceFactory {
   Hashtable consoles = new Hashtable();
   BundleContext context;
   public void start( BundleContext context )
      throws Exception {
      this.context = context;
      Hashtable local = new Hashtable();
      local.put(Constants.SERVICE_PID,"com.acme.console");
      context.registerService(
         ManagedServiceFactory.class.getName(),
         this,
         local );
   }

   public void updated( String pid, Dictionary config ){
      Console console = (Console) consoles.get(pid);
      if (console == null) {
         console = new Console(context);
         consoles.put(pid, console);
      }

      int port = getInt(config, "port", 2011);
      String network = getString(
         config,
         "network",
         null /*all*/
      );
      console.setPort(port, network);
   }
```

```
        public void deleted(String pid) {
            Console console = (Console) consoles.get(pid);
            if (console != null) {
                consoles.remove(pid);
                console.close();
            }
        }
    }
```

104.7 Configuration Admin Service

The ConfigurationAdmin interface provides methods to maintain configuration data in an OSGi environment. This configuration information is defined by a number of Configuration objects associated with specific configuration targets. Configuration objects can be created, listed, modified, and deleted through this interface. Either a remote management system or the bundles configuring their own configuration information may perform these operations.

The ConfigurationAdmin interface has methods for creating and accessing Configuration objects for a Managed Service, as well as methods for managing new Configuration objects for a Managed Service Factory.

104.7.1 Creating a Managed Service Configuration Object

A bundle can create a new Managed Service Configuration object with ConfigurationAdmin.getConfiguration. No create method is offered because doing so could introduce race conditions between different bundles trying to create a Configuration object for the same Managed Service. The getConfiguration method must atomically create and persistently store an object if it does not yet exist.

Two variants of this method are:

- getConfiguration(String) – This method is used by a bundle with a given location to configure its *own* ManagedService objects. The argument specifies the PID of the targeted service.
- getConfiguration(String,String) – This method is used by a management bundle to configure *another* bundle. Therefore, this management bundle needs the right permission. The first argument is the PID and the second argument is the location identifier of the targeted ManagedService object.

All Configuration objects have a method, getFactoryPid(), which in this case must return null because the Configuration object is associated with a Managed Service.

Creating a new Configuration object must *not* initiate a callback to the Managed Service updated method until the properties are set in the Configuration with the update method.

104.7.2 Creating a Managed Service Factory Configuration Object

The ConfigurationAdmin class provides two methods to create a new instance of a Managed Service Factory:

- createFactoryConfiguration(String) – This method is used by a bundle with a given location to configure its own ManagedServiceFactory objects. The argument specifies the PID of the targeted ManagedServiceFactory object. This *factory PID* can be obtained from the returned Configuration object with the getFactoryPid() method.
- createFactoryConfiguration(String,String)– This method is used by a management bundle to configure another bundle's ManagedServiceFactory object. The first argument is the PID and the second is the location identifier of the targeted ManagedServiceFactory object. The *factory PID* can be obtained from the returned Configuration object with getFactoryPid method.

Creating a new factory configuration must *not* initiate a callback to the Managed Service Factory updated method until the properties are set in the Configuration object with the update method.

104.7.3 Accessing Existing Configurations

The existing set of Configuration objects can be listed with listConfigurations(String). The argument is a String object with a filter expression. This filter expression has the same syntax as the Framework Filter class. For example:

```
(&(size=42)(service.factoryPid=*osgi*))
```

The Configuration Admin service must only return Configurations that are visible to the calling bundle, see *Location Binding* on page 93.

A single Configuration object is identified with a PID and can be obtained with listConfigurations(String) if it is visible. null is returned in both cases when there are no visible Configuration objects.

104.7.3.1 Updating a Configuration

The process of updating a Configuration object is the same for Managed Services and Managed Service Factories. First, listConfigurations(String) or getConfiguration(String) should be used to get a Configuration object. The properties can be obtained with Configuration.getProperties. When no update has occurred since this object was created, getProperties returns null.

New properties can be set by calling Configuration.update. The Configuration Admin service must first store the configuration information and then call all configuration targets that have visibility with the updated method: either the ManagedService.updated(Dictionary) or ManagedServiceFactory.updated(String,Dictionary) method. If a target service is not registered, the fresh configuration information must be given to the target when the configuration target service registers and it has visibility.

The update method calls in Configuration objects are not executed synchronously with the related target services updated method. The updated method must be called asynchronously. The Configuration Admin service, however, must have updated the persistent storage before the update method returns.

The update method must also asynchronously send out a Configuration Event CM_UPDATED to all registered Configuration Listeners.

104.7.4 Using Multi-Locations

Sharing configuration between different bundles can be done using multi-locations, see *Location Binding* on page 93. A multi-location for a Configuration enables this Configuration to be delivered to any bundle that has visibility to that configuration. It is also possible that Bundles are interested in multiple PIDs for one target service, for this reason they can register multiple PIDs for one service.

For example, a number of bundles require access to the URL of a remote host, associated with the PID com.acme.host. A manager, aware that this PID is used by different bundles, would need to specify a location for the Configuration that allows delivery to any bundle. A multi-location, any location starting with a question mark achieves this. The part after the question mark has only use if the system runs with security, it allows the implementation of regions, see *Regions* on page 106. In this example a single question mark is used because any Bundle can receive this Configuration. The manager's code could look like:

```
Configuration c = admin.getConfiguration( "com.acme.host", "?" );
Hashtable ht = new Hashtable();
ht.put( "host", hostURL);
c.update(ht);
```

A Bundle interested in the host configuration would register a Managed Service with the following properties:

```
service.pid     = [ "com.acme.host", "com.acme.system" ]
```

The Bundle would be called back for both the com.acme.host and com.acme.system PID and must therefore discriminate between these two cases. This Managed Service therefore would have a callback like:

```
volatile URL url;
public void updated( Dictionary d ) {
  if ( d.get("service.pid").equals("com.acme.host"))
     this.url = new URL( d.get("host"));
  if ( d.get("service.pid").equals("com.acme.system"))
      ....
}
```

104.7.5 Regions

In certain cases it is necessary to isolate bundles from each other. This will require that the configuration can be separated in *regions*. Each region can then be configured by a separate manager that is only allowed to manage bundles in its own region. Bundles can then only see configurations from their own region. Such a region based system can only be achieved with Java security as this is the only way to place bundles in a sandbox. This section describes how the Configuration's location binding can be used to implement regions if Java security is active.

Regions are groups of bundles that share location information among each other but are not willing to share this information with others. Using the multi-locations, see *Location Binding* on page 93, and security it is possible to limit access to a Configuration by using a location name. A Bundle can only receive a Configuration when it has ConfigurationPermission[location name,TARGET]. It is therefore possible to create region by choosing a region name for the location. A management agent then requires ConfigurationPermission[?region-name,CONFIGURE] and a Bundle in the region requires ConfigurationPermission[?region-name,TARGET].

To implement regions, the management agent is required to use multi-locations; without the question mark a Configuration is only visible to a Bundle that has the exact location of the Configuration. With a multi-location, the Configuration is delivered to any bundle that has the appropriate permission. Therefore, if regions are used, no manager should have ConfigurationPermission[*, CONFIGURE] because it would be able to configure anybody. This permission would enable the manager to set the location to any region or set the location to null. All managers must be restricted to a permission like ConfigurationPermission[?com.acme.region.*,CONFIGURE]. The resource name for a Configuration Permission uses substring matching as in the OSGi Filter, this facility can be used to simplify the administrative setup and implement more complex sharing schemes.

For example, a management agent works for the region com.acme. It has the following permission:

```
ConfigurationPermission[?com.acme.*,CONFIGURE]
```

The manager requires multi-location updates for com.acme.* (the last period is required in this wildcarding). For the CONFIGURE action the question mark must be specified in the resource name. The bundles in the region have the permission:

```
ConfigurationPermission["?com.acme.alpha",TARGET]
```

The question mark must be specified for the TARGET permission. A management agent that needs to configure Bundles in a region must then do this as follows:

```
Configuration c = admin.getConfiguration( "com.acme.host", "?com.acme.alpha" );
Hashtable ht = new Hashtable();
ht.put( "host", hostURL);
c.update(ht);
```

Another, similar, example with two regions:

- system

- application

There is only one manager that manages all bundles. Its permissions look like:

```
ConfigurationPermission[?system,CONFIGURE]
ConfigurationPermission[?application,CONFIGURE]
```

A Bundle in the application region can have the following permissions:

```
ConfigurationPermission[?application,TARGET]
```

This managed bundle therefore has only visibility to configurations in the application region.

104.7.6 Deletion

A Configuration object that is no longer needed can be deleted with Configuration.delete, which removes the Configuration object from the database. The database must be updated before the target service's updated or deleted method is called. Only services that have received the configuration dictionary before must be called.

If the target service is a Managed Service Factory, the factory is informed of the deleted Configuration object by a call to ManagedServiceFactory.deleted(String) method. It should then remove the associated *instance*. The ManagedServiceFactory.deleted(String) call must be done asynchronously with respect to Configuration.delete().

When a Configuration object of a Managed Service is deleted, ManagedService.updated is called with null for the properties argument. This method may be used for clean-up, to revert to default values, or to unregister a service. This method is called asynchronously from the delete method.

The update method must also asynchronously send out a Configuration Event CM_DELETED to all registered Configuration Listeners.

104.7.7 Updating a Bundle's Own Configuration

The Configuration Admin service specification does not distinguish between updates via a Management Agent and a bundle updating its own configuration information (as defined by its location). Even if a bundle updates its own configuration information, the Configuration Admin service must callback the associated target service's updated method.

As a rule, to update its own configuration, a bundle's user interface should *only* update the configuration information and never its internal structures directly. This rule has the advantage that the events, from the bundle implementation's perspective, appear similar for internal updates, remote management updates, and initialization.

104.8 Configuration Events

Configuration Admin can update interested parties of changes in its repository. The model is based on the white board pattern where a Configuration Listener service is registered with the service registry. The Configuration Listener service will receive ConfigurationEvent objects if important changes take place. The Configuration Admin service must call the ConfigurationListener.configurationEvent(ConfigurationEvent) method with such an event. This method should be called asynchronously, and on another thread, than the call that caused the event. Configuration Events must be delivered in order for each listener as they are generated. The way events must be delivered is the same as described in *Delivering Events* on page 106 of the Core specification.

The ConfigurationEvent object carries a factory PID (getFactoryPid()) and a PID (getPid()). If the factory PID is null, the event is related to a Managed Service Configuration object, else the event is related to a Managed Service Factory Configuration object.

The ConfigurationEvent object can deliver the following events from the getType() method:

- CM_DELETED – The Configuration object is deleted.

- CM_UPDATED – The Configuration object is updated.
- CM_LOCATION_CHANGED – The location of the Configuration object changed.

The Configuration Event also carries the ServiceReference object of the Configuration Admin service that generated the event.

104.8.1 Event Admin Service and Configuration Change Events

Configuration events must be delivered asynchronously by the Configuration Admin implementation, if present. The topic of a configuration event must be:

```
org/osgi/service/cm/ConfigurationEvent/<event type>
```

The <event type> can be any of the following:

CM_DELETED
CM_UPDATED
CM_LOCATION_CHANGED

The properties of a configuration event are:

- cm.factoryPid – (String) The factory PID of the associated Configuration object, if the target is a Managed Service Factory. Otherwise not set.
- cm.pid – (String) The PID of the associated Configuration object.
- service – (ServiceReference) The Service Reference of the Configuration Admin service.
- service.id – (Long) The Configuration Admin service's ID.
- service.objectClass – (String[]) The Configuration Admin service's object class (which must include org.osgi.service.cm.ConfigurationAdmin)
- service.pid – (String) The Configuration Admin service's persistent identity, if set.

104.9 Configuration Plugin

The Configuration Admin service allows third-party applications to participate in the configuration process. Bundles that register a service object under a ConfigurationPlugin interface can process the configuration dictionary just before it reaches the configuration target service.

Plug-ins allow sufficiently privileged bundles to intercept configuration dictionaries just *before* they must be passed to the intended Managed Service or Managed Service Factory but *after* the properties are stored. The changes the plug-in makes are dynamic and must not be stored. The plug-in must only be called when an update takes place while it is registered and there is a valid dictionary. The plugin is not called when a configuration is deleted.

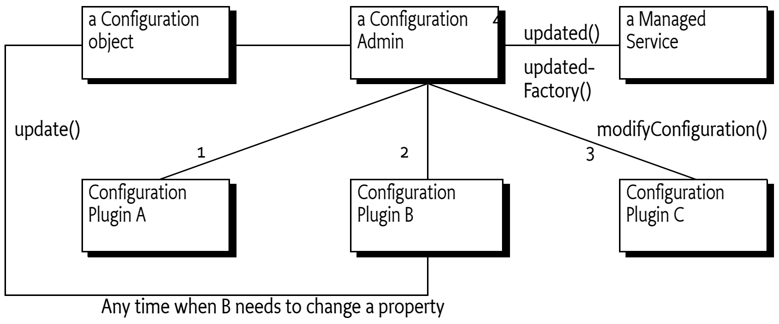
The ConfigurationPlugin interface has only one method: modifyConfiguration(ServiceReference, Dictionary). This method inspects or modifies configuration data.

All plug-ins in the service registry must be traversed and called before the properties are passed to the configuration target service. Each Configuration Plugin object gets a chance to inspect the existing data, look at the target object, which can be a ManagedService object or a ManagedServiceFactory object, and modify the properties of the configuration dictionary. The changes made by a plug-in must be visible to plugins that are called later.

ConfigurationPlugin objects should not modify properties that belong to the configuration properties of the target service unless the implications are understood. This functionality is mainly intended to provide functions that leverage the Framework service registry. The changes made by the plugin should normally not be validated. However, the Configuration Admin must ignore changes to the automatic properties as described in *Automatic Properties* on page 95.

For example, a Configuration Plugin service may add a physical location property to a service. This property can be leveraged by applications that want to know where a service is physically located. This scenario could be carried out without any further support of the service itself, except for the general requirement that the service should propagate the public properties it receives from the Configuration Admin service to the service registry.

Figure 104.8 Order of Configuration Plugin Services



104.9.1 Limiting The Targets

A ConfigurationPlugin object may optionally specify a cm.target registration property. This value is the PID of the configuration target whose configuration updates the ConfigurationPlugin object wants to intercept.

The ConfigurationPlugin object must then only be called with updates for the configuration target service with the specified PID. For a factory target service, the factory PID is used and the plugin will see all instances of the factory. Omitting the cm.target registration property means that it is called for *all* configuration updates.

104.9.2 Example of Property Expansion

Consider a Managed Service that has a configuration property service.to with the value (objectclass=com.acme.Alarm). When the Configuration Admin service sets this property on the target service, a ConfigurationPlugin object may replace the (objectclass=com.acme.Alarm) filter with an array of existing alarm systems' PIDs as follows:

```
ID "service.to=[32434,232,12421,1212]"
```

A new Alarm Service with service.pid=343 is registered, requiring that the list of the target service be updated. The bundle which registered the Configuration Plugin service, therefore, wants to set the service.to registration property on the target service. It does *not* do this by calling ManagedService.updated directly for several reasons:

- In a securely configured system, it should not have the permission to make this call or even obtain the target service.
- It could get into race conditions with the Configuration Admin service if it had the permissions in the previous bullet. Both services would compete for access simultaneously.

Instead, it must get the Configuration object from the Configuration Admin service and call the update method on it.

The Configuration Admin service must schedule a new update cycle on another thread, and sometime in the future must call ConfigurationPlugin.modifyProperties. The ConfigurationPlugin object could then set the service.to property to [32434,232,12421,1212, 343]. After that, the Configuration Admin service must call updated on the target service with the new service.to list.

104.9.3 Configuration Data Modifications

Modifications to the configuration dictionary are still under the control of the Configuration Admin service, which must determine whether to accept the changes, hide critical variables, or deny the changes for other reasons.

The ConfigurationPlugin interface must also allow plugins to detect configuration updates to the service via the callback. This ability allows them to synchronize the configuration updates with transient information.

104.9.4 Forcing a Callback

If a bundle needs to force a Configuration Plugin service to be called again, it must fetch the appropriate Configuration object from the Configuration Admin service and call the update() method (the no parameter version) on this object. This call forces an update with the current configuration dictionary so that all applicable plug-ins get called again.

104.9.5 Calling Order

The order in which the ConfigurationPlugin objects are called must depend on the service.cmRanking configuration property of the ConfigurationPlugin object. Table 104.2 shows the usage of the service.cmRanking property for the order of calling the Configuration Plugin services.

Table 104.2 service.cmRanking *Usage For Ordering*

service.cmRanking value	Description
< 0	The Configuration Plugin service should not modify properties and must be called before any modifications are made.
>= 0 && <= 1000	The Configuration Plugin service modifies the configuration data. The calling order should be based on the value of the service.cmRanking property.
> 1000	The Configuration Plugin service should not modify data and is called after all modifications are made.

104.10 Meta Typing

This section discusses how the Metatype specification is used in the context of a Configuration Admin service.

When a Managed Service or Managed Service Factory is registered, the service object may also implement the MetaTypeProvider interface.

If the Managed Service or Managed Service Factory object implements the MetaTypeProvider interface, a management bundle may assume that the associated ObjectClassDefinition object can be used to configure the service.

The ObjectClassDefinition and AttributeDefinition objects contain sufficient information to automatically build simple user interfaces. They can also be used to augment dedicated interfaces with accurate validations.

When the Metatype specification is used, care should be taken to match the capabilities of the metatype package to the capabilities of the Configuration Admin service specification. Specifically:

- The metatype specification cannot describe nested arrays and vectors or arrays/vectors of mixed type.

This specification does not address how the metatype is made available to a management system due to the many open issues regarding remote management.

104.11 Security

104.11.1 Configuration Permission

Every bundle has the implicit right to receive and configure configurations with a location that exactly matches the Bundle's location or that is null. For all other situations the Configuration Admin must verify that the configuring and to be updated bundles have a Configuration Permission that matches the Configuration's location.

The resource name of this permission maps to the location of the Configuration, the location can control the visibility of a Configuration for a bundle. The resource name is compared with the actual configuration location using the OSGi Filter sub-string matching. The question mark for multi-locations is part of the given resource name. The Configure Permission has the following actions:

- CONFIGURE – Can manage matching configurations
- TARGET – Can be updated with a matching configuration

To be able to set the location to null requires a ConfigurationPermission[*,CONFIGURE].

It is possible to deny bundles the use of multi-locations by using Conditional Permission Admin's deny model.

104.11.2 Permissions Summary

Configuration Admin service security is implemented using Service Permission and Configuration Permission. The following table summarizes the permissions needed by the Configuration Admin bundle itself, as well as the typical permissions needed by the bundles with which it interacts.

Configuration Admin:

```
ServicePermission[ ..ConfigurationAdmin, REGISTER ]
ServicePermission[ ..ManagedService, GET ]
ServicePermission[ ..ManagedServiceFactory, GET ]
ServicePermission[ ..ConfigurationPlugin, GET ]
ConfigurationPermission[ *, CONFIGURE ]
AdminPermission[ *, METADATA ]
```

Managed Service:

```
ServicePermission[ ..ConfigurationAdmin, GET ]
ServicePermission[ ..ManagedService, REGISTER ]
ConfigurationPermission[ ... , TARGET ]
```

Managed Service Factory:

```
ServicePermission[ ..ConfigurationAdmin, GET ]
ServicePermission[ ..ManagedServiceFactory, REGISTER ]
ConfigurationPermission[ ... , TARGET ]
```

Configuration Plugin:

```
ServicePermission[ ..ConfigurationPlugin, REGISTER ]
```

Configuration Listener:

```
ServicePermission[ ..ConfigurationListener, REGISTER ]
```

The Configuration Admin service must have ServicePermission[ConfigurationAdmin, REGISTER]. It will also be the only bundle that needs the ServicePermission[ManagedService | ManagedServiceFactory |ConfigurationPlugin, GET]. No other bundle should be allowed to have GET permission for these interfaces. The Configuration Admin bundle must also hold ConfigurationPermission[*,CONFIGURE].

Bundles that can be configured must have the ServicePermission[ManagedService | ManagedServiceFactory, REGISTER]. Bundles registering ConfigurationPlugin objects must have ServicePermission[ConfigurationPlugin, REGISTER]. The Configuration Admin service must trust all services registered with the ConfigurationPlugin interface. Only the Configuration Admin service should have ServicePermission[ConfigurationPlugin, GET].

If a Managed Service or Managed Service Factory is implemented by an object that is also registered under another interface, it is possible, although inappropriate, for a bundle other than the Configuration Admin service implementation to call the updated method. Security-aware bundles can avoid this problem by having their updated methods check that the caller has ConfigurationPermission[*, CONFIGURE].

Bundles that want to change their own configuration need ServicePermission[ConfigurationAdmin, GET]. A bundle with ConfigurationPermission[*,CONFIGURE]is allowed to access and modify any Configuration object.

Pre-configuration of bundles requires ConfigurationPermission[location,CONFIGURE] (location can use the sub-string matching rules of the Filter) because the methods that specify a location require this permission.

104.11.3 Configuration and Permission Administration

Configuration information has a direct influence on the permissions needed by a bundle. For example, when the Configuration Admin Bundle orders a bundle to use port 2011 for a console, that bundle also needs permission for listening to incoming connections on that port.

Both a simple and a complex solution exist for this situation.

The simple solution for this situation provides the bundle with a set of permissions that do not define specific values but allow a range of values. For example, a bundle could listen to ports above 1024 freely. All these ports could then be used for configuration.

The other solution is more complicated. In an environment where there is very strong security, the bundle would only be allowed access to a specific port. This situation requires an atomic update of both the configuration data and the permissions. If this update was not atomic, a potential security hole would exist during the period of time that the set of permissions did not match the configuration.

The following scenario can be used to update a configuration and the security permissions:

1 Stop the bundle.
2 Update the appropriate Configuration object via the Configuration Admin service.
3 Update the permissions in the Framework.
4 Start the bundle.

This scenario would achieve atomicity from the point of view of the bundle.

104.12 Changes

- Removed the remote management section.
- Allow multiple bundles to register with the same PID to access to the same configuration, see *Location Binding* on page 93.
- Extended Configuration Permission with a location name so that it can be used to prevent access to Configurations marked with a certain location, enabling regions, see *Regions* on page 106.

- Removed the reference to Configurable

104.13 org.osgi.service.cm

Configuration Admin Package Version 1.4.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.cm; version="[1.4,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.cm; version="[1.4,1.5)"
```

104.13.1 Summary

- Configuration – The configuration information for a ManagedService or ManagedServiceFactory object.
- ConfigurationAdmin – Service for administering configuration data.
- ConfigurationEvent – A Configuration Event.
- ConfigurationException – An Exception class to inform the Configuration Admin service of problems with configuration data.
- ConfigurationListener – Listener for Configuration Events.
- ConfigurationPermission – Indicates a bundle's authority to configure bundles or be updated by Configuration Admin.
- ConfigurationPlugin – A service interface for processing configuration dictionary before the update.
- ManagedService – A service that can receive configuration data from a Configuration Admin service.
- ManagedServiceFactory – Manage multiple service instances.

104.13.2 Permissions

104.13.2.1 ManagedServiceFactory

- updated(String,Dictionary)
  - ConfigurationPermission[c.location,TARGET] – Required by the bundle that registered this service

104.13.2.2 ManagedService

- updated(Dictionary)
  - ConfigurationPermission[c.location,TARGET] – Required by the bundle that registered this service

104.13.2.3 ConfigurationAdmin

- createFactoryConfiguration(String,String)
  - ConfigurationPermission[location,CONFIGURE] – if location is not null
  - ConfigurationPermission["*",CONFIGURE] – if location is null
- getConfiguration(String,String)
  - ConfigurationPermission[*,CONFIGURE] – if location is null or if the returned configuration c already exists and c.location is null
  - ConfigurationPermission[location,CONFIGURE] – if location is not null
  - ConfigurationPermission[c.location,CONFIGURE] – if the returned configuration c already exists and c.location is not null
- getConfiguration(String)

   - ConfigurationPermission[c.location,CONFIGURE] – If the configuration c already exists and c.location is not null
- listConfigurations(String)
   - ConfigurationPermission[c.location,CONFIGURE] – Only configurations c are returned for which the caller has this permission

104.13.2.4 Configuration
- setBundleLocation(String)
   - ConfigurationPermission[this.location,CONFIGURE] – if this.location is not null
   - ConfigurationPermission[location,CONFIGURE] – if location is not null
   - ConfigurationPermission["*",CONFIGURE] – if this.location is null or if location is null
- getBundleLocation()
   - ConfigurationPermission[this.location,CONFIGURE] – if this.location is not null
   - ConfigurationPermission["*",CONFIGURE] – if this.location is null

104.13.3 public interface Configuration

The configuration information for a ManagedService or ManagedServiceFactory object. The Configuration Admin service uses this interface to represent the configuration information for a ManagedService or for a service instance of a ManagedServiceFactory.

A Configuration object contains a configuration dictionary and allows the properties to be updated via this object. Bundles wishing to receive configuration dictionaries do not need to use this class - they register a ManagedService or ManagedServiceFactory. Only administrative bundles, and bundles wishing to update their own configurations need to use this class.

The properties handled in this configuration have case insensitive String objects as keys. However, case must be preserved from the last set key/value.

A configuration can be *bound* to a specific bundle or to a region of bundles using the *location*. In its simplest form the location is the location of the target bundle that registered a Managed Service or a Managed Service Factory. However, if the location starts with ? then the location indicates multiple delivery. In such a case the configuration must be delivered to all targets. If security is on, the Configuration Permission can be used to restrict the targets that receive updates. The Configuration Admin must only update a target when the configuration location matches the location of the target's bundle or the target bundle has a Configuration Permission with the action ConfigurationPermission.TARGET and a name that matches the configuration location. The name in the permission may contain wildcards ('*') to match the location using the same substring matching rules as Filter. Bundles can always create, manipulate, and be updated from configurations that have a location that matches their bundle location.

If a configuration's location is null, it is not yet bound to a location. It will become bound to the location of the first bundle that registers a ManagedService or ManagedServiceFactory object with the corresponding PID.

The same Configuration object is used for configuring both a Managed Service Factory and a Managed Service. When it is important to differentiate between these two the term "factory configuration" is used.

No Implement Consumers of this API must not implement this interface

104.13.3.1 public void delete ( ) throws IOException

□ Delete this Configuration object. Removes this configuration object from the persistent store. Notify asynchronously the corresponding Managed Service or Managed Service Factory. A ManagedService object is notified by a call to its updated method with a null properties argument. A ManagedServiceFactory object is notified by a call to its deleted method.

Also initiates an asynchronous call to all ConfigurationListeners with a ConfigurationEvent.CM_DELETED event.

Throws `IOException` – If delete fails.

`IllegalStateException` – If this configuration has been deleted.

104.13.3.2 public boolean equals (Object other)

other `Configuration` object to compare against

□ Equality is defined to have equal PIDs Two Configuration objects are equal when their PIDs are equal.

Returns `true` if equal, `false` if not a `Configuration` object or one with a different PID.

104.13.3.3 public String getBundleLocation ( )

□ Get the bundle location. Returns the bundle location or region to which this configuration is bound, or `null` if it is not yet bound to a bundle location or region. If the location starts with ? then the configuration is delivered to all targets and not restricted to a single bundle.

Returns location to which this configuration is bound, or `null`.

Throws `IllegalStateException` – If this configuration has been deleted.

`SecurityException` – when the required permissions are not available

Security `ConfigurationPermission[this.location,CONFIGURE]` – if this.location is not `null`

`ConfigurationPermission["*",CONFIGURE]` – if this.location is `null`

104.13.3.4 public String getFactoryPid ( )

□ For a factory configuration return the PID of the corresponding Managed Service Factory, else return `null`.

Returns factory PID or `null`

Throws `IllegalStateException` – If this configuration has been deleted.

104.13.3.5 public String getPid ( )

□ Get the PID for this `Configuration` object.

Returns the PID for this `Configuration` object.

Throws `IllegalStateException` – if this configuration has been deleted

104.13.3.6 public Dictionary<String,Object> getProperties ( )

□ Return the properties of this `Configuration` object. The `Dictionary` object returned is a private copy for the caller and may be changed without influencing the stored configuration. The keys in the returned dictionary are case insensitive and are always of type `String`.

If called just after the configuration is created and before update has been called, this method returns `null`.

Returns A private copy of the properties for the caller or `null`. These properties must not contain the "service.bundleLocation" property. The value of this property may be obtained from the `getBundleLocation()` method.

Throws `IllegalStateException` – If this configuration has been deleted.

104.13.3.7 public int hashCode ( )

□ Hash code is based on PID. The hash code for two Configuration objects must be the same when the Configuration PID's are the same.

Returns hash code for this Configuration object

104.13.3.8 public void setBundleLocation (String location)

location a location, region, or `null`

□ Bind this Configuration object to the specified location. If the location parameter is null then the Configuration object will not be bound to a location/region. It will be set to the bundle's location before the first time a Managed Service/Managed Service Factory receives this Configuration object via the updated method and before any plugins are called. The bundle location or region will be set persistently.

If the location starts with ? then all targets registered with the given PID must be updated.

If the location is changed then existing targets must be informed. If they can no longer see this configuration, the configuration must be deleted or updated with null. If this configuration becomes visible then they must be updated with this configuration.

Also initiates an asynchronous call to all ConfigurationListeners with a ConfigurationEvent.CM_LOCATION_CHANGED event.

Throws IllegalStateException – If this configuration has been deleted.

SecurityException – when the required permissions are not available

SecurityException – when the required permissions are not available

Security ConfigurationPermission[this.location,CONFIGURE] – if this.location is not null

ConfigurationPermission[location,CONFIGURE] – if location is not null

ConfigurationPermission["*",CONFIGURE] – if this.location is null or if location is null

104.13.3.9 public void update (Dictionary<String,?> properties) throws IOException

properties the new set of properties for this configuration

□ Update the properties of this Configuration object. Stores the properties in persistent storage after adding or overwriting the following properties:

- "service.pid" : is set to be the PID of this configuration.
- "service.factoryPid" : if this is a factory configuration it is set to the factory PID else it is not set.

These system properties are all of type String.

If the corresponding Managed Service/Managed Service Factory is registered, its updated method must be called asynchronously. Else, this callback is delayed until aforementioned registration occurs.

Also initiates an asynchronous call to all ConfigurationListeners with a ConfigurationEvent.CM_UPDATED event.

Throws IOException – if update cannot be made persistent

IllegalArgumentException – if the Dictionary object contains invalid configuration types or contains case variants of the same key name.

IllegalStateException – If this configuration has been deleted.

104.13.3.10 public void update ( ) throws IOException

□ Update the Configuration object with the current properties. Initiate the updated callback to the Managed Service or Managed Service Factory with the current properties asynchronously.

This is the only way for a bundle that uses a Configuration Plugin service to initiate a callback. For example, when that bundle detects a change that requires an update of the Managed Service or Managed Service Factory via its ConfigurationPlugin object.

Throws IOException – if update cannot access the properties in persistent storage

IllegalStateException – If this configuration has been deleted.

See Also ConfigurationPlugin

104.13.4 public interface ConfigurationAdmin

Service for administering configuration data.

The main purpose of this interface is to store bundle configuration data persistently. This information is represented in Configuration objects. The actual configuration data is a Dictionary of properties inside a Configuration object.

There are two principally different ways to manage configurations. First there is the concept of a Managed Service, where configuration data is uniquely associated with an object registered with the service registry.

Next, there is the concept of a factory where the Configuration Admin service will maintain 0 or more Configuration objects for a Managed Service Factory that is registered with the Framework.

The first concept is intended for configuration data about “things/services” whose existence is defined externally, e.g. a specific printer. Factories are intended for “things/services” that can be created any number of times, e.g. a configuration for a DHCP server for different networks.

Bundles that require configuration should register a Managed Service or a Managed Service Factory in the service registry. A registration property named service.pid (persistent identifier or PID) must be used to identify this Managed Service or Managed Service Factory to the Configuration Admin service.

When the ConfigurationAdmin detects the registration of a Managed Service, it checks its persistent storage for a configuration object whose service.pid property matches the PID service property (service.pid) of the Managed Service. If found, it calls ManagedService.updated(Dictionary) method with the new properties. The implementation of a Configuration Admin service must run these callbacks asynchronously to allow proper synchronization.

When the Configuration Admin service detects a Managed Service Factory registration, it checks its storage for configuration objects whose service.factoryPid property matches the PID service property of the Managed Service Factory. For each such Configuration objects, it calls the ManagedServiceFactory.updated method asynchronously with the new properties. The calls to the updated method of a ManagedServiceFactory must be executed sequentially and not overlap in time.

In general, bundles having permission to use the Configuration Admin service can only access and modify their own configuration information. Accessing or modifying the configuration of other bundles requires ConfigurationPermission[location,CONFIGURE], where location is the configuration location.

Configuration objects can be *bound* to a specified bundle location or to a region (configuration location starts with ?). If a location is not set, it will be learned the first time a target is registered. If the location is learned this way, the Configuration Admin service must detect if the bundle corresponding to the location is uninstalled. If this occurs, the Configuration object must be unbound, that is its location field is set back to null.

If target’s bundle location matches the configuration location it is always updated.

If the configuration location starts with ?, that is, the location is a region, then the configuration must be delivered to all targets registered with the given PID. If security is on, the target bundle must have Configuration Permission[location,TARGET], where location matches given the configuration location with wildcards as in the Filter substring match. The security must be verified using the org.osgi.framework.Bundle.hasPermission(Object) method on the target bundle.

If a target cannot be updated because the location does not match or it has no permission and security is active then the Configuration Admin service must not do the normal callback.

The method descriptions of this class refer to a concept of “the calling bundle”. This is a loose way of referring to the bundle which obtained the Configuration Admin service from the service registry. Implementations of ConfigurationAdmin must use a org.osgi.framework.ServiceFactory to support this concept.

No Implement Consumers of this API must not implement this interface

104.13.4.1 public static final String SERVICE_BUNDLELOCATION = “service.bundleLocation”

Configuration property naming the location of the bundle that is associated with a a Configuration object. This property can be searched for but must not appear in the configuration dictionary for security reason. The property’s value is of type String.

Since 1.1

104.13.4.2 public static final String SERVICE_FACTORYPID = “service.factoryPid”

Configuration property naming the Factory PID in the configuration dictionary. The property’s value is of type String.

Since 1.1

104.13.4.3 public Configuration createFactoryConfiguration (String factoryPid) throws IOException

factoryPid PID of factory (not null).

□ Create a new factory Configuration object with a new PID. The properties of the new Configuration object are null until the first time that its Configuration.update(Dictionary) method is called.

It is not required that the factoryPid maps to a registered Managed Service Factory.

The Configuration object is bound to the location of the calling bundle. It is possible that the same factoryPid has associated configurations that are bound to different bundles. Bundles should only see the factory configurations that they are bound to or have the proper permission.

Returns A new Configuration object.

Throws IOException – if access to persistent storage fails.

104.13.4.4 public Configuration createFactoryConfiguration (String factoryPid , String location) throws IOException

factoryPid PID of factory (not null).

location A bundle location string, or null.

□ Create a new factory Configuration object with a new PID. The properties of the new Configuration object are null until the first time that its Configuration.update(Dictionary) method is called.

It is not required that the factoryPid maps to a registered Managed Service Factory.

The Configuration is bound to the location specified. If this location is null it will be bound to the location of the first bundle that registers a Managed Service Factory with a corresponding PID. It is possible that the same factoryPid has associated configurations that are bound to different bundles. Bundles should only see the factory configurations that they are bound to or have the proper permission.

If the location starts with ? then the configuration must be delivered to all targets with the corresponding PID.

Returns a new Configuration object.

Throws IOException – if access to persistent storage fails.

SecurityException – when the require permissions are not available

Security ConfigurationPermission[location,CONFIGURE] – if location is not null

ConfigurationPermission[”*”,CONFIGURE] – if location is null

104.13.4.5 public Configuration getConfiguration (String pid , String location) throws IOException

pid Persistent identifier.

location The bundle location string, or null.

□ Get an existing Configuration object from the persistent store, or create a new Configuration object.

If a Configuration with this PID already exists in Configuration Admin service return it. The location parameter is ignored in this case though it is still used for a security check.

Else, return a new Configuration object. This new object is bound to the location and the properties are set to null. If the location parameter is null, it will be set when a Managed Service with the corresponding PID is registered for the first time. If the location starts with ? then the configuration is bound to all targets that are registered with the corresponding PID.

Returns An existing or new Configuration object.

Throws IOException – if access to persistent storage fails.

SecurityException – when the require permissions are not available

Security ConfigurationPermission[*,CONFIGURE] – if location is null or if the returned configuration c already exists and c.location is null

ConfigurationPermission[location,CONFIGURE] – if location is not null

ConfigurationPermission[c.location,CONFIGURE] – if the returned configuration c already exists and c.location is not null

104.13.4.6 public Configuration getConfiguration (String pid) throws IOException

pid persistent identifier.

□ Get an existing or new Configuration object from the persistent store. If the Configuration object for this PID does not exist, create a new Configuration object for that PID, where properties are null. Bind its location to the calling bundle's location.

Otherwise, if the location of the existing Configuration object is null, set it to the calling bundle's location.

Returns an existing or new Configuration matching the PID.

Throws IOException – if access to persistent storage fails.

SecurityException – when the required permission is not available

Security ConfigurationPermission[c.location,CONFIGURE] – If the configuration c already exists and c.location is not null

104.13.4.7 public Configuration[] listConfigurations (String filter) throws IOException , InvalidSyntaxException

filter A filter string, or null to retrieve all Configuration objects.

□ List the current Configuration objects which match the filter.

Only Configuration objects with non- null properties are considered current. That is, Configuration.getProperties() is guaranteed not to return null for each of the returned Configuration objects.

When there is no security on then all configurations can be returned. If security is on, the caller must have ConfigurationPermission[location,CONFIGURE].

The syntax of the filter string is as defined in the Filter class. The filter can test any configuration properties including the following:

- service.pid - the persistent identity
- service.factoryPid - the factory PID, if applicable

- service.bundleLocation - the bundle location

The filter can also be null, meaning that all Configuration objects should be returned.

Returns All matching Configuration objects, or null if there aren't any.

Throws IOException – if access to persistent storage fails

InvalidSyntaxException – if the filter string is invalid

Security ConfigurationPermission[c.location,CONFIGURE] – Only configurations c are returned for which the caller has this permission

104.13.5 public class ConfigurationEvent

A Configuration Event.

ConfigurationEvent objects are delivered to all registered ConfigurationListener service objects. ConfigurationEvents must be asynchronously delivered in chronological order with respect to each listener.

A type code is used to identify the type of event. The following event types are defined:

- CM_UPDATED
- CM_DELETED
- CM_LOCATION_CHANGED

Additional event types may be defined in the future.

Security Considerations. ConfigurationEvent objects do not provide Configuration objects, so no sensitive configuration information is available from the event. If the listener wants to locate the Configuration object for the specified pid, it must use ConfigurationAdmin.

See Also ConfigurationListener

Since 1.2

Concurrency Immutable

104.13.5.1 public static final int CM_DELETED = 2

A Configuration has been deleted.

This ConfigurationEvent type that indicates that a Configuration object has been deleted. An event is fired when a call to Configuration.delete() successfully deletes a configuration.

104.13.5.2 public static final int CM_LOCATION_CHANGED = 3

The location of a Configuration has been changed.

This ConfigurationEvent type that indicates that the location of a Configuration object has been changed. An event is fired when a call to Configuration.setBundleLocation(String) successfully changes the location.

Since 1.4

104.13.5.3 public static final int CM_UPDATED = 1

A Configuration has been updated.

This ConfigurationEvent type that indicates that a Configuration object has been updated with new properties. An event is fired when a call to Configuration.update(Dictionary) successfully changes a configuration.

104.13.5.4 public ConfigurationEvent (ServiceReference<ConfigurationAdmin> reference , int type , String factoryPid , String pid)

reference The ServiceReference object of the Configuration Admin service that created this event.

type The event type. See `getType()`.

factoryPid The factory pid of the associated configuration if the target of the configuration is a ManagedServiceFactory. Otherwise `null` if the target of the configuration is a ManagedService.

pid The pid of the associated configuration.

□ Constructs a `ConfigurationEvent` object from the given `ServiceReference` object, event type, and pids.

104.13.5.5 public String getFactoryPid ( )

□ Returns the factory pid of the associated configuration.

Returns Returns the factory pid of the associated configuration if the target of the configuration is a ManagedServiceFactory. Otherwise `null` if the target of the configuration is a ManagedService.

104.13.5.6 public String getPid ( )

□ Returns the pid of the associated configuration.

Returns Returns the pid of the associated configuration.

104.13.5.7 public ServiceReference<ConfigurationAdmin> getReference ( )

□ Return the `ServiceReference` object of the Configuration Admin service that created this event.

Returns The `ServiceReference` object for the Configuration Admin service that created this event.

104.13.5.8 public int getType ( )

□ Return the type of this event.

The type values are:

- `CM_UPDATED`
- `CM_DELETED`
- `CM_LOCATION_CHANGED`

Returns The type of this event.

104.13.6 public class ConfigurationException extends Exception

An `Exception` class to inform the Configuration Admin service of problems with configuration data.

104.13.6.1 public ConfigurationException (String property , String reason)

property name of the property that caused the problem, `null` if no specific property was the cause

reason reason for failure

□ Create a `ConfigurationException` object.

104.13.6.2 public ConfigurationException (String property , String reason , Throwable cause)

property name of the property that caused the problem, `null` if no specific property was the cause

reason reason for failure

cause The cause of this exception.

□ Create a `ConfigurationException` object.

Since 1.2

104.13.6.3 public Throwable getCause ( )

□ Returns the cause of this exception or `null` if no cause was set.

Returns The cause of this exception or null if no cause was set.

Since 1.2

104.13.6.4 public String getProperty ( )

□ Return the property name that caused the failure or null.

Returns name of property or null if no specific property caused the problem

104.13.6.5 public String getReason ( )

□ Return the reason for this exception.

Returns reason of the failure

104.13.6.6 public Throwable initCause (Throwable cause)

cause The cause of this exception.

□ Initializes the cause of this exception to the specified value.

Returns This exception.

Throws `IllegalArgumentException` – If the specified cause is this exception.

`IllegalStateException` – If the cause of this exception has already been set.

Since 1.2

104.13.7 public interface ConfigurationListener

Listener for Configuration Events. When a ConfigurationEvent is fired, it is asynchronously delivered to a ConfigurationListener.

ConfigurationListener objects are registered with the Framework service registry and are notified with a ConfigurationEvent object when an event is fired.

ConfigurationListener objects can inspect the received ConfigurationEvent object to determine its type, the pid of the Configuration object with which it is associated, and the Configuration Admin service that fired the event.

Security Considerations. Bundles wishing to monitor configuration events will require ServicePermission[ConfigurationListener,REGISTER] to register a ConfigurationListener service.

Since 1.2

104.13.7.1 public void configurationEvent (ConfigurationEvent event)

event The ConfigurationEvent.

□ Receives notification of a Configuration that has changed.

104.13.8 public final class ConfigurationPermission extends BasicPermission

Indicates a bundle's authority to configure bundles or be updated by Configuration Admin.

Since 1.2

Concurrency Thread-safe

104.13.8.1 public static final String CONFIGURE = "configure"

Provides permission to create new configurations for other bundles as well as manipulate them. The action string configure.

104.13.8.2 public static final String TARGET = "target"

The permission to be updated, that is, act as a Managed Service or Managed Service Factory. The action string target.

Since 1.4

104.13.8.3 public ConfigurationPermission (String name , String actions)

name Name of the permission. Wildcards ('*') are allowed in the name. During implies(Permission), the name is matched to the requested permission using the substring matching rules used by Filters.

actions Comma separated list of CONFIGURE, TARGET (case insensitive).

□ Create a new ConfigurationPermission.

104.13.8.4 public boolean equals (Object obj)

obj The object being compared for equality with this object.

□ Determines the equality of two ConfigurationPermission objects.

Two ConfigurationPermission objects are equal.

Returns true if obj is equivalent to this ConfigurationPermission; false otherwise.

104.13.8.5 public String getActions ( )

□ Returns the canonical string representation of the ConfigurationPermission actions.

Always returns present ConfigurationPermission actions in the following order: configure, target

Returns Canonical string representation of the ConfigurationPermission actions.

104.13.8.6 public int hashCode ( )

□ Returns the hash code value for this object.

Returns Hash code value for this object.

104.13.8.7 public boolean implies (Permission p)

p The target permission to check.

□ Determines if a ConfigurationPermission object "implies" the specified permission.

Returns true if the specified permission is implied by this object; false otherwise.

104.13.8.8 public PermissionCollection newPermissionCollection ( )

□ Returns a new PermissionCollection object suitable for storing ConfigurationPermissions.

Returns A new PermissionCollection object.

104.13.9 public interface ConfigurationPlugin

A service interface for processing configuration dictionary before the update.

A bundle registers a ConfigurationPlugin object in order to process configuration updates before they reach the Managed Service or Managed Service Factory. The Configuration Admin service will detect registrations of Configuration Plugin services and must call these services every time before it calls the ManagedService or ManagedServiceFactoryupdated method. The Configuration Plugin service thus has the opportunity to view and modify the properties before they are passed to the Managed Service or Managed Service Factory.

Configuration Plugin (plugin) services have full read/write access to all configuration information that passes through them.

The Integerservice.cmRanking registration property may be specified. Not specifying this registration property, or setting it to something other than an Integer, is the same as setting it to the Integer zero. The service.cmRanking property determines the order in which plugins are invoked. Lower ranked plugins are called before higher ranked ones. In the event of more than one plugin having the same value of service.cmRanking, then the Configuration Admin service arbitrarily chooses the order in which they are called.

By convention, plugins with service.cmRanking< 0 or service.cmRanking > 1000 should not make modifications to the properties.

The Configuration Admin service has the right to hide properties from plugins, or to ignore some or all the changes that they make. This might be done for security reasons. Any such behavior is entirely implementation defined.

A plugin may optionally specify a cm.target registration property whose value is the PID of the Managed Service or Managed Service Factory whose configuration updates the plugin is intended to intercept. The plugin will then only be called with configuration updates that are targeted at the Managed Service or Managed Service Factory with the specified PID. Omitting the cm.target registration property means that the plugin is called for all configuration updates.

104.13.9.1 public static final String CM_RANKING = "service.cmRanking"

A service property to specify the order in which plugins are invoked. This property contains an Integer ranking of the plugin. Not specifying this registration property, or setting it to something other than an Integer, is the same as setting it to the Integer zero. This property determines the order in which plugins are invoked. Lower ranked plugins are called before higher ranked ones.

Since 1.2

104.13.9.2 public static final String CM_TARGET = "cm.target"

A service property to limit the Managed Service or Managed Service Factory configuration dictionaries a Configuration Plugin service receives. This property contains a String[] of PIDs. A Configuration Admin service must call a Configuration Plugin service only when this property is not set, or the target service's PID is listed in this property.

104.13.9.3 public void modifyConfiguration (ServiceReference<?> reference , Dictionary<String,Object> properties)

reference reference to the Managed Service or Managed Service Factory

properties The configuration properties. This argument must not contain the "service.bundleLocation" property. The value of this property may be obtained from the Configuration.getBundleLocation method.

□ View and possibly modify the a set of configuration properties before they are sent to the Managed Service or the Managed Service Factory. The Configuration Plugin services are called in increasing order of their service.cmRanking property. If this property is undefined or is a non- Integer type, 0 is used.

This method should not modify the properties unless the service.cmRanking of this plugin is in the range 0 <= service.cmRanking <= 1000.

If this method throws any Exception, the Configuration Admin service must catch it and should log it.

A Configuration Plugin will only be called for properties from configurations that have a location for which the Configuration Plugin has permission when security is active. When security is not active, no filtering is done.

104.13.10 public interface ManagedService

A service that can receive configuration data from a Configuration Admin service.

A Managed Service is a service that needs configuration data. Such an object should be registered with the Framework registry with the service.pid property set to some unique identifier called a PID.

If the Configuration Admin service has a Configuration object corresponding to this PID, it will callback the updated() method of the ManagedService object, passing the properties of that Configuration object.

If it has no such Configuration object, then it calls back with a null properties argument. Registering a Managed Service will always result in a callback to the updated() method provided the Configuration Admin service is, or becomes active. This callback must always be done asynchronously.

Else, every time that either of the updated() methods is called on that Configuration object, the ManagedService.updated() method with the new properties is called. If the delete() method is called on that Configuration object, ManagedService.updated() is called with a null for the properties parameter. All these callbacks must be done asynchronously.

The following example shows the code of a serial port that will create a port depending on configuration information.

```
class SerialPort implements ManagedService {

  ServiceRegistration registration;
  Hashtable configuration;
  CommPortIdentifier id;

  synchronized void open(CommPortIdentifier id,
  BundleContext context) {
    this.id = id;
    registration = context.registerService(
      ManagedService.class.getName(),
      this,
      getDefaults()
    );
  }

  Hashtable getDefaults() {
    Hashtable defaults = new Hashtable();
    defaults.put( “port”, id.getName() );
    defaults.put( “product”, “unknown” );
    defaults.put( “baud”, “9600” );
    defaults.put( Constants.SERVICE_PID,
      “com.acme.serialport.” + id.getName() );
    return defaults;
  }

  public synchronized void updated(
    Dictionary configuration  ) {
    if ( configuration == null )
      registration.setProperties( getDefaults() );
    else {
      setSpeed( configuration.get(”baud”) );
      registration.setProperties( configuration );
    }
  }
  ...
}
```

As a convention, it is recommended that when a Managed Service is updated, it should copy all the properties it does not recognize into the service registration properties. This will allow the Configuration Admin service to set properties on services which can then be used by other applications.

Normally, a single Managed Service for a given PID is given the configuration dictionary, this is the configuration that is bound to the location of the registering bundle. However, when security is on, a Managed Service can have Configuration Permission to also be updated for other locations.

104.13.10.1 public void updated (Dictionary<String,?> properties) throws ConfigurationException

properties A copy of the Configuration properties, or `null` . This argument must not contain the "service.bundleLocation" property. The value of this property may be obtained from the `Configuration.getBundleLocation` method.

□ Update the configuration for a Managed Service.

When the implementation of `updated(Dictionary)` detects any kind of error in the configuration properties, it should create a new `ConfigurationException` which describes the problem. This can allow a management system to provide useful information to a human administrator.

If this method throws any other `Exception`, the Configuration Admin service must catch it and should log it.

The Configuration Admin service must call this method asynchronously with the method that initiated the callback. This implies that implementors of Managed Service can be assured that the callback will not take place during registration when they execute the registration in a synchronized method.

If the the location allows multiple managed services to be called back for a single configuration then the callbacks must occur in service ranking order. Changes in the location must be reflected by deleting the configuration if the configuration is no longer visible and updating when it becomes visible.

If no configuration exists for the corresponding PID, or the bundle has no access to the configuration, then the bundle must be called back with a `null` to signal that CM is active but there is no data.

Throws `ConfigurationException` – when the update fails

Security `ConfigurationPermission[c.location,TARGET]` – Required by the bundle that registered this service

104.13.11 public interface ManagedServiceFactory

Manage multiple service instances. Bundles registering this interface are giving the Configuration Admin service the ability to create and configure a number of instances of a service that the implementing bundle can provide. For example, a bundle implementing a DHCP server could be instantiated multiple times for different interfaces using a factory.

Each of these *service instances* is represented, in the persistent storage of the Configuration Admin service, by a factory `Configuration` object that has a PID. When such a `Configuration` is updated, the Configuration Admin service calls the `ManagedServiceFactory` updated method with the new properties. When `updated` is called with a new PID, the Managed Service Factory should create a new factory instance based on these configuration properties. When called with a PID that it has seen before, it should update that existing service instance with the new configuration information.

In general it is expected that the implementation of this interface will maintain a data structure that maps PIDs to the factory instances that it has created. The semantics of a factory instance are defined by the Managed Service Factory. However, if the factory instance is registered as a service object with the service registry, its PID should match the PID of the corresponding `Configuration` object (but it should **not** be registered as a Managed Service!).

An example that demonstrates the use of a factory. It will create serial ports under command of the Configuration Admin service.

```
class SerialPortFactory
  implements ManagedServiceFactory {
  ServiceRegistration registration;
  Hashtable ports;
  void start(BundleContext context) {
    Hashtable properties = new Hashtable();
    properties.put( Constants.SERVICE_PID,
      "com.acme.serialportfactory" );
    registration = context.registerService(
      ManagedServiceFactory.class.getName(),
      this,
      properties
    );
  }
  public void updated( String pid,
    Dictionary properties  ) {
    String portName = (String) properties.get("port");
    SerialPortService port =
      (SerialPort) ports.get( pid );
    if ( port == null ) {
      port = new SerialPortService();
      ports.put( pid, port );
      port.open();
    }
    if ( port.getPortName().equals(portName) )
      return;
    port.setPortName( portName );
  }
  public void deleted( String pid ) {
    SerialPortService port =
      (SerialPort) ports.get( pid );
    port.close();
    ports.remove( pid );
  }
  ...
}
```

104.13.11.1 public void deleted (String pid)

pid the PID of the service to be removed

□ Remove a factory instance. Remove the factory instance associated with the PID. If the instance was registered with the service registry, it should be unregistered. The Configuration Admin must call deleted for each instance it received in updated(String, Dictionary).

If this method throws any Exception, the Configuration Admin service must catch it and should log it.

The Configuration Admin service must call this method asynchronously.

104.13.11.2 public String getName ( )

□ Return a descriptive name of this factory.

Returns the name for the factory, which might be localized

104.13.11.3 public void updated (String pid , Dictionary<String,?> properties) throws

ConfigurationException

pid The PID for this configuration.

properties A copy of the configuration properties. This argument must not contain the service.bundleLocation" property. The value of this property may be obtained from the `Configuration.getBundleLocation` method.

□ Create a new instance, or update the configuration of an existing instance. If the PID of the `Configuration` object is new for the Managed Service Factory, then create a new factory instance, using the configuration `properties` provided. Else, update the service instance with the provided `properties`.

If the factory instance is registered with the Framework, then the configuration `properties` should be copied to its registry properties. This is not mandatory and security sensitive properties should obviously not be copied.

If this method throws any `Exception`, the Configuration Admin service must catch it and should log it.

When the implementation of updated detects any kind of error in the configuration properties, it should create a new `ConfigurationException` which describes the problem.

The Configuration Admin service must call this method asynchronously. This implies that implementors of the `ManagedServiceFactory` class can be assured that the callback will not take place during registration when they execute the registration in a synchronized method.

If the security allows multiple managed service factories to be called back for a single configuration then the callbacks must occur in service ranking order.

It is valid to create multiple factory instances that are bound to different locations. Managed Service Factory services must only be updated with configurations that are bound to their location or that start with the ? prefix and for which they have permission. Changes in the location must be reflected by deleting the corresponding configuration if the configuration is no longer visible or updating when it becomes visible.

Throws `ConfigurationException` – when the configuration properties are invalid.

Security `ConfigurationPermission[c.location,TARGET]` – Required by the bundle that registered this service

104.14

105 Metatype Service Specification

Version 1.2

105.1 Introduction

The Metatype specification defines interfaces that allow bundle developers to describe attribute types in a computer readable form using so-called *metadata*.

The purpose of this specification is to allow services to specify the type information of data that they can use as arguments. The data is based on *attributes*, which are key/value pairs like properties.

A designer in a type-safe language like Java is often confronted with the choice of using the language constructs to exchange data or using a technique based on attributes/properties that are based on key/value pairs. Attributes provide an escape from the rigid type-safety requirements of modern programming languages.

Type-safety works very well for software development environments in which multiple programmers work together on large applications or systems, but often lacks the flexibility needed to receive structured data from the outside world.

The attribute paradigm has several characteristics that make this approach suitable when data needs to be communicated between different entities which "speak" different languages. Attributes are uncomplicated, resilient to change, and allow the receiver to dynamically adapt to different types of data.

As an example, the OSGi Service Platform Specifications define several attribute types which are used in a Framework implementation, but which are also used and referenced by other OSGi specifications such as the *Configuration Admin Service Specification* on page 89. A Configuration Admin service implementation deploys attributes (key/value pairs) as configuration properties.

The Meta Type Service provides a unified access point to the Meta Type information that is associated with bundles. This Meta Type information can be defined by an XML resource in a bundle (OSGI-INF/metatype directories must be scanned for any XML resources), it can come from the Meta Type Provider service, or it can be obtained from Managed Service or Managed Service Factory services.

105.1.1 Essentials

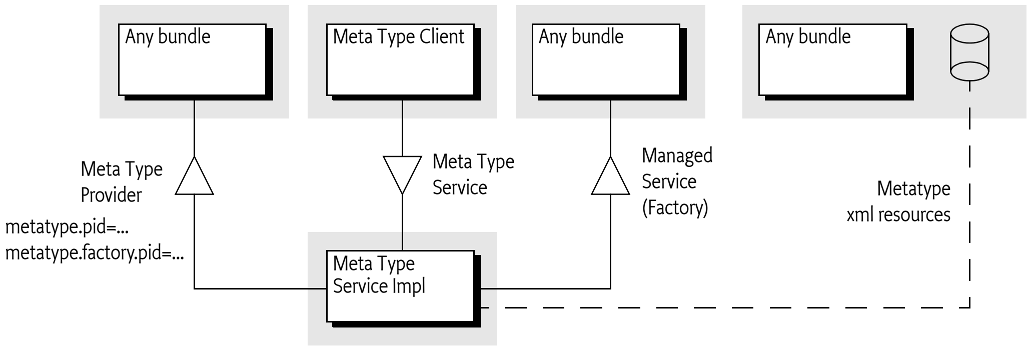
- *Conceptual model* – The specification must have a conceptual model for how classes and attributes are organized.
- *Standards* – The specification should be aligned with appropriate standards, and explained in situations where the specification is not aligned with, or cannot be mapped to, standards.
- *Remote Management* – Remote management should be taken into account.
- *Size* – Minimal overhead in size for a bundle using this specification is required.
- *Localization* – It must be possible to use this specification with different languages at the same time. This ability allows servlets to serve information in the language selected in the browser.
- *Type information* – The definition of an attribute should contain the name (if it is required), the cardinality, a label, a description, labels for enumerated values, and the Java class that should be used for the values.
- *Validation* – It should be possible to validate the values of the attributes.

105.1.2 Entities

- *Meta Type Service* – A service that provides a unified access point for meta type information.

- *Attribute* – A key/value pair.
- *PID* – A unique persistent ID, defined in configuration management.
- *Attribute Definition* – Defines a description, name, help text, and type information of an attribute.
- *Object Class Definition* – Defines the type of a datum. It contains a description and name of the type plus a set of AttributeDefinition objects.
- *Meta Type Provider* – Provides access to the object classes that are available for this object. Access uses the PID and a locale to find the best ObjectClassDefinition object.
- *Meta Type Information* – Provides meta type information for a bundle.

Figure 105.1 Class Diagram Meta Type Service, org.osgi.service.metatype



105.1.3 Operation

The Meta Type service defines a rich dynamic typing system for properties. The purpose of the type system is to allow reasonable User Interfaces to be constructed dynamically.

The type information is normally carried by the bundles themselves. Either by implementing the MetaTypeProvider interface on the Managed Service or Managed Service Factory, by carrying one or more XML resources that define a number of Meta Types in the OSGI-INF/metatype directories, or registering a Meta Type Provider as a service. Additionally, a Meta Type service could have other sources that are not defined in this specification.

The Meta Type Service provides unified access to Meta Types that are carried by the resident bundles. The Meta Type Service collects this information from the bundles and provides uniform access to it. A client can requests the Meta Type Information associated with a particular bundle. The MetaTypeInformation object provides a list of ObjectClassDefinition objects for a bundle. These objects define all the information for a specific *object class*. An object class is a some descriptive information and a set of named attributes (which are key/value pairs).

Access to Object Class Definitions is qualified by a locale and a Persistent IDentity (PID). This specification does not specify what the PID means. One application is OSGi Configuration Management where a PID is used by the Managed Service and Managed Service Factory services. In general, a PID should be regarded as the name of a variable where an Object Class Definition defines its type.

105.2 Attributes Model

The Framework uses the LDAP filter syntax for searching the Framework registry. The usage of the attributes in this specification and the Framework specification closely resemble the LDAP attribute model. Therefore, the names used in this specification have been aligned with LDAP. Consequently, the interfaces which are defined by this Specification are:

- AttributeDefinition

- `ObjectClassDefinition`
- `MetaTypeProvider`

These names correspond to the LDAP attribute model. For further information on ASN.1-defined attributes and X.500 object classes and attributes, see [2] *Understanding and Deploying LDAP Directory services*.

The LDAP attribute model assumes a global name-space for attributes, and object classes consist of a number of attributes. So, if an object class inherits the same attribute from different parents, only one copy of the attribute must become part of the object class definition. This name-space implies that a given attribute, for example `cn`, should *always* be the common name and the type must always be a `String`. An attribute `cn` cannot be an `Integer` in another object class definition. In this respect, the OSGi approach towards attribute definitions is comparable with the LDAP attribute model.

105.3 Object Class Definition

The `ObjectClassDefinition` interface is used to group the attributes which are defined in `AttributeDefinition` objects.

An `ObjectClassDefinition` object contains the information about the overall set of attributes and has the following elements:

- A name which can be returned in different locales.
- A global name-space in the registry, which is the same condition as LDAP/X.500 object classes. In these standards the OSI Object Identifier (OID) is used to uniquely identify object classes. If such an OID exists, (which can be requested at several standard organizations, and many companies already have a node in the tree) it can be returned here. Otherwise, a unique id should be returned. This id can be a Java class name (reverse domain name) or can be generated with a GUID algorithm. All LDAP-defined object classes already have an associated OID. It is strongly advised to define the object classes from existing LDAP schemes which provide many preexisting OIDs. Many such schemes exist ranging from postal addresses to DHCP parameters.
- A human-readable description of the class.
- A list of attribute definitions which can be filtered as required, or optional. Note that in X.500 the mandatory or required status of an attribute is part of the object class definition and not of the attribute definition.
- An icon, in different sizes.

105.4 Attribute Definition

The `AttributeDefinition` interface provides the means to describe the data type of attributes.

The `AttributeDefinition` interface defines the following elements:

- Defined names (final ints) for the data types as restricted in the Framework for the attributes, called the syntax in OSI terms, which can be obtained with the `getType()` method.
- `AttributeDefinition` objects should use an ID that is similar to the OID as described in the ID field for `ObjectClassDefinition`.
- A localized name intended to be used in user interfaces.
- A localized description that defines the semantics of the attribute and possible constraints, which should be usable for tooltips.
- An indication if this attribute should be stored as a unique value, a `Vector`, or an array of values, as well as the maximum cardinality of the type.
- The data type, as limited by the Framework service registry attribute types.
- A validation function to verify if a possible value is correct.
- A list of values and a list of localized labels. Intended for popup menus in GUIs, allowing the user to choose from a set.

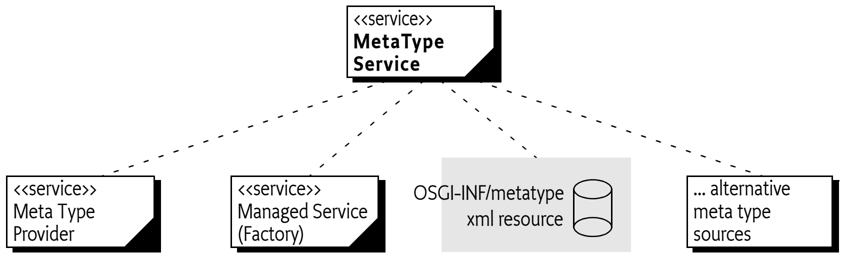
- A default value (String[]). The return depends on the following cases:
  - *not specified* – Return null if this attribute is not specified.
  - *cardinality = 0* – Return an array with one element.
  - *otherwise* – Return an array with less or equal than the absolute value of cardinality, possibly empty if the value is an empty string.

105.5 Meta Type Service

The Meta Type Service provides unified access to Meta Type information that is associated with a Bundle. It can get this information through the following means:

- *Meta Type Resource* – A bundle can provide one or more XML resources that are contained in its JAR file. These resources contain an XML definition of meta types as well as to what PIDs these Meta Types apply. These XML resources must reside in the OSGI-INF/metatype directories of the bundle (including any fragments).
- *ManagedService[Factory] objects* – As defined in the configuration management specification, ManagedService and ManagedServiceFactory service objects can optionally implement the MetaTypeProvider interface. The Meta Type Service will only search for MetaTypeProvider objects if no meta type resources are found in the bundle.
- *Meta Type Provider service* – Bundles can register Meta Type Provider services to dynamically provide meta types for PIDs and factory PIDs.

Figure 105.2 Sources for Meta Types


<<service>>
MetaType Service
<<service>>
Meta Type Provider
<<service>>
Managed Service (Factory)
OSGI-INF/metatype xml resource
... alternative meta type sources

This model is depicted in Figure 105.2.

The Meta Type Service can therefore be used to retrieve meta type information for bundles which contain Meta Type resources or which provide their own MetaTypeProvider objects. The MetaTypeService interface has a single method:

- getMetaTypeInformation(Bundle) – Given a bundle, it must return the Meta Type Information for that bundle, even if there is no meta type information available at the moment of the call.

The returned MetaTypeInformation object maintains a map of PID to ObjectClassDefinition objects. The map is keyed by locale and PID. The list of maintained PIDs is available from the MetaTypeInformation object with the following methods:

- getPids() – PIDs for which Meta Types are available.
- getFactoryPids() – PIDs associated with Managed Service Factory services.

These methods and their interaction with the Meta Type resource are described in *Use of the Designate Element* on page 138.

The MetaTypeInformation interface extends the MetaTypeProvider interface. The MetaTypeProvider interface is used to access meta type information. It supports locale dependent information so that the text used in AttributeDefinition and ObjectClassDefinition objects can be adapted to different locales.

Which locales are supported by the MetaTypeProvider object are defined by the implementer or the meta type resources.The list of available locales can be obtained from the MetaTypeProvider object.

The MetaTypeProvider interface provides the following methods:

- getObjectClassDefinition(String,String) – Get access to an ObjectClassDefinition object for the given PID. The second parameter defines the locale.
- getLocales() – List the locales that are available.

Locale objects are represented in String objects because not all profiles support Locale. The String holds the standard Locale presentation of:

```
locale = language ( '_' country ( '_' variation) )
language ::= < defined by ISO 3166 >
country  ::= < defined by ISO 639 >
```

For example, en, nl_BE, en_CA_posix are valid locales. The use of null for locale indicates that java.util.Locale.getDefault() must be used.

The Meta Type Service implementation class is the main class. It registers the org.osgi.service.metatype.MetaTypeService service and has a method to get a MetaTypeInformation object for a bundle.

Following is some sample code demonstrating how to print out all the Object Class Definitions and Attribute Definitions contained in a bundle:

```
void printMetaTypes( MetaTypeService mts, Bundle b ) {
   MetaTypeInformation mti =
      mts.getMetaTypeInformation(b);
   String [] pids = mti.getPids();
   String [] locales = mti.getLocales();

   for ( int locale = 0; locale<locales.length; locale++ ) {
      System.out.println("Locale " + locales[locale] );
      for (int i=0; i< pids.length; i++) {
         ObjectClassDefinition ocd =
            mti.getObjectClassDefinition(pids[i], null);
         AttributeDefinition[] ads =
            ocd.getAttributeDefinitions(
               ObjectClassDefinition.ALL);
         for (int j=0; j< ads.length; j++) {
           System.out.println("OCD="+ocd.getName()
               + "AD="+ads[j].getName());
         }
      }
   }
}
```

105.6 Meta Type Provider Service

A Meta Type Provider service allows third party contributions to the internal Object Class Definition repository. A Meta Type Provider can contribute multiple PIDs, both factory and singleton PIDs. A Meta Type Provider service must register with both or one of the following service properties:

- METATYPE_PID – (String+) Provides a list of PIDs that this Meta Type Provider can provide Object Class Definitions for. The listed PIDs are intended to be used as normal singleton PIDs used by Managed Services.

- METATYPE_FACTORY_PID – (String+) Provides a list of factory PIDs that this Meta Type Provider can provide Object Class Definitions for. The listed PIDs are intended to be used as factory PIDs used by Managed Service Factories.

The Object Class Definitions must originate from the bundle that registered the Meta Type Provider service. Third party extenders should therefore use the bundle of their extendee. A Meta Type Service must report these Object Class Definitions on the Meta Type Information of the registering bundle, merged with any other information from that bundle.

The Meta Type Service must track these Meta Type Provider services and make their Meta Types available as if they were provided on the Managed Service (Factory) services. The Meta Types must become unavailable when the Meta Type Provider service is unregistered.

105.7 Using the Meta Type Resources

A bundle that wants to provide meta type resources must place these resources in the OSGI-INF/metatype directory. The name of the resource must be a valid JAR path. All resources in that directory must be meta type documents. Fragments can contain additional meta type resources in the same directory and they must be taken into account when the meta type resources are searched. A meta type resource must be encoded in UTF-8.

The MetaType Service must support localization of the

- name
- icon
- description
- label attributes

The localization mechanism must be identical using the same mechanism as described in the Core module layer, section *Localization* on page 64, using the same property resource. However, it is possible to override the property resource in the meta type definition resources with the localization attribute of the MetaData element.

The Meta Type Service must examine the bundle and its fragments to locate all localization resources for the localization base name. From that list, the Meta Type Service derives the list of locales which are available for the meta type information. This list can then be returned by MetaTypeInformation.getLocales method. This list can change at any time because the bundle could be refreshed. Clients should be prepared that this list changes after they received it.

105.7.1 XML Schema of a Meta Type Resource

This section describes the schema of the meta type resource. This schema is not intended to be used during runtime for validating meta type resources. The schema is intended to be used by tools and external management systems.

The XML namespace for meta type documents must be:

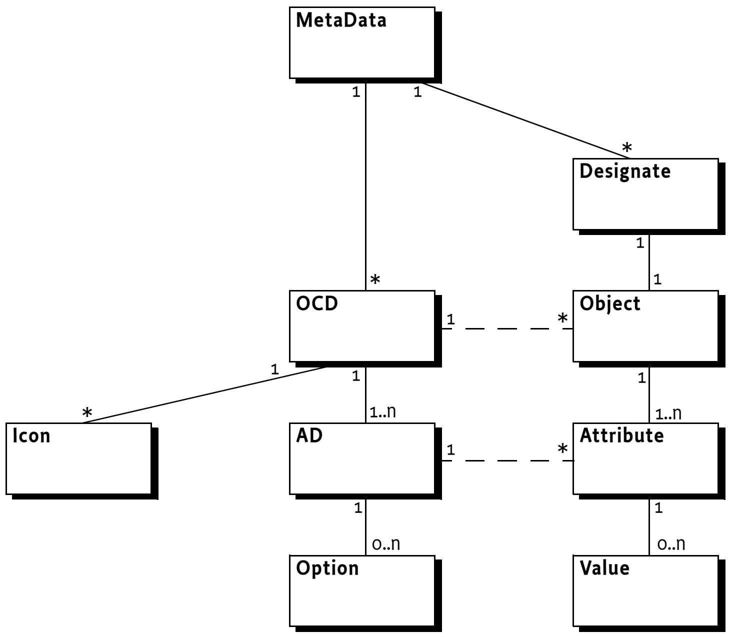
```
http://www.osgi.org/xmlns/metatype/v1.2.0
```

The namespace abbreviation should be metatype. I.e. the following header should be:

```
<metatype:MetaData
   xmlns:metatype=
      "http://www.osgi.org/xmlns/metatype/v1.2.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   >
```

The file can be found in the osgi.jar file that can be downloaded from the www.osgi.org web site.

Figure 105.3 XML Schema Instance Structure (Type name = Element name)



The element structure of the XML file is:

```
MetaData   ::= OCD* Designate*

OCD        ::= AD+  Icon
AD         ::= Option*

Designate  ::= Object
Object     ::= Attribute *

Attribute  ::= Value *
```

The different elements are described in Table 105.1.

Table 105.1 XML Schema for Meta Type resources

	Attribute	Deflt	Type	Method	Description
MetaData					Top Element
	localization		string		Points to the Properties file that can localize this XML. See *Localization* on page 64 of the Core book.
OCD					Object Class Definition
	name	<>	string	getName()	A human readable name that can be localized.
	description			getDescription()	A human readable description of the Object Class Definition that can be localized.
	id	<>		getID()	A unique id, cannot be localized.

Table 105.1 XML Schema for Meta Type resources

	Attribute	Deflt	Type	Method	Description
Designate					An association between one PID and an Object Class Definition. This element *designates* a PID to be of a certain *type*.
	pid	<>	string		The PID that is associated with an OCD. This can be a reference to a factory or singleton configuration object. Either pid or factoryPid must be specified. See *Use of the Designate Element* on page 138.
	factoryPid		string		If the factoryPid attribute is set, this Designate element defines a factory configuration for the given factory, if it is not set or empty, it designates a singleton configuration. Either pid or factoryPid must be specified. See *Use of the Designate Element* on page 138.
	bundle		string		Location of the bundle that implements the PID. This binds the PID to the bundle. I.e. no other bundle using the same PID may use this designation. In a Meta Type resource this field may be set to an wildcard (\u002A, "*") to indicate the bundle where the resource comes from. This is an optional attribute but can be mandatory in certain usage schemes, for example the Autoconf Resource Processor.
	optional	false	boolean		If true, then this Designate element is optional, errors during processing must be ignored.
	merge	false	boolean		If the PID refers to an existing variable, then merge the properties with the existing properties if this attribute is true. Otherwise, replace the properties.
AD					Attribute Definition
	name		string	getName()	A localizable name for the Attribute Definition. description
	description		string	getDescription()	A localizable description for the Attribute Definition.
	id			getID()	The unique ID of the Attribute Definition.

Table 105.1 XML Schema for Meta Type resources

Attribute	Deflt	Type	Method	Description
type		string	getType()	The type of an attribute is an enumeration of the different scalar types. The string is mapped to one of the constants on the AttributeDefinition interface. Valid values, which are defined in the Scalar type, are: String ↔ STRING Long ↔ LONG Double ↔ DOUBLE Float ↔ FLOAT Integer ↔ INTEGER Byte ↔ BYTE Char ↔ CHARACTER Boolean ↔ BOOLEAN Short ↔ SHORT Password ↔ PASSWORD
cardinality	0		getCardinality()	The number of elements an instance can take. Positive numbers describe an array ([]) and negative numbers describe a Vector object.
min		string	validate(String)	A validation value. This value is not directly available from the AttributeDefinition interface. However, the validate(String) method must verify this. The semantics of this field depend on the type of this Attribute Definition.
max		string	validate(String)	A validation value. Similar to the min field.
default		string	getDefaultValue()	The default value. A default is an array of String objects. The XML attribute must contain a comma delimited list. The default value is trimmed and escaped in the same way as described in the validate(String) method. The empty string is a valid value. If the empty string specifies the default for an attribute with cardinality != 0 then it must be seen as an empty Vector or array.
required	true	boolean		Required attribute. The required attribute indicates whether or not the attribute key must appear within the configuration dictionary to be valid.
Option				One option label/value for the options in an AD.
label	<>	string	getOptionLabels()	The label
value	<>	string	getOptionValues()	The value

Table 105.1 XML Schema for Meta Type resources

Attribute	Deflt	Type	Method	Description
Icon				An icon definition.
resource	<>	string	getIcon(int)	The resource is a URL. The base URL is assumed to be the XML file with the definition. I.e. if the XML is a resource in the JAR file, then this URL can reference another resource in that JAR file using a relative URL.
size	<>	string	getIcon(int)	The number of pixels of the icon, maps to the size parameter of the getIcon(int) method.
Object				A definition of an instance.
ocdref	<>	string		A reference to the id attribute of an OCD element. I.e. this attribute defines the OCD type of this object.
Attribute				A value for an attribute of an object.
adref	<>	string		A reference to the id of the AD in the OCD as referenced by the parent Object.
content		string		The content of the attributes. If this is an array, the content must be separated by commas (',' \u002C). Commas must be escaped as described at the default attribute of the AD element. See default on page 137.
Value				Holds a single value. This element can be repeated multiple times under an Attribute

105.7.2 Use of the Designate Element

For the MetaType Service, the Designate definition is used to declare the available PIDs and factory PIDs; the Attribute elements are never used by the MetaType service.

The getPids() method returns an array of PIDs that were specified in the pid attribute of the Object elements. The getFactoryPids() method returns an array of the factoryPid attributes. For factories, the related pid attribute is ignored because all instances of a factory must share the same meta type.

The following example shows a metatype reference to a singleton configuration and a factory configuration.

```
<Designate pid="com.acme.designate.1">
   <Object ocdref="com.acme.designate"./>
</Designate>
<Designate factoryPid="com.acme.designate.factory"
   bundle="*">
   <Object ocdref="com.acme.designate"/>
</Designate>
```

Other schemes can embed the Object element in the Designate element to define actual instances for the Configuration Admin service. In that case the pid attribute must be used together with the factoryPid attribute. However, in that case an aliasing model is required because the Configuration Admin service does not allow the creator to choose the Configuration object's PID.

105.7.3 Example Metadata File

This example defines a meta type file for a Person record, based on ISO attribute types. The ids that are used are derived from ISO attributes.

```
<xml version="1.0" encoding="UTF-8">
<MetaData
   xmlns=
      "http://www.osgi.org/xmlns/metatype/v1.2.0"
    localization="person">
  <OCD name="%person" id="2.5.6.6"
      description="%Person Record">
    <AD name="%sex" id="2.5.4.12" type="Integer">
       <Option label="%male" value="1"/>
       <Option label="%female" value="0"/>
    </AD>
    <AD name="%sn" id="2.5.4.4" type="String"/>
    <AD name="%cn" id="2.5.4.3" type="String"/>
    <AD name="%seeAlso" id="2.5.4.34" type="String"
        cardinality="8" default="http://www.google.com,
             http://www.yahoo.com"/>
    <AD name="%telNumber" id="2.5.4.20" type="String"/>
 </OCD>

 <Designate pid="com.acme.addressbook">
  <Object ocdref="2.5.6.6"/>
 </Designate>
</MetaData>
```

Translations for this file, as indicated by the localization attribute must be stored in the root directory (e.g. person_du_NL.properties). The default localization base name for the properties is OSGI-INF/l10n/bundle, but can be overridden by the manifest Bundle-Localization header and the localization attribute of the Meta Data element. The property files have the base name of person. The Dutch, French and English translations could look like:

```
person_du_NL.properties:
person=Persoon
person\ record=Persoons beschrijving
cn=Naam
sn=Voornaam
seeAlso=Zie ook
telNumber=Tel. Nummer
sex=Geslacht
male=Mannelijk
female=Vrouwelijk

person_fr.properties
person=Personne
person\ record=Description de la personne
cn=Nom
sn=Surnom
seeAlso=Reference
telNumber=Tel.
sex=Sexe
male=Homme
female=Femme
```

```
person_en_US.properties
person=Person
person\ record=Person Record
cn=Name
sn=Sur Name
seeAlso=See Also
telNumber=Tel.
sex=Sex
male=Male
female=Female
```

105.8 Object

The OCD element can be used to describe the possible contents of a Dictionary object. In this case, the attribute name is the key. The Object element can be used to assign a value to a Dictionary object.

For example:

```
<Designate pid="com.acme.b">
   <Object ocdref="b">
      <Attribute adref="foo" content="Zaphod Beeblebrox"/>
      <Attribute adref="bar">
         <Value>1</Value>
         <Value>2</Value>
         <Value>3</Value>
         <Value>4</Value>
         <Value>5</Value>
      </Attribute>
   </Object>
  </Designate>
```

105.9 XML Schema

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:metatype="http://www.osgi.org/xmlns/metatype/v1.2.0"
   targetNamespace="http://www.osgi.org/xmlns/metatype/v1.2.0"
   version="1.2.0">

   <element name="MetaData" type="metatype:Tmetadata" />

   <complexType name="Tmetadata">
      <sequence>
         <element name="OCD" type="metatype:Tocd" minOccurs="0"
            maxOccurs="unbounded" />
         <element name="Designate" type="metatype:Tdesignate"
            minOccurs="0" maxOccurs="unbounded" />
         <!-- It is non-deterministic, per W3C XML Schema 1.0: http://www.w3.org/TR/xmlschema-1/#cos-nonambig
            to use namespace="##any" below. -->
         <any namespace="##other" processContents="lax" minOccurs="0"
            maxOccurs="unbounded" />
      </sequence>
      <attribute name="localization" type="string" use="optional" />
      <anyAttribute />
   </complexType>

   <complexType name="Tocd">
      <sequence>
         <element name="AD" type="metatype:Tad" minOccurs="1"
            maxOccurs="unbounded" />
         <element name="Icon" type="metatype:Ticon" minOccurs="0"
            maxOccurs="unbounded" />
         <!-- It is non-deterministic, per W3C XML Schema 1.0: http://www.w3.org/TR/xmlschema-1/#cos-nonambig
```

```
            to use namespace="##any" below. -->
        <any namespace="##other" processContents="lax" minOccurs="0"
            maxOccurs="unbounded" />
    </sequence>
    <attribute name="name" type="string" use="required" />
    <attribute name="description" type="string" use="optional" />
    <attribute name="id" type="string" use="required" />
    <anyAttribute />
</complexType>

<complexType name="Tad">
    <sequence>
        <element name="Option" type="metatype:Toption" minOccurs="0"
            maxOccurs="unbounded" />
        <!-- It is non-deterministic, per W3C XML Schema 1.0: http://www.w3.org/TR/xmlschema-1/#cos-nonambig
            to use namespace="##any" below. -->
        <any namespace="##other" processContents="lax" minOccurs="0"
            maxOccurs="unbounded" />
    </sequence>
    <attribute name="name" type="string" use="optional" />
    <attribute name="description" type="string" use="optional" />
    <attribute name="id" type="string" use="required" />
    <attribute name="type" type="metatype:Tscalar" use="required" />
    <attribute name="cardinality" type="int" use="optional"
        default="0" />
    <attribute name="min" type="string" use="optional" />
    <attribute name="max" type="string" use="optional" />
    <attribute name="default" type="string" use="optional" />
    <attribute name="required" type="boolean" use="optional"
        default="true" />
    <anyAttribute />
</complexType>

<complexType name="Tobject">
    <sequence>
        <element name="Attribute" type="metatype:Tattribute"
            minOccurs="0" maxOccurs="unbounded" />
        <!-- It is non-deterministic, per W3C XML Schema 1.0: http://www.w3.org/TR/xmlschema-1/#cos-nonambig
            to use namespace="##any" below. -->
        <any namespace="##other" processContents="lax" minOccurs="0"
            maxOccurs="unbounded" />
    </sequence>
    <attribute name="ocdref" type="string" use="required" />
    <anyAttribute />
</complexType>

<complexType name="Tattribute">
    <sequence>
        <element name="Value" type="string" minOccurs="0"
            maxOccurs="unbounded" />
        <!-- It is non-deterministic, per W3C XML Schema 1.0: http://www.w3.org/TR/xmlschema-1/#cos-nonambig
            to use namespace="##any" below. -->
        <any namespace="##other" processContents="lax" minOccurs="0"
            maxOccurs="unbounded" />
    </sequence>
    <attribute name="adref" type="string" use="required" />
    <attribute name="content" type="string" use="optional" />
    <anyAttribute />
</complexType>

<complexType name="Tdesignate">
    <sequence>
        <element name="Object" type="metatype:Tobject" minOccurs="1"
            maxOccurs="1" />
        <any namespace="##any" processContents="lax" minOccurs="0"
            maxOccurs="unbounded" />
    </sequence>
    <attribute name="pid" type="string" use="optional" />
    <attribute name="factoryPid" type="string" use="optional" />
    <attribute name="bundle" type="string" use="optional" />
    <attribute name="optional" type="boolean" default="false"
        use="optional" />
    <attribute name="merge" type="boolean" default="false"
        use="optional" />
    <anyAttribute />
    </complexType>

    <simpleType name="Tscalar">
        <restriction base="string">
            <enumeration value="String" />
            <enumeration value="Long" />
            <enumeration value="Double" />
            <enumeration value="Float" />
            <enumeration value="Integer" />
            <enumeration value="Byte" />
            <enumeration value="Char" />
            <enumeration value="Boolean" />
            <enumeration value="Short" />
            <enumeration value="Password" />
        </restriction>
    </simpleType>

    <complexType name="Toption">
        <sequence>
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="label" type="string" use="required" />
        <attribute name="value" type="string" use="required" />
        <anyAttribute />
    </complexType>

    <complexType name="Ticon">
        <sequence>
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="resource" type="string" use="required" />
        <attribute name="size" type="positiveInteger" use="required" />
        <anyAttribute />
    </complexType>

    <attribute name="must-understand" type="boolean">
        <annotation>
            <documentation xml:lang="en">
                This attribute should be used by extensions to documents
                to require that the document consumer understand the
                extension.
            </documentation>
        </annotation>
    </attribute>
</schema>
```

105.10 Limitations

The OSGi MetaType specification is intended to be used for simple applications. It does not, therefore, support recursive data types, mixed types in arrays/vectors, or nested arrays/vectors.

105.11 Related Standards

One of the primary goals of this specification is to make metatype information available at run-time with minimal overhead. Many related standards are applicable to metatypes; except for Java beans, however, all other metatype standards are based on document formats (e.g. XML). In the OSGi Service Platform, document format standards are deemed unsuitable due to the overhead required in the execution environment (they require a parser during run-time).

Another consideration is the applicability of these standards. Most of these standards were developed for management systems on platforms where resources are not necessarily a concern. In this case, a metatype standard is normally used to describe the data structures needed to control some other computer via a network. This other computer, however, does not require the metatype information as it is *implementing* this information.

In some traditional cases, a management system uses the metatype information to control objects in an OSGi Service Platform. Therefore, the concepts and the syntax of the metatype information must be mappable to these popular standards. Clearly, then, these standards must be able to describe objects in an OSGi Service Platform. This ability is usually not a problem, because the metatype languages used by current management systems are very powerful.

105.12 Changes

- Added the possibility to dynamically add meta types with the Meta Type Provider service, see *Meta Type Provider Service* on page 133.
- Added a PASSWORD type.
- Added METATYPE_PID and METATYPE_FACTORY_PID constants to the MetaTypProvider class.

105.13 Security Considerations

Special security issues are not applicable for this specification.

105.14 org.osgi.service.metatype

Metatype Package Version 1.2.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.metatype; version="[1.2,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.metatype; version="[1.2,1.3)"
```

105.14.1 Summary

- AttributeDefinition – An interface to describe an attribute.
- MetaTypeInformation – A MetaType Information object is created by the MetaTypeService to return meta type information for a specific bundle.
- MetaTypeProvider – Provides access to metatypes.
- MetaTypeService – The MetaType Service can be used to obtain meta type information for a bundle.
- ObjectClassDefinition – Description for the data type information of an objectclass.

105.14.2 Permissions

105.14.3 public interface AttributeDefinition

An interface to describe an attribute.

An AttributeDefinition object defines a description of the data type of a property/attribute.

Concurrency Thread-safe

105.14.3.1 public static final int BIGDECIMAL = 10

The BIGDECIMAL (10) type. Attributes of this type should be stored as BigDecimal, Vector with BigDecimal or BigDecimal[] objects depending on getCardinality().

Deprecated As of 1.1.

105.14.3.2 public static final int BIGINTEGER = 9

The BIGINTEGER (9) type. Attributes of this type should be stored as BigInteger, Vector with BigInteger or BigInteger[] objects, depending on the getCardinality() value.

Deprecated As of 1.1.

105.14.3.3 public static final int BOOLEAN = 11

The BOOLEAN (11) type. Attributes of this type should be stored as Boolean, Vector with Boolean or boolean[] objects depending on getCardinality().

105.14.3.4 public static final int BYTE = 6

The BYTE (6) type. Attributes of this type should be stored as Byte, Vector with Byte or byte[] objects, depending on the getCardinality() value.

105.14.3.5 public static final int CHARACTER = 5

The CHARACTER (5) type. Attributes of this type should be stored as Character, Vector with Character or char[] objects, depending on the getCardinality() value.

105.14.3.6 public static final int DOUBLE = 7

The DOUBLE (7) type. Attributes of this type should be stored as Double, Vector with Double or double[] objects, depending on the getCardinality() value.

105.14.3.7 public static final int FLOAT = 8

The FLOAT (8) type. Attributes of this type should be stored as Float, Vector with Float or float[] objects, depending on the getCardinality() value.

105.14.3.8 public static final int INTEGER = 3

The INTEGER (3) type. Attributes of this type should be stored as Integer, Vector with Integer or int[] objects, depending on the getCardinality() value.

105.14.3.9 public static final int LONG = 2

The LONG (2) type. Attributes of this type should be stored as Long, Vector with Long or long[] objects, depending on the getCardinality() value.

105.14.3.10 public static final int PASSWORD = 12

The PASSWORD (12) type. Attributes of this type must be stored as String, Vector with String or String[] objects depending on {link getCardinality()}. A PASSWORD must be treated as a string but the type can be used to disguise the information when displayed to a user to prevent others from seeing it.

Since 1.2

105.14.3.11 public static final int SHORT = 4

The SHORT (4) type. Attributes of this type should be stored as Short, Vector with Short or short[] objects, depending on the getCardinality() value.

105.14.3.12 public static final int STRING = 1

The STRING (1) type.

Attributes of this type should be stored as String, Vector with String or String[] objects, depending on the getCardinality() value.

105.14.3.13 public int getCardinality ( )

□ Return the cardinality of this attribute. The OSGi environment handles multi valued attributes in arrays ([]) or in Vector objects. The return value is defined as follows:

```
x = Integer.MIN_VALUE    no limit, but use Vector
x < 0                    -x = max occurrences, store in Vector
x > 0                     x = max occurrences, store in array []
x = Integer.MAX_VALUE    no limit, but use array []
x = 0                     1 occurrence required
```

Returns The cardinality of this attribute.

105.14.3.14 public String[] getDefaultValue ( )

□ Return a default for this attribute. The object must be of the appropriate type as defined by the cardinality and getType(). The return type is a list of String objects that can be converted to the appropriate type. The cardinality of the return array must follow the absolute cardinality of this type. E.g. if the cardinality = 0, the array must contain 1 element. If the cardinality is 1, it must contain 0 or 1 elements. If it is -5, it must contain from 0 to max 5 elements. Note that the special case of a 0 cardinality, meaning a single value, does not allow arrays or vectors of 0 elements.

Returns Return a default value or null if no default exists.

105.14.3.15 public String getDescription ( )

□ Return a description of this attribute. The description may be localized and must describe the semantics of this type and any constraints.

Returns The localized description of the definition.

105.14.3.16 public String getID ( )

□ Unique identity for this attribute. Attributes share a global namespace in the registry. E.g. an attribute cn or commonName must always be a String and the semantics are always a name of some object. They share this aspect with LDAP/X.500 attributes. In these standards the OSI Object Identifier (OID) is used to uniquely identify an attribute. If such an OID exists, (which can be requested at several standard organisations and many companies already have a node in the tree) it can be returned here. Otherwise, a unique id should be returned which can be a Java class name (reverse domain name) or generated with a GUID algorithm. Note that all LDAP defined attributes already have an OID. It is strongly advised to define the attributes from existing LDAP schemes which will give the OID. Many such schemes exist ranging from postal addresses to DHCP parameters.

Returns The id or oid

105.14.3.17 public String getName ( )

□ Get the name of the attribute. This name may be localized.

Returns The localized name of the definition.

105.14.3.18 public String[] getOptionLabels ( )

□ Return a list of labels of option values.

The purpose of this method is to allow menus with localized labels. It is associated with getOptionValues. The labels returned here are ordered in the same way as the values in that method.

If the function returns null, there are no option labels available.

This list must be in the same sequence as the getOptionValues() method. I.e. for each index i in getOptionLabels, i in getOptionValues() should be the associated value.

For example, if an attribute can have the value male, female, unknown, this list can return (for dutch) new String[] { “Man”, “Vrouw”, “Onbekend” }.

Returns A list values

105.14.3.19 public String[] getOptionValues ( )

□ Return a list of option values that this attribute can take.

If the function returns null, there are no option values available.

Each value must be acceptable to validate() (return "") and must be a String object that can be converted to the data type defined by getType() for this attribute.

This list must be in the same sequence as getOptionLabels(). I.e. for each index i in getOptionValues, i in getOptionLabels() should be the label.

For example, if an attribute can have the value male, female, unknown, this list can return new String[] { "male", "female", "unknown" }.

Returns A list values

105.14.3.20 public int getType ( )

□ Return the type for this attribute.

Defined in the following constants which map to the appropriate Java type. STRING,LONG,INTEGER, CHAR, BYTE,DOUBLE,FLOAT, BOOLEAN.

Returns The type for this attribute.

105.14.3.21 public String validate (String value)

value The value before turning it into the basic data type. If the cardinality indicates a multi valued attribute then the given string must be escap

□ Validate an attribute in String form. An attribute might be further constrained in value. This method will attempt to validate the attribute according to these constraints. It can return three different values:

```
null          No validation present
""            No problems detected
"..."         A localized description of why the value is wrong
```

If the cardinality of this attribute is multi-valued then this string must be interpreted as a comma delimited string. The complete value must be trimmed from white space as well as spaces around commas. Commas (',' ,) and spaces (' ') and back-slashes ('\' \) can be escaped with another back-slash. Escaped spaces must not be trimmed. For example:

```
value="  a\,b,b\,c,\ c\\,d   " => [ "a,b", "b,c", " c\", "d" ]
```

Returns null, "", or another string

105.14.4 public interface MetaTypeInformation extends MetaTypeProvider

A MetaType Information object is created by the MetaTypeService to return meta type information for a specific bundle.

Since 1.1

Concurrency Thread-safe

No Implement Consumers of this API must not implement this interface

105.14.4.1 public Bundle getBundle ( )

□ Return the bundle for which this object provides meta type information.

Returns Bundle for which this object provides meta type information.

105.14.4.2 **public String[] getFactoryPids ( )**

☐ Return the Factory PIDs (for ManagedServiceFactories) for which ObjectClassDefinition information is available.

Returns Array of Factory PIDs.

105.14.4.3 **public String[] getPids ( )**

☐ Return the PIDs (for ManagedServices) for which ObjectClassDefinition information is available.

Returns Array of PIDs.

105.14.5 public interface MetaTypeProvider

Provides access to metatypes. This interface can be implemented on a Managed Service or Managed Service Factory as well as registered as a service. When registered as a service, it must be registered with a METATYPE_FACTORY_PID or METATYPE_PID service property (or both). Any PID mentioned in either of these factories must be a valid argument to the getObjectClassDefinition(String, String) method.

Concurrency Thread-safe

105.14.5.1 **public static final String METATYPE_FACTORY_PID = "metatype.factory.pid"**

Service property to signal that this service has ObjectClassDefinition objects for the given factory PIDs. The type of this service property is String+.

Since 1.2

105.14.5.2 **public static final String METATYPE_PID = "metatype.pid"**

Service property to signal that this service has ObjectClassDefinition objects for the given PIDs. The type of this service property is String+.

Since 1.2

105.14.5.3 **public String[] getLocales ( )**

☐ Return a list of available locales. The results must be names that consists of language [_ country [_ variation]] as is customary in the Locale class.

Returns An array of locale strings or null if there is no locale specific localization can be found.

105.14.5.4 **public ObjectClassDefinition getObjectClassDefinition (String id , String locale)**

id The ID of the requested object class. This can be a pid or factory pid returned by getPids or getFactoryPids.

locale The locale of the definition or null for default locale.

☐ Returns an object class definition for the specified id localized to the specified locale.

The locale parameter must be a name that consists of language["_" country["_" variation]] as is customary in the Locale class. This Locale class is not used because certain profiles do not contain it.

Returns A ObjectClassDefinition object.

Throws IllegalArgumentException – If the id or locale arguments are not valid

105.14.6 public interface MetaTypeService

The MetaType Service can be used to obtain meta type information for a bundle. The MetaType Service will examine the specified bundle for meta type documents to create the returned MetaTypeInformation object.

If the specified bundle does not contain any meta type documents, then a MetaTypeInformation object will be returned that wrappers any ManagedService or ManagedServiceFactory services registered by the specified bundle that implement MetaTypeProvider. Thus the MetaType Service can be used to retrieve meta type information for bundles which contain a meta type documents or which provide their own MetaTypeProvider objects.

Since 1.1

Concurrency Thread-safe

No Implement Consumers of this API must not implement this interface

105.14.6.1 public static final String METATYPE_DOCUMENTS_LOCATION = "OSGI-INF/metatype"

Location of meta type documents. The MetaType Service will process each entry in the meta type documents directory.

105.14.6.2 public MetaTypeInformation getMetaTypeInformation (Bundle bundle)

bundle The bundle for which meta type information is requested.

□ Return the MetaType information for the specified bundle.

Returns A MetaTypeInformation object for the specified bundle.

105.14.7 public interface ObjectClassDefinition

Description for the data type information of an objectclass.

Concurrency Thread-safe

105.14.7.1 public static final int ALL = -1

Argument for getAttributeDefinitions(int).

ALL indicates that all the definitions are returned. The value is -1.

105.14.7.2 public static final int OPTIONAL = 2

Argument for getAttributeDefinitions(int).

OPTIONAL indicates that only the optional definitions are returned. The value is 2.

105.14.7.3 public static final int REQUIRED = 1

Argument for getAttributeDefinitions(int).

REQUIRED indicates that only the required definitions are returned. The value is 1.

105.14.7.4 public AttributeDefinition[] getAttributeDefinitions (int filter)

filter ALL,REQUIRED,OPTIONAL

□ Return the attribute definitions for this object class.

Return a set of attributes. The filter parameter can distinguish between ALL,REQUIRED or the OPTIONAL attributes.

Returns An array of attribute definitions or null if no attributes are selected

105.14.7.5 public String getDescription ( )

□ Return a description of this object class. The description may be localized.

Returns The description of this object class.

105.14.7.6 public InputStream getIcon (int size) throws IOException

size Requested size of an icon, e.g. a 16x16 pixels icon then size = 16

□ Return an `InputStream` object that can be used to create an icon from.

Indicate the size and return an `InputStream` object containing an icon. The returned icon maybe larger or smaller than the indicated size.

The icon may depend on the localization.

Returns An InputStream representing an icon or `null`

Throws `IOException` – If the `InputStream` cannot be returned.

105.14.7.7 public String getID ( )

□ Return the id of this object class.

`ObjectDefintion` objects share a global namespace in the registry. They share this aspect with LDAP/X.500 attributes. In these standards the OSI Object Identifier (OID) is used to uniquely identify object classes. If such an OID exists, (which can be requested at several standard organisations and many companies already have a node in the tree) it can be returned here. Otherwise, a unique id should be returned which can be a java class name (reverse domain name) or generated with a GUID algorithm. Note that all LDAP defined object classes already have an OID associated. It is strongly advised to define the object classes from existing LDAP schemes which will give the OID for free. Many such schemes exist ranging from postal addresses to DHCP parameters.

Returns The id of this object class.

105.14.7.8 public String getName ( )

□ Return the name of this object class. The name may be localized.

Returns The name of this object class.

105.15 References

[1] *LDAP.*
http://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol

[2] *Understanding and Deploying LDAP Directory services*
Timothy Howes et. al. ISBN 1-57870-070-1, MacMillan Technical publishing.

107 User Admin Service Specification

Version 1.1

107.1 Introduction

OSGi Service Platforms are often used in places where end users or devices initiate actions. These kinds of actions inevitably create a need for authenticating the initiator. Authenticating can be done in many different ways, including with passwords, one-time token cards, bio-metrics, and certificates.

Once the initiator is authenticated, it is necessary to verify that this principal is authorized to perform the requested action. This authorization can only be decided by the operator of the OSGi environment, and thus requires administration.

The User Admin service provides this type of functionality. Bundles can use the User Admin service to authenticate an initiator and represent this authentication as an Authorization object. Bundles that execute actions on behalf of this user can use the Authorization object to verify if that user is authorized.

The User Admin service provides authorization based on who runs the code, instead of using the Java code-based permission model. See [1] *The Java Security Architecture for JDK 1.2*. It performs a role similar to [2] *Java Authentication and Authorization Service*.

107.1.1 Essentials

- *Authentication* – A large number of authentication schemes already exist, and more will be developed. The User Admin service must be flexible enough to adapt to the many different authentication schemes that can be run on a computer system.
- *Authorization* – All bundles should use the User Admin service to authenticate users and to find out if those users are authorized. It is therefore paramount that a bundle can find out authorization information with little effort.
- *Security* – Detailed security, based on the Framework security model, is needed to provide safe access to the User Admin service. It should allow limited access to the credentials and other properties.
- *Extensibility* – Other bundles should be able to build on the User Admin service. It should be possible to examine the information from this service and get real-time notifications of changes.
- *Properties* – The User Admin service must maintain a persistent database of users. It must be possible to use this database to hold more information about this user.
- *Administration* – Administering authorizations for each possible action and initiator is time-consuming and error-prone. It is therefore necessary to have mechanisms to group end users and make it simple to assign authorizations to all members of a group at one time.

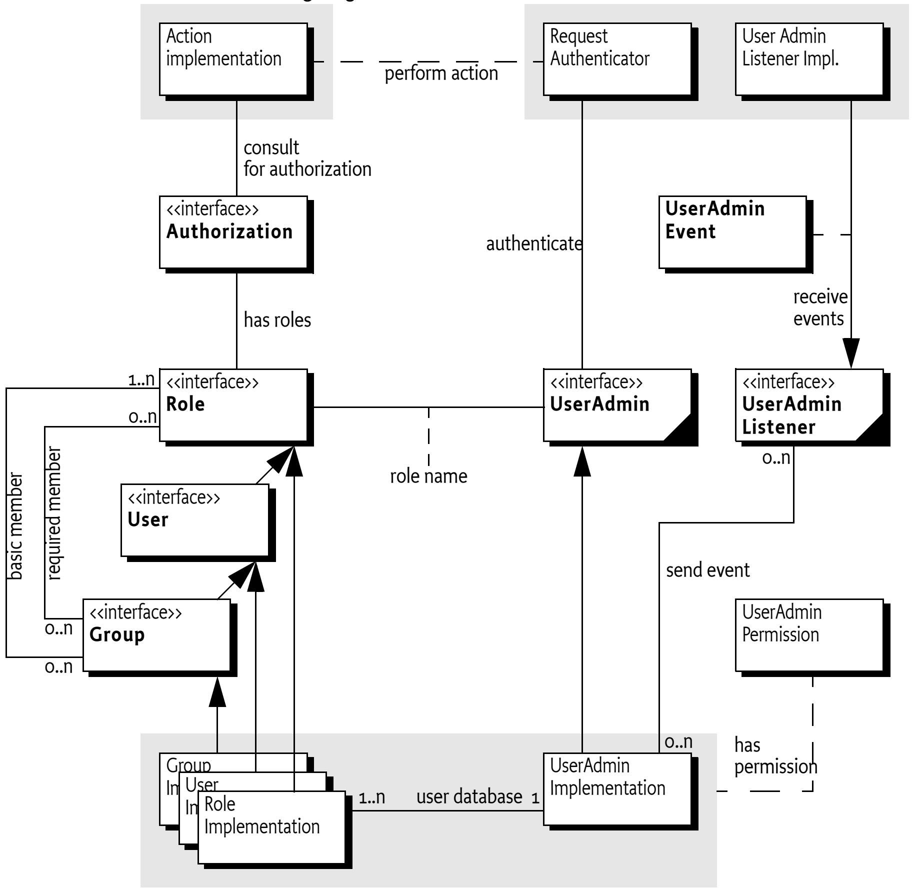
107.1.2 Entities

This Specification defines the following User Admin service entities:

- *UserAdmin* – This interface manages a database of named roles which can be used for authorization and authentication purposes.
- *Role* – This interface exposes the characteristics shared by all roles: a name, a type, and a set of properties.

- *User* – This interface (which extends Role) is used to represent any entity which may have credentials associated with it. These credentials can be used to authenticate an initiator.
- *Group* – This interface (which extends User) is used to contain an aggregation of named Role objects (Group or User objects).
- *Authorization* – This interface encapsulates an authorization context on which bundles can base authorization decisions.
- *UserAdminEvent* – This class is used to represent a role change event.
- *UserAdminListener* – This interface provides a listener for events of type UserAdminEvent that can be registered as a service.
- *UserAdminPermission* – This permission is needed to configure and access the roles managed by a User Admin service.
- *Role.USER_ANYONE* – This is a special User object that represents *any* user, it implies all other User objects. It is also used when a Group is used with only basic members. The Role.USER_ANYONE is then the only required member.

Figure 107.1 *User Admin Service*, org.osgi.service.useradmin



107.1.3 Operation

An Operator uses the User Admin service to define OSGi Service Platform users and configure them with properties, credentials, and *roles*.

A Role object represents the initiator of a request (human or otherwise). This specification defines two types of roles:

- *User* – A User object can be configured with credentials, such as a password, and properties, such as address, telephone number, and so on.
- *Group* – A Group object is an aggregation of *basic* and *required* roles. Basic and required roles are used in the authorization phase.

An OSGi Service Platform can have several entry points, each of which will be responsible for authenticating incoming requests. An example of an entry point is the Http Service, which delegates authentication of incoming requests to the handleSecurity method of the HttpContext object that was specified when the target servlet or resource of the request was registered.

The OSGi Service Platform entry points should use the information in the User Admin service to authenticate incoming requests, such as a password stored in the private credentials or the use of a certificate.

A bundle can determine if a request for an action is authorized by looking for a Role object that has the name of the requested action.

The bundle may execute the action if the Role object representing the initiator *implies* the Role object representing the requested action.

For example, an initiator Role object *X* implies an action Group object *A* if:

- *X* implies at least one of *A*'s basic members, and
- *X* implies all of *A*'s required members.

An initiator Role object *X* implies an action User object *A* if:

- *A* and *X* are equal.

The Authorization class handles this non-trivial logic. The User Admin service can capture the privileges of an authenticated User object into an Authorization object. The Authorization.hasRole method checks if the authenticate User object has (or implies) a specified action Role object.

For example, in the case of the Http Service, the HttpContext object can authenticate the initiator and place an Authorization object in the request header. The servlet calls the hasRole method on this Authorization object to verify that the initiator has the authority to perform a certain action. See *Authentication* on page 54.

107.2 Authentication

The authentication phase determines if the initiator is actually the one it says it is. Mechanisms to authenticate always need some information related to the user or the OSGi Service Platform to authenticate an external user. This information can consist of the following:

- A secret known only to the initiator.
- Knowledge about cards that can generate a unique token.
- Public information like certificates of trusted signers.
- Information about the user that can be measured in a trusted way.
- Other specific information.

107.2.1 Repository

The User Admin service offers a repository of Role objects. Each Role object has a unique name and a set of properties that are readable by anyone, and are changeable when the changer has the UserAdminPermission. Additionally, User objects, a sub-interface of Role, also have a set of private protected properties called credentials. Credentials are an extra set of properties that are used to authenticate users and that are protected by UserAdminPermission.

Properties are accessed with the Role.getProperties() method and credentials with the User.getCredentials()method. Both methods return a Dictionary object containing key/value pairs. The keys are String objects and the values of the Dictionary object are limited to String or byte[ ] objects.

This specification does not define any standard keys for the properties or credentials. The keys depend on the implementation of the authentication mechanism and are not formally defined by OSGi specifications.

The repository can be searched for objects that have a unique property (key/value pair) with the method UserAdmin.getUser(String,String). This makes it easy to find a specific user related to a specific authentication mechanism. For example, a secure card mechanism that generates unique tokens could have a serial number identifying the user. The owner of the card could be found with the method

```
User owner = useradmin.getUser(
   "secure-card-serial", "132456712-1212" );
```

If multiple User objects have the same property (key *and* value), a null is returned.

There is a convenience method to verify that a user has a credential without actually getting the credential. This is the User.hasCredential(String,Object) method.

Access to credentials is protected on a name basis by UserAdminPermission. Because properties can be read by anyone with access to a User object, UserAdminPermission only protects change access to properties.

107.2.2 Basic Authentication

The following example shows a very simple authentication algorithm based on passwords.

The vendor of the authentication bundle uses the property "com.acme.basic-id" to contain the name of a user as it logs in. This property is used to locate the User object in the repository. Next, the credential "com.acme.password" contains the password and is compared to the entered password. If the password is correct, the User object is returned. In all other cases a SecurityException is thrown.

```
public User authenticate(
      UserAdmin ua, String name, String pwd )
   throws SecurityException {
   User user = ua.getUser("com.acme.basicid",
      username);
   if (user == null)
      throw new SecurityException( "No such user" );

   if (!user.hasCredential("com.acme.password", pwd))
      throw new SecurityException(
         "Invalid password" );
   return user;
}
```

107.2.3 Certificates

Authentication based on certificates does not require a shared secret. Instead, a certificate contains a name, a public key, and the signature of one or more signers.

The name in the certificate can be used to locate a User object in the repository. Locating a User object, however, only identifies the initiator and does not authenticate it.

1. The first step to authenticate the initiator is to verify that it has the private key of the certificate.
2. Next, the User Admin service must verify that it has a User object with the right property, for example "com.acme.certificate"="Fudd".
3. The next step is to see if the certificate is signed by a trusted source. The bundle could use a central list of trusted signers and only accept certificates signed by those sources. Alternatively, it could require that the certificate itself is already stored in the repository under a unique key as a byte[] in the credentials.
4. In any case, once the certificate is verified, the associated User object is authenticated.

107.3 Authorization

The User Admin service authorization architecture is a *role-based model*. In this model, every action that can be performed by a bundle is associated with a *role*. Such a role is a Group object (called group from now on) from the User Admin service repository. For example, if a servlet could be used to activate the alarm system, there should be a group named AlarmSystemActivation.

The operator can administrate authorizations by populating the group with User objects (users) and other groups. Groups are used to minimize the amount of administration required. For example, it is easier to create one Administrators group and add administrative roles to it rather than individually administer all users for each role. Such a group requires only one action to remove or add a user as an administrator.

The authorization decision can now be made in two fundamentally different ways:

An initiator could be allowed to carry out an action (represented by a Group object) if it implied any of the Group object's members. For example, the AlarmSystemActivation Group object contains an Administrators and a Family Group object:

```
Administrators            = { Elmer, Pepe, Bugs }
Family                    = { Elmer, Pepe, Daffy }

AlarmSystemActivation     = { Administrators, Family }
```

Any of the four members Elmer, Pepe, Daffy, or Bugs can activate the alarm system.

Alternatively, an initiator could be allowed to perform an action (represented by a Group object) if it implied *all* the Group object's members. In this case, using the same AlarmSystemActivation group, only Elmer and Pepe would be authorized to activate the alarm system, since Daffy and Bugs are *not* members of *both* the Administrators and Family Group objects.

The User Admin service supports a combination of both strategies by defining both a set of *basic members* (any) and a set of *required members* (all).

```
Administrators  = { Elmer, Pepe, Bugs }
Family          = { Elmer, Pepe, Daffy }

AlarmSystemActivation
  required      = { Administrators }
  basic         = { Family }
```

The difference is made when Role objects are added to the Group object. To add a basic member, use the Group.addMember(Role) method. To add a required member, use the Group.addRequiredMember(Role) method.

Basic members define the set of members that can get access and required members reduce this set by requiring the initiator to *imply* each required member.

A User object implies a Group object if it implies the following:

- *All* of the Group's required members, and
- At *least* one of the Group's basic members

A User object always implies itself.

If only required members are used to qualify the implication, then the standard user Role.USER_ANYONE can be obtained from the User Admin service and added to the Group object. This Role object is implied by anybody and therefore does not affect the required members.

107.3.1 The Authorization Object

The complexity of authorization is hidden in an Authorization class. Normally, the authenticator should retrieve an Authorization object from the User Admin service by passing the authenticated User object as an argument. This Authorization object is then passed to the bundle that performs the action. This bundle checks the authorization with the Authorization.hasRole(String) method. The performing bundle must pass the name of the action as an argument. The Authorization object checks whether the authenticated user implies the Role object, specifically a Group object, with the given name. This is shown in the following example.

```
public void activateAlarm(Authorization auth) {
   if ( auth.hasRole( "AlarmSystemActivation" ) ) {
      // activate the alarm
      ...
   }
   else throw new SecurityException(
      "Not authorized to activate alarm" );
}
```

107.3.2 Authorization Example

This section demonstrates a possible use of the User Admin service. The service has a flexible model and many other schemes are possible.

Assume an Operator installs an OSGi Service Platform. Bundles in this environment have defined the following action groups:

```
AlarmSystemControl
InternetAccess
TemperatureControl
PhotoAlbumEdit
PhotoAlbumView
PortForwarding
```

Installing and uninstalling bundles could potentially extend this set. Therefore, the Operator also defines a number of groups that can be used to contain the different types of system users.

```
Administrators
Buddies
Children
Adults
Residents
```

In a particular instance, the Operator installs it in a household with the following residents and buddies:

```
Residents:          Elmer, Fudd, Marvin, Pepe
Buddies:            Daffy, Foghorn
```

First, the residents and buddies are assigned to the system user groups. Second, the user groups need to be assigned to the action groups.

The following tables show how the groups could be assigned.

Table 107.1 Example Groups with Basic and Required Members

Groups	Elmer	Fudd	Marvin	Pepe	Daffy	Foghorn
Residents	Basic	Basic	Basic	Basic	-	-
Buddies	-	-	-	-	Basic	Basic
Children	-	-	Basic	Basic	-	-
Adults	Basic	Basic	-	-	-	-
Administrators	Basic	-	-	-	-	-

Table 107.2 Example Action Groups with their Basic and Required Members

Groups	Residents	Buddies	Children	Adults	Admin
AlarmSystemControl	Basic	-	-	-	Required
InternetAccess	Basic	-	-	Required	-
TemperatureControl	Basic	-	-	Required	-
PhotoAlbumEdit	Basic	-	Basic	Basic	-
PhotoAlbumView	Basic	Basic	-	-	-
PortForwarding	Basic	-	-	-	Required

107.4 Repository Maintenance

The UserAdmin interface is a straightforward API to maintain a repository of User and Group objects. It contains methods to create new Group and User objects with the createRole(String,int) method. The method is prepared so that the same signature can be used to create new types of roles in the future. The interface also contains a method to remove a Role object.

The existing configuration can be obtained with methods that list all Role objects using a filter argument. This filter, which has the same syntax as the Framework filter, must only return the Role objects for which the filter matches the properties.

Several utility methods simplify getting User objects depending on their properties.

107.5 User Admin Events

Changes in the User Admin service can be determined in real time. Each User Admin service implementation must send a UserAdminEvent object to any service in the Framework service registry that is registered under the UserAdminListener interface. This event must be send asynchronously from the cause of the event. The way events must be delivered is the same as described in *Delivering Events* on page 106 of the Core specification.

This procedure is demonstrated in the following code sample.

```
class Listener implements UserAdminListener {
   public void roleChanged( UserAdminEvent event ) {
      ...
   }
}
public class MyActivator
   implements BundleActivator {
   public void start( BundleContext context ) {
      context.registerService(
           UserAdminListener.class.getName(),
           new Listener(), null );
   }
   public void stop( BundleContext context ) {}
}
```

It is not necessary to unregister the listener object when the bundle is stopped because the Framework automatically unregisters it. Once registered, the UserAdminListener object must be notified of all changes to the role repository.

107.5.1 Event Admin and User Admin Change Events

User admin events must be delivered asynchronously to the Event Admin service by the implementation, if present. The topic of a User Admin Event is:

```
org/osgi/service/useradmin/UserAdmin/<event type>
```

The following event types are supported:

```
ROLE_CREATED
ROLE_CHANGED
ROLE_REMOVED
```

All User Admin Events must have the following properties:

- event – (UserAdminEvent) The event that was broadcast by the User Admin service.
- role – (Role) The Role object that was created, modified or removed.
- role.name – (String) The name of the role.
- role.type – (Integer) One of ROLE, USER or GROUP.
- service – (ServiceReference) The Service Reference of the User Admin service.
- service.id – (Long) The User Admin service's ID.
- service.objectClass – (String[]) The User Admin service's object class (which must include org.osgi.service.useradmin.UserAdmin)
- service.pid – (String) The User Admin service's persistent identity

107.6 Security

The User Admin service is related to the security model of the OSGi Service Platform, but is complementary to the [1] *The Java Security Architecture for JDK 1.2*. The final permission of most code should be the intersection of the Java 2 Permissions, which are based on the code that is executing, and the User Admin service authorization, which is based on the user for whom the code runs.

107.6.1 UserAdminPermission

The User Admin service defines the UserAdminPermission class that can be used to restrict bundles in accessing credentials. This permission class has the following actions:

- changeProperty – This permission is required to modify properties. The name of the permission is the prefix of the property name.
- changeCredential – This action permits changing credentials. The name of the permission is the prefix of the name of the credential.
- getCredential – This action permits getting credentials. The name of the permission is the prefix of the credential.

If the name of the permission is "admin", it allows the owner to administer the repository. No action is associated with the permission in that case.

Otherwise, the permission name is used to match the property name. This name may end with a ".*" string to indicate a wildcard. For example, com.acme.*matches com.acme.fudd.elmer and com.acme.bugs.

107.7 Relation to JAAS

At a glance, the Java Authorization and Authentication Service (JAAS) seems to be a very suitable model for user administration. The OSGi organization, however, decided to develop an independent User Admin service because JAAS was not deemed applicable. The reasons for this include dependency on Java SE version 1.3 ("JDK 1.3") and existing mechanisms in the previous OSGi Service Gateway 1.0 specification.

107.7.1 JDK 1.3 Dependencies

The authorization component of JAAS relies on the java.security.DomainCombiner interface, which provides a means to dynamically update the ProtectionDomain objects affiliated with an AccessControlContext object.

This interface was added in JDK 1.3. In the context of JAAS, the SubjectDomainCombiner object, which implements the DomainCombiner interface, is used to update ProtectionDomain objects. The permissions of ProtectionDomain objects depend on where code came from and who signed it, with permissions based on who is running the code.

Leveraging JAAS would have resulted in user-based access control on the OSGi Service Platform being available only with JDK 1.3, which was not deemed acceptable.

107.7.2 Existing OSGi Mechanism

JAAS provides a pluggable authentication architecture, which enables applications and their underlying authentication services to remain independent from each other.

The Http Service already provides a similar feature by allowing servlet and resource registrations to be supported by an HttpContext object, which uses a callback mechanism to perform any required authentication checks before granting access to the servlet or resource. This way, the registering bundle has complete control on a per-servlet and per-resource basis over which authentication protocol to use, how the credentials presented by the remote requestor are to be validated, and who should be granted access to the servlet or resource.

107.7.3 Future Road Map

In the future, the main barrier of 1.3 compatibility will be removed. JAAS could then be implemented in an OSGi environment. At that time, the User Admin service will still be needed and will provide complementary services in the following ways:

- The authorization component relies on group membership information to be stored and managed outside JAAS. JAAS does not manage persistent information, so the User Admin service can be a provider of group information when principals are assigned to a Subject object.
- The authorization component allows for credentials to be collected and verified, but a repository is needed to actually validate the credentials.

In the future, the User Admin service can act as the back-end database to JAAS. The only aspect JAAS will remove from the User Admin service is the need for the Authorization interface.

107.8 org.osgi.service.useradmin

User Admin Package Version 1.1.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.useradmin; version="[1.1,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.useradmin; version="[1.1,1.2)"

107.8.1 Summary

- Authorization – The Authorization interface encapsulates an authorization context on which bundles can base authorization decisions, where appropriate.
- Group – A named grouping of roles (Role objects).
- Role – The base interface for Role objects managed by the User Admin service.
- User – A User role managed by a User Admin service.
- UserAdmin – This interface is used to manage a database of named Role objects, which can be used for authentication and authorization purposes.
- UserAdminEvent – Role change event.
- UserAdminListener – Listener for UserAdminEvents.
- UserAdminPermission – Permission to configure and access the Role objects managed by a User Admin service.

107.8.2 Permissions

107.8.3 public interface Authorization

The Authorization interface encapsulates an authorization context on which bundles can base authorization decisions, where appropriate.

Bundles associate the privilege to access restricted resources or operations with roles. Before granting access to a restricted resource or operation, a bundle will check if the Authorization object passed to it possess the required role, by calling its hasRole method.

Authorization contexts are instantiated by calling the UserAdmin.getAuthorization(User) method.

Trusting Authorization objects

There are no restrictions regarding the creation of Authorization objects. Hence, a service must only accept Authorization objects from bundles that has been authorized to use the service using code based (or Java 2) permissions.

In some cases it is useful to use ServicePermission to do the code based access control. A service basing user access control on Authorization objects passed to it, will then require that a calling bundle has the ServicePermission to get the service in question. This is the most convenient way. The OSGi environment will do the code based permission check when the calling bundle attempts to get the service from the service registry.

Example: A servlet using a service on a user's behalf. The bundle with the servlet must be given the ServicePermission to get the Http Service.

However, in some cases the code based permission checks need to be more fine-grained. A service might allow all bundles to get it, but require certain code based permissions for some of its methods.

Example: A servlet using a service on a user's behalf, where some service functionality is open to anyone, and some is restricted by code based permissions. When a restricted method is called (e.g., one handing over an Authorization object), the service explicitly checks that the calling bundle has permission to make the call.

No Implement Consumers of this API must not implement this interface

107.8.3.1 public String getName ( )

□ Gets the name of the User that this Authorization context was created for.

Returns The name of the User object that this Authorization context was created for, or null if no user was specified when this Authorization context was created.

107.8.3.2 public String[] getRoles ( )

□ Gets the names of all roles implied by this Authorization context.

Returns The names of all roles implied by this Authorization context, or null if no roles are in the context. The predefined role user.anyone will not be included in this list.

107.8.3.3 public boolean hasRole (String name)

name The name of the role to check for.

□ Checks if the role with the specified name is implied by this Authorization context.

Bundles must define globally unique role names that are associated with the privilege of accessing restricted resources or operations. Operators will grant users access to these resources, by creating a Group object for each role and adding User objects to it.

Returns true if this Authorization context implies the specified role, otherwise false.

107.8.4 public interface Group extends User

A named grouping of roles (Role objects).

Whether or not a given Authorization context implies a Group object depends on the members of that Group object.

A Group object can have two kinds of members: *basic* and *required* . A Group object is implied by an Authorization context if all of its required members are implied and at least one of its basic members is implied.

A Group object must contain at least one basic member in order to be implied. In other words, a Group object without any basic member roles is never implied by any Authorization context.

A User object always implies itself.

No loop detection is performed when adding members to Group objects, which means that it is possible to create circular implications. Loop detection is instead done when roles are checked. The semantics is that if a role depends on itself (i.e., there is an implication loop), the role is not implied.

The rule that a Group object must have at least one basic member to be implied is motivated by the following example:

```
group foo
  required members: marketing
  basic members: alice, bob
```

Privileged operations that require membership in “foo” can be performed only by “alice” and “bob”, who are in marketing.

If “alice” and “bob” ever transfer to a different department, anybody in marketing will be able to assume the “foo” role, which certainly must be prevented. Requiring that “foo” (or any Group object for that matter) must have at least one basic member accomplishes that.

However, this would make it impossible for a Group object to be implied by just its required members. An example where this implication might be useful is the following declaration: “Any citizen who is an adult is allowed to vote.” An intuitive configuration of “voter” would be:

```
group voter
```

```
  required members: citizen, adult
     basic members:
```

However, according to the above rule, the "voter" role could never be assumed by anybody, since it lacks any basic members. In order to address this issue a predefined role named "user.anyone" can be specified, which is always implied. The desired implication of the "voter" group can then be achieved by specifying "user.anyone" as its basic member, as follows:

```
group voter
  required members: citizen, adult
     basic members: user.anyone
```

No Implement Consumers of this API must not implement this interface

107.8.4.1 public boolean addMember (Role role)

role The role to add as a basic member.

□ Adds the specified Role object as a basic member to this Group object.

Returns true if the given role could be added as a basic member, and false if this Group object already contains a Role object whose name matches that of the specified role.

Throws SecurityException – If a security manager exists and the caller does not have the UserAdminPermission with name admin.

107.8.4.2 public boolean addRequiredMember (Role role)

role The Role object to add as a required member.

□ Adds the specified Role object as a required member to this Group object.

Returns true if the given Role object could be added as a required member, and false if this Group object already contains a Role object whose name matches that of the specified role.

Throws SecurityException – If a security manager exists and the caller does not have the UserAdminPermission with name admin.

107.8.4.3 public Role[] getMembers ( )

□ Gets the basic members of this Group object.

Returns The basic members of this Group object, or null if this Group object does not contain any basic members.

107.8.4.4 public Role[] getRequiredMembers ( )

□ Gets the required members of this Group object.

Returns The required members of this Group object, or null if this Group object does not contain any required members.

107.8.4.5 public boolean removeMember (Role role)

role The Role object to remove from this Group object.

□ Removes the specified Role object from this Group object.

Returns true if the Role object could be removed, otherwise false.

Throws SecurityException – If a security manager exists and the caller does not have the UserAdminPermission with name admin.

107.8.5 public interface Role

The base interface for Role objects managed by the User Admin service.

This interface exposes the characteristics shared by all Role classes: a name, a type, and a set of properties.

Properties represent public information about the Role object that can be read by anyone. Specific UserAdminPermission objects are required to change a Role object's properties.

Role object properties are Dictionary objects. Changes to these objects are propagated to the User Admin service and made persistent.

Every User Admin service contains a set of predefined Role objects that are always present and cannot be removed. All predefined Role objects are of type ROLE. This version of the org.osgi.service.useradmin package defines a single predefined role named "user.anyone", which is inherited by any other role. Other predefined roles may be added in the future. Since "user.anyone" is a Role object that has properties associated with it that can be read and modified. Access to these properties and their use is application specific and is controlled using UserAdminPermission in the same way that properties for other Role objects are.

No Implement Consumers of this API must not implement this interface

107.8.5.1 public static final int GROUP = 2

The type of a Group role.

The value of GROUP is 2.

107.8.5.2 public static final int ROLE = 0

The type of a predefined role.

The value of ROLE is 0.

107.8.5.3 public static final int USER = 1

The type of a User role.

The value of USER is 1.

107.8.5.4 public static final String USER_ANYONE = "user.anyone"

The name of the predefined role, user.anyone, that all users and groups belong to.

Since 1.1

107.8.5.5 public String getName ( )

□ Returns the name of this role.

Returns The role's name.

107.8.5.6 public Dictionary getProperties ( )

□ Returns a Dictionary of the (public) properties of this Role object. Any changes to the returned Dictionary will change the properties of this Role object. This will cause a UserAdminEvent object of type UserAdminEvent.ROLE_CHANGED to be broadcast to any UserAdminListener objects.

Only objects of type String may be used as property keys, and only objects of type String or byte[] may be used as property values. Any other types will cause an exception of type IllegalArgumentException to be raised.

In order to add, change, or remove a property in the returned Dictionary, a UserAdminPermission named after the property name (or a prefix of it) with action changeProperty is required.

Returns Dictionary containing the properties of this Role object.

107.8.5.7 public int getType ( )

□ Returns the type of this role.

Returns The role's type.

107.8.6 public interface User extends Role

A User role managed by a User Admin service.

In this context, the term "user" is not limited to just human beings. Instead, it refers to any entity that may have any number of credentials associated with it that it may use to authenticate itself.

In general, User objects are associated with a specific User Admin service (namely the one that created them), and cannot be used with other User Admin services.

A User object may have credentials (and properties, inherited from the Role class) associated with it. Specific UserAdminPermission objects are required to read or change a User object's credentials.

Credentials are Dictionary objects and have semantics that are similar to the properties in the Role class.

No Implement Consumers of this API must not implement this interface

107.8.6.1 public Dictionary getCredentials ( )

□ Returns a Dictionary of the credentials of this User object. Any changes to the returned Dictionary object will change the credentials of this User object. This will cause a UserAdminEvent object of type UserAdminEvent.ROLE_CHANGED to be broadcast to any UserAdminListeners objects.

Only objects of type String may be used as credential keys, and only objects of type String or of type byte[] may be used as credential values. Any other types will cause an exception of type IllegalArgumentException to be raised.

In order to retrieve a credential from the returned Dictionary object, a UserAdminPermission named after the credential name (or a prefix of it) with action getCredential is required.

In order to add or remove a credential from the returned Dictionary object, a UserAdminPermission named after the credential name (or a prefix of it) with action changeCredential is required.

Returns Dictionary object containing the credentials of this User object.

107.8.6.2 public boolean hasCredential (String key , Object value)

key The credential key.

value The credential value.

□ Checks to see if this User object has a credential with the specified key set to the specified value.

If the specified credential value is not of type String or byte[], it is ignored, that is, false is returned (as opposed to an IllegalArgumentException being raised).

Returns true if this user has the specified credential; false otherwise.

Throws SecurityException – If a security manager exists and the caller does not have the UserAdminPermission named after the credential key (or a prefix of it) with action getCredential.

107.8.7 public interface UserAdmin

This interface is used to manage a database of named Role objects, which can be used for authentication and authorization purposes.

This version of the User Admin service defines two types of Role objects: "User" and "Group". Each type of role is represented by an int constant and an interface. The range of positive integers is reserved for new types of roles that may be added in the future. When defining proprietary role types, negative constant values must be used.

Every role has a name and a type.

A User object can be configured with credentials (e.g., a password) and properties (e.g., a street address, phone number, etc.).

A Group object represents an aggregation of User and Group objects. In other words, the members of a Group object are roles themselves.

Every User Admin service manages and maintains its own namespace of Role objects, in which each Role object has a unique name.

No Implement Consumers of this API must not implement this interface

107.8.7.1 public Role createRole (String name , int type)

name The name of the Role object to create.

type The type of the Role object to create. Must be either a Role.USER type or Role.GROUP type.

□ Creates a Role object with the given name and of the given type.

If a Role object was created, a UserAdminEvent object of type UserAdminEvent.ROLE_CREATED is broadcast to any UserAdminListener object.

Returns The newly created Role object, or null if a role with the given name already exists.

Throws IllegalArgumentException – if type is invalid.

SecurityException – If a security manager exists and the caller does not have the UserAdminPermission with name admin.

107.8.7.2 public Authorization getAuthorization (User user)

user The User object to create an Authorization object for, or null for the anonymous user.

□ Creates an Authorization object that encapsulates the specified User object and the Role objects it possesses. The null user is interpreted as the anonymous user. The anonymous user represents a user that has not been authenticated. An Authorization object for an anonymous user will be unnamed, and will only imply groups that user.anyone implies.

Returns the Authorization object for the specified User object.

107.8.7.3 public Role getRole (String name)

name The name of the Role object to get.

□ Gets the Role object with the given name from this User Admin service.

Returns The requested Role object, or null if this User Admin service does not have a Role object with the given name.

107.8.7.4 public Role[] getRoles (String filter) throws InvalidSyntaxException

filter The filter criteria to match.

□ Gets the Role objects managed by this User Admin service that have properties matching the specified LDAP filter criteria. See org.osgi.framework.Filter for a description of the filter syntax. If a null filter is specified, all Role objects managed by this User Admin service are returned.

Returns The Role objects managed by this User Admin service whose properties match the specified filter criteria, or all Role objects if a null filter is specified. If no roles match the filter, null will be returned.

Throws InvalidSyntaxException – If the filter is not well formed.

107.8.7.5 public User getUser (String key , String value)

key The property key to look for.

value The property value to compare with.

□ Gets the user with the given property key-value pair from the User Admin service database. This is a convenience method for retrieving a User object based on a property for which every User object is supposed to have a unique value (within the scope of this User Admin service), such as for example a X.500 distinguished name.

Returns A matching user, if *exactly* one is found. If zero or more than one matching users are found, null is returned.

107.8.7.6 public boolean removeRole (String name)

name The name of the Role object to remove.

□ Removes the Role object with the given name from this User Admin service and all groups it is a member of.

If the Role object was removed, a UserAdminEvent object of type UserAdminEvent.ROLE_REMOVED is broadcast to any UserAdminListener object.

Returns true If a Role object with the given name is present in this User Admin service and could be removed, otherwise false.

Throws SecurityException – If a security manager exists and the caller does not have the UserAdminPermission with name admin.

107.8.8 public class UserAdminEvent

Role change event.

UserAdminEvent objects are delivered asynchronously to any UserAdminListener objects when a change occurs in any of the Role objects managed by a User Admin service.

A type code is used to identify the event. The following event types are defined: ROLE_CREATED type, ROLE_CHANGED type, and ROLE_REMOVED type. Additional event types may be defined in the future.

See Also UserAdmin , UserAdminListener

107.8.8.1 public static final int ROLE_CHANGED = 2

A Role object has been modified.

The value of ROLE_CHANGED is 0x00000002.

107.8.8.2 public static final int ROLE_CREATED = 1

A Role object has been created.

The value of ROLE_CREATED is 0x00000001.

107.8.8.3 public static final int ROLE_REMOVED = 4

A Role object has been removed.

The value of ROLE_REMOVED is 0x00000004.

107.8.8.4 public UserAdminEvent (ServiceReference ref , int type , Role role)

ref The ServiceReference object of the User Admin service that generated this event.

type The event type.

role The Role object on which this event occurred.

□ Constructs a UserAdminEvent object from the given ServiceReference object, event type, and Role object.

107.8.8.5 public Role getRole ( )

□ Gets the Role object this event was generated for.

Returns The Role object this event was generated for.

107.8.8.6 public ServiceReference getServiceReference ( )

□ Gets the ServiceReference object of the User Admin service that generated this event.

Returns The User Admin service's ServiceReference object.

107.8.8.7 public int getType ( )

□ Returns the type of this event.

The type values are ROLE_CREATED type, ROLE_CHANGED type, and ROLE_REMOVED type.

Returns The event type.

107.8.9 public interface UserAdminListener

Listener for UserAdminEvents.

UserAdminListener objects are registered with the Framework service registry and notified with a UserAdminEvent object when a Role object has been created, removed, or modified.

UserAdminListener objects can further inspect the received UserAdminEvent object to determine its type, the Role object it occurred on, and the User Admin service that generated it.

See Also UserAdmin , UserAdminEvent

107.8.9.1 public void roleChanged (UserAdminEvent event)

event The UserAdminEvent object.

□ Receives notification that a Role object has been created, removed, or modified.

107.8.10 public final class UserAdminPermission extends BasicPermission

Permission to configure and access the Role objects managed by a User Admin service.

This class represents access to the Role objects managed by a User Admin service and their properties and credentials (in the case of User objects).

The permission name is the name (or name prefix) of a property or credential. The naming convention follows the hierarchical property naming convention. Also, an asterisk may appear at the end of the name, following a ".", or by itself, to signify a wildcard match. For example: "org.osgi.security.protocol.*" or "*" is valid, but "*protocol" or "a*b" are not valid.

The UserAdminPermission with the reserved name "admin" represents the permission required for creating and removing Role objects in the User Admin service, as well as adding and removing members in a Group object. This UserAdminPermission does not have any actions associated with it.

The actions to be granted are passed to the constructor in a string containing a list of one or more comma-separated keywords. The possible keywords are: changeProperty,changeCredential, and getCredential. Their meaning is defined as follows:

```
action
changeProperty    Permission to change (i.e., add and remove)
                  Role object properties whose names start with
                  the name argument specified in the constructor.
changeCredential  Permission to change (i.e., add and remove)
                  User object credentials whose names start
                  with the name argument specified in the constructor.
```

```
getCredential     Permission to retrieve and check for the
                  existence of User object credentials whose names
                  start with the name argument specified in the
                  constructor.
```

The action string is converted to lowercase before processing.

Following is a PermissionInfo style policy entry which grants a user administration bundle a number of UserAdminPermission object:

```
  (org.osgi.service.useradmin.UserAdminPermission “admin”)
  (org.osgi.service.useradmin.UserAdminPermission “com.foo.*” “changeProperty,get-
Credential,changeCredential”)
  (org.osgi.service.useradmin.UserAdminPermission “user.*”, “changeProperty,
changeCredential”)
```

The first permission statement grants the bundle the permission to perform any User Admin service operations of type “admin”, that is, create and remove roles and configure Group objects.

The second permission statement grants the bundle the permission to change any properties as well as get and change any credentials whose names start with com.foo..

The third permission statement grants the bundle the permission to change any properties and credentials whose names start with user.. This means that the bundle is allowed to change, but not retrieve any credentials with the given prefix.

The following policy entry empowers the Http Service bundle to perform user authentication:

```
grant codeBase “${jars}http.jar” {
  permission org.osgi.service.useradmin.UserAdminPermission
    “user.password”, “getCredential”;
};
```

The permission statement grants the Http Service bundle the permission to validate any password credentials (for authentication purposes), but the bundle is not allowed to change any properties or credentials.

Concurrency Thread-safe

107.8.10.1 public static final String ADMIN = “admin”

The permission name “admin”.

107.8.10.2 public static final String CHANGE_CREDENTIAL = “changeCredential”

The action string “changeCredential”.

107.8.10.3 public static final String CHANGE_PROPERTY = “changeProperty”

The action string “changeProperty”.

107.8.10.4 public static final String GET_CREDENTIAL = “getCredential”

The action string “getCredential”.

107.8.10.5 public UserAdminPermission (String name , String actions)

name the name of this UserAdminPermission

actions the action string.

□ Creates a new UserAdminPermission with the specified name and actions. name is either the reserved string “admin” or the name of a credential or property, and actions contains a comma-separated list of the actions granted on the specified name. Valid actions are changeProperty, changeCredential, and getCredential.

Throws IllegalArgumentException – If name equals "admin" and actions are specified.

107.8.10.6 public boolean equals (Object obj)

obj the object to be compared for equality with this object.

□ Checks two UserAdminPermission objects for equality. Checks that obj is a UserAdminPermission, and has the same name and actions as this object.

Returns true if obj is a UserAdminPermission object, and has the same name and actions as this UserAdminPermission object.

107.8.10.7 public String getActions ( )

□ Returns the canonical string representation of the actions, separated by comma.

Returns the canonical string representation of the actions.

107.8.10.8 public int hashCode ( )

□ Returns the hash code value for this object.

Returns A hash code value for this object.

107.8.10.9 public boolean implies (Permission p)

p the permission to check against.

□ Checks if this UserAdminPermission object "implies" the specified permission.

More specifically, this method returns true if:

- *p* is an instanceof UserAdminPermission,
- *p* 's actions are a proper subset of this object's actions, and
- *p* 's name is implied by this object's name. For example, "java.*" implies "java.home".

Returns true if the specified permission is implied by this object; false otherwise.

107.8.10.10 public PermissionCollection newPermissionCollection ( )

□ Returns a new PermissionCollection object for storing UserAdminPermission objects.

Returns a new PermissionCollection object suitable for storing UserAdminPermission objects.

107.8.10.11 public String toString ( )

□ Returns a string describing this UserAdminPermission object. This string must be in PermissionInfo encoded format.

Returns The PermissionInfo encoded string for this UserAdminPermission object.

See Also org.osgi.service.permissionadmin.PermissionInfo.getEncoded()

107.9 References

[1] *The Java Security Architecture for JDK 1.2*
Version 1.0, Sun Microsystems, October 1998

[2] *Java Authentication and Authorization Service*
http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html

110 Initial Provisioning Specification

Version 1.2

110.1 Introduction

To allow freedom regarding the choice of management protocol, the OSGi Specifications assumes an architecture to remotely manage a Service Platform with a Management Agent. The Management Agent is implemented with a Management Bundle that can communicate with an unspecified management protocol.

This specification defines how the Management Agent can make its way to the Service Platform, and gives a structured view of the problems and their corresponding resolution methods.

The purpose of this specification is to enable the management of a Service Platform by an Operator, and (optionally) to hand over the management of the Service Platform later to another Operator. This approach is in accordance with the OSGi remote management reference architecture.

This bootstrapping process requires the installation of a Management Agent, with appropriate configuration data, in the Service Platform.

This specification consists of a prologue, in which the principles of the Initial Provisioning are outlined, and a number of mappings to different mechanisms.

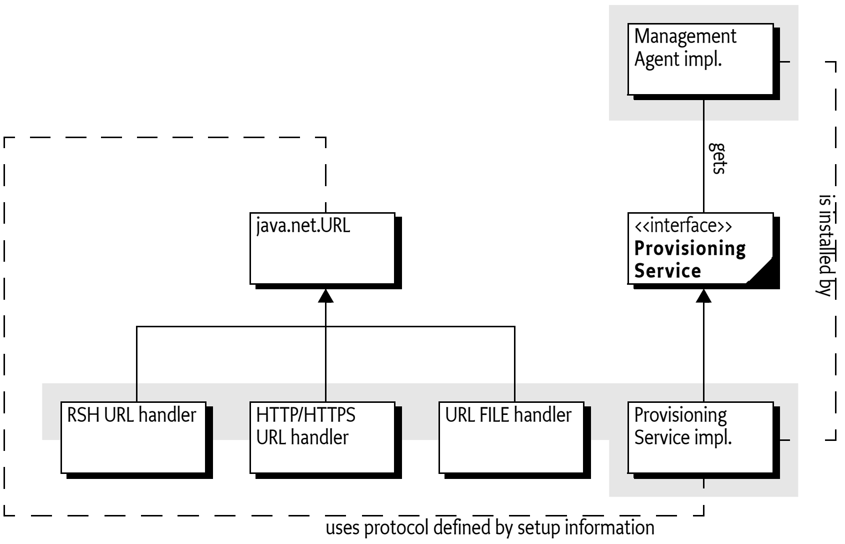
110.1.1 Essentials

- *Policy Free* – The proposed solution must be business model agnostic; none of the affected parties (Operators, SPS Manufacturers, etc.) should be forced into any particular business model.
- *Inter-operability* – The Initial Provisioning must permit arbitrary inter-operability between management systems and Service Platforms. Any compliant Remote Manager should be able to manage any compliant Service Platform, even in the absence of a prior business relationship. Adhering to this requirement allows a particular Operator to manage a variety of makes and models of Service Platform Servers using a single management system of the Operator's choice. This rule also gives the consumer the greatest choice when selecting an Operator.
- *Flexible* – The management process should be as open as possible, to allow innovation and specialization while still achieving interoperability.

110.1.2 Entities

- *Provisioning Service* – A service registered with the Framework that provides information about the initial provisioning to the Management Agent.
- *Provisioning Dictionary* – A `Dictionary` object that is filled with information from the ZIP files that are loaded during initial setup.
- *RSH Protocol* – An OSGi specific secure protocol based on HTTP.
- *Management Agent* – A bundle that is responsible for managing a Service Platform under control of a Remote Manager.

Figure 110.1 Initial Provisioning



110.2 Procedure

The following procedure should be executed by an OSGi Framework implementation that supports this Initial Provisioning specification.

When the Service Platform is first brought under management control, it must be provided with an initial request URL in order to be provisioned. Either the end user or the manufacturer may provide the initial request URL. How the initial request URL is transferred to the Framework is not specified, but a mechanism might, for example, be a command line parameter when the framework is started.

When asked to start the Initial Provisioning, the Service Platform will send a request to the management system. This request is encoded in a URL, for example:

```
http://osgi.acme.com/remote-manager
```

This URL may use any protocol that is available on the Service Platform Server. Many standard protocols exist, but it is also possible to use a proprietary protocol. For example, software could be present which can communicate with a smart card and could handle, for example, this URL:

```
smart-card://com1:0/7F20/6F38
```

Before the request URL is executed, the Service Platform information is appended to the URL. This information includes at least the Service Platform Identifier, but may also contain proprietary information, as long as the keys for this information do not conflict. Different URL schemes may use different methods of appending parameters; these details are specified in the mappings of this specification to concrete protocols.

The result of the request must be a ZIP file (The content type should be application/zip). It is the responsibility of the underlying protocol to guarantee the integrity and authenticity of this ZIP file.

This ZIP file is unpacked and its entries (except bundle and bundle-url entries, described in Table 110.2) are placed in a Dictionary object. This Dictionary object is called the *Provisioning Dictionary*. It must be made available from the Provisioning Service in the service registry. The names of the entries in the ZIP file must not start with a slash ('/').

The ZIP file may contain only four types of dictionary entries: text, binary, bundle, or bundle-url. The type of an entry can be specified in different ways. An Initial Provisioning service must look in the following places to find the information about an entry's (MIME) type (in the given order):

1 The manifest header InitialProvisioning-Entries of the given ZIP file. This header is defined in *InitialProvisioning-Entries Manifest Header* on page 175. If this header is present, but a given entry's path is not named then try the next step.
2 The extension of the entry path name if one of .txt, .jar, .url extensions. See *Content types of provisioning ZIP file* on page 173 for the mapping of types, MIME types, and extensions.
3 The entry is assumed to be a binary type

The types can optionally be specified as a MIME type as defined in [7] *MIME Types*. The text and bundle-url entries are translated into a String object from an UTF-8 encoded byte array. All other entries must be stored as a byte[].

Table 110.1 Content types of provisioning ZIP file

Type	MIME Type	Ext	Description
text	MIME_STRING text/ plain;charset=utf-8	.txt	Must be represented as a String object
binary	MIME_BYTE_ARRAY application/octet-stream	not .txt, .url, or .jar	Must be represented as a byte array (byte[]).
bundle	MIME_BUNDLE application/ vnd.osgi.bundle MIME_BUNDLE_ALT application/ x-osgi-bundle	.jar	Entries must be installed using BundleContext.installBundle(String,InputStream), with the InputStream object constructed from the contents of the ZIP entry. The location must be the name of the ZIP entry without leading slash. This entry must not be stored in the Provisioning Dictionary. If a bundle with this location name is already installed in this system, then this bundle must be updated instead of installed. The MIME_BUNDLE_ALT version is intended for backward compatibility, it specifies the original MIME type for bundles before there was an official IANA MIME type.
bundle-url	MIME_BUNDLE_URL text/ x-osgi-bundle-url; charset=utf-8	.url	The content of this entry is a string coded in utf-8. Entries must be installed using BundleContext.installBundle(String, InputStream), with the InputStream object created from the given URL. The location must be the name of the ZIP entry without leading slash. This entry must not be stored in the Provisioning Dictionary. If a bundle with this location url is already installed in this system, then this bundle must be updated instead of installed.

The Provisioning Service must install (but not start) all entries in the ZIP file that are typed with bundle or bundle-url.

If an entry named PROVISIONING_START_BUNDLE is present in the Provisioning Dictionary, then its content type must be text as defined in Table 110.1. The content of this entry must match the bundle location of a previously loaded bundle. This designated bundle must be given AllPermission and started.

If no PROVISIONING_START_BUNDLE entry is present in the Provisioning Dictionary, the Provisioning Dictionary should contain a reference to another ZIP file under the PROVISIONING_REFERENCE key. If both keys are absent, no further action must take place.

If this PROVISIONING_REFERENCE key is present and holds a String object that can be mapped to a valid URL, then a new ZIP file must be retrieved from this URL. The PROVISIONING_REFERENCE link may be repeated multiple times in successively loaded ZIP files.

Referring to a new ZIP file with such a URL allows a manufacturer to place a fixed reference inside the Service Platform Server (in a file or smart card) that will provide some platform identifying information and then also immediately load the information from the management system. The PROVISIONING_REFERENCE link may be repeated multiple times in successively loaded ZIP files. The entry PROVISIONING_UPDATE_COUNT must be an Integer object that must be incremented on every iteration.

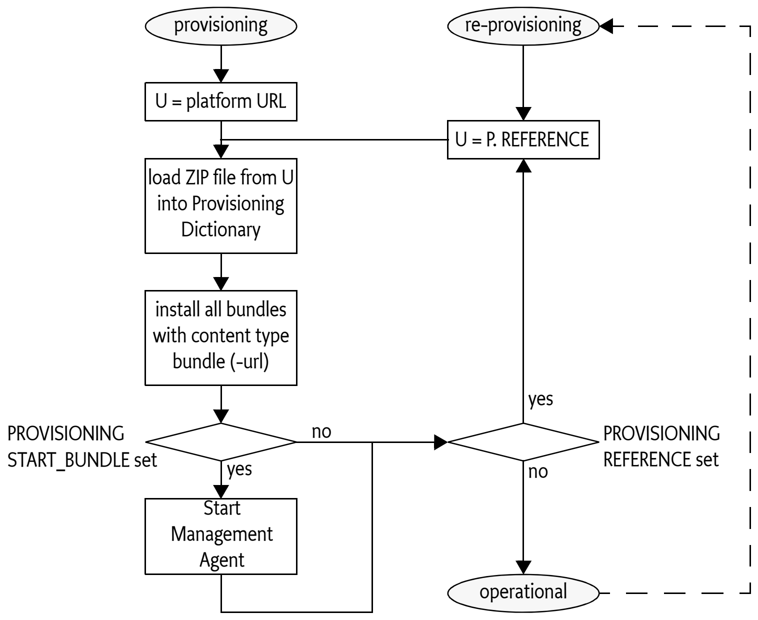
Information retrieved while loading subsequent PROVISIONING_REFERENCE URLs may replace previous key/values in the Provisioning Dictionary, but must not erase unrecognized key/values. For example, if an assignment has assigned the key proprietary-x, with a value '3', then later assignments must not override this value, unless the later loaded ZIP file contains an entry with that name. All these updates to the Provisioning Dictionary must be stored persistently. At the same time, each entry of type bundle or bundle-url (see Table 110.1) must be installed and not started.

Once the Management Agent has been started, the Initial Provisioning service has become operational. In this state, the Initial Provisioning service must react when the Provisioning Dictionary is updated with a new PROVISIONING_REFERENCE property. If this key is set, it should start the cycle again. For example, if the control of a Service Platform needs to be transferred to another Remote Manager, the Management Agent should set the PROVISIONING_REFERENCE to the location of this new Remote Manager's Initial Provisioning ZIP file.This process is called *re-provisioning*.

If errors occur during this process, the Initial Provisioning service should try to notify the Service User of the problem.

The previous description is depicted in Figure 110.2 as a flow chart.

Figure 110.2 Flow chart installation Management Agent bundle


provisioning
re-provisioning
U = platform URL
U = P. REFERENCE
load ZIP file from U into Provisioning Dictionary
install all bundles with content type bundle (-url)
PROVISIONING START_BUNDLE set
no
yes
yes
PROVISIONING REFERENCE set
no
Start Management Agent
operational

The Management Agent may require configuration data that is specific to the Service Platform instance. If this data is available outside the Management Agent bundle, the merging of this data with the Management Agent may take place in the Service Platform. Transferring the data separately will make it possible to simplify the implementation on the server side, as it is not necessary to create *personalized* Service Platform bundles. The PROVISIONING_AGENT_CONFIG key is reserved for this purpose, but the Management Agent may use another key or mechanisms if so desired.

The PROVISIONING_SPID key must contain the Service Platform Identifier.

110.2.1 InitialProvisioning-Entries Manifest Header

The InitialProvisioning-Entries manifest header optionally specifies the type of the entries in the ZIP file. The syntax for this header is:

```
InitialProvisioning-Entries ::= ip-entry ( ',' ip-entry ) *
ip-entry                    ::= path ( ';' parameter ) *
```

The entry is the path name of a resource in the ZIP file. This InitialProvisioning-Entries header recognizes the following attribute:

- type – Gives the type of the dictionary entry. The type can have one of the following values: text, binary, bundle, or bundle-url

If the type parameter entry is not specified for an entry, then the type will be inferred from the extension of the entry, as defined in table *Content types of provisioning ZIP file* on page 173.

110.3 Special Configurations

The next section shows some examples of specially configured types of Service Platform Servers and how they are treated with the respect to the specifications in this document.

110.3.1 Branded Service Platform Server

If a Service Platform Operator is selling Service Platform Servers branded exclusively for use with their service, the provisioning will most likely be performed prior to shipping the Service Platform Server to the User. Typically the Service Platform is configured with the Dictionary entry PROVISIONING_REFERENCE pointing at a location controlled by the Operator.

Up-to-date bundles and additional configuration data must be loaded from that location at activation time. The Service Platform is probably equipped with necessary security entities, like certificates, to enable secure downloads from the Operator's URL over open networks, if necessary.

110.3.2 Non-connected Service Platform

Circumstances might exist in which the Service Platform Server has no WAN connectivity, or prefers not to depend on it for the purposes not covered by this specification.

The non-connected case can be implemented by specifying a file:// URL for the initial ZIP file (PROVISIONING_REFERENCE). That file:// URL would name a local file containing the response that would otherwise be received from a remote server.

The value for the Management Agent PROVISIONING_REFERENCE found in that file will be used as input to the load process. The PROVISIONING_REFERENCE may point to a bundle file stored either locally or remotely. No code changes are necessary for the non-connected scenario. The file:// URLs must be specified, and the appropriate files must be created on the Service Platform.

110.4 The Provisioning Service

Provisioning information is conveyed between bundles using the Provisioning Service, as defined in the ProvisioningService interface. The Provisioning Dictionary is retrieved from the ProvisioningService object using the getInformation() method. This is a read-only Dictionary object, any changes to this Dictionary object must throw an UnsupportedOperationException.

The Provisioning Service provides a number of methods to update the Provisioning Dictionary.

- addInformation(Dictionary) – Add all key/value pairs in the given Dictionary object to the Provisioning Dictionary.
- addInformation(ZipInputStream) – It is also possible to add a ZIP file to the Provisioning Service immediately. This will unpack the ZIP file and add the entries to the Provisioning Dictionary. This method must install the bundles contained in the ZIP file as described in *Procedure* on page 172.
- setInformation(Dictionary) – Set a new Provisioning Dictionary. This will remove all existing entries.

Each of these method will increment the PROVISIONING_UPDATE_COUNT entry.

110.5 Management Agent Environment

The Management Agent should be written with great care to minimize dependencies on other packages and services, as *all* services in OSGi are optional. Some Service Platforms may have other bundles pre-installed, so it is possible that there may be exported packages and services available. Mechanisms outside the current specification, however, must be used to discover these packages and services before the Management Agent is installed.

The Provisioning Service must ensure that the Management Agent is running with AllPermission. The Management Agent should check to see if the Permission Admin service is available, and establish the initial permissions as soon as possible to insure the security of the device when later bundles are installed. As the PermissionAdmin interfaces may not be present (it is an optional service), the Management Agent should export the PermissionAdmin interfaces to ensure they can be resolved.

Once started, the Management Agent may retrieve its configuration data from the Provisioning Service by getting the byte[] object that corresponds to the PROVISIONING_AGENT_CONFIG key in the Provisioning Dictionary. The structure of the configuration data is implementation specific.

The scope of this specification is to provide a mechanism to transmit the raw configuration data to the Management Agent. The Management Agent bundle may alternatively be packaged with its configuration data in the bundle, so it may not be necessary for the Management Agent bundle to use the Provisioning Service at all.

Most likely, the Management Agent bundle will install other bundles to provision the Service Platform. Installing other bundles might even involve downloading a more full featured Management Agent to replace the initial Management Agent.

110.6 Mapping To File Scheme

The file: scheme is the simplest and most completely supported scheme which can be used by the Initial Provisioning specification. It can be used to store the configuration data and Management Agent bundle on the Service Platform Server, and avoids any outside communication.

If the initial request URL has a file scheme, no parameters should be appended, because the file: scheme does not accept parameters.

110.6.1 Example With File Scheme

The manufacturer should prepare a ZIP file containing only one entry named PROVISIONING_START_BUNDLE that contains a location string of an entry of type bundle or bundle-url. For example, the following ZIP file demonstrates this:

```
provisioning.start.bundle  text       agent
agent                      bundle     C0AF0E9B2AB..
```

The bundle may also be specified with a URL:

```
provisioning.start.bundle  text       http://acme.com/a.jar
agent                      bundle-url http://acme.com/a.jar
```

Upon startup, the framework is provided with the URL with the file: scheme that points to this ZIP file:

```
file:/opt/osgi/ma.zip
```

110.7 Mapping To HTTP(S) Scheme

This section defines how HTTP and HTTPS URLs must be used with the Initial Provisioning specification.

- HTTP – May be used when the data exchange takes place over networks that are secured by other means, such as a Virtual Private Network (VPN) or a physically isolated network. Otherwise, HTTP is not a valid scheme because no authentication takes place.
- HTTPS – May be used if the Service Platform is equipped with appropriate certificates.

HTTP and HTTPS share the following qualities:

- Both are well known and widely used
- Numerous implementations of the protocols exist
- Caching of the Management Agent will be desired in many implementations where limited bandwidth is an issue. Both HTTP and HTTPS already contain an accepted protocol for caching.

Both HTTP and HTTPS must be used with the GET method. The response is a ZIP file, implying that the response header Content-Type header must contain application/zip.

110.7.1 HTTPS Certificates

In order to use HTTPS, certificates must be in place. These certificates, that are used to establish trust towards the Operator, may be made available to the Service Platform using the Provisioning Service. The root certificate should be assigned to the Provisioning Dictionary before the HTTPS provider is used. Additionally, the Service Platform should be equipped with a Service Platform certificate that allows the Service Platform to properly authenticate itself towards the Operator. This specification does not state how this certificate gets installed into the Service Platform.

The root certificate is stored in the Provisioning Dictionary under the key:

PROVISIONING_ROOTX509

The Root X.509 Certificate holds certificates used to represent a handle to a common base for establishing trust. The certificates are typically used when authenticating a Remote Manager to the Service Platform. In this case, a Root X.509 certificate must be part of a certificate chain for the Operator's certificate. The format of the certificate is defined in *Certificate Encoding* on page 178.

110.7.2 Certificate Encoding

Root certificates are X.509 certificates. Each individual certificate is stored as a byte[] object. This byte[] object is encoded in the default Java manner, as follows:

- The original, binary certificate data is DER encoded
- The DER encoded data is encoded into base64 to make it text.
- The base64 encoded data is prefixed with
  -----BEGIN CERTIFICATE-----
  and suffixed with:
  -----END CERTIFICATE-----
- If a record contains more than one certificate, they are simply appended one after the other, each with a delimiting prefix and suffix.

The decoding of such a certificate may be done with the java.security.cert.CertificateFactory class:

```
InputStream bis = new ByteArrayInputStream(x509); // byte[]
CertificateFactory cf =
  CertificateFactory.getInstance("X.509");
Collection c = cf.generateCertificates(bis);
Iterator i = c.iterator();
while (i.hasNext()) {
  Certificate cert = (Certificate)i.next();
  System.out.println(cert);
}
```

110.7.3 URL Encoding

The URL must contain the Service Platform Identity, and may contain more parameters. These parameters are encoded in the URL according to the HTTP(S) URL scheme. A base URL may be set by an end user but the Provisioning Service must add the Service Platform Identifier.

If the request URL already contains HTTP parameters (if there is a '' in the request), the service_platform_id is appended to this URL as an additional parameter. If, on the other hand, the request URL does not contain any HTTP parameters, the service_platform_id will be appended to the URL after a '', becoming the first HTTP parameter. The following two examples show these two variants:

```
http://server.operator.com/service-x «
  foo=bar&service_platform_id=VIN:123456789

http://server.operator.com/service-x «
```

```
service_platform_id=VIN:123456789
```

Proper URL encoding must be applied when the URL contains characters that are not allowed. See [6] *RFC 2396 - Uniform Resource Identifier (URI)*.

110.8 Mapping To RSH Scheme

The RSH protocol is an OSGi-specific protocol, and is included in this specification because it is optimized for Initial Provisioning. It requires a shared secret between the management system and the Service Platform that is small enough to be entered by the Service User.

RSH bases authentication and encryption on Message Authentication Codes (MACs) that have been derived from a secret that is shared between the Service Platform and the Operator prior to the start of the protocol execution.

The protocol is based on an ordinary HTTP GET request/response, in which the request must be *signed* and the response must be *encrypted* and *authenticated*. Both the *signature* and *encryption key* are derived from the shared secret using Hashed Message Access Codes (HMAC) functions.

As additional input to the HMAC calculations, one client-generated nonce and one server-generated nonce are used to prevent replay attacks. The nonces are fairly large random numbers that must be generated in relation to each invocation of the protocol, in order to guarantee freshness. These nonces are called `clientfg` (client-generated freshness guarantee) and `serverfg` (server-generated freshness guarantee).

In order to separate the HMAC calculations for authentication and encryption, each is based on a different constant value. These constants are called the *authentication constant* and the *encryption constant*.

From an abstract perspective, the protocol may be described as follows.

- δ – Shared secret, 160 bits or more
- s – Server nonce, called `servercfg`, 128 bits
- c – Client nonce, called `clientfg`, 128 bits
- K_a – Authentication key, 160 bits
- K_e – Encryption key, 192 bits
- r – Response data
- e – Encrypted data
- E – Encryption constant, a `byte[]` of 05, 36, 54, 70, 00 (hex)
- A – Authentication constant, a `byte[]` of 00, 4f, 53, 47, 49 (hex)
- M – Message material, used for K_e calculation.
- m – The calculated message authentication code.
- *3DES* – Triple DES, encryption function, see [8] *3DES*. The bytes of the key must be set to odd parity. CBC mode must be used where the padding method is defined in [9] *RFC 1423 Part III: Algorithms, Modes, and Identifiers*. In [11] *Java Cryptography API (part of Java 1.4)* this is addressed as `PKCS5Padding`.
- *IV* – Initialization vector for 3DES.
- *SHA1* – Secure Hash Algorithm to generate the Hashed Message Authentication Code, see [12] *SHA-1*. The function takes a single parameter, the block to be worked upon.
- *HMAC* – The function that calculates a message authentication code, which must HMAC-SHA1. HMAC-SHA1 is defined in [1] *HMAC: Keyed-Hashing for Message Authentication*. The HMAC function takes a key and a block to be worked upon as arguments. Note that the lower 16 bytes of the result must be used.
- *{}* – Concatenates its arguments
- *[]* – Indicates access to a sub-part of a variable, in bytes. Index starts at one, not zero.

In each step, the emphasized server or client indicates the context of the calculation. If both are used at the same time, each variable will have server or client as a subscript.

1. The *client* generates a random nonce, stores it and denotes it clientfg

   $c = nonce$

2. The client sends the request with the clientfg to the server.

   $c_{server} \Leftarrow c_{client}$

3. The *server* generates a nonce and denotes it serverfg.

   $s = nonce$

4. The *server* calculates an authentication key based on the SHA1 function, the shared secret, the received clientfg, the serverfg and the authentication constant.

   $K_a \leftarrow SHA1(\{\delta, c, s, A\})$

5. The *server* calculates an encryption key using an SHA-1 function, the shared secret, the received clientfg, the serverfg and the encryption constant. It must first calculate the *key material* M.

   $M[1, 20] \leftarrow SHA1(\{\delta, c, s, E\})$

   $M[21, 40] \leftarrow SHA1(\{\delta, M[1, 20], c, s, E\})$

6. The key for DES consists K_e and IV.

   $K_e \leftarrow M[1, 24]$

   $IV \leftarrow M[25, 32]$

   The *server* encrypts the response data using the encryption key derived in 5. The encryption algorithm that must be used to encrypt/decrypt the response data is 3DES. 24 bytes (192 bits) from M are used to generate K_e, but the low order bit of each byte must be used as an odd parity bit. This means that before using K_e, each byte must be processed to set the low order bit so that the byte has odd parity.

   The encryption/decryption key used is specified by the following:

   $e \leftarrow 3DES(K_e, IV, r)$

7. The *server* calculates a MAC *m* using the HMAC function, the encrypted response data and the authentication key derived in 4.

   $m \leftarrow HMAC(K_a, e)$

8. The *server* sends a response to the *client* containing the serverfg, the MAC *m* and the encrypted response data

   $s_{client} \Leftarrow s_{server}$

   $m_{client} \Leftarrow m_{server}$

   $e_{client} \Leftarrow e_{server}$

   The *client* calculates the encryption key K_e the same way the server did in step 5 and 6, and uses this to decrypt the encrypted response data. The serverfg value received in the response is used in the calculation.

   $r \leftarrow 3DES(K_e, IV, e)$

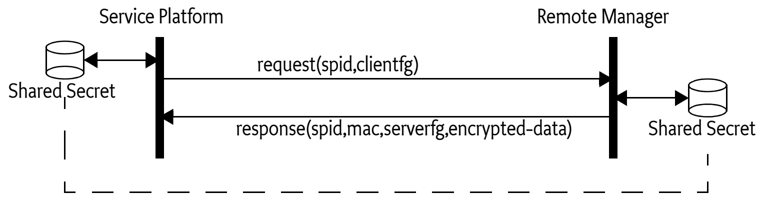
9. The *client* performs the calculation of the MAC *m'* in the same way the server did, and checks that the results match the received MAC *m*. If they do not match, further processing is discarded. The serverfg value received in the response is used in the calculation.

   $K_a \leftarrow SHA1(\{\delta, c, s, A\})$

   $m' \leftarrow HMAC(K_a, e)$

   $m' = m$

Figure 110.3 *Action Diagram for RSH*



110.8.1 Shared Secret

The *shared secret* should be a key of length 160 bits (20 bytes) or more. The length is selected to match the output of the selected hash algorithm [2] *NIST, FIPS PUB 180-1: Secure Hash Standard, April 1995.*.

In some scenarios, the shared secret is generated by the Operator and communicated to the User, who inserts the secret into the Service Platform through some unspecified means.

The opposite is also possible: the shared secret can be stored within the Service Platform, extracted from it, and then communicated to the Operator. In this scenario, the source of the shared secret could be either the Service Platform or the Operator.

In order for the server to calculate the authentication and encryption keys, it requires the proper shared secret. The server must have access to many different shared secrets, one for each Service Platform it is to support. To be able to resolve this issue, the server must typically also have access to the Service Platform Identifier of the Service Platform. The normal way for the server to know the Service Platform Identifier is through the application protocol, as this value is part of the URL encoded parameters of the HTTP, HTTPS, or RSH mapping of the Initial Provisioning.

In order to be able to switch Operators, a new shared secret must be used. The new secret may be generated by the new Operator and then inserted into the Service Platform device using a mechanism not covered by this specification. Or the device itself may generate the new secret and convey it to the owner of the device using a display device or read-out, which is then communicated to the new operator out-of-band. Additionally, the generation of the new secret may be triggered by some external event, like holding down a button for a specified amount of time.

110.8.2 Request Coding

RSH is mapped to HTTP or HTTPS. Thus, the request parameters are URL encoded as discussed in 110.7.3 *URL Encoding*. RSH requires an additional parameter in the URL: the `clientfg` parameter. This parameter is a nonce that is used to counter replay attacks. See also *RSH Transport* on page 182.

110.8.3 Response Coding

The server's response to the client is composed of three parts:

- A header containing the protocol version and the `serverfg`
- The MAC
- The encrypted response

These three items are packaged into a binary container according to Table 110.2.

Table 110.2 *RSH Header description*

Bytes	Description	Value hex
4	Number of bytes in header	2E
1	Major version number	01
1	Minor version number	00
16	`serverfg`	...

Table 110.2 RSH Header description

Bytes	Description	Value hex
4	Number of bytes in MAC	10
16	Message Authentication Code	MAC
4	Number of bytes of encrypted ZIP file	N
N	Encrypted ZIP file	...

The response content type is an RSH-specific encrypted ZIP file, implying that the response header Content-Type must be application/x-rsh for the HTTP request. When the content file is decrypted, the content must be a ZIP file.

110.8.4 RSH URL

The RSH URL must be used internally within the Service Platform to indicate the usage of RSH for initial provisioning. The RSH URL format is identical to the HTTP URL format, except that the scheme is rsh: instead of http:. For example (« means line continues on next line):

```
rsh://server.operator.com/service-x
```

110.8.5 Extensions to the Provisioning Service Dictionary

RSH specifies one additional entry for the Provisioning Dictionary:

PROVISIONING_RSH_SECRET

The value of this entry is a byte[] containing the shared secret used by the RSH protocol.

110.8.6 RSH Transport

RSH is mapped to HTTP or HTTPS and follows the same URL encoding rules, except that the clientfg is additionally appended to the URL. The key in the URL must be clientfg and the value must be encoded in base 64 format:

The clientfg parameter is transported as an HTTP parameter that is appended after the service_platform_id parameter. The second example above would then be:

```
rsh://server.operator.com/service-x
```

Which, when mapped to HTTP, must become:

```
http://server.operator.com/service-x «
   service_platform_id=VIN:123456789& «
   clientfg=AHPmWcw%2FsiWYC37xZNdKvQ%3D%3D
```

110.9 Exception Handling

The Initial Provisioning process is a a sensitive process that must run without user supervision. There is therefore a need to handle exceptional cases in a well defined way to simplify trouble shooting.

There are only 2 types of problems that halt the provisioning process. They are:

- IO Exception when reading or writing provisioning information.
- IO Exception when retrieving or processing a provisioning zip file.

Other exceptions can occur and the Provisioning Service must do any attempt to log these events.

In the cases that the provisioning process stops, it is important that the clients of the provisioning service have a way to find out that the process is stopped. The mechanism that is used for this is a special entry in the provisioning dictionary. The name of the entry must be `provisioning.error`. The value is a String object with the following format:

- Numeric error code
- Space
- A human readable string describing the error.

Permitted error codes are:

- 0 – Unknown error
- 1 – Couldn't load or save provisioning information
- 2 – Malformed URL Exception
- 3 – IO Exception when retrieving document of a URL
- 4 – Corrupted Zip Input Stream

The provisioning.update.count will be incremented as normal when a `provisioning.error` entry is added to the provisioning information. After, the provisioning service will take no further action.

Some examples:

```
0 SIM card removed
2 "http://www.acme.com/secure/blib/ifa.zip"
```

110.10 Security

The security model for the Service Platform is based on the integrity of the Management Agent deployment. If any of the mechanisms used during the deployment of management agents are weak, or can be compromised, the whole security model becomes weak.

From a security perspective, one attractive means of information exchange would be a smart card. This approach enables all relevant information to be stored in a single place. The Operator could then provide the information to the Service Platform by inserting the smart card into the Service Platform.

110.10.1 Concerns

The major security concerns related to the deployment of the Management Agent are:

- The Service Platform is controlled by the intended Operator
- The Operator controls the intended Service Platform(s)
- The integrity and confidentiality of the information exchange that takes place during these processes must be considered

In order to address these concerns, an implementation of the OSGi Remote Management Architecture must assure that:

- The Operator authenticates itself to the Service Platform
- The Service Platform authenticates itself to the Operator
- The integrity and confidentiality of the Management Agent, certificates, and configuration data are fully protected if they are transported over public transports.

Each mapping of the Initial Provisioning specification to a concrete implementation must describe how these goals are met.

110.10.2 Service Platform Long-Term Security

Secrets for long-term use may be exchanged during the Initial Provisioning procedures. This way, one or more secrets may be shared securely, assuming that the Provisioning Dictionary assignments used are implemented with the proper security characteristics.

110.10.3 Permissions

The provisioning information may contain sensitive information. Also, the ability to modify provisioning information can have drastic consequences. Thus, only trusted bundles should be allowed to register, or get the Provisioning Service. This restriction can be enforced using `ServicePermission[ ProvisioningService, GET]`.

No `Permission` classes guard reading or modification of the Provisioning Dictionary, so care must be taken not to leak the `Dictionary` object received from the Provisioning Service to bundles that are not trusted.

Whether message-based or connection-based, the communications used for Initial Provisioning must support mutual authentication and message integrity checking, at a minimum.

By using both server and client authentication in HTTPS, the problem of establishing identity is solved. In addition, HTTPS will encrypt the transmitted data. HTTPS requires a Public Key Infrastructure implementation in order to retrieve the required certificates.

When RSH is used, it is vital that the shared secret is shared only between the Operator and the Service Platform, and no one else.

110.11 org.osgi.service.provisioning

Provisioning Package Version 1.2.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.provisioning; version="[1.2,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.provisioning; version="[1.2,1.3)"
```

110.11.1 public interface ProvisioningService

Service for managing the initial provisioning information.

Initial provisioning of an OSGi device is a multi step process that culminates with the installation and execution of the initial management agent. At each step of the process, information is collected for the next step. Multiple bundles may be involved and this service provides a means for these bundles to exchange information. It also provides a means for the initial Management Bundle to get its initial configuration information.

The provisioning information is collected in a `Dictionary` object, called the Provisioning Dictionary. Any bundle that can access the service can get a reference to this object and read and update provisioning information. The key of the dictionary is a `String` object and the value is a `String` or `byte[]` object. The single exception is the PROVISIONING_UPDATE_COUNT value which is an Integer. The `provisioning` prefix is reserved for keys defined by OSGi, other key names may be used for implementation dependent provisioning systems.

Any changes to the provisioning information will be reflected immediately in all the dictionary objects obtained from the Provisioning Service.

Because of the specific application of the Provisioning Service, there should be only one Provisioning Service registered. This restriction will not be enforced by the Framework. Gateway operators or manufactures should ensure that a Provisioning Service bundle is not installed on a device that already has a bundle providing the Provisioning Service.

The provisioning information has the potential to contain sensitive information. Also, the ability to modify provisioning information can have drastic consequences. Thus, only trusted bundles should be allowed to register and get the Provisioning Service. The ServicePermission is used to limit the bundles that can gain access to the Provisioning Service. There is no check of Permission objects to read or modify the provisioning information, so care must be taken not to leak the Provisioning Dictionary received from getInformation method.

No Implement Consumers of this API must not implement this interface

110.11.1.1 public static final String INITIALPROVISIONING_ENTRIES = "InitialProvisioning-Entries"

Name of the header that specifies the type information for the ZIP file entries.

Since 1.2

110.11.1.2 public static final String MIME_BUNDLE = "application/vnd.osgi.bundle"

MIME type to be stored in the extra field of a ZipEntry object for an installable bundle file. Zip entries of this type will be installed in the framework, but not started. The entry will also not be put into the information dictionary.

110.11.1.3 public static final String MIME_BUNDLE_ALT = "application/x-osgi-bundle"

Alternative MIME type to be stored in the extra field of a ZipEntry object for an installable bundle file. Zip entries of this type will be installed in the framework, but not started. The entry will also not be put into the information dictionary. This alternative entry is only for backward compatibility, new applications are recommended to use MIME_BUNDLE, which is an official IANA MIME type.

Since 1.2

110.11.1.4 public static final String MIME_BUNDLE_URL = "text/x-osgi-bundle-url"

MIME type to be stored in the extra field of a ZipEntry for a String that represents a URL for a bundle. Zip entries of this type will be used to install (but not start) a bundle from the URL. The entry will not be put into the information dictionary.

110.11.1.5 public static final String MIME_BYTE_ARRAY = "application/octet-stream"

MIME type to be stored stored in the extra field of a ZipEntry object for byte[] data.

110.11.1.6 public static final String MIME_STRING = "text/plain;charset=utf-8"

MIME type to be stored in the extra field of a ZipEntry object for String data.

110.11.1.7 public static final String PROVISIONING_AGENT_CONFIG = "provisioning.agent.config"

The key to the provisioning information that contains the initial configuration information of the initial Management Agent. The value will be of type byte[].

110.11.1.8 public static final String PROVISIONING_REFERENCE = "provisioning.reference"

The key to the provisioning information that contains the location of the provision data provider. The value must be of type String.

110.11.1.9 public static final String PROVISIONING_ROOTX509 = "provisioning.rootx509"

The key to the provisioning information that contains the root X509 certificate used to establish trust with operator when using HTTPS.

110.11.1.10 public static final String PROVISIONING_RSH_SECRET = "provisioning.rsh.secret"

The key to the provisioning information that contains the shared secret used in conjunction with the RSH protocol.

110.11.1.11 public static final String PROVISIONING_SPID = "provisioning.spid"

The key to the provisioning information that uniquely identifies the Service Platform. The value must be of type String.

110.11.1.12 public static final String PROVISIONING_START_BUNDLE = "provisioning.start.bundle"

The key to the provisioning information that contains the location of the bundle to start with AllPermission. The bundle must have be previously installed for this entry to have any effect.

110.11.1.13 public static final String PROVISIONING_UPDATE_COUNT = "provisioning.update.count"

The key to the provisioning information that contains the update count of the info data. Each set of changes to the provisioning information must end with this value being incremented. The value must be of type Integer. This key/value pair is also reflected in the properties of the ProvisioningService in the service registry.

110.11.1.14 public void addInformation (Dictionary info)

info the set of Provisioning Information key/value pairs to add to the Provisioning Information dictionary. Any keys are values that are of an invalid type will be silently ignored.

□ Adds the key/value pairs contained in info to the Provisioning Information dictionary. This method causes the PROVISIONING_UPDATE_COUNT to be incremented.

110.11.1.15 public void addInformation (ZipInputStream zis) throws IOException

zis the ZipInputStream that will be used to add key/value pairs to the Provisioning Information dictionary and install and start bundles. If a ZipEntry does not have an Extra field that corresponds to one of the four defined MIME types (MIME_STRING, MIME_BYTE_ARRAY,MIME_BUNDLE, and MIME_BUNDLE_URL) in will be silently ignored.

□ Processes the ZipInputStream and extracts information to add to the Provisioning Information dictionary, as well as, install/update and start bundles. This method causes the PROVISIONING_UPDATE_COUNT to be incremented.

Throws IOException – if an error occurs while processing the ZipInputStream. No additions will be made to the Provisioning Information dictionary and no bundles must be started or installed.

110.11.1.16 public Dictionary getInformation ( )

□ Returns a reference to the Provisioning Dictionary. Any change operations (put and remove) to the dictionary will cause an UnsupportedOperationException to be thrown. Changes must be done using the setInformation and addInformation methods of this service.

Returns A reference to the Provisioning Dictionary.

110.11.1.17 public void setInformation (Dictionary info)

info the new set of Provisioning Information key/value pairs. Any keys are values that are of an invalid type will be silently ignored.

□ Replaces the Provisioning Information dictionary with the key/value pairs contained in info. Any key/value pairs not in info will be removed from the Provisioning Information dictionary. This method causes the PROVISIONING_UPDATE_COUNT to be incremented.

110.12 References

[1] *HMAC:* Keyed-Hashing for Message Authentication
http://www.ietf.org/rfc/rfc2104.txt Krawczyk ,et. al. 1997.

[2] *NIST, FIPS PUB 180-1: Secure Hash Standard, April 1995.*

[3] *Hypertext Transfer Protocol - HTTP/1.1*
http://www.ietf.org/rfc/rfc2616.txt *Fielding, R., et. al.*

[4] *Rescorla, E., HTTP over TLS, IETF RFC 2818, May 2000*
http://www.ietf.org/rfc/rfc2818.txt.

[5] *ZIP Archive format*
http://www.pkware.com/support/zip-app-note/archives

[6] *RFC 2396 - Uniform Resource Identifier (URI)*
http://www.ietf.org/rfc/rfc2396.txt

[7] *MIME Types*
http://www.ietf.org/rfc/rfc2046.txt and http://www.iana.org/assignments/media-types

[8] *3DES*
W/ Tuchman, "Hellman Presents No Shortcut Solution to DES," IEEE Spectrum, v. 16, n. 7 July 1979, pp40-41.

[9] *RFC 1423 Part III: Algorithms, Modes, and Identifiers*
http://www.ietf.org/rfc/rfc1423.txt

[10] *PKCS 5*
ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2

[11] *Java Cryptography API (part of Java 1.4)*
http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html/

[12] *SHA-1*
U.S. Government, Proposed Federal Information Processing Standard for Secure Hash Standard, January 1992

[13] *Transport Layer Security*
http://www.ietf.org/rfc/rfc2246.txt, January 1999, The TLS Protocol Version 1.0, T. Dierks & C. Allen.

111 UPnP™ Device Service Specification

Version 1.2

111.1 Introduction

The UPnP Device Architecture specification provides the protocols for a peer-to-peer network. It specifies how to join a network and how devices can be controlled using XML messages sent over HTTP. The OSGi specifications address how code can be download and managed in a remote system. Both standards are therefore fully complimentary. Using an OSGi Service Platform to work with UPnP enabled devices is therefore a very successful combination.

This specification specifies how OSGi bundles can be developed that interoperate with UPnP™ (Universal Plug and Play) devices and UPnP control points. The specification is based on the UPnP Device Architecture and does not further explain the UPnP specifications. The UPnP specifications are maintained by [1] *UPnP Forum*.

UPnP™ is a trademark of the UPnP Implementers Corporation.

111.1.1 Essentials

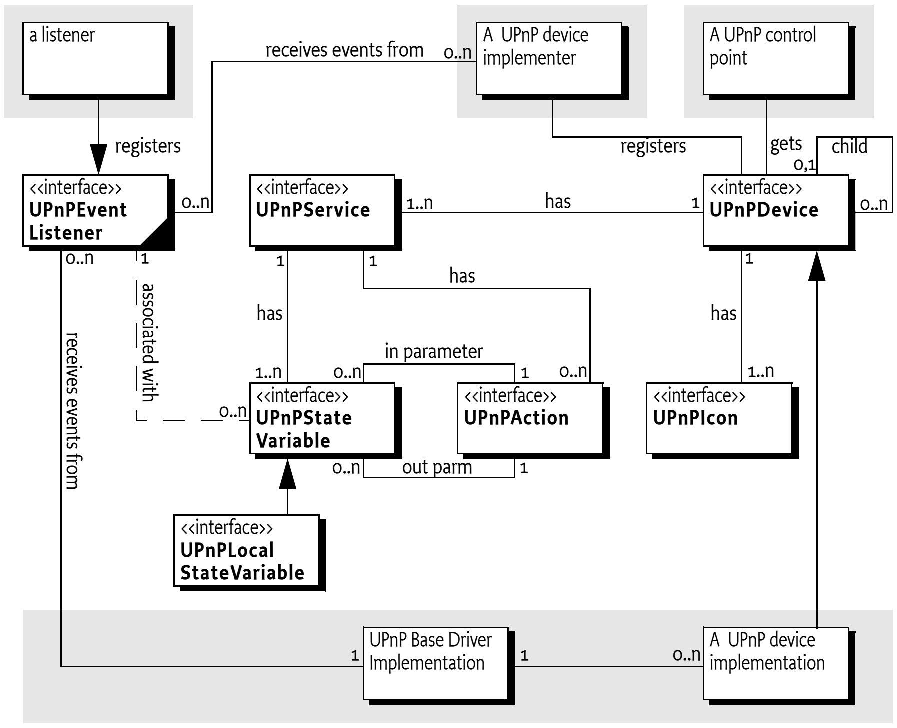
- *Scope* – This specification is limited to device control aspects of the UPnP specifications. Aspects concerning the TCP/IP layer, like DHCP and limited TTL, are not addressed.
- *Transparency* – OSGi services should be made available to networks with UPnP enabled devices in a transparent way.
- *Network Selection* – It must be possible to restrict the use of the UPnP protocols to a selection of the connected networks. For example, in certain cases OSGi services that are UPnP enabled should not be published to the Wide Area Network side of a gateway, nor should UPnP devices be detected on this WAN.
- *Event handling* – Bundles must be able to listen to UPnP events.
- *Export OSGi services as UPnP devices* – Enable bundles that make a service available to UPnP control points.
- *Implement UPnP Control Points* – Enable bundles that control UPnP devices.

111.1.2 Entities

- *UPnP Base Driver* – The bundle that implements the bridge between OSGi and UPnP networks. This entity is not represented as a service.
- *UPnP Root Device* –A physical device can contain one or more root devices. Root devices contain one ore more devices. A root device is modelled with a `UPnPDevice` object, there is no separate interface defined for root devices.
- *UPnP Device* – The representation of a UPnP device. A UPnP device may contain other UPnP devices and UPnP services. This entity is represented by a `UPnPDevice` object. A device can be local (implemented in the Framework) or external (implemented by another device on the net).
- *UPnP Service* –A UPnP device consists of a number of services. A UPnP service has a number of UPnP state variables that can be queried and modified with actions. This concept is represented by a `UPnPService` object.

- *UPnP Action* – A UPnP service is associated with a number of actions that can be performed on that service and that may modify the UPnP state variables. This entity is represented by a UPnPAction object.
- *UPnP State Variable* – A variable associated with a UPnP service, represented by a UPnPStateVariable object.
- *UPnP Local State Variable* – Extends the UPnPStateVariable interface when the state variable is implemented locally. This interface provides access to the actual value.
- *UPnP Event Listener Service* – A listener to events coming from UPnP devices.
- *UPnP Host* – The machine that hosts the code to run a UPnP device or control point.
- *UPnP Control Point* – A UPnP device that is intended to control UPnP devices over a network. For example, a UPnP remote controller.
- *UPnP Icon* – A representation class for an icon associated with a UPnP device.
- *UPnP Exception* – An exception that delivers errors that were discovered in the UPnP layer.
- *UDN* – Unique Device Name, a name that uniquely identifies the a specific device.

Figure 111.1 UPnP Service Specification class Diagram org.osgi.service.upnp package



111.1.3 Operation Summary

To make a UPnP service available to UPnP control points on a network, an OSGi service object must be registered under the UPnPDevice interface with the Framework. The UPnP driver bundle must detect these UPnP Device services and must make them available to the network as UPnP devices using the UPnP protocol.

UPnP devices detected on the local network must be detected and automatically registered under the UPnPDevice interface with the Framework by the UPnP driver implementation bundle.

A bundle that wants to control UPnP devices, for example to implement a UPnP control point, should track UPnP Device services in the OSGi service registry and control them appropriately. Such bundles should not distinguish between resident or remote UPnP Device services.

111.2 UPnP Specifications

The UPnP DA is intended to be used in a broad range of device from the computing (PCs printers), consumer electronics (DVD, TV, radio), communication (phones) to home automation (lighting control, security) and home appliances (refrigerators, coffee makers) domains.

For example, a UPnP TV might announce its existence on a network by broadcasting a message. A UPnP control point on that network can then discover this TV by listening to those announce messages. The UPnP specifications allow the control point to retrieve information about the user interface of the TV. This information can then be used to allow the end user to control the remote TV from the control point, for example turn it on or change the channels.

The UPnP specification supports the following features:

- *Detect and control a UPnP standardized device.* In this case the control point and the remote device share a priori knowledge about how the device should be controlled. The UPnP Forum intends to define a large number of these standardized devices.
- *Use a user interface description.* A UPnP control point receives enough information about a device and its services to automatically build a user interface for it.
- *Programmatic Control.* A program can directly control a UPnP device without a user interface. This control can be based on detected information about the device or through a priori knowledge of the device type.
- *Allows the user to browse a web page supplied by the device.* This web page contains a user interface for the device that be directly manipulated by the user. However, this option is not well defined in the UPnP Device Architecture specification and is not tested for compliance.

The UPnP Device Architecture specification and the OSGi Service Platform provide *complementary* functionality. The UPnP Device Architecture specification is a data communication protocol that does not specify where and how programs execute. That choice is made by the implementations. In contrast, the OSGi Service Platform specifies a (managed) execution point and does not define what protocols or media are supported. The UPnP specification and the OSGi specifications are fully complementary and do not overlap.

From the OSGi perspective, the UPnP specification is a communication protocol that can be implemented by one or more bundles. This specification therefore defines the following:

- How an OSGi bundle can implement a service that is exported to the network via the UPnP protocols.
- How to find and control services that are available on the local network.

The UPnP specifications related to the assignment of IP addresses to new devices on the network or auto-IP self configuration should be handled at the operating system level. Such functions are outside the scope of this specification.

111.2.1 UPnP Base Driver

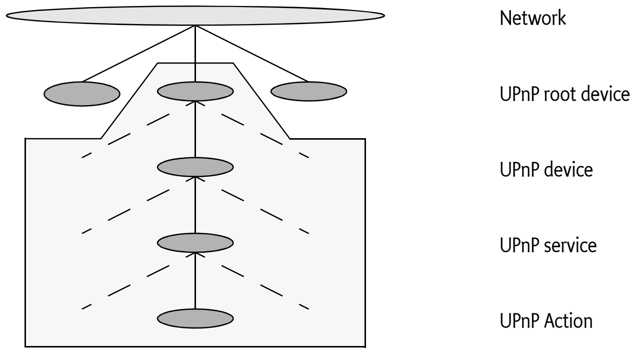
The functionality of the UPnP service is implemented in a UPnP *base driver.* This is a bundle that implements the UPnP protocols and handles the interaction with bundles that use the UPnP devices. A UPnP base driver bundle must provide the following functions:

- Discover UPnP devices on the network and map each discovered device into an OSGi registered UPnP Device service.
- Present UPnP marked services that are registered with the OSGi Framework on one or more networks to be used by other computers.

111.3 UPnP Device

The principle entity of the UPnP specification is the UPnP device. There is a UPnP *root device* that represents a physical appliance, such as a complete TV. The root device contains a number of sub-devices. These might be the tuner, the monitor, and the sound system. Each sub-device is further composed of a number of UPnP services. A UPnP service represents some functional unit in a device. For example, in a TV tuner it can represent the TV channel selector. Figure 111.2 on page 192 illustrates this hierarchy.

Figure 111.2 UPnP device hierarchy



Each UPnP service can be manipulated with a number of UPnP actions. UPnP actions can modify the state of a UPnP state variable that is associated with a service. For example, in a TV there might be a state variable *volume*. There are then actions to set the volume, to increase the volume, and to decrease the volume.

111.3.1 Root Device

The UPnP root device is registered as a UPnP Device service with the Framework, as well as all its sub-devices. Most applications will work with sub-devices, and, as a result, the children of the root device are registered under the UPnPDevice interface.

UPnP device properties are defined per sub-device in the UPnP specification. These properties must be registered with the OSGi Framework service registry so they are searchable.

Bundles that want to handle the UPnP device hierarchy can use the registered service properties to find the parent of a device (which is another registered UPnPDevice).

The following service registration properties can be used to discover this hierarchy:

- PARENT_UDN – (String) The Universal Device Name (UDN) of the parent device. A root device most not have this property registered. Type is a String object.
- CHILDREN_UDN – (String[]) An array of UDNs of this device's children.

111.3.2 Exported Versus Imported Devices

Both imported (from the network to the OSGi service registry) and exported (from the service registry to the network) UPnPDevice services must have the same representation in the OSGi Service Platform for identical devices. For example, if an OSGi UPnP Device service is exported as a UPnP device from an OSGi Service Platform to the network, and it is imported into another OSGi Service Platform, the object representation should be equal. Application bundles should therefore be able to interact with imported and exported forms of the UPnP device in the same manner.

Imported and exported UPnP devices differ only by two marker properties that can be added to the service registration. One marker, DEVICE_CATEGORY, should typically be set only on imported devices. By not setting DEVICE_CATEGORY on internal UPnP devices, the Device Manager does not try to refine these devices (See the *Device Access Specification* on page 63 for more information about the Device Manager). If the device service does not implement the Device interface and does not have the DEVICE_CATEGORY property set, it is not considered a *device* according to the Device Access Specification.

The other marker, UPNP_EXPORT, should only be set on internally created devices that the bundle developer wants to export. By not setting UPNP_EXPORT on registered UPnP Device services, the UPnP Device service can be used by internally created devices that should not be exported to the network. This allows UPnP devices to be simulated within an OSGi Service Platform without announcing all of these devices to any networks.

The UPNP_EXPORT service property has no defined type, any value is correct.

111.3.3 Icons

A UPnP device can optionally support an icon. The purpose of this icon is to identify the device on a UPnP control point. UPnP control points can be implemented in large computers like PC's or simple devices like a remote control. However, the graphic requirements for these UPnP devices differ tremendously. The device can, therefore, export a number of icons of different size and depth.

In the UPnP specifications, an icon is represented by a URL that typically refers to the device itself. In this specification, a list of icons is available from the UPnP Device service.

In order to obtain localized icons, the method getIcons(String) can be used to obtain different versions. If the locale specified is a null argument, then the call returns the icons of the default locale of the called device (not the default locale of the UPnP control point).When a bundle wants to access the icon of an imported UPnP device, the UPnP driver gets the data and presents it to the application through an input stream.

A bundle that needs to export a UPnP Device service with one ore more icons must provide an implementation of the UPnPIcon interface. This implementation must provide an InputStream object to the actual icon data. The UPnP driver bundle must then register this icon with an HTTP server and include the URL to the icon with the UPnP device data at the appropriate place.

111.4 Device Category

UPnP Device services are devices in the context of the Device Manager. This means that these services need to register with a number of properties to participate in driver refinement. The value for UPnP devices is defined in the UPnPDevice constant DEVICE_CATEGORY. The value is UPnP. The UPnPDevice interface contains a number of constants for matching values. Refer to *MATCH_GENERIC* on page 201 for further information.

111.5 UPnPService

A UPnP Device contains a number of UPnPService objects. UPnPService objects combine zero or more actions and one or more state variables.

111.5.1 State Variables

The UPnPStateVariable interface encapsulates the properties of a UPnP state variable. In addition to the properties defined by the UPnP specification, a state variable is also mapped to a Java data type. The Java data type is used when an event is generated for this state variable and when an action is performed containing arguments related to this state variable. There must be a strict correspondence between the UPnP data type and the Java data type so that bundles using a particular UPnP device profile can predict the precise Java data type.

The function QueryStateVariable defined in the UPnP specification has been deprecated and is therefore not implemented. It is recommended to use the UPnP event mechanism to track UPnP state variables.

Additionally, a UPnPStateVariableobject can also implement the UPnPLocalStateVariable interface if the device is implemented locally. That is, the device is not imported from the network. The UPnPLocalStateVariable interface provides a getCurrentValue() method that provides direct access to the actual value of the state variable.

111.6 Working With a UPnP Device

The UPnP driver must register all discovered UPnP devices in the local networks. These devices are registered under a UPnPDevice interface with the OSGi Framework.

Using a remote UPnP device thus involves tracking UPnP Device services in the OSGi service registry. The following code illustrates how this can be done. The sample Controller class extends the ServiceTracker class so that it can track all UPnP Device services and add them to a user interface, such as a remote controller application.

```
class Controller extends ServiceTracker {
   UI      ui;

   Controller( BundleContext context ) {
      super( context, UPnPDevice.class.getName(), null );
   }
   public Object addingService( ServiceReference ref ) {
      UPnPDevice dev = (UPnPDevice)super.addingService(ref);
      ui.addDevice( dev );
      return dev;
   }
   public void removedService( ServiceReference ref,
      Object dev ) {
      ui.removeDevice( (UPnPDevice) dev );
   }
   ...
}
```

111.7 Implementing a UPnP Device

OSGi services can also be exported as UPnP devices to the local networks, in a way that is transparent to typical UPnP devices. This allows developers to bridge legacy devices to UPnP networks. A bundle should perform the following to export an OSGi service as a UPnP device:

- Register an UPnP Device service with the registration property UPNP_EXPORT.
- Use the registration property PRESENTATION_URL to provide the presentation page. The service implementer must register its own servlet with the Http Service to serve out this interface. This URL must point to that servlet.

There can be multiple UPnP root devices hosted by one OSGi platform. The relationship between the UPnP devices and the OSGi platform is defined by the PARENT_UDN and CHILDREN_UDN service properties. The bundle registering those device services must make sure these properties are set accordingly.

Devices that are implemented on the OSGi Service Platform (in contrast with devices that are imported from the network) should use the UPnPLocalStateVariable interface for their state variables instead of the UPnPStateVariable interface. This interface provides programmatic access to the actual value of the state variable as maintained by the device specific code.

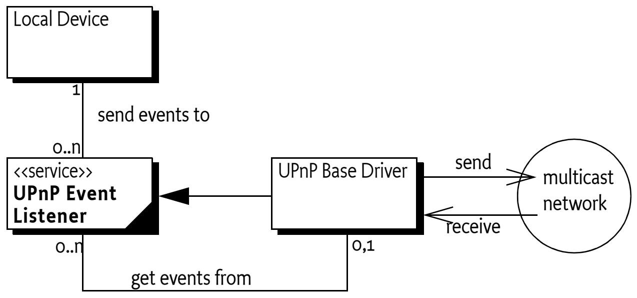
111.8 Event API

There are two distinct event directions for the UPnP Service specification.

- External events from the network must be dispatched to listeners inside the OSGi Service Platforms. The UPnP Base driver is responsible for mapping the network events to internal listener events.
- Implementations of UPnP devices must send out events to local listeners as well as cause the transmission of the UPnP network events.

UPnP events are sent using the whiteboard model, in which a bundle interested in receiving the UPnP events registers an object implementing the UPnPEventListener interface. A filter can be set to limit the events for which a bundle is notified. The UPnP Base driver must register a UPnP Event Lister without filter that receives all events.

Figure 111.3 Event Dispatching for Local and External Devices



If a service is registered with a property named upnp.filter with the value of an instance of an Filter object, the listener is only notified for matching events (This is a Filter object and not a String object because it allows the InvalidSyntaxException to be thrown in the client and not the UPnP driver bundle).

The filter might refer to any valid combination of the following pseudo properties for event filtering:

- UPnPDevice.UDN – (UPnP.device.UDN/String) Only events generated by services contained in the specific device are delivered. For example: (UPnP.device.UDN=uuid:Upnp-TVEmulator-1_0-1234567890001)
- UPnPDevice.TYPE– (UPnP.device.type/String or String[]) Only events generated by services contained in a device of the given type are delivered. For example: (UPnP.device.type=urn:schemas-upnp-org:device:tvdevice:1)
- UPnPService.ID – (UPnP.service.id/String) Service identity. Only events generated by services matching the given service ID are delivered.
- UPnPService.TYPE – (UPnP.service.type/String or String[]) Only events generated by services of of the given type are delivered.

If an event is generated by either a local device or via the base driver for an external device, the notifyUPnPEvent(String,String,Dictionary) method is called on all registered UPnPEventListener services for which the optional filter matches for that event. If no filter is specified, all events must be delivered. If the filter does not match, the UPnP Driver must not call the UPnP Event Listener service. The way events must be delivered is the same as described in *Delivering Events* on page 106 of the Core specification.

One or multiple events are passed as parameters to the notifyUPnPEvent(String,String,Dictionary) method. The Dictionary object holds a pair of UpnPStateVariable objects that triggered the event and an Object for the new value of the state variable.

111.8.1 Initial Event Delivery

Special care must be taken with the initial subscription to events. According to the UPnP specification, when a client subscribes for notification of events for the first time, the device sends out a number of events for each state variable, indicating the current value of each state variable. This behavior simplifies the synchronization of a device and an event-driven client.

The UPnP Base Driver must mimic this event distribution on behalf of external devices. It must therefore remember the values of the state variables of external devices. A UPnP Device implementation must send out these initial events for each state variable they have a value for.

The UPnP Base Driver must have stored the last event from the device and retransmit the value over the multicast network. The UPnP Driver must register an event listener without any filter for this purpose.

The call to the listener's notification method must be done asynchronously.

111.9 UPnP Events and Event Admin service

UPnP events must be delivered asynchronously to the Event Admin service by the UPnP implementation, if present. UPnP events have the following topic:

```
org/osgi/service/upnp/UPnPEvent
```

The properties of a UPnP event are the following:

- upnp.deviceId – (String) The identity as defined by UPnPDevice.UDN of the device sending the event.
- upnp.serviceId – (String) The identity of the service sending the events.
- upnp.events – (Dictionary) A Dictionary object containing the new values for the state variables that have changed.

111.10 Localization

All values of the UPnP properties are obtained from the device using the device's default locale. If an application wants to query a set of localized property values, it has to use the method getDescriptions(String). For localized versions of the icons, the method getIcons(String) is to be used.

111.11 Dates and Times

The UPnP specification uses different types for date and time concepts. An overview of these types is given in Table 111.1 on page 197.

Table 111.1 Mapping UPnP Date/Time types to Java

UPnP Type	Class	Example	Value (TZ=CEST= +0200)
date	Date	1985-04-12	Sun April 12 00:00:00 CEST 1985
dateTime	Date	1985-04-12T10:15:30	Sun April 12 10:15:30 CEST 1985
dateTime.tz	Date	1985-04-12T10:15:30+0400	Sun April 12 08:15:30 CEST 1985
time	Long	23:20:50	84.050.000 (ms)
time.tz	Long	23:20:50+0300	1.250.000 (ms)

The UPnP specification points to [2] *XML Schema*. In this standard, [3] *ISO 8601 Date And Time formats* are referenced. The mapping is not completely defined which means that the this OSGi UPnP specification defines a complete mapping to Java classes. The UPnP types date, dateTime and dateTime.tz are represented as a Date object. For the date type, the hours, minutes and seconds must all be zero.

The UPnP types time and time.tz are represented as a Long object that represents the number of ms since midnight. If the time wraps to the next day due to a time zone value, then the final value must be truncated to modulo 86.400.000.

See also *TYPE_DATE* on page 209 and further.

111.12 UPnP Exception

The UPnP Exception can be thrown when a UPnPAction is invoked. This exception contains information about the different UPnP layers. The following errors are defined:

INVALID_ACTION – (401) No such action could be found.

INVALID_ARGS – (402) Invalid argument.

INVALID_SEQUENCE_NUMBER – (403) Out of synchronization.

INVALID_VARIABLE – (404) State variable not found.

DEVICE_INTERNAL_ERROR – (501) Internal error.

Further errors are categorized as follows:

- *Common Action Errors* – In the range of 600-69, defined by the UPnP Forum Technical Committee.
- *Action Specific Errors* – In the range of 700-799, defined by the UPnP Forum Working Committee.
- *Non-Standard Action Specific Errors* – In the range of 800-899. Defined by vendors.

111.13 Configuration

In order to provide a standardized way to configure a UPnP driver bundle, the Configuration Admin property upnp.ssdp.address is defined.

The value is a String[] with a list of IP addresses, optionally followed with a colon (':', \u003A) and a port number. For example:

```
239.255.255.250:1900
```

Those addresses define the interfaces which the UPnP driver is operating on. If no SSDP address is specified, the default assumed will be 239.255.255.250:1900. If no port is specified, port 1900 is assumed as default.

111.14 Networking considerations

111.14.1 The UPnP Multicasts

The operating system must support multicasting on the selected network device. In certain cases, a multicasting route has to be set in the operating system routing table.

These configurations are highly dependent on the underlying operating system and beyond the scope of this specification.

111.15 Security

The UPnP specification is based on HTTP and uses plain text SOAP (XML) messages to control devices. For this reason, it does not provide any inherent security mechanisms. However, the UPnP specification is based on the exchange of XML files and not code. This means that at least worms and viruses cannot be implemented using the UPnP protocols.

However, a bundle registering a UPnP Device service is represented on the outside network and has the ability to communicate. The same is true for getting a UPnP Device service. It is therefore recommended that `ServicePermission[UPnPDevice|UPnPEventListener, REGISTER|GET]` be used sparingly and only for bundles that are trusted.

111.16 Changes

- Added a new constructor for a UPnP Exception with a nested cause
- Renamed the `getUPnPError_Code` method to `getUPnPErrorCode`, the old one is left deprecated.

111.17 org.osgi.service.upnp

UPnP Package Version 1.2.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.upnp; version="[1.2,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.upnp; version="[1.2,1.3)"
```

111.17.1 Summary

- `UPnPAction` – A UPnP action.
- `UPnPDevice` – Represents a UPnP device.
- `UPnPEventListener` – UPnP Events are mapped and delivered to applications according to the OSGi whiteboard model.
- `UPnPException` – There are several defined error situations describing UPnP problems while a control point invokes actions to UPnPDevices.
- `UPnPIcon` – A UPnP icon representation.
- `UPnPLocalStateVariable` – A local UPnP state variable which allows the value of the state variable to be queried.
- `UPnPService` – A representation of a UPnP Service.

- UPnPStateVariable – The meta-information of a UPnP state variable as declared in the device's service state table (SST).

111.17.2 Permissions

111.17.3 public interface UPnPAction

A UPnP action. Each UPnP service contains zero or more actions. Each action may have zero or more UPnP state variables as arguments.

111.17.3.1 public String[] getInputArgumentNames ( )

□ Lists all input arguments for this action.

Each action may have zero or more input arguments.

This method must continue to return the action input argument names after the UPnP action has been removed from the network.

Returns Array of input argument names or null if no input arguments.

See Also UPnPStateVariable

111.17.3.2 public String getName ( )

□ Returns the action name. The action name corresponds to the name field in the actionList of the service description.

- For standard actions defined by a UPnP Forum working committee, action names must not begin with X_ nor A_.
- For non-standard actions specified by a UPnP vendor and added to a standard service, action names must begin with X_.

This method must continue to return the action name after the UPnP action has been removed from the network.

Returns Name of action, must not contain a hyphen character or a hash character

111.17.3.3 public String[] getOutputArgumentNames ( )

□ List all output arguments for this action.

This method must continue to return the action output argument names after the UPnP action has been removed from the network.

Returns Array of output argument names or null if there are no output arguments.

See Also UPnPStateVariable

111.17.3.4 public String getReturnArgumentName ( )

□ Returns the name of the designated return argument.

One of the output arguments can be flagged as a designated return argument.

This method must continue to return the action return argument name after the UPnP action has been removed from the network.

Returns The name of the designated return argument or null if none is marked.

111.17.3.5 public UPnPStateVariable getStateVariable (String argumentName)

argumentName The name of the UPnP action argument.

□ Finds the state variable associated with an argument name. Helps to resolve the association of state variables with argument names in UPnP actions.

Returns State variable associated with the named argument or null if there is no such argument.

Throws IllegalStateException – if the UPnP action has been removed from the network.

See Also UPnPStateVariable

111.17.3.6 public Dictionary invoke (Dictionary args) throws Exception

args A Dictionary of arguments. Must contain the correct set and type of arguments for this action. May be null if no input arguments exist.

□ Invokes the action. The input and output arguments are both passed as Dictionary objects. Each entry in the Dictionary object has a String object as key representing the argument name and the value is the argument itself. The class of an argument value must be assignable from the class of the associated UPnP state variable. The input argument Dictionary object must contain exactly those arguments listed by getInputArguments method. The output argument Dictionary object will contain exactly those arguments listed by getOutputArguments method.

Returns A Dictionary with the output arguments. null if the action has no output arguments.

Throws UPnPException – A UPnP error has occurred.

IllegalStateException – if the UPnP action has been removed from the network.

Exception – The execution fails for some reason.

See Also UPnPStateVariable

111.17.4 public interface UPnPDevice

Represents a UPnP device. For each UPnP root and embedded device, an object is registered with the framework under the UPnPDevice interface.

The relationship between a root device and its embedded devices can be deduced using the UPnPDevice.CHILDREN_UDN and UPnPDevice.PARENT_UDN service registration properties.

The values of the UPnP property names are defined by the UPnP Forum.

All values of the UPnP properties are obtained from the device using the device's default locale.

If an application wants to query for a set of localized property values, it has to use the method UPnPDevice.getDescriptions(String locale).

111.17.4.1 public static final String CHILDREN_UDN = "UPnP.device.childrenUDN"

The property key that must be set for all devices containing other embedded devices.

The value is an array of UDNs for each of the device's children (String[]). The array contains UDNs for the immediate descendants only.

If an embedded device in turn contains embedded devices, the latter are not included in the array.

The UPnP Specification does not encourage more than two levels of nesting.

The property is not set if the device does not contain embedded devices.

The property is of type String[]. Value is "UPnP.device.childrenUDN"

111.17.4.2 public static final String DEVICE_CATEGORY = "UPnP"

Constant for the value of the service property DEVICE_CATEGORY used for all UPnP devices. Value is "UPnP".

See Also org.osgi.service.device.Constants.DEVICE_CATEGORY

111.17.4.3 public static final String FRIENDLY_NAME = "UPnP.device.friendlyName"

Mandatory property key for a short user friendly version of the device name. The property value holds a String object with the user friendly name of the device. Value is "UPnP.device.friendlyName".

111.17.4.4 public static final String ID = "UPnP.device.UDN"

Property key for the Unique Device ID property. This property is an alias to UPnPDevice.UDN. It is merely provided for reasons of symmetry with the UPnPService.ID property. The value of the property is a String object of the Device UDN. The value of the key is "UPnP.device.UDN".

111.17.4.5 public static final String MANUFACTURER = "UPnP.device.manufacturer"

Mandatory property key for the device manufacturer's property. The property value holds a String representation of the device manufacturer's name. Value is "UPnP.device.manufacturer".

111.17.4.6 public static final String MANUFACTURER_URL = "UPnP.device.manufacturerURL"

Optional property key for a URL to the device manufacturers Web site. The value of the property is a String object representing the URL. Value is "UPnP.device.manufacturerURL".

111.17.4.7 public static final int MATCH_GENERIC = 1

Constant for the UPnP device match scale, indicating a generic match for the device. Value is 1.

111.17.4.8 public static final int MATCH_MANUFACTURER_MODEL = 7

Constant for the UPnP device match scale, indicating a match with the device model. Value is 7.

111.17.4.9 public static final int MATCH_MANUFACTURER_MODEL_REVISION = 15

Constant for the UPnP device match scale, indicating a match with the device revision. Value is 15.

111.17.4.10 public static final int MATCH_MANUFACTURER_MODEL_REVISION_SERIAL = 31

Constant for the UPnP device match scale, indicating a match with the device revision and the serial number. Value is 31.

111.17.4.11 public static final int MATCH_TYPE = 3

Constant for the UPnP device match scale, indicating a match with the device type. Value is 3.

111.17.4.12 public static final String MODEL_DESCRIPTION = "UPnP.device.modelDescription"

Optional (but recommended) property key for a String object with a long description of the device for the end user. The value is "UPnP.device.modelDescription".

111.17.4.13 public static final String MODEL_NAME = "UPnP.device.modelName"

Mandatory property key for the device model name. The property value holds a String object giving more information about the device model. Value is "UPnP.device.modelName".

111.17.4.14 public static final String MODEL_NUMBER = "UPnP.device.modelNumber"

Optional (but recommended) property key for a String class typed property holding the model number of the device. Value is "UPnP.device.modelNumber".

111.17.4.15 public static final String MODEL_URL = "UPnP.device.modelURL"

Optional property key for a String typed property holding a string representing the URL to the Web site for this model. Value is "UPnP.device.modelURL".

111.17.4.16 public static final String PARENT_UDN = "UPnP.device.parentUDN"

The property key that must be set for all embedded devices. It contains the UDN of the parent device. The property is not set for root devices. The value is "UPnP.device.parentUDN".

111.17.4.17 public static final String PRESENTATION_URL = "UPnP.presentationURL"

Optional (but recommended) property key for a `String` typed property holding a string representing the URL to a device representation Web page. Value is "UPnP.presentationURL".

111.17.4.18 public static final String SERIAL_NUMBER = "UPnP.device.serialNumber"

Optional (but recommended) property key for a `String` typed property holding the serial number of the device. Value is "UPnP.device.serialNumber".

111.17.4.19 public static final String TYPE = "UPnP.device.type"

Property key for the UPnP Device Type property. Some standard property values are defined by the Universal Plug and Play Forum. The type string also includes a version number as defined in the UPnP specification. This property must be set.

For standard devices defined by a UPnP Forum working committee, this must consist of the following components in the given order separated by colons:

- `urn`
- schemas-upnp-org
- `device`
- a device type suffix
- an integer device version

For non-standard devices specified by UPnP vendors following components must be specified in the given order separated by colons:

- `urn`
- an ICANN domain name owned by the vendor
- `device`
- a device type suffix
- an integer device version

To allow for backward compatibility the UPnP driver must automatically generate additional Device Type property entries for smaller versions than the current one. If for example a device announces its type as version 3, then properties for versions 2 and 1 must be automatically generated.

In the case of exporting a UPnPDevice, the highest available version must be announced on the network.

Syntax Example: `urn:schemas-upnp-org:device:deviceType:v`

The value is "UPnP.device.type".

111.17.4.20 public static final String UDN = "UPnP.device.UDN"

Property key for the Unique Device Name (UDN) property. It is the unique identifier of an instance of a `UPnPDevice`. The value of the property is a `String` object of the Device UDN. Value of the key is "UPnP.device.UDN". This property must be set.

111.17.4.21 public static final String UPC = "UPnP.device.UPC"

Optional property key for a `String` typed property holding the Universal Product Code (UPC) of the device. Value is "UPnP.device.UPC".

111.17.4.22 public static final String UPNP_EXPORT = "UPnP.export"

The `UPnP.export` service property is a hint that marks a device to be picked up and exported by the UPnP Service. Imported devices do not have this property set. The registered property requires no value.

The UPNP_EXPORT string is "UPnP.export".

111.17.4.23 public Dictionary getDescriptions (String locale)

locale A language tag as defined by RFC 1766 and maintained by ISO 639. Examples include "de", "en" or "en-US". The default locale of the device is specified by passing a null argument.

□ Get a set of localized UPnP properties. The UPnP specification allows a device to present different device properties based on the client's locale. The properties used to register the UPnPDevice service in the OSGi registry are based on the device's default locale. To obtain a localized set of the properties, an application can use this method.

Not all properties might be available in all locales. This method does **not** substitute missing properties with their default locale versions.

This method must continue to return the properties after the UPnP device has been removed from the network.

Returns Dictionary mapping property name Strings to property value Strings

111.17.4.24 public UPnPIcon[] getIcons (String locale)

locale A language tag as defined by RFC 1766 and maintained by ISO 639. Examples include "de", "en" or "en-US". The default locale of the device is specified by passing a null argument.

□ Lists all icons for this device in a given locale. The UPnP specification allows a device to present different icons based on the client's locale.

Returns Array of icons or null if no icons are available.

Throws IllegalStateException – if the UPnP device has been removed from the network.

111.17.4.25 public UPnPService getService (String serviceId)

serviceId The service id

□ Locates a specific service by its service id.

Returns The requested service or null if not found.

Throws IllegalStateException – if the UPnP device has been removed from the network.

111.17.4.26 public UPnPService[] getServices ( )

□ Lists all services provided by this device.

Returns Array of services or null if no services are available.

Throws IllegalStateException – if the UPnP device has been removed from the network.

111.17.5 public interface UPnPEventListener

UPnP Events are mapped and delivered to applications according to the OSGi whiteboard model. An application that wishes to be notified of events generated by a particular UPnP Device registers a service extending this interface.

The notification call from the UPnP Service to any UPnPEventListener object must be done asynchronous with respect to the originator (in a separate thread).

Upon registration of the UPnP Event Listener service with the Framework, the service is notified for each variable which it listens for with an initial event containing the current value of the variable. Subsequent notifications only happen on changes of the value of the variable.

A UPnP Event Listener service filter the events it receives. This event set is limited using a standard framework filter expression which is specified when the listener service is registered.

The filter is specified in a property named "upnp.filter" and has as a value an object of type org.osgi.framework.Filter.

When the Filter is evaluated, the folowing keywords are recognized as defined as literal constants in the UPnPDevice class.

The valid subset of properties for the registration of UPnP Event Listener services are:

- UPnPDevice.TYPE-- Which type of device to listen for events.
- UPnPDevice.ID-- The ID of a specific device to listen for events.
- UPnPService.TYPE-- The type of a specific service to listen for events.
- UPnPService.ID-- The ID of a specific service to listen for events.

111.17.5.1 public static final String UPNP_FILTER = "upnp.filter"

Key for a service property having a value that is an object of type org.osgi.framework.Filter and that is used to limit received events.

111.17.5.2 public void notifyUPnPEvent (String deviceId , String serviceId , Dictionary events)

deviceId ID of the device sending the events

serviceId ID of the service sending the events

events Dictionary object containing the new values for the state variables that have changed.

□ Callback method that is invoked for received events. The events are collected in a Dictionary object. Each entry has a String key representing the event name (= state variable name) and the new value of the state variable. The class of the value object must match the class specified by the UPnP State Variable associated with the event. This method must be called asynchronously

111.17.6 public class UPnPException extends Exception

There are several defined error situations describing UPnP problems while a control point invokes actions to UPnPDevices.

Since 1.1

111.17.6.1 public static final int DEVICE_INTERNAL_ERROR = 501

The invoked action failed during execution.

111.17.6.2 public static final int INVALID_ACTION = 401

No Action found by that name at this service.

111.17.6.3 public static final int INVALID_ARGS = 402

Not enough arguments, too many arguments with a specific name, or one of more of the arguments are of the wrong type.

111.17.6.4 public static final int INVALID_SEQUENCE_NUMBER = 403

The different end-points are no longer in synchronization.

111.17.6.5 public static final int INVALID_VARIABLE = 404

Refers to a non existing variable.

111.17.6.6 public UPnPException (int errorCode , String errorDescription)

errorCode error code which defined by UPnP Device Architecture V1.0.

errorDescription error description which explain the type of problem.

□ This constructor creates a UPnPException on the specified error code and error description.

111.17.6.7 public UPnPException (int errorCode , String errorDescription , Throwable errorCause)

errorCode error code which defined by UPnP Device Architecture V1.0.

errorDescription error description which explain the type of the problem.

errorCause cause of that `UPnPException`.

□ This constructor creates a `UPnPException` on the specified error code, error description and error cause.

Since 1.2

111.17.6.8 public int getUPnPError_Code ( )

□ Returns the UPnPError Code occurred by UPnPDevices during invocation.

Returns The UPnPErrorCode defined by a UPnP Forum working committee or specified by a UPnP vendor.

Deprecated As of version 1.2, replaced by `getUPnPErrorCode()`

111.17.6.9 public int getUPnPErrorCode ( )

□ Returns the UPnP Error Code occurred by UPnPDevices during invocation.

Returns The UPnPErrorCode defined by a UPnP Forum working committee or specified by a UPnP vendor.

Since 1.2

111.17.7 public interface UPnPIcon

A UPnP icon representation. Each UPnP device can contain zero or more icons.

111.17.7.1 public int getDepth ( )

□ Returns the color depth of the icon in bits.

This method must continue to return the icon depth after the UPnP device has been removed from the network.

Returns The color depth in bits. If the actual color depth of the icon is unknown, -1 is returned.

111.17.7.2 public int getHeight ( )

□ Returns the height of the icon in pixels. If the actual height of the icon is unknown, -1 is returned.

This method must continue to return the icon height after the UPnP device has been removed from the network.

Returns The height in pixels, or -1 if unknown.

111.17.7.3 public InputStream getInputStream ( ) throws IOException

□ Returns an `InputStream` object for the icon data. The `InputStream` object provides a way for a client to read the actual icon graphics data. The number of bytes available from this `InputStream` object can be determined via the `getSize()` method. The format of the data encoded can be determined by the MIME type available via the `getMimeType()` method.

Returns An InputStream to read the icon graphics data from.

Throws `IOException` – If the `InputStream` cannot be returned.

`IllegalStateException` – if the UPnP device has been removed from the network.

See Also `UPnPIcon.getMimeType()`

111.17.7.4 public String getMimeType ( )

□ Returns the MIME type of the icon. This method returns the format in which the icon graphics, read from the `InputStream` object obtained by the `getInputStream()` method, is encoded.

The format of the returned string is in accordance to RFC2046. A list of valid MIME types is maintained by the IANA (http://www.iana.org/assignments/media-types/) .

Typical values returned include: “image/jpeg” or “image/gif”

This method must continue to return the icon MIME type after the UPnP device has been removed from the network.

Returns The MIME type of the encoded icon.

111.17.7.5 public int getSize ( )

□ Returns the size of the icon in bytes. This method returns the number of bytes of the icon available to read from the `InputStream` object obtained by the `getInputStream()` method. If the actual size can not be determined, -1 is returned.

Returns The icon size in bytes, or -1 if the size is unknown.

Throws `IllegalStateException` – if the UPnP device has been removed from the network.

111.17.7.6 public int getWidth ( )

□ Returns the width of the icon in pixels. If the actual width of the icon is unknown, -1 is returned.

This method must continue to return the icon width after the UPnP device has been removed from the network.

Returns The width in pixels, or -1 if unknown.

111.17.8 public interface UPnPLocalStateVariable extends UPnPStateVariable

A local UPnP state variable which allows the value of the state variable to be queried.

Since 1.1

111.17.8.1 public Object getCurrentValue ( )

□ This method will keep the current values of UPnPStateVariables of a UPnPDevice whenever UPnP-StateVariable’s value is changed , this method must be called.

Returns `Object` current value of UPnPStateVariable. if the current value is initialized with the default value defined UPnP service description.

Throws `IllegalStateException` – if the UPnP state variable has been removed.

111.17.9 public interface UPnPService

A representation of a UPnP Service. Each UPnP device contains zero or more services. The UPnP description for a service defines actions, their arguments, and event characteristics.

111.17.9.1 public static final String ID = “UPnP.service.id”

Property key for the optional service id. The service id property is used when registering UPnP Device services or UPnP Event Listener services. The value of the property contains a `String` array (`String[]`) of service ids. A UPnP Device service can thus announce what service ids it contains. A UPnP Event Listener service can announce for what UPnP service ids it wants notifications. A service id does **not** have to be universally unique. It must be unique only within a device. A `null` value is a wildcard, matching **all** services. The value is “UPnP.service.id”.

111.17.9.2 public static final String TYPE = "UPnP.service.type"

Property key for the optional service type uri. The service type property is used when registering UPnP Device services and UPnP Event Listener services. The property contains a `String` array (`String[]`) of service types. A UPnP Device service can thus announce what types of services it contains. A UPnP Event Listener service can announce for what type of UPnP services it wants notifications. The service version is encoded in the type string as specified in the UPnP specification. A `null` value is a wildcard, matching **all** service types. Value is "UPnP.service.type".

See Also `UPnPService.getType()`

111.17.9.3 public UPnPAction getAction (String name)

name Name of action. Must not contain hyphen or hash characters. Should be < 32 characters.

□ Locates a specific action by name. Looks up an action by its name.

Returns The requested action or `null` if no action is found.

Throws `IllegalStateException` – if the UPnP service has been removed from the network.

111.17.9.4 public UPnPAction[] getActions ( )

□ Lists all actions provided by this service.

Returns Array of actions (`UPnPAction[]`)or `null` if no actions are defined for this service.

Throws `IllegalStateException` – if the UPnP service has been removed from the network.

111.17.9.5 public String getId ( )

□ Returns the `serviceId` field in the UPnP service description.

For standard services defined by a UPnP Forum working committee, the serviceId must contain the following components in the indicated order:

- `urn:upnp-org:serviceId:`
- service ID suffix

Example: `urn:upnp-org:serviceId:serviceID`.

Note that `upnp-org` is used instead of `schemas-upnp-org` in this example because an XML schema is not defined for each serviceId.

For non-standard services specified by UPnP vendors, the serviceId must contain the following components in the indicated order:

- `urn:`
- ICANN domain name owned by the vendor
- `:serviceId:`
- service ID suffix

Example: `urn:domain-name:serviceId:serviceID`.

This method must continue to return the service id after the UPnP service has been removed from the network.

Returns The service ID suffix defined by a UPnP Forum working committee or specified by a UPnP vendor. Must be <= 64 characters. Single URI.

111.17.9.6 public UPnPStateVariable getStateVariable (String name)

name Name of the State Variable

□ Gets a `UPnPStateVariable` objects provided by this service by name

Returns State variable or `null` if no such state variable exists for this service.

Throws `IllegalStateException` – if the UPnP service has been removed from the network.

111.17.9.7 public UPnPStateVariable[] getStateVariables ( )

□ Lists all `UPnPStateVariable` objects provided by this service.

Returns Array of state variables or `null` if none are defined for this service.

Throws `IllegalStateException` – if the UPnP service has been removed from the network.

111.17.9.8 public String getType ( )

□ Returns the `serviceType` field in the UPnP service description.

For standard services defined by a UPnP Forum working committee, the serviceType must contain the following components in the indicated order:

- `urn:schemas-upnp-org:service:`
- service type suffix:
- integer service version

Example: `urn:schemas-upnp-org:service:serviceType:v`.

For non-standard services specified by UPnP vendors, the `serviceType` must contain the following components in the indicated order:

- `urn:`
- ICANN domain name owned by the vendor
- `:service:`
- service type suffix:
- integer service version

Example: `urn:domain-name:service:serviceType:v`.

This method must continue to return the service type after the UPnP service has been removed from the network.

Returns The service type suffix defined by a UPnP Forum working committee or specified by a UPnP vendor. Must be <= 64 characters, not including the version suffix and separating colon. Single URI.

111.17.9.9 public String getVersion ( )

□ Returns the version suffix encoded in the `serviceType` field in the UPnP service description.

This method must continue to return the service version after the UPnP service has been removed from the network.

Returns The integer service version defined by a UPnP Forum working committee or specified by a UPnP vendor.

111.17.10 public interface UPnPStateVariable

The meta-information of a UPnP state variable as declared in the device's service state table (SST).

Method calls to interact with a device (e.g. `UPnPAction.invoke(...);`) use this class to encapsulate meta information about the input and output arguments.

The actual values of the arguments are passed as Java objects. The mapping of types from UPnP data types to Java data types is described with the field definitions.

111.17.10.1 public static final String TYPE_BIN_BASE64 = "bin.base64"

MIME-style Base64 encoded binary BLOB.

Takes 3 Bytes, splits them into 4 parts, and maps each 6 bit piece to an octet. (3 octets are encoded as 4.) No limit on size.

Mapped to `byte[]` object. The Java byte array will hold the decoded content of the BLOB.

111.17.10.2 public static final String TYPE_BIN_HEX = “bin.hex”

Hexadecimal digits representing octets.

Treats each nibble as a hex digit and encodes as a separate Byte. (1 octet is encoded as 2.) No limit on size.

Mapped to byte[] object. The Java byte array will hold the decoded content of the BLOB.

111.17.10.3 public static final String TYPE_BOOLEAN = “boolean”

True or false.

Mapped to Boolean object.

111.17.10.4 public static final String TYPE_CHAR = “char”

Unicode string.

One character long.

Mapped to Character object.

111.17.10.5 public static final String TYPE_DATE = “date”

A calendar date.

Date in a subset of ISO 8601 format without time data.

See http://www.w3.org/TR/ xmlschema-2/#date (http://www.w3.org/TR/xmlschema-2/#date) .

Mapped to java.util.Date object. Always 00:00 hours.

111.17.10.6 public static final String TYPE_DATETIME = “dateTime”

A specific instant of time.

Date in ISO 8601 format with optional time but no time zone.

See http://www.w3.org /TR/xmlschema-2/#dateTime (http://www.w3.org/TR/xmlschema-2/#dateTime) .

Mapped to java.util.Date object using default time zone.

111.17.10.7 public static final String TYPE_DATETIME_TZ = “dateTime.tz”

A specific instant of time.

Date in ISO 8601 format with optional time and optional time zone.

See http://www.w3.org /TR/xmlschema-2/#dateTime (http://www.w3.org/TR/xmlschema-2/#dateTime) .

Mapped to java.util.Date object adjusted to default time zone.

111.17.10.8 public static final String TYPE_FIXED_14_4 = “fixed.14.4”

Same as r8 but no more than 14 digits to the left of the decimal point and no more than 4 to the right.

Mapped to Double object.

111.17.10.9 public static final String TYPE_FLOAT = “float”

Floating-point number.

Mantissa (left of the decimal) and/or exponent may have a leading sign. Mantissa and/or exponent may have leading zeros. Decimal character in mantissa is a period, i.e., whole digits in mantissa separated from fractional digits by period. Mantissa separated from exponent by E. (No currency symbol.) (No grouping of digits in the mantissa, e.g., no commas.)

Mapped to Float object.

111.17.10.10 public static final String TYPE_I1 = "i1"

1 Byte int.

Mapped to Integer object.

111.17.10.11 public static final String TYPE_I2 = "i2"

2 Byte int.

Mapped to Integer object.

111.17.10.12 public static final String TYPE_I4 = "i4"

4 Byte int.

Must be between -2147483648 and 2147483647

Mapped to Integer object.

111.17.10.13 public static final String TYPE_INT = "int"

Integer number.

Mapped to Integer object.

111.17.10.14 public static final String TYPE_NUMBER = "number"

Same as r8.

Mapped to Double object.

111.17.10.15 public static final String TYPE_R4 = "r4"

4 Byte float.

Same format as float. Must be between 3.40282347E+38 to 1.17549435E-38.

Mapped to Float object.

111.17.10.16 public static final String TYPE_R8 = "r8"

8 Byte float.

Same format as float. Must be between -1.79769313486232E308 and -4.94065645841247E-324 for negative values, and between 4.94065645841247E-324 and 1.79769313486232E308 for positive values, i.e., IEEE 64-bit (8-Byte) double.

Mapped to Double object.

111.17.10.17 public static final String TYPE_STRING = "string"

Unicode string.

No limit on length.

Mapped to String object.

111.17.10.18 public static final String TYPE_TIME = "time"

An instant of time that recurs every day.

Time in a subset of ISO 8601 format with no date and no time zone.

See http://www.w3.org /TR/xmlschema-2/#time (http://www.w3.org/TR/xmlschema-2/#dateTime) .

Mapped to Long. Converted to milliseconds since midnight.

111.17.10.19 **public static final String TYPE_TIME_TZ = "time.tz"**

An instant of time that recurs every day.

Time in a subset of ISO 8601 format with optional time zone but no date.

See http://www.w3.org /TR/xmlschema-2/#time (http://www.w3.org/TR/xmlschema-2/#dateTime) .

Mapped to Long object. Converted to milliseconds since midnight and adjusted to default time zone, wrapping at 0 and 24∗60∗60∗1000.

111.17.10.20 **public static final String TYPE_UI1 = "ui1"**

Unsigned 1 Byte int.

Mapped to an Integer object.

111.17.10.21 **public static final String TYPE_UI2 = "ui2"**

Unsigned 2 Byte int.

Mapped to Integer object.

111.17.10.22 **public static final String TYPE_UI4 = "ui4"**

Unsigned 4 Byte int.

Mapped to Long object.

111.17.10.23 **public static final String TYPE_URI = "uri"**

Universal Resource Identifier.

Mapped to String object.

111.17.10.24 **public static final String TYPE_UUID = "uuid"**

Universally Unique ID.

Hexadecimal digits representing octets. Optional embedded hyphens are ignored.

Mapped to String object.

111.17.10.25 **public String[] getAllowedValues ( )**

☐ Returns the allowed values, if defined. Allowed values can be defined only for String types.

This method must continue to return the state variable allowed values after the UPnP state variable has been removed from the network.

Returns The allowed values or null if not defined. Should be less than 32 characters.

111.17.10.26 **public Object getDefaultValue ( )**

☐ Returns the default value, if defined.

This method must continue to return the state variable default value after the UPnP state variable has been removed from the network.

Returns The default value or null if not defined. The type of the returned object can be determined by getJavaDataType.

111.17.10.27 **public Class getJavaDataType ( )**

☐ Returns the Java class associated with the UPnP data type of this state variable.

Mapping between the UPnP data types and Java classes is performed according to the schema mentioned above.

```
Integer             ui1, ui2, i1, i2, i4, int
Long                ui4, time, time.tz
Float               r4, float
Double              r8, number, fixed.14.4
Character           char
String              string, uri, uuid
Date                date, dateTime, dateTime.tz
Boolean             boolean
byte[]              bin.base64, bin.hex
```

This method must continue to return the state variable java type after the UPnP state variable has been removed from the network.

Returns A class object corresponding to the Java type of this argument.

111.17.10.28 public Number getMaximum ( )

□ Returns the maximum value, if defined. Maximum values can only be defined for numeric types.

This method must continue to return the state variable maximum value after the UPnP state variable has been removed from the network.

Returns The maximum value or null if not defined.

111.17.10.29 public Number getMinimum ( )

□ Returns the minimum value, if defined. Minimum values can only be defined for numeric types.

This method must continue to return the state variable minimum value after the UPnP state variable has been removed from the network.

Returns The minimum value or null if not defined.

111.17.10.30 public String getName ( )

□ Returns the variable name.

- All standard variables defined by a UPnP Forum working committee must not begin with X_ nor A_.
- All non-standard variables specified by a UPnP vendor and added to a standard service must begin with X_.

This method must continue to return the state variable name after the UPnP state variable has been removed from the network.

Returns Name of state variable. Must not contain a hyphen character nor a hash character. Should be < 32 characters.

111.17.10.31 public Number getStep ( )

□ Returns the size of an increment operation, if defined. Step sizes can be defined only for numeric types.

This method must continue to return the step size after the UPnP state variable has been removed from the network.

Returns The increment size or null if not defined.

111.17.10.32 public String getUPnPDataType ( )

□ Returns the UPnP type of this state variable. Valid types are defined as constants.

This method must continue to return the state variable UPnP data type after the UPnP state variable has been removed from the network.

Returns The UPnP data type of this state variable, as defined in above constants.

111.17.10.33 public boolean sendsEvents ( )

□ Tells if this StateVariable can be used as an event source. If the StateVariable is eventable, an event listener service can be registered to be notified when changes to the variable appear.

This method must continue to return the correct value after the UPnP state variable has been removed from the network.

Returns `true` if the `StateVariable` generates events, `false` otherwise.

111.18 References

[1] *UPnP Forum*
http://www.upnp.org

[2] *XML Schema*
http://www.w3.org/TR/xmlschema-2

[3] *ISO 8601 Date And Time formats*
http://www.iso.ch

112 Declarative Services Specification

Version 1.2

112.1 Introduction

The OSGi Framework contains a procedural service model which provides a publish/find/bind model for using *services*. This model is elegant and powerful, it enables the building of applications out of bundles that communicate and collaborate using these services.

This specification addresses some of the complications that arise when the OSGi service model is used for larger systems and wider deployments, such as:

- *Startup Time* – The procedural service model requires a bundle to actively register and acquire its services. This is normally done at startup time, requiring all present bundles to be initialized with a Bundle Activator. In larger systems, this quickly results in unacceptably long startup times.
- *Memory Footprint* – A service registered with the Framework implies that the implementation, and related classes and objects, are loaded in memory. If the service is never used, this memory is unnecessarily occupied. The creation of a class loader may therefore cause significant overhead.
- *Complexity* – Service can come and go at any time. This dynamic behavior makes the service programming model more complex than more traditional models. This complexity negatively influences the adoption of the OSGi service model as well as the robustness and reliability of applications because these applications do not always handle the dynamicity correctly.

The *service component* model uses a declarative model for publishing, finding and binding to OSGi services. This model simplifies the task of authoring OSGi services by performing the work of registering the service and handling service dependencies. This minimizes the amount of code a programmer has to write; it also allows service components to be loaded only when they are needed. As a result, bundles need not provide a `BundleActivator` class to collaborate with others through the service registry.

From a system perspective, the service component model means reduced startup time and potentially a reduction of the memory footprint. From a programmer's point of view the service component model provides a simplified programming model.

The Service Component model makes use of concepts described in [1] *Automating Service Dependency Management in a Service-Oriented Component Model.*

112.1.1 Essentials

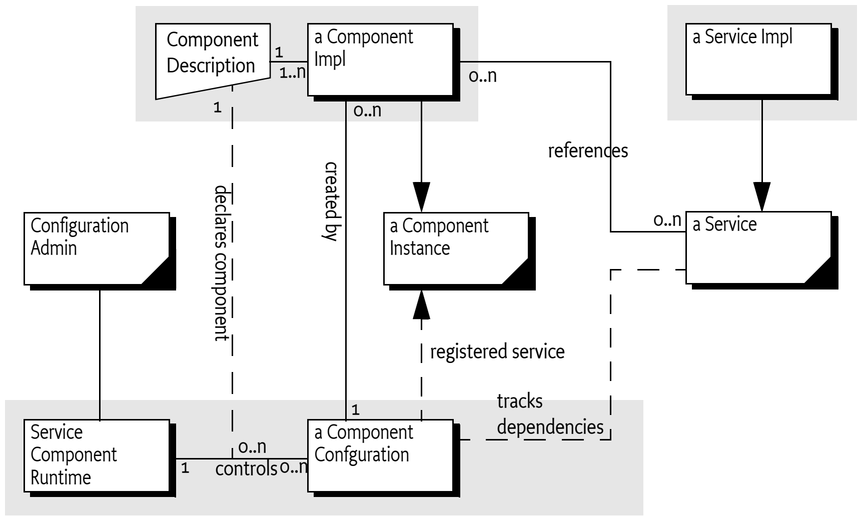
- *Backward Compatibility* – The service component model must operate seamlessly with the existing service model.
- *Size Constraints* – The service component model must not require memory and performance intensive subsystems. The model must also be applicable on resource constrained devices.
- *Delayed Activation* – The service component model must allow delayed activation of a service component. Delayed activation allows for delayed class loading and object creation until needed, thereby reducing the overall memory footprint.
- *Simplicity* – The programming model for using declarative services must be very simple and not require the programmer to learn a complicated API or XML sub-language.

- *Reactive* – It must be possible to react to changes in the external dependencies with different policies.
- *Annotations* – Annotations must be provided that can leverage the type information to create the XML descriptor.

112.1.2 Entities

- *Service Component* – A service component contains a description that is interpreted at run time to create and dispose objects depending on the availability of other services, the need for such an object, and available configuration data. Such objects can optionally provide a service. This specification also uses the generic term *component* to refer to a service component.
- *Component Description* – The declaration of a service component. It is contained within an XML document in a bundle.
- *Component Properties* – A set of properties which can be specified by the component description, Configuration Admin service and from the component factory.
- *Component Configuration* – A component configuration represents a component description parameterized by component properties. It is the entity that tracks the component dependencies and manages a component instance. An activated component configuration has a component context.
- *Component Instance* – An instance of the component implementation class. A component instance is created when a component configuration is activated and discarded when the component configuration is deactivated. A component instance is associated with exactly one component configuration.
- *Delayed Component* – A component whose component configurations are activated when their service is requested.
- *Immediate Component* – A component whose component configurations are activated immediately upon becoming satisfied.
- *Factory Component* – A component whose component configurations are created and activated through the component's component factory.
- *Reference* – A specified dependency of a component on a set of target services.
- *Service Component Runtime (SCR)* – The actor that manages the components and their life cycle.
- *Target Services* – The set of services that is defined by the reference interface and target property filter.
- *Bound Services* – The set of target services that are bound to a component configuration.
- *Event methods* – The bind, updated, and unbind methods associated with a Reference.

Figure 112.1 Service Component Runtime, org.osgi.service.component package



112.1.3 Synopsis

The Service Component Runtime reads component descriptions from started bundles. These descriptions are in the form of XML documents which define a set of components for a bundle. A component can refer to a number of services that must be available before a component configuration becomes satisfied. These dependencies are defined in the descriptions and the specific target services can be influenced by configuration information in the Configuration Admin service. After a component configuration becomes satisfied, a number of different scenarios can take place depending on the component type:

- *Immediate Component* – The component configuration of an immediate component must be activated immediately after becoming satisfied. Immediate components may provide a service.
- *Delayed Component* – When a component configuration of a delayed component becomes satisfied, SCR will register the service specified by the `service` element without activating the component configuration. If this service is requested, SCR must activate the component configuration creating an instance of the component implementation class that will be returned as the service object. If the `servicefactory` attribute of the `service` element is `true`, then, for each distinct bundle that requests the service, a different component configuration is created and activated and a new instance of the component implementation class is returned as the service object.
- *Factory Component* – If a component's description specifies the `factory` attribute of the `component` element, SCR will register a Component Factory service. This service allows client bundles to create and activate multiple component configurations and dispose of them. If the component's description also specifies a `service` element, then as each component configuration is activated, SCR will register it as a service.

112.1.4 Readers

- *Architects* – The chapter, *Components* on page 217, gives a comprehensive introduction to the capabilities of the component model. It explains the model with a number of examples. The section about *Component Life Cycle* on page 234 provides some deeper insight in the life cycle of components.
- *Service Programmers* – Service programmers should read *Components* on page 217. This chapter should suffice for the most common cases. For the more advanced possibilities, they should consult *Component Description* on page 227 for the details of the XML grammar for component descriptions.
- *Deployers* – Deployers should consult *Deployment* on page 244.

112.2 Components

A component is a normal Java class contained within a bundle. The distinguishing aspect of a component is that it is *declared* in an XML document. Component configurations are activated and deactivated under the full control of SCR. SCR bases its decisions on the information in the component's description. This information consists of basic component information like the name and type, optional services that are implemented by the component, and *references*. References are dependencies that the component has on other services.

SCR must *activate* a component configuration when the component is enabled and the component configuration is satisfied and a component configuration is needed. During the life time of a component configuration, SCR can notify the component of changes in its bound references.

SCR will *deactivate* a previously activated component configuration when the component becomes disabled, the component configuration becomes unsatisfied, or the component configuration is no longer needed.

If an activated component configuration's configuration properties change, SCR must deactivate the component configuration and then attempt to reactivate the component configuration using the new configuration information.

112.2.1 Declaring a Component

A component requires the following artifacts in the bundle:

- An XML document that contains the component description.
- The Service-Component manifest header which names the XML documents that contain the component descriptions.
- An implementation class that is specified in the component description.

The elements in the component's description are defined in *Component Description* on page 227. The XML grammar for the component declaration is defined by the XML Schema, see *Component Description Schema* on page 248.

112.2.2 Immediate Component

An *immediate component* is activated as soon as its dependencies are satisfied. If an immediate component has no dependencies, it is activated immediately. A component is an immediate component if it is not a factory component and either does not specify a service or specifies a service and the immediate attribute of the component element set to true. If an immediate component configuration is satisfied and specifies a service, SCR must register the component configuration as a service in the service registry and then activate the component configuration.

For example, the bundle entry /OSGI-INF/activator.xml contains:

```
<xml version="1.0" encoding="UTF-8">
<scr:component name="example.activator"
   xmlns:scr="http://www.osgi.org/xmlns/scr/v1.2.0">
   <implementation class="com.acme.Activator"/>
</scr:component>
```

The manifest header Service-Component must also be specified in the bundle manifest. For example:

```
Service-Component: OSGI-INF/activator.xml
```

An example class for this component could look like:

```
public class Activator {
   public Activator() {...}
   private void activate(BundleContext context) {...}
   private void deactivate() {...}
}
```

This example component is virtually identical to a Bundle Activator. It has no references to other services so it will be satisfied immediately. It publishes no service so SCR will activate a component configuration immediately.

The activate method is called when SCR activates the component configuration and the deactivate method is called when SCR deactivates the component configuration. If the activate method throws an Exception, then the component configuration is not activated and will be discarded.

112.2.3 Delayed Component

A *delayed component* specifies a service, is not specified to be a factory component and does not have the immediate attribute of the component element set to true. If a delayed component configuration is satisfied, SCR must register the component configuration as a service in the service registry but the activation of the component configuration is delayed until the registered service is requested. The registered service of a delayed component looks like a normal registered service but does not incur the overhead of an ordinarily registered service that require a service's bundle to be initialized to register the service.

For example, a bundle needs to see events of a specific topic. The Event Admin uses the white board pattern, receiving the events is therefore as simple as registering a Event Handler service. The example XML for the delayed component looks like:

```
<xml version="1.0" encoding="UTF-8">
<scr:component name="example.handler"
   xmlns:scr="http://www.osgi.org/xmlns/scr/v1.2.0">
   <implementation class="com.acme.HandlerImpl"/>
   <property name="event.topics">some/topic</property>
   <service>
      <provide interface=
         "org.osgi.service.event.EventHandler"/>
   </service>
</scr:component>
```

The associated component class looks like:

```
public class HandlerImpl implements EventHandler {
   public void handleEvent(Event evt ) {
      ...
  }
}
```

The component configuration will only be activated once the Event Admin service requires the service because it has an event to deliver on the topic to which the component subscribed.

112.2.4 Factory Component

Certain software patterns require the creation of component configurations on demand. For example, a component could represent an application that can be launched multiple times and each application instance can then quit independently. Such a pattern requires a factory that creates the instances. This pattern is supported with a *factory component*. A factory component is used if the factory attribute of the component element is set to a *factory identifier*. This identifier can be used by a bundle to associate the factory with externally defined information.

SCR must register a Component Factory service on behalf of the component as soon as the component factory is satisfied. The service properties must be:

- component.name – The name of the component.
- component.factory – The factory identifier.

The service properties of the Component Factory service must not include the component properties.

New configurations of the component can be created and activated by calling the newInstance method on this Component Factory service. The newInstance(Dictionary) method has a Dictionary object as argument. This Dictionary object is merged with the component properties as described in *Component Properties* on page 243. If the component specifies a service, then the service is registered after the created component configuration is satisfied with the component properties. Then the component configuration is activated.

For example, a component can provide a connection to a USB device. Such a connection should normally not be shared and should be created each time such a service is needed. The component description to implement this pattern looks like:

```
<xml version="1.0" encoding="UTF-8">
<scr:component name="example.factory"
   factory="usb.connection"
   xmlns:scr="http://www.osgi.org/xmlns/scr/v1.2.0">
   <implementation class="com.acme.USBConnectionImpl"/>
</scr:component>
```

The component class looks like:

```
public class USBConnectionImpl implements USBConnection {
  private void activate(Map properties) {
    ...
  }
}
```

A factory component can be associated with a service. In that case, such a service is registered for each component configuration. For example, the previous example could provide a USB Connection service.

```
<xml version="1.0" encoding="UTF-8">
<scr:component name="example.factory"
  factory="usb.connection"
  xmlns:scr="http://www.osgi.org/xmlns/scr/v1.2.0">
  <implementation class="com.acme.USBConnectionImpl"/>
  <service>
    <provide interface="com.acme.USBConnection"/>
  </service>
</scr:component>
```

The associated component class looks like:

```
public class USBConnectionImpl implements USBConnection {
  private void activate(Map properties) {...}
  public void connect() { ... }
  ...
  public void close() { ... }
}
```

A new service will be registered each time a new component configuration is created and activated with the newInstance method. This allows a bundle other than the one creating the component configuration to utilize the service. If the component configuration is deactivated, the service must be unregistered.

112.3 References to Services

Most bundles will require access to other services from the service registry. The dynamics of the service registry require care and attention of the programmer because referenced services, once acquired, could be unregistered at any moment. The component model simplifies the handling of these service dependencies significantly.

The services that are selected by a reference are called the *target services*. These are the services selected by the BundleContext.getServiceReferences method where the first argument is the reference's interface and the second argument is the reference's target property, which must be a valid filter.

A component configuration becomes *satisfied* when each specified reference is satisfied. A reference is *satisfied* if it specifies optional cardinality or when the target services contains at least one member. An activated component configuration that becomes *unsatisfied* must be deactivated.

During the activation of a component configuration, SCR must bind some or all of the target services of a reference to the component configuration. Any target service that is bound to the component configuration is called a *bound* service. See *Binding Services* on page 238.

112.3.1 Accessing Services

A component instance must be able to use the services that are referenced by the component configuration, that is, the bound services of the references. There are two strategies for a component instance to acquire these bound services:

- *Event strategy* – SCR calls a method on the component instance when a service becomes bound, when a service becomes unbound, or when its properties are updated. These methods are the bind, updated, and unbind methods specified by the reference. The event strategy is useful if the component needs to be notified of changes to the bound services for a dynamic reference.
- *Lookup strategy* – A component instance can use one of the `locateService` methods of `ComponentContext` to locate a bound service. These methods take the `name` of the reference as a parameter. If the reference has a dynamic policy, it is important to not store the returned service object(s) but look it up every time it is needed.

A component may use either or both strategies to access bound services.

112.3.2 Event Methods

When using the event strategy the SCR must callback the components at the appropriate time. The SCR must callback on the following events:

- `bind` – The bind method is called to bind a new service to the component that matches the selection criteria. If the `policy` is `dynamic` then the bind method of a replacement service can be called before its corresponding unbind method.
- `updated` – The updated method is called when the service properties of a bound services are modified and the resulting properties do not cause the service to become unbound because it is no longer selected by the `target` filter.
- `unbind` – The unbind method is called when the SCR needs to unbind the service.

Together these methods are called the *event methods*. Event methods must have one of the following prototypes:

```
void <method-name>(ServiceReference);
void <method-name>(<parameter-type>);
void <method-name>(<parameter-type>, Map);
```

If an event method has the first prototype, then a Service Reference to the bound service will be passed to the method. This Service Reference may later be passed to the `locateService(String, ServiceReference)` method to obtain the actual service object. This approach is useful when the service properties need to be examined before accessing the service object. It also allows for the delayed activation of bound services when using the event strategy.

If an event method has the second prototype, then the service object of the bound service is passed to the method. The method's parameter type must be assignable from the type specified by the reference's `interface` attribute. That is, the service object of the bound service must be castable to the method's parameter type.

If an event method has the third prototype, then the service object of the bound service is passed to the method as the first argument and an unmodifiable Map containing the service properties of the bound service is passed as the second argument. The method's first parameter type must be assignable from the type specified by the reference's `interface` attribute. That is, the service object of the bound service must be castable to the method's first parameter type.

The bind and unbind methods must be called once for each bound service. This implies that if the reference has multiple cardinality, then the methods may be called multiple times. The updated method can be called multiple times per service.

A suitable method is selected using the following priority:

1 The method takes a single argument and the type of the argument is `org.osgi.framework.ServiceReference`.

2 The method takes a single argument and the type of the argument is the type specified by the reference's `interface` attribute.
3 The method takes a single argument and the type of the argument is assignable from the type specified by the reference's `interface` attribute. If multiple methods match this rule, this implies the method name is overloaded and SCR may choose any of the methods to call.
4 The method takes two argument and the type of the first argument is the type specified by the reference's `interface` attribute and the type of the second argument is `java.util.Map`.
5 The method takes two argument and the type of the first argument is assignable from the type specified by the reference's `interface` attribute and the type of the second argument is `java.util.Map`. If multiple methods match this rule, this implies the method name is overloaded and SCR may choose any of the methods to call.

When searching for an event method to call, SCR must locate a suitable method as specified in *Locating Component Methods* on page 247. If no suitable method is located, SCR must log an error message with the Log Service, if present, and there will be no bind, updated, or unbind notification.

When the service object for a bound service is first provided to a component instance, that is passed to an event method or returned by a locate service method, SCR must get the service object from the OSGi Framework's service registry using the `getService` method on the component's Bundle Context. If the service object for a bound service has been obtained and the service becomes unbound, SCR must unget the service object using the `ungetService` method on the component's Bundle Context and discard all references to the service object.

For example, a component requires the Log Service and uses the lookup strategy. The reference is declared without any bind, updated, and unbind methods:

```
<xml version="1.0" encoding="UTF-8">
<scr:component name="example.listen"
   xmlns:scr="http://www.osgi.org/xmlns/scr/v1.2.0">
   <implementation class="com.acme.LogLookupImpl"/>
   <reference name="LOG"
      interface="org.osgi.service.log.LogService"/>
</scr:component>
```

The component implementation class must now lookup the service. This looks like:

```
public class LogLookupImpl {
   private void activate(ComponentContext ctxt) {
      LogService log = (LogService)
         ctxt.locateService("LOG");
      log.log(LogService.LOG_INFO, "Hello Components!"));
   }
}
```

Alternatively, the component could use the event strategy and ask to be notified with the Log Service by declaring bind, updated, and unbind methods.

```
<xml version="1.0" encoding="UTF-8">
<scr:component name="example.listen"
   xmlns:scr="http://www.osgi.org/xmlns/scr/v1.2.0">
   <implementation class="com.acme.LogEventImpl"/>
   <reference name="LOG"
      interface="org.osgi.service.log.LogService"
      bind="setLog"
      updated="updatedLog"
      unbind="unsetLog"
   />
</scr:component>
```

The component implementation class looks like:

```
public class LogEventImpl {
   LogService       log;
   Integer          level;
   void setLog( LogService l, Map<String,?> ref ) {
      log = l;
      updatedLog(ref);
   }
   void updatedLog( Map<String,?> ref) {
      level = (Integer) ref.get("level");
   }
   void unsetLog( LogService l ) { log = null; }

   private void activate() {
      log.log(LogService.LOG_INFO, "Hello Components!"));
   }
}
```

Event methods can be declared private in the component class but are only looked up in the inheritance chain when they are protected, public, or have default access. See *Locating Component Methods* on page 247.

112.3.3 Reference Cardinality

A component implementation is always written with a certain *cardinality* in mind. The cardinality represents two important concepts:

- *Multiplicity* – Does the component implementation assume a single service or does it explicitly handle multiple occurrences For example, when a component uses the Log Service, it only needs to bind to one Log Service to function correctly. Alternatively, when the Configuration Admin uses the Configuration Listener services it needs to bind to all target services present in the service registry to dispatch its events correctly.
- *Optionality* – Can the component function without any bound service present Some components can still perform useful tasks even when no target service is available, other components must bind to at least one target service before they can be useful. For example, the Configuration Admin in the previous example must still provide its functionality even if there are no Configuration Listener services present. Alternatively, an application that solely presents a Servlet page has little to do when the Http Service is not present, it should therefore use a reference with a mandatory cardinality.

The cardinality is expressed with the following syntax:

```
cardinality  ::= optionality '..' multiplicity
optionality  ::= '0' | '1'
multiplicity ::= '1' | 'n'
```

A reference is *satisfied* if the number of target services is equal to or more than the optionality. The multiplicity is irrelevant for the satisfaction of the reference. The multiplicity only specifies if the component implementation is written to handle being bound to multiple services (n) or requires SCR to select and bind to a single service (1).

The cardinality for a reference can be specified as one of four choices:

- 0..1 – Optional and unary.
- 1..1 – Mandatory and unary (Default) .
- 0..n – Optional and multiple.
- 1..n – Mandatory and multiple.

When a satisfied component configuration is activated, there must be at most one bound service for each reference with a unary cardinality and at least one bound service for each reference with a mandatory cardinality. If the cardinality constraints cannot be maintained after a component configuration is activated, that is the reference becomes unsatisfied, the component configuration must be deactivated. If the reference has a unary cardinality and there is more than one target service for the reference, then the bound service must be the target service with the highest service ranking as specified by the `service.ranking` property. If there are multiple target services with the same service ranking, then the bound service must be the target service with the highest service ranking and the lowest service ID as specified by the `service.id` property.

For example, a component wants to register a resource with all Http Services that are available. Such a scenario has the cardinality of `0..n`. The code must be prepared to handle multiple calls to the bind method for each Http Service in such a case. In this example, the code uses the `registerResources` method to register a directory for external access.

```
<xml version="1.0" encoding="UTF-8">
<scr:component name="example.listen"
   xmlns:scr="http://www.osgi.org/xmlns/scr/v1.2.0">
   <implementation class="com.acme.HttpResourceImpl"/>
   <reference name="HTTP"
      interface="org.osgi.service.http.HttpService"
      cardinality="0..n"
      bind="setPage"
      unbind="unsetPage"
   />
</scr:component>

public class HttpResourceImpl {
   private void setPage(HttpService http) {
      http.registerResources("/scr", "scr", null );
   }
   private void unsetPage(HttpService http) {
      http.unregister("/scr");
   }
}
```

112.3.4 Reference Policy

Once all the references of a component are satisfied, a component configuration can be activated and therefore bound to target services. However, the dynamic nature of the OSGi service registry makes it likely that services are registered, modified and unregistered after target services are bound. These changes in the service registry could make one or more bound services no longer a target service thereby making obsolete any object references that the component has to these service objects. Components therefore must specify a *policy* how to handle these changes in the set of bound services. A *policy-option* can further refine how changes affect bound services.

The *static policy* is the most simple policy and is the default policy. A component instance never sees any of the dynamics. Component configurations are deactivated before any bound service for a reference having a static policy becomes unavailable. If a target service is available to replace the bound service which became unavailable, the component configuration must be reactivated and bound to the replacement service.

If the `policy-option` is `reluctant` then the registration of an additional target service for a reference must not result in deactivating and reactivating a component configuration. If the `policy-option` is `greedy` then the component must be reactivated when new applicable services become available, see Table 112.1 on page 226. A reference with a static policy is called a *static reference*. A static reference can still be updated dynamically if it specifies an updated method.

The static policy can be very expensive if it depends on services that frequently unregister and re-register or if the cost of activating and deactivating a component configuration is high. Static policy is usually also not applicable if the cardinality specifies multiple bound services.

The *dynamic policy* is slightly more complex since the component implementation must properly handle changes in the set of bound services that can occur on any thread. With the dynamic policy, SCR can change the set of bound services without deactivating a component configuration. If the component uses the event strategy to access services, then the component instance will be notified of changes in the set of bound services by calls to the bind, and unbind methods.

If the policy-option is reluctant then a bound reference is not rebound even if a more suitable service becomes available for a 1..1 or 0..1 reference. If the policy-option is greedy then the component must be unbound and rebound for that reference.

A reference with a dynamic policy is called a *dynamic reference*.

The previous example with the registering of a resource directory used a static policy. This implied that the component configurations are deactivated when there is a change in the bound set of Http Services. The code in the example can be seen to easily handle the dynamics of Http Services that come and go. The component description can therefore be updated to:

```
<xml version="1.0" encoding="UTF-8">
<scr:component name="example.listen"
   xmlns:scr="http://www.osgi.org/xmlns/scr/v1.2.0">
   <implementation class="com.acme.HttpResourceImpl"/>
   <reference name="HTTP"
      interface="org.osgi.service.http.HttpService"
      cardinality="0..n"
      policy="dynamic"
      bind="setPage"
      unbind="unsetPage"
   />
</scr:component>
```

The code is identical to the previous example.

112.3.5 Policy Option

The policy-option defines how eager the reference is to rebind when a new, potentially with a higher ranking, target service becomes available. It can have the following values:

- reluctant – Minimize rebinding and reactivating.
- greedy – Maximize the use of the best service by deactivating static references or rebinding dynamic references.

Table 112.1 defines the actions that are taken when a *better* target service becomes available. In this context, better is when the reference is not bound or when the new target service has a higher ranking than the bound service.

Table 112.1 Action taken for policy-option when a new or higher ranking service becomes available

Cardinality	**static reluctant**	**static greedy**	**dynamic reluctant**	**dynamic greedy**
0..1	Ignore	Reactivate to bind the better target service.	If no service is bound, bind to new target service. Otherwise, ignore new target service.	If no service is bound, bind to better target service. Otherwise, unbind the bound service and bind the better target service.
1..1	Ignore	Reactivate to bind the better target service.	Ignore	Unbind the bound service, then bind the new service.
0..n	Ignore	Reactivate	Bind new target service	Bind new target service
1..n	Ignore	Reactivate	Bind new target service	Bind new target service

112.3.6 Selecting Target Services

The target services for a reference are constrained by the reference's interface name and target property. By specifying a filter in the target property, the programmer and deployer can constrain the set of services that should be part of the target services.

For example, a component wants to track all Component Factory services that have a factory identification of acme.application. The following component description shows how this can be done.

```
<xml version="1.0" encoding="UTF-8">
<scr:component name="example.listen"
   xmlns:scr="http://www.osgi.org/xmlns/scr/v1.2.0">
   <implementation class="com.acme.FactoryTracker"/>
   <reference name="FACTORY"
      interface=
         "org.osgi.service.component.ComponentFactory"
      target="(component.factory=acme.application)"
   />
</scr:component>
```

The filter is manifested as a component property called the *target property*. The target property can also be set by property and properties elements, see *Property Element* on page 230. The deployer can also set the target property by establishing a configuration for the component which sets the value of the target property. This allows the deployer to override the target property in the component description. See *Component Properties* on page 243 for more information.

112.3.7 Circular References

It is possible for a set of component descriptions to create a circular dependency. For example, if component A references a service provided by component B and component B references a service provided by component A then a component configuration of one component cannot be satisfied without accessing a partially activated component instance of the other component. SCR must ensure that a component instance is never accessible to another component instance or as a service until it has been fully activated, that is it has returned from its activate method if it has one.

Circular references must be detected by SCR when it attempts to satisfy component configurations and SCR must fail to satisfy the references involved in the cycle and log an error message with the Log Service, if present. However, if one of the references in the cycle has optional cardinality SCR must break the cycle. The reference with the optional cardinality can be satisfied and bound to zero target services. Therefore the cycle is broken and the other references may be satisfied.

112.4 Component Description

Component descriptions are defined in XML documents contained in a bundle and any attached fragments.

If SCR detects an error when processing a component description, it must log an error message with the Log Service, if present, and ignore the component description. Errors can include XML parsing errors and ill-formed component descriptions.

112.4.1 Annotations

A number of CLASS retention annotations have been provided to allow tools to construct the XML from the Java class files. These annotations will be discussed with the appropriate elements an attributes. Since the naming rules between XML and Java differ, some name changes are necessary.

- *Elements* – The annotation class that corresponds to an element starts with an upper case. For example the component element is represented by the @Component annotation.
- *Attributes* – Multi word attributes that use a minus sign ('-' \u002D) are changed to camel case. For example, the component element configuration-pid attribute is the configurationPid member in the @Component annotation.

Some elements do not have a corresponding annotation since the annotations can be parameterized by the type information in the Java class. For example, the @Component annotation synthesizes the implement element's class attribute from the type it is applied to.

These annotations are intended to be used during build time to generate the XML and are not recognized by SCR at runtime.

112.4.2 Service Component Header

XML documents containing component descriptions must be specified by the Service-Component header in the manifest. The value of the header is a comma separated list of paths to XML entries within the bundle.

```
Service-Component ::= header // 3.2.4
```

The Service-Component header has no architected directives or properties. The header can be left empty.

The last component of each path in the Service-Component header may use wildcards so that Bundle.findEntries can be used to locate the XML document within the bundle and its fragments. For example:

```
Service-Component: OSGI-INF/*.xml
```

A Service-Component manifest header specified in a fragment is ignored by SCR. However, XML documents referenced by a bundle's Service-Component manifest header may be contained in attached fragments.

SCR must process each XML document specified in this header. If an XML document specified by the header cannot be located in the bundle and its attached fragments, SCR must log an error message with the Log Service, if present, and continue.

112.4.3 XML Document

A component description must be in a well-formed XML document [4] stored in a UTF-8 encoded bundle entry. The namespace for component descriptions is:

```
http://www.osgi.org/xmlns/scr/v1.2.0
```

The recommended prefix for this namespace is scr. This prefix is used by examples in this specification. XML documents containing component descriptions may contain a single, root component element or one or more component elements embedded in a larger document. Use of the namespace for component descriptions is mandatory. The attributes and sub-elements of a component element are always unqualified.

If an XML document contains a single, root component element which does not specify a namespace, then the http://www.osgi.org/xmlns/scr/v1.0.0 namespace is assumed. Component descriptions using the http://www.osgi.org/xmlns/scr/v1.0.0 namespace must be treated according to version 1.0 of this specification.

SCR must parse all component elements in the namespace. Elements not in this namespace must be ignored. Ignoring elements that are not recognized allows component descriptions to be embedded in any XML document. For example, an entry can provide additional information about components. These additional elements are parsed by another sub-system.

See *Component Description Schema* on page 248 for component description schema.

112.4.4 Component Element

The component element specifies the component description. The following text defines the structure of the XML grammar using a form that is similar to the normal grammar used in OSGi specifications. In this case the grammar should be mapped to XML elements:

```
<component>          ::= <implementation>
                         (<properties> | <property>) *
                         <service>
                         <reference> *
```

SCR must not require component descriptions to specify the elements in the order listed above and as required by the XML schema. SCR must allow other orderings since arbitrary orderings of these elements do not affect the meaning of the component description. Only the relative ordering of property and properties element have meaning.

The component element has the attributes and @Component annotations defined in Table 112.2.

Table 112.2 Component Element and Annotations

Attribute	Annotation	Description
name	name	The *name* of a component must be unique within a bundle. The component name is used as a PID to retrieve component properties from the OSGi Configuration Admin service if present, unless a configuration-pid attribute has been defined. See *Deployment* on page 244 for more information. If the component name is used as a PID then it should be unique within the framework. The XML schema allows the use of component names which are not valid PIDs. Care must be taken to use a valid PID for a component name if the component should be configured by the Configuration Admin service. This attribute is optional. The default value of this attribute is the value of the class attribute of the nested implementation element. If multiple component elements in a bundle use the same value for the class attribute of their nested implementation element, then using the default value for this attribute will result in duplicate component names. In this case, this attribute must be specified with a unique value.
enabled	enabled	Controls whether the component is *enabled* when the bundle is started. The default value is true. If enabled is set to false, the component is disabled until the method enableComponent is called on the ComponentContext object. This allows some initialization to be performed by some other component in the bundle before this component can become satisfied. See *Enabled* on page 234.
factory	factory	If set to a non-empty string, it indicates that this component is a *factory component*. SCR must register a Component Factory service for each factory component. See *Factory Component* on page 219.
immediate	immediate	Controls whether component configurations must be immediately activated after becoming satisfied or whether activation should be delayed. The default value is false if the factory attribute or if the service element is specified and true otherwise. If this attribute is specified, its value must be false if the factory attribute is also specified or must be true unless the service element is also specified.
configuration-policy	configurationPolicy (OPTIONAL, REQUIRE, or IGNORE)	Controls whether component configurations must be satisfied depending on the presence of a corresponding Configuration object in the OSGi Configuration Admin service. A corresponding configuration is a Configuration object where the PID is the name of the component. • optional – (default) Use the corresponding Configuration object if present but allow the component to be satisfied even if the corresponding Configuration object is not present. • require – There must be a corresponding Configuration object for the component configuration to become satisfied. • ignore – Always allow the component configuration to be satisfied and do not use the corresponding Configuration object even if it is present.
configuration-pid	configurationPid	The configuration PID to be used for the component in conjunction with Configuration Admin. The default value for this attribute is the name of the component, or if this is also not specified, the implementation class name.
activate	Activate	Specifies the name of the method to call when a component configuration is activated. The default value of this attribute is activate. See *Activate Method* on page 238 for more information. The annotation must be applied to the activate method and can only be used once.

Table 112.2 Component Element and Annotations

Attribute	Annotation	Description
deactivate	Deactivate	Specifies the name of the method to call when a component configuration is deactivated. The default value of this attribute is deactivate. See *Deactivate Method* on page 241 for more information. The annotation must be applied to the deactivate method and can only be used once.
modified	Modified	Specifies the name of the method to call when the configuration properties for a component configuration is using a Configuration object from the Configuration Admin service and that Configuration object is modified without causing the component configuration to become unsatisfied. If this attribute is not specified, then the component configuration will become unsatisfied if its configuration properties use a Configuration object that is modified in any way. See *Modified Method* on page 240 for more information. The annotation must be applied to the modified method and can only be used once.

112.4.5 Implementation Element

The implementation element is required and defines the name of the component implementation class. The single attribute is defined in Table 112.3.

Table 112.3 Implementation Element and Annotations

Attribute	Annotation	Description
class	Component	The Java fully qualified name of the implementation class. The component Component annotation will define the implementation element automatically from the type it is applied to.

The class is retrieved with the loadClass method of the component's bundle. The class must be public and have a public constructor without arguments (this is normally the default constructor) so component instances may be created by SCR with the newInstance method on Class.

If the component description specifies a service, the class must implement all interfaces that are provided by the service.

112.4.6 Property Element

A component description can define a number of properties. These can defined inline or from a resource in the bundle. The property and properties elements can occur multiple times and they can be interleaved. This interleaving is relevant because the properties are processed from top to bottom. Later properties override earlier properties that have the same name.

Properties can also be overridden by a Configuration Admin service's Configuration object before they are exposed to the component or used as service properties. This is described in *Component Properties* on page 243 and *Deployment* on page 244.

The property element has the attributes and annotations defined in Table 112.4.

For example, a component that needs an array of hosts can use the following property definition:

```
<property name="hosts">
     www.acme.com
     backup.acme.com
</property>
```

This property declaration results in the property hosts, with a value of String[] { "www.acme.com", "backup.acme.com" }.

Table 112.4 Property Element and Annotations

Attribute	Annotation	Description
name	Component property	The name of the property.
value		The value of the property. This value is parsed according to the property type. If the value attribute is specified, the body of the element is ignored. If the type of the property is not String, parsing of the value is done by the static valueOf(String) method in the given type. For Character types, the conversion must be handled by Integer.valueOf method, a Character is always represented by its Unicode value.
type		• The type of the property. Defines how to interpret the value. The type must be one of the following Java types: • String (default) • Long • Double • Float • Integer • Byte • Character • Boolean • Short
‹body›		If the value attribute is not specified, the body of the property element must contain one or more values. The value of the property is then an array of the specified type. Except for String objects, the result will be translated to an array of primitive types. For example, if the type attribute specifies Integer, then the resulting array must be int[]. Values must be placed one per line and blank lines are ignored. Parsing of the value is done by the parse methods in the class identified by the type, after trimming the line of any beginning and ending white space. String values are also trimmed of beginning and ending white space before being placed in the array.

A property can also be set with the property annotation element of Component. This element is an array of strings that must follow the following syntax:

```
property ::= name ( ':' type )? '=' value
```

In this case name, type, and value parts map to the attributes of the property element. If multiple values must be specified then the same name can be repeated multiple times. The annotation does not support ordering of properties. For example:

```
@Component(property={"foo:Integer=1","foo:Integer=2","foo:Integer=3"})
public class FooImpl {
  ...
}
```

The properties element references an entry in the bundle whose contents conform to a standard [3] *Java Properties File.*

The entry is read and processed to obtain the properties and their values. The properties element attributes are defined in Table 112.5.

Table 112.5 Properties Element and Annotations

Attribute	Annotation	Description
entry	Component properties	The entry path relative to the root of the bundle

For example, to include vendor identification properties that are stored in the OSGI-INF directory, the following definition could be used:

```
<properties entry="OSGI-INF/vendor.properties" />
```

The Component properties element can be used to provide the same information, this element consists of an array of strings where each string defines an entry. The order within the array is the order that must be used for the XML. However, the annotations do not allow mixing of the property and properties element.

For example:

```
@Component(properties="OSGI-INF/vendor.properties")
```

112.4.7 Service Element

The service element is optional. It describes the service information to be used when a component configuration is to be registered as a service.

A service element has the following attribute Table 112.6.

Table 112.6 Service Element and Annotations

Attribute	Annotation	Description
servicefactory	Component servicefactory	Controls whether the service uses the ServiceFactory concept of the OSGi Framework. The default value is false. If servicefactory is set to true, a different component configuration is created, activated and its component instance returned as the service object for each distinct bundle that requests the service. Each of these component configurations has the same component properties. Otherwise, the same component instance from the single component configuration is returned as the service object for all bundles that request the service.

The servicefactory attribute must not be true if the component is a factory component or an immediate component. This is because SCR is not free to create component configurations as necessary to support servicefactory. A component description is ill-formed if it specifies that the component is a factory component or an immediate component and servicefactory is set to true.

The service element must have one or more provide elements that define the service interfaces. The provide element has the attribute defined in Table 112.7.

Table 112.7 Provide Element and Annotations

Attribute	Annotation	Description
interface	Component service	The name of the interface that this service is registered under. This name must be the fully qualified name of a Java class. For example, org.osgi.service.log.LogService. The specified Java class should be an interface rather than a class, however specifying a class is supported. The component implementation class must implement all the specified service interfaces. The Component annotation can specify the provided services, if this element is not specified all directly implemented interfaces on the component's type are defined as service interfaces. Specifying an empty array indicates that no service should be registered.

For example, a component implements an Event Handler service.

```
<service>
   <provide interface=
      "org.osgi.service.eventadmin.EventHandler"/>
</service>
```

This previous example can be generated with the following annotation:

```
@Component
public class Foo implements EventHandler { ... }
```

112.4.8 Reference Element

A *reference* declares a dependency that a component has on a set of target services. A component configuration is not satisfied, unless all its references are satisfied. A reference specifies target services by specifying their interface and an optional target filter.

A reference element has the attributes defined in Table 112.8.

Table 112.8 Reference Element and Annotations

Attribute	Annotation	Description
name	name	The name of the reference. This name is local to the component and can be used to locate a bound service of this reference with one of the locateService methods of ComponentContext. Each reference element within the component must have a unique name. This name attribute is optional. The default value of this attribute is the value of the interface attribute of this element. If multiple reference elements in the component use the same interface name, then using the default value for this attribute will result in duplicate reference names. In this case, this attribute must be specified with a unique name for the reference to avoid an error. The Reference annotation will use the full method name as the default reference name.
interface	service	Fully qualified name of the class that is used by the component to access the service. The service provided to the component must be type compatible with this class. That is, the component must be able to cast the service object to this class. A service must be registered under this name to be considered for the set of target services. The Reference annotation will use the type of the first argument of the method it is applied for the service value.
cardinality	cardinality MANDATORY OPTIONAL MULTIPLE AT_LEAST_ONE	Specifies if the reference is optional and if the component implementation support a single bound service or multiple bound services. See *Reference Cardinality* on page 223.
policy	policy STATIC DYNAMIC	The policy declares the assumption of the component about dynamicity. See *Reference Policy* on page 224.
policy-option	policyOption RELUCTANT GREEDY	Defines the policy when a better service becomes available. See *Reference Policy* on page 224.
target	target	An optional OSGi Framework filter expression that further constrains the set of target services. The default is no filter, limiting the set of matched services to all service registered under the given reference interface. The value of this attribute is used to set a target property. See *Selecting Target Services* on page 226.

Table 112.8 Reference Element and Annotations

Attribute	Annotation	Description
bind	Reference	The name of a method in the component implementation class that is used to notify that a service is bound to the component configuration. For static references, this method is only called before the activate method. For dynamic references, this method can also be called while the component configuration is active. See *Accessing Services* on page 221. The Reference annotation will use the method it is applied to as the bind method.
updated	updated	The name of a method in the component implementation class that is used to notify that a bound service has modified its properties.
unbind	unbind	Same as bind, but is used to notify the component configuration that the service is unbound. For static references, the method is only called after the deactivate method. For dynamic references, this method can also be called while the component configuration is active. See *Accessing Services* on page 221.

The following code demonstrates the use of the Reference annotation.

```
@Component
public class FooImpl implements Foo {
  @Activate
  void open() { ... }
  @Deactivate
  void close() { ... }

  @Reference(
    policy = DYNAMIC,
    policyOption=GREEDY,
    cardinality=MANDATORY )
  void setLog( LogService log) { ... }
  void unsetLog( LogService log) { ... }
  void updatedLog( Map<String,?> ref ) { ... }
}
```

112.5 Component Life Cycle

112.5.1 Enabled

A component must first be *enabled* before it can be used. A component cannot be enabled unless the component's bundle is started. See *Starting Bundles* on page 91 of the Core specification. All components in a bundle become disabled when the bundle is stopped. So the life cycle of a component is contained within the life cycle of its bundle.

Every component can be enabled or disabled. The initial enabled state of a component is specified in the component description via the enabled attribute of the component element. See *Component Element* on page 228. Component configurations can be created, satisfied and activated only when the component is enabled.

The enabled state of a component can be controlled with the Component Context enableComponent(String) and disableComponent(String) methods. The purpose of later enabling a component is to be able to decide programmatically when a component can become enabled. For example, an immediate component can perform some initialization work before other components in the bundle are enabled. The component descriptions of all other components in the bundle can be disabled by having enabled set to false in their component descriptions. After any necessary initialization work is complete, the immediate component can call enableComponent to enable the remaining components.

The enableComponent and disableComponent methods must return after changing the enabled state of the named component. Any actions that result from this, such as activating or deactivating a component configuration, must occur asynchronously to the method call. Therefore a component can disable itself.

All components in a bundle can be enabled by passing a null as the argument to enableComponent.

112.5.2 Satisfied

Component configurations can only be activated when the component configuration is *satisfied.* A component configuration becomes satisfied when the following conditions are all satisfied:

- The component is *enabled.*
- If the component description specifies configuration-policy=required, then a Configuration object for the component is present in the Configuration Admin service.
- Using the component properties of the component configuration, all the component's references are satisfied. A reference is satisfied when the reference specifies optional cardinality or there is at least one target service for the reference.

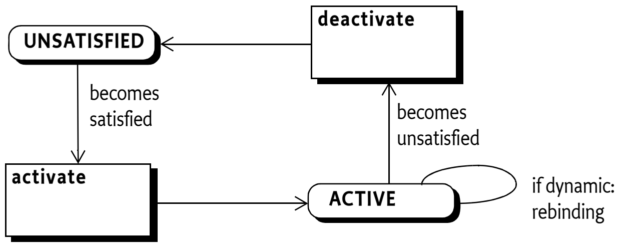
Once any of the listed conditions are no longer true, the component configuration becomes *unsatisfied.* An activated component configuration that becomes unsatisfied must be deactivated.

112.5.3 Immediate Component

A component is an immediate component when it must be activated as soon as its dependencies are satisfied. Once the component configuration becomes unsatisfied, the component configuration must be deactivated. If an immediate component configuration is satisfied and specifies a service, SCR must register the component configuration as a service in the service registry and then activate the component configuration. The service properties for this registration consist of the component properties as defined in *Service Properties* on page 244.

The state diagram is shown in Figure 112.2.

Figure 112.2 Immediate Component Configuration



112.5.4 Delayed Component

A key attribute of a delayed component is the delaying of class loading and object creation. Therefore, the activation of a delayed component configuration does not occur until there is an actual request for a service object. A component is a delayed component when it specifies a service but it is not a factory component and does not have the immediate attribute of the component element set to true.

SCR must register a service after the component configuration becomes satisfied. The registration of this service must look to observers of the service registry as if the component's bundle actually registered this service. This strategy makes it possible to register services without creating a class loader for the bundle and loading classes, thereby allowing reduction in initialization time and a delay in memory footprint.

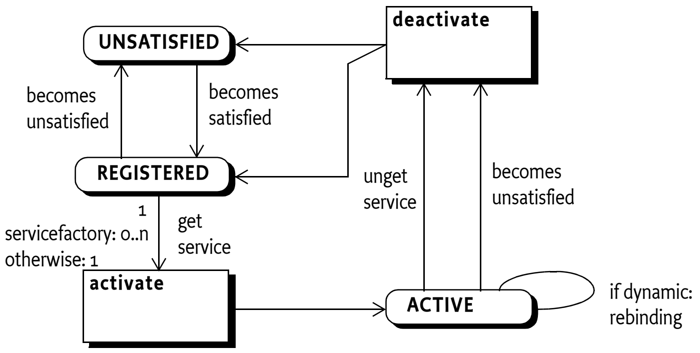
When SCR registers the service on behalf of a component configuration, it must avoid causing a class load to occur from the component's bundle. SCR can ensure this by registering a `ServiceFactory` object with the Framework for that service. By registering a `ServiceFactory` object, the actual service object is not needed until the `ServiceFactory` is called to provide the service object. The service properties for this registration consist of the component properties as defined in *Service Properties* on page 244.

The activation of a component configuration must be delayed until its service is requested. When the service is requested, if the service has the `servicefactory` attribute set to `true`, SCR must create and activate a unique component configuration for each bundle requesting the service. Otherwise, SCR must activate a single component configuration which is used by all bundles requesting the service. A component instance can determine the bundle it was activated for by calling the `getUsingBundle()` method on the Component Context.

The activation of delayed components is depicted in a state diagram in Figure 112.3. Notice that multiple component configurations can be created from the `REGISTERED` state if a delayed component specifies `servicefactory` set to `true`.

If the service registered by a component configuration becomes unused because there are no more bundles using it, then SCR should deactivate that component configuration. This allows SCR implementations to eagerly reclaim activated component configurations.

Figure 112.3 Delayed Component Configuration



112.5.5 Factory Component

SCR must register a Component Factory service as soon as the *component factory* becomes satisfied. The component factory is satisfied when the following conditions are all satisfied:

- The component is enabled.
- Using the component properties specified by the component description, all the component's references are satisfied. A reference is satisfied when the reference specifies optional cardinality or there is at least one target service for the reference

The component factory, however, does not use any of the target services and does not bind to them.

Once any of the listed conditions are no longer true, the component factory becomes unsatisfied and the Component Factory service must be unregistered. Any component configurations activated via the component factory are unaffected by the unregistration of the Component Factory service, but may themselves become unsatisfied for the same reason.

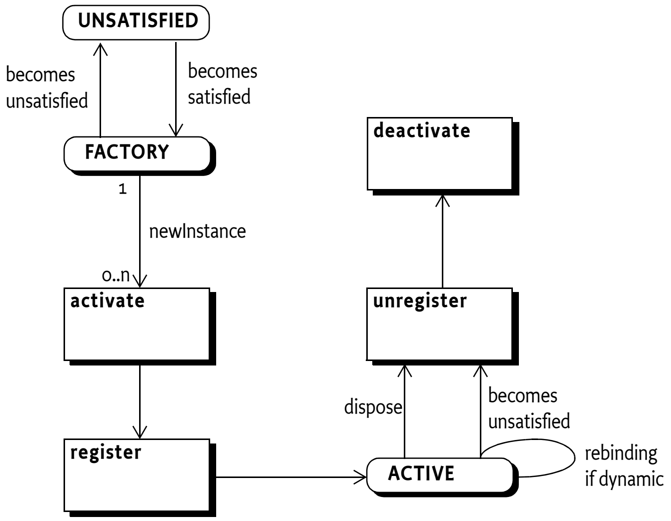
The Component Factory service must be registered under the name org.osgi.service.component.ComponentFactory with the following service properties:

- component.name – The name of the component.
- component.factory – The value of the factory attribute.

The service properties of the Component Factory service must not include the component properties.

New component configurations are created and activated when the newInstance method of the Component Factory service is called. If the component description specifies a service, the component configuration is registered as a service under the provided interfaces. The service properties for this registration consist of the component properties as defined in *Service Properties* on page 244. The service registration must take place before the component configuration is activated. Service unregistration must take place before the component configuration is deactivated.

Figure 112.4 *Factory Component*



A Component Factory service has a single method: newInstance(Dictionary). This method must create, satisfy and activate a new component configuration and register its component instance as a service if the component description specifies a service. It must then return a ComponentInstance object. This ComponentInstance object can be used to get the component instance with the getInstance() method.

SCR must attempt to satisfy the component configuration created by newInstance before activating it. If SCR is unable to satisfy the component configuration given the component properties and the Dictionary argument to newInstance, the newInstance method must throw a ComponentException.

The client of the Component Factory service can also deactivate a component configuration with the dispose() method on the ComponentInstance object. If the component configuration is already deactivated, or is being deactivated, then this method is ignored. Also, if the component configuration becomes unsatisfied for any reason, it must be deactivated by SCR.

Once a component configuration created by the Component Factory has been deactivated, that component configuration will not be reactivated or used again.

112.5.6 Activation

Activating a component configuration consists of the following steps:

1 Load the component implementation class.
2 Create the component instance and component context.

3 Bind the target services. See *Binding Services* on page 238.
4 Call the activate method, if present. See *Activate Method* on page 238.

Component instances must never be reused. Each time a component configuration is activated, SCR must create a new component instance to use with the activated component configuration. A component instance must complete activation before it can be deactivated. Once the component configuration is deactivated or fails to activate due to an exception, SCR must unbind all the component's bound services and discard all references to the component instance associated with the activation.

112.5.7 Binding Services

When a component configuration's reference is satisfied, there is a set of zero or more target services for that reference. When the component configuration is activated, a subset of the target services for each reference are bound to the component configuration. The subset is chosen by the cardinality of the reference. See *Reference Cardinality* on page 223.

When binding services, the references are processed in the order in which they are specified in the component description. That is, target services from the first specified reference are bound before services from the next specified reference.

For each reference using the event strategy, the bind method must be called for each bound service of that reference. This may result in activating a component configuration of the bound service which could result in an exception. If the loss of the bound service due to the exception causes the reference's cardinality constraint to be violated, then activation of this component configuration will fail. Otherwise the bound service which failed to activate will be considered unbound. If a bind method throws an exception, SCR must log an error message containing the exception with the Log Service, if present, but the activation of the component configuration does not fail.

112.5.8 Activate Method

A component instance can have an activate method. The name of the activate method can be specified by the `activate` attribute. See *Component Element* on page 228. If the `activate` attribute is not specified, the default method name of `activate` is used. The prototype of the activate method is:

```
void <method-name>(<arguments>);
```

The activate method can take zero or more arguments. Each argument must be of one of the following types:

- `ComponentContext` – The component instance will be passed the Component Context for the component configuration.
- `BundleContext` – The component instance will be passed the Bundle Context of the component's bundle.
- `Map` – The component instance will be passed an unmodifiable Map containing the component properties.

A suitable method is selected using the following priority:

1 The method takes a single argument and the type of the argument is `org.osgi.service.component.ComponentContext`.
2 The method takes a single argument and the type of the argument is `org.osgi.framework.BundleContext`.
3 The method takes a single argument and the type of the argument is the `java.util.Map`.
4 The method takes two or more arguments and the type of each argument must be `org.osgi.service.component.ComponentContext`, `org.osgi.framework.BundleContext` or `java.util.Map`. If multiple methods match this rule, this implies the method name is overloaded and SCR may choose any of the methods to call.
5 The method takes zero arguments.

When searching for the activate method to call, SCR must locate a suitable method as specified in *Locating Component Methods* on page 247. If the activate attribute is specified and no suitable method is located, SCR must log an error message with the Log Service, if present, and the component configuration is not activated.

If an activate method is located, SCR must call this method to complete the activation of the component configuration. If the activate method throws an exception, SCR must log an error message containing the exception with the Log Service, if present, and the component configuration is not activated.

112.5.9 Component Context

The Component Context is made available to a component instance via the activate and deactivate methods. It provides the interface to the execution context of the component, much like the Bundle Context provides a bundle the interface to the Framework. A Component Context should therefore be regarded as a capability and not shared with other components or bundles.

Each distinct component instance receives a unique Component Context. Component Contexts are not reused and must be discarded when the component configuration is deactivated.

112.5.10 Bound Service Replacement

If an active component configuration has a dynamic reference with unary cardinality and the bound service is modified or unregistered and ceases to be a target service, or the policy-option is greedy and a better target service becomes available then SCR must attempt to replace the bound service with a new target service. SCR must first bind a replacement target service and then unbind the outgoing service. This reversed order allows the component to not having to handle the inevitable gap between the unbind and bind methods. However, this means that in the unbind method care must be taken to not overwrite the newly bound service. For example, the following code handles the associated concurrency issues and simplify handling the reverse order.

```
final AtomicReference<LogService> log = new AtomicReference<LogService>();

void setLogService( LogService log ) {
    this.log.set(log);
}
void unsetLogService( LogService log ) {
   this.log.compareAndSet(log,null);
}
```

If the dynamic reference has a mandatory cardinality and no replacement target service is available, the component configuration must be deactivated because the cardinality constraints will be violated.

If a component configuration has a static reference and a bound service is modified or unregistered and ceases to be a target service, or the policy-option is greedy and a better target service becomes available then SCR must deactivate the component configuration. Afterwards, SCR must attempt to activate the component configuration again if another target service can be used as a replacement for the outgoing service.

112.5.11 Updated method

If an active component is bound to a service that modifies it properties then the component can be notified with the update method specified on the reference element. This method can be called with a Service Reference or a Map to obtain the service properties.

112.5.12 Modification

Modifying a component configuration can occur if the component description specifies the modified attribute and the component properties of the component configuration use a Configuration object from the Configuration Admin service and that Configuration object is modified without causing the component configuration to become unsatisfied. If this occurs, the component instance will be notified of the change in the component properties.

If the modified attribute is not specified, then the component configuration will become unsatisfied if its component properties use a Configuration object and that Configuration object is modified in any way.

Modifying a component configuration consists of the following steps:

1 Update the component context for the component configuration with the modified configuration properties.
2 Call the modified method. See *Modified Method* on page 240.
3 Modify the bound services for the dynamic references if the set of target services changed due to changes in the target properties. See *Bound Service Replacement* on page 239.
4 If the component configuration is registered as a service, modify the service properties.

A component instance must complete activation, or a previous modification, before it can be modified.

See *Modified Configurations* on page 244 for more information.

112.5.13 Modified Method

The name of the modified method is specified by the modified attribute. See *Component Element* on page 228. The prototype and selection priority of the modified method is identical to that of the activate method. See *Activate Method* on page 238.

SCR must locate a suitable method as specified in *Locating Component Methods* on page 247. If the modified attribute is specified and no suitable method is located, SCR must log an error message with the Log Service, if present, and the component configuration becomes unsatisfied and is deactivated as if the modified attribute was not specified.

If a modified method is located, SCR must call this method to notify the component configuration of changes to the component properties. If the modified method throws an exception, SCR must log an error message containing the exception with the Log Service, if present and continue processing the modification.

112.5.14 Deactivation

Deactivating a component configuration consists of the following steps:

1 Call the deactivate method, if present. See *Deactivate Method* on page 241.
2 Unbind any bound services. See *Unbinding* on page 241.
3 Release all references to the component instance and component context.

A component instance must complete activation or modification before it can be deactivated. A component configuration can be deactivated for a variety of reasons. The deactivation reason can be received by the deactivate method. The following reason values are defined:

- 0 – Unspecified.
- 1 – The component was disabled.
- 2 – A reference became unsatisfied.
- 3 – A configuration was changed.
- 4 – A configuration was deleted.
- 5 – The component was disposed.
- 6 – The bundle was stopped.

Once the component configuration is deactivated, SCR must discard all references to the component instance and component context associated with the activation.

112.5.15 Deactivate Method

A component instance can have a deactivate method. The name of the deactivate method can be specified by the `deactivate` attribute. See *Component Element* on page 228. If the `deactivate` attribute is not specified, the default method name of `deactivate` is used. The prototype of the deactivate method is:

```
void <method-name>(<arguments>);
```

The deactivate method can take zero or more arguments. Each argument must be assignable from one of the following types:

- `ComponentContext` – The component instance will be passed the Component Context for the component.
- `BundleContext` – The component instance will be passed the Bundle Context of the component's bundle.
- `Map` – The component instance will be passed an unmodifiable Map containing the component properties.
- `int` or `Integer` – The component instance will be passed the reason the component configuration is being deactivated. See *Deactivation* on page 240.

A suitable method is selected using the following priority:

1. The method takes a single argument and the type of the argument is `org.osgi.service.component.ComponentContext`.
2. The method takes a single argument and the type of the argument is `org.osgi.framework.BundleContext`.
3. The method takes a single argument and the type of the argument is the `java.util.Map`.
4. The method takes a single argument and the type of the argument is the `int`.
5. The method takes a single argument and the type of the argument is the `java.lang.Integer`.
6. The method takes two or more arguments and the type of each argument must be `org.osgi.service.component.ComponentContext`, `org.osgi.framework.BundleContext`, `java.util.Map`, `int` or `java.lang.Integer`. If multiple methods match this rule, this implies the method name is overloaded and SCR may choose any of the methods to call.
7. The method takes zero arguments.

When searching for the deactivate method to call, SCR must locate a suitable method as specified in *Locating Component Methods* on page 247. If the `deactivate` attribute is specified and no suitable method is located, SCR must log an error message with the Log Service, if present, and the deactivation of the component configuration will continue.

If a deactivate method is located, SCR must call this method to commence the deactivation of the component configuration. If the deactivate method throws an exception, SCR must log an error message containing the exception with the Log Service, if present, and the deactivation of the component configuration will continue.

112.5.16 Unbinding

When a component configuration is deactivated, the bound services are unbound from the component configuration.

When unbinding services, the references are processed in the reverse order in which they are specified in the component description. That is, target services from the last specified reference are unbound before services from the previous specified reference.

For each reference using the event strategy, the unbind method must be called for each bound service of that reference. If an unbind method throws an exception, SCR must log an error message containing the exception with the Log Service, if present, and the deactivation of the component configuration will continue.

112.5.17 Life Cycle Example

A component could declare a dependency on the Http Service to register some resources.

```
<xml version="1.0" encoding="UTF-8">
<scr:component name="example.binding"
   xmlns:scr="http://www.osgi.org/xmlns/scr/v1.2.0">
   <implementation class="example.Binding"/>
   <reference name="LOG"
      interface="org.osgi.service.log.LogService"
      cardinality="1..1"
      policy="static"
   />
   <reference name="HTTP"
      interface="org.osgi.service.http.HttpService"
      cardinality="0..1"
      policy="dynamic"
      bind="setHttp"
      unbind="unsetHttp"
   />
</scr:component>
```

The component implementation code looks like:

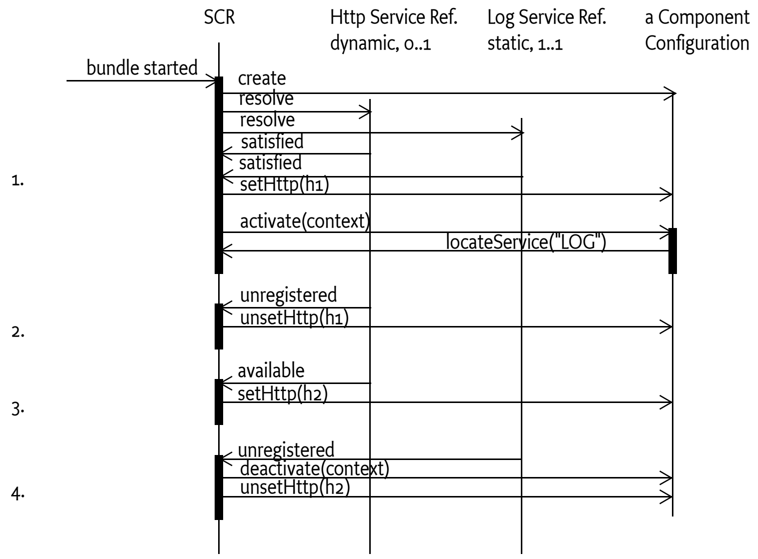
```
public class Binding {
   LogService  log;
   HttpService http;

   private void setHttp(HttpService h) {
      this.http = h;
      // register servlet
   }
    private void unsetHttp(HttpService h){
      this.h = null;
      // unregister servlet
   }
   private void activate(ComponentContext context ) {.
     log = (LogService) context.locateService("LOG");
    }
   private void deactivate(ComponentContext context ){...}
}
```

This example is depicted in a sequence diagram in Figure 112.5. with the following scenario:

1 A bundle with the example.Binding component is started. At that time there is a Log Service l1 and a Http Service h1 registered.
2 The Http Service h1 is unregistered
3 A new Http Service h2 is registered
4 The Log Service h1 is unregistered.

Figure 112.5 *Sequence Diagram for binding*



112.6 Component Properties

Each component configuration is associated with a set of component properties. The component properties are specified in the following places (in order of precedence):

1 Properties specified in the argument of ComponentFactory.newInstance method. This is only applicable for factory components.
2 Properties retrieved from the OSGi Configuration Admin service with a Configuration object that has a PID equal to the *configuration PID*. The configuration PID is the component name, or when specified, the configuration-pid attribute.
3 Properties specified in the component description. Properties specified later in the component description override properties that have the same name specified earlier. Properties can be specified in the component description in the following ways:
   - target attribute of reference elements – Sets a component property called the *target property* of the reference. The key of a target property is the name of the reference appended with .target. The value of a target property is the value of the target attribute. For example, a reference with the name http whose target attribute has the value "(http.port=80)" results in the component property having the name http.target and value "(http.port=80)". See *Selecting Target Services* on page 226. The target property can also be set wherever component properties can be set.
   - property and properties elements – See *Property Element* on page 230.

The precedence behavior allows certain default values to be specified in the component description while allowing properties to be replaced and extended by:

- A configuration in Configuration Admin
- The argument to ComponentFactory.newInstance method

SCR always adds the following component properties, which cannot be overridden:

- component.name – The component name.
- component.id – A unique value (Long) that is larger than all previously assigned values. These values are not persistent across restarts of SCR.

112.6.1 Service Properties

When SCR registers a service on behalf of a component configuration, SCR must follow the recommendations in *Property Propagation* on page 95 and must not propagate private configuration properties. That is, the service properties of the registered service must be all the component properties of the component configuration whose property names do not start with full stop ('.' \u002E).

Component properties whose names start with full stop are available to the component instance but are not available as service properties of the registered service.

112.7 Deployment

A component description contains default information to select target services for each reference. However, when a component is deployed, it is often necessary to influence the target service selection in a way that suits the needs of the deployer. Therefore, SCR uses Configuration objects from Configuration Admin to replace and extend the component properties for a component configuration. That is, through Configuration Admin, a deployer can configure component properties.

The name of the component is used as the key for obtaining additional component properties from Configuration Admin. The following situations can arise:

- *No Configuration* – If the component's configuration-policy is set to ignore or there is no Configuration with a PID or factory PID equal to the configuration PID, then component configurations will not obtain component properties from Configuration Admin. Only component properties specified in the component description or via the ComponentFactory.newInstance method will be used.
- *Not Satisfied* – If the component's configuration-policy is set to require and there is no Configuration with a PID or factory PID equal to the configuration PID, then the component configuration is not satisfied and will not be activated.
- *Single Configuration* – If there exists a Configuration with a PID equal to the configuration PID, then component configurations will obtain additional component properties from Configuration Admin.
- *Factory Configuration* – If a factory PID exists, with zero or more Configurations, that is equal to the configuration PID, then for each Configuration, a component configuration must be created that will obtain additional component properties from Configuration Admin.

A factory configuration must not be used if the component is a factory component. This is because SCR is not free to create component configurations as necessary to support multiple Configurations. When SCR detects this condition, it must log an error message with the Log Service, if present, and ignore the component description.

SCR must obtain the Configuration objects from the Configuration Admin service using the Bundle Context of the bundle containing the component.

For example, there is a component named com.acme.client with a reference named HTTP that requires an Http Service which must be bound to a component com.acme.httpserver which provides an Http Service. A deployer can establish the following configuration:

```
[PID=com.acme.client, factoryPID=null]
HTTP.target = (component.name=com.acme.httpserver)
```

112.7.1 Modified Configurations

SCR must track changes in the Configuration objects used in the component properties of a component configuration. If a Configuration object that is used by a component configuration is deleted, then the component configuration will become unsatisfied and SCR must deactivate that component configuration.

If a Configuration object that is used by a component configuration changes, then SCR must take action based upon whether the component configuration has been activated and whether the component description specifies the modified attribute.

If a component configuration has not been activated and it has a service registered, then a Configuration object change that leaves the component configuration satisfied will only cause the service properties of the service to be modified.

If a component description specifies the modified attribute and the changes to the target properties for the component configuration do not cause any references of the component configuration to become unsatisfied, SCR must modify the component properties for the component configuration. See *Modification* on page 240. A reference can become unsatisfied by a target property change if either:

- A bound service of a static reference is no longer a target service, or
- There are no target services for a mandatory dynamic reference.

Otherwise, the component configuration will become unsatisfied and SCR must deactivate that component configuration. SCR must attempt to satisfy the component configuration with the updated component properties.

112.8 Use of the Annotations

The Declarative Services Annotations provide a convenient way to create the component description XML during build time. Since annotations are placed in the source file and can use types, fields, and methods they can significantly simplify the use of Declarative Services.

The Declarative Services Annotations are build time annotations because one of the key aspect of Declarative Services is its laziness. Implementations can easily read the component description XML from the bundle, pre-process it, and cache the results between framework invocations. This is way it is unnecessary to create a class on the bundle when the bundle is started and/or scan the classes for annotations.

The Declarative Services Annotations are not inherited, they can only be used on a given class, annotations on its super class hierarchy or interfaces are not taken into account.

The primary annotation is the Component annotation. It indicates that a class is a component. It's defaults create the easiest to use component:

- Its name is the class name
- It registers all directly implemented interfaces as services
- The instance will be shared by all bundles
- It is enabled
- It is immediate if it has no services, otherwise it is delayed
- It has an optional configuration policy
- The configuration PID is the class name

For example, the following class registers a Speech service that can run on a Macintosh:

```
pubic interface Speech {
  void say(String what) throws Exception;
}

@Component
public class MacSpeech implements Speech {
   ScriptEngine engine =
     new ScriptEngineManager().getEngineByName("AppleScript");

    public void say(String message) throws Exception {
      engine.eval("say \"" + message.replace('"','\'' + "\"");
        }
    }
```

The previous example must generate the following XML:

```
<scr:component name='com.example.MacSpeech'>
  <implementation class='com.example.MacSpeech'/>
  <service>
    <provide interface='com.example.service.speech.Speech'/>
  </service>
</component>
```

It is possible to add activate and deactivate methods on the component with the Activate and Deactivate annotations. If the component wants to be updated for changes in the configuration properties than it can also indicated the modified method with the Modified annotation. For example:

```
@Activate
void open(Map<String,?> properties) { ... }

@Deactivate
void close() { ... }

@Modified
void modified(Map<String,?> properties) { ... }
```

If a component has dependencies on other services then they can be referenced with the Reference annotation that is applied to the bind method. The defaults for the reference annotations are:

- The name of the bind method is used for the name of the reference.
- 1:1 Cardinality.
- Static reluctant policy.
- The requested service is the type of the first argument of the method the Reference annotation is applied to.
- It will infer a default unset method and updated method based on the method name it is applied to.

For example:

```
@Reference(cardinality=MULTIPLE, policy=DYNAMIC)
void setLogService( LogService log, Map<String,?> props ) { ... }
void unsetLogService( LogService log ) {  ... }
void updatedLogService( Map<String,?> map ) { ...}
```

112.9 Service Component Runtime

112.9.1 Relationship to OSGi Framework

The SCR must have access to the Bundle Context of any bundle that contains a component. The SCR needs access to the Bundle Context for the following reasons:

- To be able to register and get services on behalf of a bundle with components.
- To interact with the Configuration Admin on behalf of a bundle with components.
- To provide a component its Bundle Context when the Component Context getBundleContext method is called.

The SCR should use the Bundle.getBundleContext() method to obtain the Bundle Context reference.

112.9.2 Starting and Stopping SCR

When SCR is implemented as a bundle, any component configurations activated by SCR must be deactivated when the SCR bundle is stopped. When the SCR bundle is started, it must process any components that are declared in bundles that are started. This includes bundles which are started and are awaiting lazy activation.

112.9.3 Logging Error Messages

When SCR must log an error message to the Log Service, it must use a Log Service obtained using the component's Bundle Context so that the resulting Log Entry is associated with the component's bundle.

If SCR is unable to obtain, or use, a Log Service using the component's Bundle Context, then SCR must log the error message to a Log Service obtained using SCR's bundle context to ensure the error message is logged.

112.9.4 Locating Component Methods

SCR will need to locate activate, deactivate, modified, bind, updated, and unbind methods for a component instance. These methods will be located, and called, using reflection. The declared methods of each class in the component implementation class' hierarchy are examined for a suitable method. If a suitable method is found in a class, and it is accessible to the component implementation class, then that method must be used. If suitable methods are found in a class but none of the suitable methods are accessible by the component implementation class, then the search for suitable methods terminates with no suitable method having been located. If no suitable methods are found in a class, the search continues in the superclass.

Only methods that are accessible, [5] *Access Control Java Language Specification*, to the component implementation class will be used. If the method has the `public` or `protected` access modifier, then access is permitted. Otherwise, if the method has the `private` access modifier, then access is permitted only if the method is declared in the component implementation class. Otherwise, if the method has default access, also known as package private access, then access is permitted only if the method is declared in the component implementation class or if the method is declared in a superclass and all classes in the hierarchy from the component implementation class to the superclass, inclusive, are in the same package and loaded by the same class loader.

It is recommended that these methods should not be declared with the `public` access modifier so that they do not appear as public methods on the component instance when it is used as a service object. Having these methods declared `public` allows any code to call the methods with reflection, even if a Security Manager is installed. These methods are generally intended to only be called by SCR.

112.9.5 Bundle Activator Interaction

A bundle containing components may also declare a Bundle Activator. Such a bundle may also be marked for lazy activation. Since components are activated by SCR and Bundle Activators are called by the OSGi Framework, a bundle using both components and a Bundle Activator must take care. The Bundle Activator's start method must not rely upon SCR having activated any of the bundle's components. However, the components can rely upon the Bundle Activator's start method having been called. That is, there is a *happens-before* relationship between the Bundle Activator's start method being run and the components being activated.

112.10 Security

112.10.1 Service Permissions

Declarative services are built upon the existing OSGi service infrastructure. This means that Service Permission applies regarding the ability to publish, find or bind services.

If a component specifies a service, then component configurations for the component cannot be satisfied unless the component's bundle has `ServicePermission[<provides>, REGISTER]` for each provided interface specified for the service.

If a component's reference does not specify optional cardinality, the reference cannot be satisfied unless the component's bundle has `ServicePermission[<interface>, GET]` for the specified interface in the reference. If the reference specifies optional cardinality but the component's bundle does not have `ServicePermission[<interface>, GET]` for the specified interface in the reference, no service must be bound for this reference.

If a component is a factory component, then the above Service Permission checks still apply. But the component's bundle is not required to have `ServicePermission[ComponentFactory, REGISTER]` as the Component Factory service is registered by SCR.

112.10.2 Required Admin Permission

The SCR requires `AdminPermission[*,CONTEXT]` because it needs access to the bundle's Bundle Context object with the `Bundle.getBundleContext()` method.

112.10.3 Using hasPermission

SCR does all publishing, finding and binding of services on behalf of the component using the Bundle Context of the component's bundle. This means that normal stack-based permission checks will check SCR and not the component's bundle. Since SCR is registering and getting services on behalf of a component's bundle, SCR must call the `Bundle.hasPermission` method to validate that a component's bundle has the necessary permission to register or get a service.

112.11 Component Description Schema

This XML Schema defines the component description grammar.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:scr="http://www.osgi.org/xmlns/scr/v1.2.0"
   targetNamespace="http://www.osgi.org/xmlns/scr/v1.2.0"
   elementFormDefault="unqualified"
   attributeFormDefault="unqualified"
   version="1.2.0">

   <annotation>
      <documentation xml:lang="en">
         This is the XML Schema for component descriptions used by
         the Service Component Runtime (SCR). Component description
         documents may be embedded in other XML documents. SCR will
         process all XML documents listed in the Service-Component
         manifest header of a bundle. XML documents containing
         component descriptions may contain a single, root component
         element or one or more component elements embedded in a
         larger document. Use of the namespace for component
         descriptions is mandatory. The attributes and subelements
         of a component element are always unqualified.
      </documentation>
   </annotation>
   <element name="component" type="scr:Tcomponent" />
   <complexType name="Tcomponent">
      <sequence>
```

```
            <annotation>
                <documentation xml:lang="en">
                    Implementations of SCR must not require component
                    descriptions to specify the subelements of the component
                    element in the order as required by the schema. SCR
                    implementations must allow other orderings since
                    arbitrary orderings do not affect the meaning of the
                    component description. Only the relative ordering of
                    property and properties element have meaning.
                </documentation>
            </annotation>
            <choice minOccurs="0" maxOccurs="unbounded">
                <element name="property" type="scr:Tproperty" />
                <element name="properties" type="scr:Tproperties" />
            </choice>
            <element name="service" type="scr:Tservice" minOccurs="0"
                maxOccurs="1" />
            <element name="reference" type="scr:Treference"
                minOccurs="0" maxOccurs="unbounded" />
            <element name="implementation" type="scr:Timplementation"
                minOccurs="1" maxOccurs="1" />
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="enabled" type="boolean" default="true"
            use="optional" />
        <attribute name="name" type="token" use="optional">
            <annotation>
                <documentation xml:lang="en">
                    The default value of this attribute is the value of
                    the class attribute of the nested implementation
                    element. If multiple component elements use the same
                    value for the class attribute of their nested
                    implementation element, then using the default value
                    for this attribute will result in duplicate names.
                    In this case, this attribute must be specified with
                    a unique value.
                </documentation>
            </annotation>
        </attribute>
        <attribute name="factory" type="string" use="optional" />
        <attribute name="immediate" type="boolean" use="optional" />
        <attribute name="configuration-policy"
            type="scr:Tconfiguration-policy" default="optional" use="optional" />
        <attribute name="activate" type="token" use="optional"
            default="activate" />
        <attribute name="deactivate" type="token" use="optional"
            default="deactivate" />
        <attribute name="modified" type="token" use="optional" />
        <attribute name="configuration-pid" type="token" use="optional">
            <annotation>
                <documentation xml:lang="en">
                    The default value of this attribute is the value of
                    the name attribute of this element.
                </documentation>
            </annotation>
        </attribute>
        <anyAttribute />
    </complexType>
    <complexType name="Timplementation">
        <sequence>
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="class" type="token" use="required" />
        <anyAttribute />
    </complexType>
    <complexType name="Tproperty">
        <simpleContent>
            <extension base="string">
                <attribute name="name" type="string" use="required" />
                <attribute name="value" type="string" use="optional" />
                <attribute name="type" type="scr:Tjava-types"
                    default="String" use="optional" />
                <anyAttribute />
            </extension>
        </simpleContent>
    </complexType>
    <complexType name="Tproperties">
        <sequence>
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="entry" type="string" use="required" />
        <anyAttribute />
    </complexType>
    <complexType name="Tservice">
        <sequence>
            <element name="provide" type="scr:Tprovide" minOccurs="1"
                maxOccurs="unbounded" />
            <!-- It is non-deterministic, per W3C XML Schema 1.0:
            http://www.w3.org/TR/xmlschema-1/#cos-nonambig
            to use name space="##any" below. -->
            <any namespace="##other" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="servicefactory" type="boolean" default="false"
            use="optional" />
        <anyAttribute />
    </complexType>
    <complexType name="Tprovide">
        <sequence>
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="interface" type="token" use="required" />
        <anyAttribute />
    </complexType>
    <complexType name="Treference">
        <sequence>
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="name" type="token" use="optional">
            <annotation>
                <documentation xml:lang="en">
                    The default value of this attribute is the value of
                    the interface attribute of this element. If multiple
                    instances of this element within a component element
                    use the same value for the interface attribute, then
                    using the default value for this attribute will result
                    in duplicate names. In this case, this attribute
                    must be specified with a unique value.
                </documentation>
            </annotation>
        </attribute>
        <attribute name="interface" type="token" use="required" />
        <attribute name="cardinality" type="scr:Tcardinality"
            default="1..1" use="optional" />
        <attribute name="policy" type="scr:Tpolicy" default="static"
            use="optional" />
        <attribute name="policy-option" type="scr:Tpolicy-option"
            default="reluctant" use="optional" />
        <attribute name="target" type="string" use="optional" />
        <attribute name="bind" type="token" use="optional" />
        <attribute name="unbind" type="token" use="optional" />
        <attribute name="updated" type="token" use="optional" />
        <anyAttribute />
    </complexType>
    <simpleType name="Tjava-types">
        <restriction base="string">
            <enumeration value="String" />
            <enumeration value="Long" />
            <enumeration value="Double" />
            <enumeration value="Float" />
            <enumeration value="Integer" />
            <enumeration value="Byte" />
            <enumeration value="Character" />
            <enumeration value="Boolean" />
            <enumeration value="Short" />
```

```
			</restriction>
		</simpleType>
		<simpleType name="Tcardinality">
			<restriction base="string">
				<enumeration value="0..1" />
				<enumeration value="0..n" />
				<enumeration value="1..1" />
				<enumeration value="1..n" />
			</restriction>
		</simpleType>
		<simpleType name="Tpolicy">
			<restriction base="string">
				<enumeration value="static" />
				<enumeration value="dynamic" />
			</restriction>
		</simpleType>
		<simpleType name="Tpolicy-option">
			<restriction base="string">
				<enumeration value="reluctant" />
				<enumeration value="greedy" />
			</restriction>
		</simpleType>
		<simpleType name="Tconfiguration-policy">
			<restriction base="string">
				<enumeration value="optional" />
				<enumeration value="require" />
				<enumeration value="ignore" />
			</restriction>
		</simpleType>
		<attribute name="must-understand" type="boolean">
			<annotation>
				<documentation xml:lang="en">
					This attribute should be used by extensions to documents
					to require that the document consumer understand the
					extension. This attribute must be qualified when used.
				</documentation>
			</annotation>
		</attribute>
	</schema>
```

SCR must not require component descriptions to specify the elements in the order required by the schema. SCR must allow other orderings since arbitrary orderings of these elements do not affect the meaning of the component description. Only the relative ordering of `property`, `properties` and `reference` elements have meaning for overriding previously set property values.

The schema is also available in digital form from [6] *OSGi XML Schemas*.

112.12 Changes

- Added a section to clarify the interaction between component activation and Bundle Activator execution.
- Added an updated method to the `reference` element to receive services updates for bound references
- Added a `configuration-pid` attribute to the component element to allow the configuration PID to be separate from the component name.
- Added a new policy-option attribute to the reference element to allow references to be greedy for rebinding.
- Added build time annotations

112.13 org.osgi.service.component

Service Component Package Version 1.2.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.component; version="[1.2,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.component; version="[1.2,1.3)"

112.13.1 Summary

- ComponentConstants – Defines standard names for Service Component constants.
- ComponentContext – A Component Context object is used by a component instance to interact with its execution context including locating services by reference name.
- ComponentException – Unchecked exception which may be thrown by the Service Component Runtime.
- ComponentFactory – When a component is declared with the factory attribute on its component element, the Service Component Runtime will register a Component Factory service to allow new component configurations to be created and activated rather than automatically creating and activating component configuration as necessary.
- ComponentInstance – A ComponentInstance encapsulates a component instance of an activated component configuration.

112.13.2 Permissions

112.13.3 public interface ComponentConstants

Defines standard names for Service Component constants.

No Implement Consumers of this API must not implement this interface

112.13.3.1 public static final String COMPONENT_FACTORY = "component.factory"

A service registration property for a Component Factory that contains the value of the factory attribute. The value of this property must be of type String.

112.13.3.2 public static final String COMPONENT_ID = "component.id"

A component property that contains the generated id for a component configuration. The value of this property must be of type Long.

The value of this property is assigned by the Service Component Runtime when a component configuration is created. The Service Component Runtime assigns a unique value that is larger than all previously assigned values since the Service Component Runtime was started. These values are NOT persistent across restarts of the Service Component Runtime.

112.13.3.3 public static final String COMPONENT_NAME = "component.name"

A component property for a component configuration that contains the name of the component as specified in the name attribute of the component element. The value of this property must be of type String.

112.13.3.4 public static final int DEACTIVATION_REASON_BUNDLE_STOPPED = 6

The component configuration was deactivated because the bundle was stopped.

Since 1.1

112.13.3.5 public static final int DEACTIVATION_REASON_CONFIGURATION_DELETED = 4

The component configuration was deactivated because its configuration was deleted.

Since 1.1

112.13.3.6 public static final int DEACTIVATION_REASON_CONFIGURATION_MODIFIED = 3

The component configuration was deactivated because its configuration was changed.

Since 1.1

112.13.3.7 public static final int DEACTIVATION_REASON_DISABLED = 1

The component configuration was deactivated because the component was disabled.

Since 1.1

112.13.3.8 public static final int DEACTIVATION_REASON_DISPOSED = 5

The component configuration was deactivated because the component was disposed.

Since 1.1

112.13.3.9 public static final int DEACTIVATION_REASON_REFERENCE = 2

The component configuration was deactivated because a reference became unsatisfied.

Since 1.1

112.13.3.10 public static final int DEACTIVATION_REASON_UNSPECIFIED = 0

The reason the component configuration was deactivated is unspecified.

Since 1.1

112.13.3.11 public static final String REFERENCE_TARGET_SUFFIX = ".target"

The suffix for reference target properties. These properties contain the filter to select the target services for a reference. The value of this property must be of type `String`.

112.13.3.12 public static final String SERVICE_COMPONENT = "Service-Component"

Manifest header specifying the XML documents within a bundle that contain the bundle's Service Component descriptions.

The attribute value may be retrieved from the `Dictionary` object returned by the `Bundle.getHeaders` method.

112.13.4 public interface ComponentContext

A Component Context object is used by a component instance to interact with its execution context including locating services by reference name. Each component instance has a unique Component Context.

A component instance may have an activate method. If a component instance has a suitable and accessible activate method, this method will be called when a component configuration is activated. If the activate method takes a `ComponentContext` argument, it will be passed the component instance's Component Context object. If the activate method takes a `BundleContext` argument, it will be passed the component instance's Bundle Context object. If the activate method takes a `Map` argument, it will be passed an unmodifiable Map containing the component properties.

A component instance may have a deactivate method. If a component instance has a suitable and accessible deactivate method, this method will be called when the component configuration is deactivated. If the deactivate method takes a `ComponentContext` argument, it will be passed the component instance's Component Context object. If the deactivate method takes a `BundleContext` argument, it will be passed the component instance's Bundle Context object. If the deactivate method takes a `Map` argument, it will be passed an unmodifiable Map containing the component properties. If the deactivate method takes an `int` or `Integer` argument, it will be passed the reason code for the component instance's deactivation.

Concurrency Thread-safe

No Implement Consumers of this API must not implement this interface

112.13.4.1 public void disableComponent (String name)

name The name of a component.

□ Disables the specified component name. The specified component name must be in the same bundle as this component.

112.13.4.2 public void enableComponent (String name)

name The name of a component or null to indicate all components in the bundle.

□ Enables the specified component name. The specified component name must be in the same bundle as this component.

112.13.4.3 public BundleContext getBundleContext ( )

□ Returns the BundleContext of the bundle which contains this component.

Returns The BundleContext of the bundle containing this component.

112.13.4.4 public ComponentInstance getComponentInstance ( )

□ Returns the Component Instance object for the component instance associated with this Component Context.

Returns The Component Instance object for the component instance.

112.13.4.5 public Dictionary getProperties ( )

□ Returns the component properties for this Component Context.

Returns The properties for this Component Context. The Dictionary is read only and cannot be modified.

112.13.4.6 public ServiceReference getServiceReference ( )

□ If the component instance is registered as a service using the service element, then this method returns the service reference of the service provided by this component instance.

This method will return null if the component instance is not registered as a service.

Returns The ServiceReference object for the component instance or null if the component instance is not registered as a service.

112.13.4.7 public Bundle getUsingBundle ( )

□ If the component instance is registered as a service using the servicefactory="true" attribute, then this method returns the bundle using the service provided by the component instance.

This method will return null if:

- The component instance is not a service, then no bundle can be using it as a service.
- The component instance is a service but did not specify the servicefactory="true" attribute, then all bundles using the service provided by the component instance will share the same component instance.
- The service provided by the component instance is not currently being used by any bundle.

Returns The bundle using the component instance as a service or null.

112.13.4.8 public Object locateService (String name)

name The name of a reference as specified in a reference element in this component's description.

□ Returns the service object for the specified reference name.

If the cardinality of the reference is 0..n or 1..n and multiple services are bound to the reference, the service with the highest ranking (as specified in its Constants.SERVICE_RANKING property) is returned. If there is a tie in ranking, the service with the lowest service ID (as specified in its Constants.SERVICE_ID property); that is, the service that was registered first is returned.

Returns A service object for the referenced service or null if the reference cardinality is 0..1 or 0..n and no bound service is available.

Throws ComponentException – If the Service Component Runtime catches an exception while activating the bound service.

112.13.4.9 public Object locateService (String name , ServiceReference reference)

name The name of a reference as specified in a reference element in this component's description.

reference The ServiceReference to a bound service. This must be a ServiceReference provided to the component via the bind or unbind method for the specified reference name.

□ Returns the service object for the specified reference name and ServiceReference.

Returns A service object for the referenced service or null if the specified ServiceReference is not a bound service for the specified reference name.

Throws ComponentException – If the Service Component Runtime catches an exception while activating the bound service.

112.13.4.10 public Object[] locateServices (String name)

name The name of a reference as specified in a reference element in this component's description.

□ Returns the service objects for the specified reference name.

Returns An array of service objects for the referenced service or null if the reference cardinality is 0..1 or 0..n and no bound service is available. If the reference cardinality is 0..1 or 1..1 and a bound service is available, the array will have exactly one element.

Throws ComponentException – If the Service Component Runtime catches an exception while activating a bound service.

112.13.5 public class ComponentException extends RuntimeException

Unchecked exception which may be thrown by the Service Component Runtime.

112.13.5.1 public ComponentException (String message , Throwable cause)

message The message for the exception.

cause The cause of the exception. May be null.

□ Construct a new ComponentException with the specified message and cause.

112.13.5.2 public ComponentException (String message)

message The message for the exception.

□ Construct a new ComponentException with the specified message.

112.13.5.3 public ComponentException (Throwable cause)

cause The cause of the exception. May be null.

□ Construct a new ComponentException with the specified cause.

112.13.5.4 public Throwable getCause ( )

□ Returns the cause of this exception or null if no cause was set.

Returns The cause of this exception or `null` if no cause was set.

112.13.5.5 public Throwable initCause (Throwable cause)

cause The cause of this exception.

□ Initializes the cause of this exception to the specified value.

Returns This exception.

Throws `IllegalArgumentException` – If the specified cause is this exception.

`IllegalStateException` – If the cause of this exception has already been set.

112.13.6 public interface ComponentFactory

When a component is declared with the `factory` attribute on its `component` element, the Service Component Runtime will register a Component Factory service to allow new component configurations to be created and activated rather than automatically creating and activating component configuration as necessary.

Concurrency Thread-safe

No Implement Consumers of this API must not implement this interface

112.13.6.1 public ComponentInstance newInstance (Dictionary properties)

properties Additional properties for the component configuration or `null` if there are no additional properties.

□ Create and activate a new component configuration. Additional properties may be provided for the component configuration.

Returns A `ComponentInstance` object encapsulating the component instance of the component configuration. The component configuration has been activated and, if the component specifies a `service` element, the component instance has been registered as a service.

Throws `ComponentException` – If the Service Component Runtime is unable to activate the component configuration.

112.13.7 public interface ComponentInstance

A ComponentInstance encapsulates a component instance of an activated component configuration. ComponentInstances are created whenever a component configuration is activated.

ComponentInstances are never reused. A new ComponentInstance object will be created when the component configuration is activated again.

Concurrency Thread-safe

No Implement Consumers of this API must not implement this interface

112.13.7.1 public void dispose ( )

□ Dispose of the component configuration for this component instance. The component configuration will be deactivated. If the component configuration has already been deactivated, this method does nothing.

112.13.7.2 public Object getInstance ( )

□ Returns the component instance of the activated component configuration.

Returns The component instance or `null` if the component configuration has been deactivated.

112.14 org.osgi.service.component.annotations

Service Component Annotations Package Version 1.2.

This package is not used at runtime. Annotated classes are processed by tools to generate Component Descriptions which are used at runtime.

112.14.1 Summary

- `Activate` – Identify the annotated method as the `activate` method of a Service Component.
- `Component` – Identify the annotated class as a Service Component.
- `ConfigurationPolicy` – Configuration Policy for the `Component` annotation.
- `Deactivate` – Identify the annotated method as the `deactivate` method of a Service Component.
- `Modified` – Identify the annotated method as the `modified` method of a Service Component.
- `Reference` – Identify the annotated method as a `bind` method of a Service Component.
- `ReferenceCardinality` – Cardinality for the `Reference` annotation.
- `ReferencePolicy` – Policy for the `Reference` annotation.
- `ReferencePolicyOption` – Policy option for the `Reference` annotation.

112.14.2 Permissions

112.14.3 @Activate

Identify the annotated method as the `activate` method of a Service Component.

The annotated method is the activate method of the Component.

This annotation is not processed at runtime by a Service Component Runtime implementation. It must be processed by tools and used to add a Component Description to the bundle.

See Also `The activate attribute of the component element of a Component Description.`

Since 1.1

Retention `CLASS`

Target `METHOD`

112.14.4 @Component

Identify the annotated class as a Service Component.

The annotated class is the implementation class of the Component.

This annotation is not processed at runtime by a Service Component Runtime implementation. It must be processed by tools and used to add a Component Description to the bundle.

See Also `The component element of a Component Description.`

Retention `CLASS`

Target `TYPE`

112.14.4.1 String name default “”

□ The name of this Component.

If not specified, the name of this Component is the fully qualified type name of the class being annotated.

See Also `The name attribute of the component element of a Component Description.`

112.14.4.2 Class<?>[] service default {}

□ The types under which to register this Component as a service.

If no service should be registered, the empty value `{}` must be specified.

If not specified, the service types for this Component are all the *directly* implemented interfaces of the class being annotated.

See Also `The service element of a Component Description.`

112.14.4.3 String factory default ""

□ The factory identifier of this Component. Specifying a factory identifier makes this Component a Factory Component.

If not specified, the default is that this Component is not a Factory Component.

See Also `The factory attribute of the component element of a Component Description.`

112.14.4.4 boolean servicefactory default false

□ Declares whether this Component uses the OSGi ServiceFactory concept and each bundle using this Component's service will receive a different component instance.

If `true`, this Component uses the OSGi ServiceFactory concept. If `false` or not specified, this Component does not use the OSGi ServiceFactory concept.

See Also `The servicefactory attribute of the service element of a Component Description.`

112.14.4.5 boolean enabled default true

□ Declares whether this Component is enabled when the bundle containing it is started.

If `true`, this Component is enabled. If `false` or not specified, this Component is disabled.

See Also `The enabled attribute of the component element of a Component Description.`

112.14.4.6 boolean immediate default false

□ Declares whether this Component must be immediately activated upon becoming satisfied or whether activation should be delayed.

If `true`, this Component must be immediately activated upon becoming satisfied. If `false`, activation of this Component is delayed. If this property is specified, its value must be `false` if the `factory` property is also specified or must be `true` if the `service` property is specified with an empty value.

If not specified, the default is `false` if the `factory` property is specified or the `service` property is not specified or specified with a non-empty value and `true` otherwise.

See Also `The immediate attribute of the component element of a Component Description.`

112.14.4.7 String[] property default {}

□ Properties for this Component.

Each property string is specified as "`key=value`". The type of the property value can be specified in the key as `key:type=value`. The type must be one of the property types supported by the type attribute of the property element of a Component Description.

To specify a property with multiple values, use multiple key, value pairs. For example, "`foo=bar`", "`foo=baz`".

See Also `The property element of a Component Description.`

112.14.4.8 String[] properties default {}

□ Property entries for this Component.

Specifies the name of an entry in the bundle whose contents conform to a standard Java Properties File. The entry is read and processed to obtain the properties and their values.

See Also `The properties element of a Component Description.`

112.14.4.9 String xmlns default “”

- The XML name space of the Component Description for this Component.

  If not specified, the XML name space of the Component Description for this Component should be the lowest Declarative Services XML name space which supports all the specification features used by this Component.

See Also `The XML name space specified for a Component Description.`

112.14.4.10 ConfigurationPolicy configurationPolicy default ConfigurationPolicy.OPTIONAL

- The configuration policy of this Component.

  Controls whether component configurations must be satisfied depending on the presence of a corresponding Configuration object in the OSGi Configuration Admin service. A corresponding configuration is a Configuration object where the PID equals the name of the component.

  If not specified, the `OPTIONAL` configuration policy is used.

See Also `The configuration-policy attribute of the component element of a Component Description.`

Since 1.1

112.14.4.11 String configurationPid default “”

- The configuration PID for the configuration of this Component.

  Allows the configuration PID for this Component to be different than the name of this Component.

  If not specified, the name of this Component is used as the configuration PID of this Component.

See Also `The configuration-pid attribute of the component element of a Component Description.`

Since 1.2

112.14.5 enum ConfigurationPolicy

Configuration Policy for the `Component` annotation.

Controls whether component configurations must be satisfied depending on the presence of a corresponding Configuration object in the OSGi Configuration Admin service. A corresponding configuration is a Configuration object where the PID is the name of the component.

Since 1.1

112.14.5.1 OPTIONAL

Use the corresponding Configuration object if present but allow the component to be satisfied even if the corresponding Configuration object is not present.

112.14.5.2 REQUIRE

There must be a corresponding Configuration object for the component configuration to become satisfied.

112.14.5.3 IGNORE

Always allow the component configuration to be satisfied and do not use the corresponding Configuration object even if it is present.

112.14.6 @Deactivate

Identify the annotated method as the `deactivate` method of a Service Component.

The annotated method is the deactivate method of the Component.

This annotation is not processed at runtime by a Service Component Runtime implementation. It must be processed by tools and used to add a Component Description to the bundle.

See Also `The deactivate attribute of the component element of a Component Description.`

Since 1.1

Retention CLASS

Target METHOD

112.14.7 @Modified

Identify the annotated method as the `modified` method of a Service Component.

The annotated method is the modified method of the Component.

This annotation is not processed at runtime by a Service Component Runtime implementation. It must be processed by tools and used to add a Component Description to the bundle.

See Also `The modified attribute of the component element of a Component Description.`

Since 1.1

Retention CLASS

Target METHOD

112.14.8 @Reference

Identify the annotated method as a `bind` method of a Service Component.

The annotated method is a bind method of the Component.

This annotation is not processed at runtime by a Service Component Runtime implementation. It must be processed by tools and used to add a Component Description to the bundle.

In the generated Component Description for a component, the references must be ordered in ascending lexicographical order (using `String.compareTo`) of the reference `name`s.

See Also `The reference element of a Component Description.`

Retention CLASS

Target METHOD

112.14.8.1 String name default “”

□ The name of this reference.

If not specified, the name of this reference is based upon the name of the method being annotated. If the method name begins with `bind`, `set` or `add`, that is removed.

See Also `The name attribute of the reference element of a Component Description.`

112.14.8.2 Class<?> service default Object.class

□ The type of the service to bind to this reference.

If not specified, the type of the service to bind is based upon the type of the first argument of the method being annotated.

See Also `The interface attribute of the reference element of a Component Description.`

112.14.8.3 ReferenceCardinality cardinality default ReferenceCardinality.MANDATORY

□ The cardinality of the reference.

If not specified, the reference has a `1..1` cardinality.

See Also `The cardinality attribute of the reference element of a Component Description.`

112.14.8.4 ReferencePolicy policy default ReferencePolicy.STATIC

- The policy for the reference.

  If not specified, the `STATIC` reference policy is used.

See Also `The policy attribute of the reference element of a Component Description.`

112.14.8.5 String target default “”

- The target filter for the reference.

See Also `The target attribute of the reference element of a Component Description.`

112.14.8.6 String unbind default “”

- The name of the unbind method which is associated with the annotated bind method.

  To declare no unbind method, the value “`-`” must be used.

  If not specified, the name of the unbind method is derived from the name of the annotated bind method. If the annotated method name begins with `bind`, `set` or `add`, that is replaced with `unbind`, `unset` or `remove`, respectively, to derive the unbind method name. Otherwise, `un` is prefixed to the annotated method name to derive the unbind method name. The unbind method is only set if the component type contains a method with the derived name.

See Also `The unbind attribute of the reference element of a Component Description.`

112.14.8.7 ReferencePolicyOption policyOption default ReferencePolicyOption.RELUCTANT

- The policy option for the reference.

  If not specified, the `RELUCTANT` reference policy option is used.

See Also `The policy-option attribute of the reference element of a Component Description.`

Since 1.2

112.14.8.8 String updated default “”

- The name of the updated method which is associated with the annotated bind method.

  To declare no updated method, the value “`-`” must be used.

  If not specified, the name of the updated method is derived from the name of the annotated bind method. If the annotated method name begins with `bind`, `set` or `add`, that is replaced with `updated` to derive the updated method name. Otherwise, `updated` is prefixed to the annotated method name to derive the updated method name. The updated method is only set if the component type contains a method with the derived name.

See Also `The updated attribute of the reference element of a Component Description.`

Since 1.2

112.14.9 enum ReferenceCardinality

Cardinality for the `Reference` annotation.

Specifies if the reference is optional and if the component implementation support a single bound service or multiple bound services.

112.14.9.1 OPTIONAL

The reference is optional and unary. That is, the reference has a cardinality of `0..1`.

112.14.9.2 MANDATORY

The reference is mandatory and unary. That is, the reference has a cardinality of `1..1`.

112.14.9.3 MULTIPLE

The reference is optional and multiple. That is, the reference has a cardinality of 0..n.

112.14.9.4 AT_LEAST_ONE

The reference is mandatory and multiple. That is, the reference has a cardinality of 1..n.

112.14.10 enum ReferencePolicy

Policy for the Reference annotation.

112.14.10.1 STATIC

The static policy is the most simple policy and is the default policy. A component instance never sees any of the dynamics. Component configurations are deactivated before any bound service for a reference having a static policy becomes unavailable. If a target service is available to replace the bound service which became unavailable, the component configuration must be reactivated and bound to the replacement service.

112.14.10.2 DYNAMIC

The dynamic policy is slightly more complex since the component implementation must properly handle changes in the set of bound services. With the dynamic policy, SCR can change the set of bound services without deactivating a component configuration. If the component uses the event strategy to access services, then the component instance will be notified of changes in the set of bound services by calls to the bind and unbind methods.

112.14.11 enum ReferencePolicyOption

Policy option for the Reference annotation.

Since 1.2

112.14.11.1 RELUCTANT

The reluctant policy option is the default policy option for both static and dynamic reference policies. When a new target service for a reference becomes available, references having the reluctant policy option for the static policy or the dynamic policy with a unary cardinality will ignore the new target service. References having the dynamic policy with a multiple cardinality will bind the new target service.

112.14.11.2 GREEDY

The greedy policy option is a valid policy option for both static and dynamic reference policies. When a new target service for a reference becomes available, references having the greedy policy option will bind the new target service.

112.15 References

[1] *Automating Service Dependency Management in a Service-Oriented Component Model*
Humberto Cervantes, Richard S. Hall, Proceedings of the Sixth Component-Based Software Engineering Workshop, May 2003, pp. 91-96.
http://www-adele.imag.fr/Les.Publications/intConferences/CBSE2003Cer.pdf

[2] *Service Binder*
Humberto Cervantes, Richard S. Hall, http://gravity.sourceforge.net/servicebinder

[3] *Java Properties File*
http://java.sun.com/j2se/1.4.2/docs/api/java/util/Properties.html#load(java.io.InputStream)

[4] *Extensible Markup Language (XML) 1.0*
http://www.w3.org/TR/REC-xml/

[5] *Access Control Java Language Specification*
http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html#104285

[6] *OSGi XML Schemas*
http://www.osgi.org/Release4/XMLSchemas

113 Event Admin Service Specification

Version 1.3

113.1 Introduction

Nearly all the bundles in an OSGi framework must deal with events, either as an event publisher or as an event handler. So far, the preferred mechanism to disperse those events have been the service interface mechanism.

Dispatching events for a design related to X, usually involves a service of type `XListener`. However, this model does not scale well for fine grained events that must be dispatched to many different handlers. Additionally, the dynamic nature of the OSGi environment introduces several complexities because both event publishers and event handlers can appear and disappear at any time.

The Event Admin service provides an inter-bundle communication mechanism. It is based on a event *publish* and *subscribe* model, popular in many message based systems.

This specification defines the details for the participants in this event model.

113.1.1 Essentials

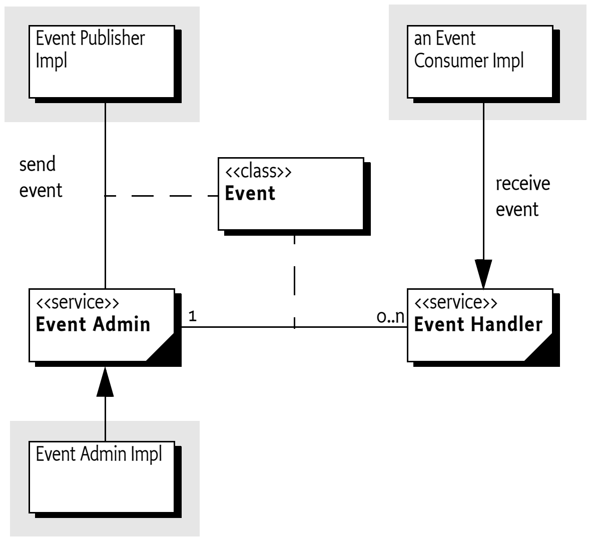
- *Simplifications* – The model must significantly simplify the process of programming an event source and an event handler.
- *Dependencies* – Handle the myriad of dependencies between event sources and event handlers for proper cleanup.
- *Synchronicity* – It must be possible to deliver events asynchronously or synchronously with the caller.
- *Event Window* – Only event handlers that are active when an event is published must receive this event, handlers that register later must not see the event.
- *Performance* – The event mechanism must impose minimal overhead in delivering events.
- *Selectivity* – Event listeners must only receive notifications for the event types for which they are interested
- *Reliability* – The Event Admin must ensure that events continue to be delivered regardless the quality of the event handlers.
- *Security* – Publishing and receiving events are sensitive operations that must be protected per event type.
- *Extensibility* – It must be possible to define new event types with their own data types.
- *Native Code* – Events must be able to be passed to native code or come from native code.
- *OSGi Events* – The OSGi Framework, as well as a number of OSGi services, already have number of its own events defined. For uniformity of processing, these have to be mapped into generic event types.

113.1.2 Entities

- *Event* – An `Event` object has a topic and a `Dictionary` object that contains the event properties. It is an immutable object.
- *Event Admin* – The service that provides the publish and subscribe model to Event Handlers and Event Publishers.
- *Event Handler* – A service that receives and handles `Event` objects.

- *Event Publisher* – A bundle that sends event through the Event Admin service.
- *Event Subscriber* – Another name for an Event Handler.
- *Topic* – The name of an Event type.
- *Event Properties* – The set of properties that is associated with an Event.

Figure 113.1 The Event Admin service org.osgi.service.event package



113.1.3 Synopsis

The Event Admin service provides a place for bundles to publish events, regardless of their destination. It is also used by Event Handlers to subscribe to specific types of events.

Events are published under a topic, together with a number of event properties. Event Handlers can specify a filter to control the Events they receive on a very fine grained basis.

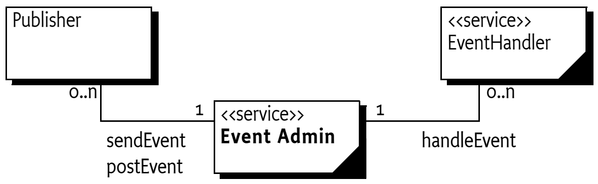
113.1.4 What To Read

- *Architects* – The *Event Admin Architecture* on page 266 provides an overview of the Event Admin service.
- *Event Publishers* – The *Event Publisher* on page 269 provides an introduction of how to write an Event Publisher. The *Event Admin Architecture* on page 266 provides a good overview of the design.
- *Event Subscribers/Handlers* – The *Event Handler* on page 268 provides the rules on how to subscribe and handle events.

113.2 Event Admin Architecture

The Event Admin is based on the *Publish-Subscribe* pattern. This pattern decouples sources from their handlers by interposing an *event channel* between them. The publisher posts events to the channel, which identifies which handlers need to be notified and then takes care of the notification process. This model is depicted in Figure 113.2.

Figure 113.2 Channel Pattern



In this model, the event source and event handler are completely decoupled because neither has any direct knowledge of the other. The complicated logic of monitoring changes in the event publishers and event handlers is completely contained within the event channel. This is highly advantageous in an OSGi environment because it simplifies the process of both sending and receiving events.

113.3 The Event

Events have the following attributes:

- *Topic* – A topic that defines what happened. For example, when a bundle is started an event is published that has a topic of `org/osgi/framework/BundleEvent/STARTED`.
- *Properties* – Zero or more properties that contain additional information about the event. For example, the previous example event has a property of `bundle.id` which is set to a `Long` object, among other properties.

113.3.1 Topics

The topic of an event defines the *type* of the event. It is fairly granular in order to give handlers the opportunity to register for just the events they are interested in. When a topic is designed, its name should not include any other information, such as the publisher of the event or the data associated with the event, those parts are intended to be stored in the event properties.

The topic is intended to serve as a first-level filter for determining which handlers should receive the event. Event Admin service implementations use the structure of the topic to optimize the dispatching of the events to the handlers.

Topics are arranged in a hierarchical namespace. Each level is defined by a token and levels are separated by slashes. More precisely, the topic must conform to the following grammar:

```
topic ::= token ( '/' token ) *    // See 1.3.2 Core book
```

Topics should be designed to become more specific when going from left to right. Handlers can provide a prefix that matches a topic, using the preferred order allows a handler to minimize the number of prefixes it needs to register.

Topics are case-sensitive. As a convention, topics should follow the reverse domain name scheme used by Java packages to guarantee uniqueness. The separator must be slashes ('/' `\u002F`) instead of the full stop ('.' `\u002E`).

This specification uses the convention `fully/qualified/package/ClassName/ACTION`. If necessary, a pseudo-class-name is used.

113.3.2 Properties

Information about the actual event is provided as properties. The property name is a case-sensitive string and the value can be any object. Although any Java object can be used as a property value, only `String` objects and the eight primitive types (plus their wrappers) should be used. Other types cannot be passed to handlers that reside external from the Java VM.

Another reason that arbitrary classes should not be used is the mutability of objects. If the values are not immutable, then any handler that receives the event could change the value. Any handlers that received the event subsequently would see the altered value and not the value as it was when the event was sent.

The topic of the event is available as a property with the key EVENT_TOPIC. This allows filters to include the topic as a condition if necessary.

113.3.3 High Performance

An event processing system can become a bottleneck in large systems. One expensive aspect of the Event object is its properties and its immutability. This combination requires the Event object to create a copy of the properties for each object. There are many situations where the same properties are dispatched through Event Admin, the topic is then used to signal the information. Creating the copy of the properties can therefore take unnecessary CPU time and memory. However, the immutability of the Event object requires the properties to be immutable.

For this reason, this specification also provides an immutable Map with the Event Properties class. This class implements an immutable map that is recognized and trusted by the Event object to not mutate. Using an Event Properties object allows a client to create many different Event objects with different topics but sharing the same properties object.

The following example shows how an event poster can limit the copying of the properties.

```
void foo(EventAdmin eventAdmin) {
   Map<String,Object> props = new HashMap<String,Object>();
   props.put("foo", 1);
   EventProperties eventProps = new EventProperties( props );

   for ( int i=0; i<1000; i++)
      eventAdmin.postEvent( new Event( "my/topic/" + i, eventProps) );
}
```

113.4 Event Handler

Event handlers must be registered as services with the OSGi framework under the object class org.osgi.service.event.EventHandler.

Event handlers should be registered with a property (constant from the EventConstants class) EVENT_TOPIC. The value being a String or String[] object that describes which *topics* the handler is interested in. A wildcard ('*' \u002A) may be used as the last token of a topic name, for example com/action/*. This matches any topic that shares the same first tokens. For example, com/action/* matches com/action/listen.

Event Handlers which have not specified the EVENT_TOPIC service property must not receive events.

The value of each entry in the EVENT_TOPIC service registration property must conform to the following grammar:

```
topic-scope ::= '*' | ( topic '/*'? )
```

Event handlers can also be registered with a service property named EVENT_FILTER. The value of this property must be a string containing a Framework filter specification. Any of the event's properties can be used in the filter expression.

```
event-filter ::= filter         // 3.2.7 Core book
```

Each Event Handler is notified for any event which belongs to the topics the handler has expressed an interest in. If the handler has defined a EVENT_FILTER service property then the event properties must also match the filter expression. If the filter is an error, then the Event Admin service should log a warning and further ignore the Event Handler.

For example, a bundle wants to see all Log Service events with a level of WARNING or ERROR, but it must ignore the INFO and DEBUG events. Additionally, the only events of interest are when the bundle symbolic name starts with com.acme.

```
public AcmeWatchDog implements BundleActivator,
      EventHandler {
   final static String [] topics = new String[] {
      "org/osgi/service/log/LogEntry/LOG_WARNING",
      "org/osgi/service/log/LogEntry/LOG_ERROR" };

   public void start(BundleContext context) {
      Dictionary d = new Hashtable();
      d.put(EventConstants.EVENT_TOPIC, topics );
      d.put(EventConstants.EVENT_FILTER,
         "(bundle.symbolicName=com.acme.*)" );
      context.registerService( EventHandler.class.getName(),
         this, d );
   }
   public void stop( BundleContext context) {}

   public void handleEvent(Event event ) {
      //...
   }
}
```

If there are multiple Event Admin services registered with the Framework then all Event Admin services must send their published events to all registered Event Handlers.

113.4.1 Ordering

In the default case, an Event Handler will receive posted (asynchronous) events from a single thread in the same order as they were posted. Maintaining this ordering guarantee requires the Event Admin to serialize the delivery of events instead of, for example, delivering the events on different worker threads. There are many scenarios where this ordering is not really required. For this reason, an Event Handler can signal to the Event Admin that events can be delivered out of order. This is notified with the EVENT_DELIVERY service property. This service property can be used in the following way:

- Not set or set to both – The Event Admin must deliver the events in the proper order.
- DELIVERY_ASYNC_ORDERED – Events must be delivered in order.
- DELIVERY_ASYNC_UNORDERED – Allow the events to be delivered in any order.

113.5 Event Publisher

To fire an event, the event source must retrieve the Event Admin service from the OSGi service registry. Then it creates the event object and calls one of the Event Admin service's methods to fire the event either synchronously or asynchronously.

The following example is a class that publishes a time event every 60 seconds.

```
public class TimerEvent extends Thread
   implements BundleActivator {
   Hashtable          time = new Hashtable();
   ServiceTracker     tracker;
```

```
    public TimerEvent() { super("TimerEvent"); }

    public void start(BundleContext context ) {
      tracker = new ServiceTracker(context,
        EventAdmin.class.getName(), null );
      tracker.open();
      start();
    }

    public void stop( BundleContext context ) {
      interrupt();
      tracker.close();
    }

    public void run() {
        while ( ! Thread.interrupted() ) try {
          Calendar   c = Calendar.getInstance();
          set(c,Calendar.MINUTE,"minutes");
          set(c,Calendar.HOUR,"hours");
          set(c,Calendar.DAY_OF_MONTH,"day");
          set(c,Calendar.MONTH,"month");
          set(c,Calendar.YEAR,"year");

          EventAdmin ea =
            (EventAdmin) tracker.getService();
          if ( ea != null )
            ea.sendEvent(new Event("com/acme/timer",
              time ));
          Thread.sleep(60000-c.get(Calendar.SECOND)*1000);
        } catch( InterruptedException e ) {
          return;
        }
    }

    void set( Calendar c, int field, String key ) {
      time.put( key, new Integer(c.get(field)) );
    }
  }
```

113.6 Specific Events

113.6.1 General Conventions

Some handlers are more interested in the contents of an event rather than what actually happened. For example, a handler wants to be notified whenever an Exception is thrown anywhere in the system. Both Framework Events and Log Entry events may contain an exception that would be of interest to this hypothetical handler. If both Framework Events and Log Entries use the same property names then the handler can access the Exception in exactly the same way. If some future event type follows the same conventions then the handler can receive and process the new event type even though it had no knowledge of it when it was compiled.

The following properties are suggested as conventions. When new event types are defined they should use these names with the corresponding types and values where appropriate. These values should be set only if they are not null

A list of these property names can be found in Table 113.1..

Table 113.1 *General property names for events*

Name	Type	Notes
BUNDLE_SIGNER	String | Collection <String>	A bundle's signers DN
BUNDLE_VERSION	Version	A bundle's version
BUNDLE_SYMBOLICNAME	String	A bundle's symbolic name
EVENT	Object	The actual event object. Used when rebroadcasting an event that was sent via some other event mechanism
EXCEPTION	Throwable	An exception or error
EXCEPTION_MESSAGE	String	Must be equal to exception.getMessage().
EXCEPTION_CLASS	String	Must be equal to the name of the Exception class.
MESSAGE	String	A human-readable message that is usually not localized.
SERVICE	Service Reference	A Service Reference
SERVICE_ID	Long	A service's id
SERVICE_OBJECTCLASS	String[]	A service's objectClass
SERVICE_PID	String | Collection <String>	A service's persistent identity. A PID that is specified with a String[] must be coerced into a Collection<String>.
TIMESTAMP	Long	The time when the event occurred, as reported by System.currentTimeMillis()

The topic of an OSGi event is constructed by taking the fully qualified name of the event class, substituting a slash for every period, and appending a slash followed by the name of the constant that defines the event type. For example, the topic of

```
BundleEvent.STARTED
```

Event becomes

```
org/osgi/framework/BundleEvent/STARTED
```

If a type code for the event is unknown then the event must be ignored.

113.6.2 OSGi Events

In order to present a consistent view of all the events occurring in the system, the existing Framework-level events are mapped to the Event Admin's publish-subscribe model. This allows event subscribers to treat framework events exactly the same as other events.

It is the responsibility of the Event Admin service implementation to map these Framework events to its queue.

The properties associated with the event depends on its class as outlined in the following sections.

113.6.3 Framework Event

Framework Events must be delivered asynchronously with a topic of:

```
org/osgi/framework/FrameworkEvent/<event type>
```

The following event types are supported:

```
STARTED
ERROR
PACKAGES_REFRESHED
STARTLEVEL_CHANGED
WARNING
INFO
```

Other events are ignored, no event will be send by the Event Admin. The following event properties must be set for a Framework Event.

- event – (FrameworkEvent) The original event object.

If the FrameworkEvent getBundle method returns a non-null value, the following fields must be set:

- bundle.id – (Long) The source's bundle id.
- bundle.symbolicName – (String) The source bundle's symbolic name. Only set if the bundle's symbolic name is not null.
- bundle.version – (Version) The version of the bundle, if set.
- bundle.signer – (String|Collection<String>) The DNs of the signers.
- bundle – (Bundle) The source bundle.

If the FrameworkEvent getThrowable method returns a non- null value:

- exception.class – (String) The fully-qualified class name of the attached Exception.
- exception.message –(String) The message of the attached exception. Only set if the Exception message is not null.
- exception – (Throwable) The Exception returned by the getThrowable method.

113.6.4 Bundle Event

Framework Events must be delivered asynchronously with a topic of:

```
org/osgi/framework/BundleEvent/<event type>
```

The following event types are supported:

```
INSTALLED
STARTED
STOPPED
UPDATED
UNINSTALLED
RESOLVED
UNRESOLVED
```

Unknown events must be ignored.

The following event properties must be set for a Bundle Event. If listeners require synchronous delivery then they should register a Synchronous Bundle Listener with the Framework.

- event – (BundleEvent) The original event object.
- bundle.id – (Long) The source's bundle id.
- bundle.symbolicName – (String) The source bundle's symbolic name. Only set if the bundle's symbolic name is not null.
- bundle.version – (Version) The version of the bundle, if set.
- bundle.signer – (String|Collection<String>) The DNs of the signers.

- bundle – (Bundle) The source bundle.

113.6.5 Service Event

Service Events must be delivered asynchronously with the topic:

```
org/osgi/framework/ServiceEvent/<event type>
```

The following event types are supported:

```
REGISTERED
MODIFIED
UNREGISTERING
```

Unknown events must be ignored.

- event – (ServiceEvent) The original Service Event object.
- service – (ServiceReference) The result of the getServiceReference method
- service.id – (Long) The service's ID.
- service.pid – (String or Collection<String>) The service's persistent identity. Only set if not null. If the PID is specified as a String[] then it must be coerced into a Collection<String>.
- service.objectClass – (String[]) The service's object class.

113.6.6 Other Event Sources

Several OSGi service specifications define their own event model. It is the responsibility of these services to map their events to Event Admin events. Event Admin is seen as a core service that will be present in most devices. However, if there is no Event Admin service present, applications are not mandated to buffer events.

113.7 Event Admin Service

The Event Admin service must be registered as a service with the object class org.osgi.service.event.EventAdmin. Multiple Event Admin services can be registered. Publishers should publish their event on the Event Admin service with the highest value for the SERVICE_RANKING service property. This is the service selected by the getServiceReference method.

The Event Admin service is responsible for tracking the registered handlers, handling event notifications and providing at least one thread for asynchronous event delivery.

113.7.1 Synchronous Event Delivery

Synchronous event delivery is initiated by the sendEvent method. When this method is invoked, the Event Admin service determines which handlers must be notified of the event and then notifies each one in turn. The handlers can be notified in the caller's thread or in an event-delivery thread, depending on the implementation. In either case, all notifications must be completely handled before the sendEvent method returns to the caller.

Synchronous event delivery is significantly more expensive than asynchronous delivery. All things considered equal, the asynchronous delivery should be preferred over the synchronous delivery.

Callers of this method will need to be coded defensively and assume that synchronous event notifications could be handled in a separate thread. That entails that they must not be holding any monitors when they invoke the sendEvent method. Otherwise they significantly increase the likelihood of deadlocks because Java monitors are not reentrant from another thread by definition. Not holding monitors is good practice even when the event is dispatched in the same thread.

113.7.2 Asynchronous Event Delivery

Asynchronous event delivery is initiated by the postEvent method. When this method is invoked, the Event Admin service must determine which handlers are interested in the event. By collecting this list of handlers during the method invocation, the Event Admin service ensures that only handlers that were registered at the time the event was posted will receive the event notification. This is the same as described in *Delivering Events* on page 106 of the Core specification.

The Event Admin service can use more than one thread to deliver events. If it does then it must guarantee that each handler receives the events in the same order as the events were posted, unless this handler allows unordered deliver, see *Ordering* on page 269. This ensures that handlers see events in their expected order. For example, for some handlers it would be an error to see a destroyed event before the corresponding created event.

Before notifying each handler, the event delivery thread must ensure that the handler is still registered in the service registry. If it has been unregistered then the handler must not be notified.

113.7.3 Order of Event Delivery

Asynchronous events are delivered in the order in which they arrive in the event queue. Thus if two events are posted by the same thread then they will be delivered in the same order (though other events may come between them). However, if two or more events are posted by different threads then the order in which they arrive in the queue (and therefore the order in which they are delivered) will depend very much on subtle timing issues. The event delivery system cannot make any guarantees in this case. An Event Handler can indicate that the ordering is not relevant, allowing the Event Admin to more aggressively parallelize the event deliver, see *Ordering* on page 269.

Synchronous events are delivered as soon as they are sent. If two events are sent by the same thread, one after the other, then they must be guaranteed to be processed serially and in the same order. However, if two events are sent by different threads then no guarantees can be made. The events can be processed in parallel or serially, depending on whether or not the Event Admin service dispatches synchronous events in the caller's thread or in a separate thread.

Note that if the actions of a handler trigger a synchronous event, then the delivery of the first event will be paused and delivery of the second event will begin. Once delivery of the second event has completed, delivery of the first event will resume. Thus some handlers may observe the second event before they observe the first one.

113.8 Reliability

113.8.1 Exceptions in callbacks

If a handler throws an Exception during delivery of an event, it must be caught by the Event Admin service and handled in some implementation specific way. If a Log Service is available the exception should be logged. Once the exception has been caught and dealt with, the event delivery must continue with the next handlers to be notified, if any.

As the Log Service can also forward events through the Event Admin service there is a potential for a loop when an event is reported to the Log Service.

113.8.2 Dealing with Stalled Handlers

Event handlers should not spend too long in the handleEvent method. Doing so will prevent other handlers in the system from being notified. If a handler needs to do something that can take a while, it should do it in a different thread.

An event admin implementation can attempt to detect stalled or deadlocked handlers and deal with them appropriately. Exactly how it deals with this situation is left as implementation specific. One allowed implementation is to mark the current event delivery thread as invalid and spawn a new event delivery thread. Event delivery must resume with the next handler to be notified.

Implementations can choose to blacklist any handlers that they determine are misbehaving. Blacklisted handlers must not be notified of any events. If a handler is blacklisted, the event admin should log a message that explains the reason for it.

113.9 Inter-operability with Native Applications

Implementations of the Event Admin service can support passing events to, and/or receiving events from native applications.

If the implementation supports native inter-operability, it must be able to pass the topic of the event and its properties to/from native code. Implementations must be able to support property values of the following types:

- String objects, including full Unicode support
- Integer, Long, Byte, Short, Float, Double, Boolean, Character objects
- Single-dimension arrays of the above types (including String)
- Single-dimension arrays of Java's eight primitive types (int, long, byte, short, float, double, boolean, char)

Implementations can support additional types. Property values of unsupported types must be silently discarded.

113.10 Security

113.10.1 Topic Permission

The TopicPermission class allows fine-grained control over which bundles may post events to a given topic and which bundles may receive those events.

The target parameter for the permission is the topic name. TopicPermission classes uses a wildcard matching algorithm similar to the BasicPermission class, except that slashes are used as separators instead of periods. For example, a name of a/b/* implies a/b/c but not x/y/z or a/b.

There are two available actions: PUBLISH and SUBSCRIBE. These control a bundle's ability to either publish or receive events, respectively. Neither one implies the other.

113.10.2 Required Permissions

Bundles that need to register an event handler must be granted ServicePermission[org.osgi.service.event.EventHandler, REGISTER]. In addition, handlers require TopicPermission[<topic>, SUBSCRIBE] for each topic they want to be notified about.

Bundles that need to publish an event must be granted ServicePermission[org.osgi.service.event.EventAdmin, GET] so that they may retrieve the Event Admin service and use it. In addition, event sources require TopicPermission[<topic>, PUBLISH] for each topic they want to send events to.

Bundles that need to iterate the handlers registered with the system must be granted ServicePermission[org.osgi.service.event.EventHandler, GET] to retrieve the event handlers from the service registry.

Only a bundle that contains an Event Admin service implementation should be granted ServicePermission[org.osgi.service.event.EventAdmin, REGISTER] to register the event channel admin service.

113.10.3 Security Context During Event Callbacks

During an event notification, the Event Admin service's Protection Domain will be on the stack above the handler's Protection Domain. In the case of a synchronous event, the event publisher's protection domain can also be on the stack.

Therefore, if a handler needs to perform a secure operation using its own privileges, it must invoke the doPrivileged method to isolate its security context from that of its caller.

The event delivery mechanism must not wrap event notifications in a doPrivileged call.

113.11 Changes

- Added a containsProperty method to the Event object to detect of a property was set and not just null.
- Added a new *Ordering* on page 269 section to define the out of order delivery.
- Added the Event Properties Map to optimize not having to always create a new Map for each event.

113.12 org.osgi.service.event

Event Admin Package Version 1.3.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.event; version=”[1.3,2.0)”
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.event; version=”[1.3,1.4)”
```

113.12.1 Summary

- Event – An event.
- EventAdmin – The Event Admin service.
- EventConstants – Defines standard names for EventHandler properties.
- EventHandler – Listener for Events.
- EventProperties – The properties for an Event.
- TopicPermission – A bundle's authority to publish or subscribe to event on a topic.

113.12.2 Permissions

113.12.3 public class Event

An event. Event objects are delivered to EventHandler services which subscribe to the topic of the event.

Concurrency Immutable

113.12.3.1 public Event (String topic , Map<String,?> properties)

topic The topic of the event.

properties The event's properties (may be null). A property whose key is not of type String will be ignored.

□ Constructs an event.

Throws IllegalArgumentException – If topic is not a valid topic name.

Since 1.2

113.12.3.2 public Event (String topic , Dictionary<String,?> properties)

topic The topic of the event.

properties The event's properties (may be null). A property whose key is not of type String will be ignored.

□ Constructs an event.

Throws IllegalArgumentException – If topic is not a valid topic name.

113.12.3.3 public final boolean containsProperty (String name)

name The name of the property.

□ Indicate the presence of an event property. The event topic is present using the property name "event.topics".

Returns true if a property with the specified name is in the event. This property may have a null value. false otherwise.

Since 1.3

113.12.3.4 public boolean equals (Object object)

object The Event object to be compared.

□ Compares this Event object to another object.

An event is considered to be **equal to** another event if the topic is equal and the properties are equal. The properties are compared using the java.util.Map.equals() rules which includes identity comparison for array values.

Returns true if object is a Event and is equal to this object; false otherwise.

113.12.3.5 public final Object getProperty (String name)

name The name of the property to retrieve.

□ Retrieve the value of an event property. The event topic may be retrieved with the property name "event.topics".

Returns The value of the property, or null if not found.

113.12.3.6 public final String[] getPropertyNames ( )

□ Returns a list of this event's property names. The list will include the event topic property name "event.topics".

Returns A non-empty array with one element per property.

113.12.3.7 public final String getTopic ( )

□ Returns the topic of this event.

Returns The topic of this event.

113.12.3.8 public int hashCode ( )

□ Returns a hash code value for this object.

Returns An integer which is a hash code value for this object.

113.12.3.9 public final boolean matches (Filter filter)

filter The filter to test.

□ Tests this event's properties against the given filter using a case sensitive match.

Returns true If this event's properties match the filter, false otherwise.

113.12.3.10 public String toString ( )

□ Returns the string representation of this event.

Returns The string representation of this event.

113.12.4 public interface EventAdmin

The Event Admin service. Bundles wishing to publish events must obtain the Event Admin service and call one of the event delivery methods.

Concurrency Thread-safe

No Implement Consumers of this API must not implement this interface

113.12.4.1 public void postEvent (Event event)

event The event to send to all listeners which subscribe to the topic of the event.

□ Initiate asynchronous, ordered delivery of an event. This method returns to the caller before delivery of the event is completed. Events are delivered in the order that they are received by this method.

Throws `SecurityException` – If the caller does not have `TopicPermission[topic,PUBLISH]` for the topic specified in the event.

113.12.4.2 public void sendEvent (Event event)

event The event to send to all listeners which subscribe to the topic of the event.

□ Initiate synchronous delivery of an event. This method does not return to the caller until delivery of the event is completed.

Throws `SecurityException` – If the caller does not have `TopicPermission[topic,PUBLISH]` for the topic specified in the event.

113.12.5 public interface EventConstants

Defines standard names for `EventHandler` properties.

No Implement Consumers of this API must not implement this interface

113.12.5.1 public static final String BUNDLE = "bundle"

The Bundle object of the bundle relevant to the event. The type of the value for this event property is `Bundle`.

Since 1.1

113.12.5.2 public static final String BUNDLE_ID = "bundle.id"

The Bundle id of the bundle relevant to the event. The type of the value for this event property is `Long`.

Since 1.1

113.12.5.3 public static final String BUNDLE_SIGNER = "bundle.signer"

The Distinguished Names of the signers of the bundle relevant to the event. The type of the value for this event property is `String` or `Collection` of `String`.

113.12.5.4 public static final String BUNDLE_SYMBOLICNAME = "bundle.symbolicName"

The Bundle Symbolic Name of the bundle relevant to the event. The type of the value for this event property is `String`.

113.12.5.5 public static final String BUNDLE_VERSION = "bundle.version"

The version of the bundle relevant to the event. The type of the value for this event property is `Version`.

Since 1.2

113.12.5.6 public static final String DELIVERY_ASYNC_ORDERED = "async.ordered"

Event Handler delivery quality value specifying the Event Handler requires asynchronously delivered events be delivered in order. Ordered delivery is the default for asynchronously delivered events.

This delivery quality value is mutually exclusive with `DELIVERY_ASYNC_UNORDERED`. However, if both this value and `DELIVERY_ASYNC_UNORDERED` are specified for an event handler, this value takes precedence.

See Also `EVENT_DELIVERY`

Since 1.3

113.12.5.7 public static final String DELIVERY_ASYNC_UNORDERED = "async.unordered"

Event Handler delivery quality value specifying the Event Handler does not require asynchronously delivered events be delivered in order. This may allow an Event Admin implementation to optimize asynchronous event delivery by relaxing ordering requirements.

This delivery quality value is mutually exclusive with `DELIVERY_ASYNC_ORDERED`. However, if both this value and `DELIVERY_ASYNC_ORDERED` are specified for an event handler, `DELIVERY_ASYNC_ORDERED` takes precedence.

See Also `EVENT_DELIVERY`

Since 1.3

113.12.5.8 public static final String EVENT = "event"

The forwarded event object. Used when rebroadcasting an event that was sent via some other event mechanism. The type of the value for this event property is `Object`.

113.12.5.9 public static final String EVENT_DELIVERY = "event.delivery"

Service Registration property specifying the delivery qualities requested by an Event Handler service.

Event handlers MAY be registered with this property. Each value of this property is a string specifying a delivery quality for the Event handler.

The value of this property must be of type `String`, `String[]`, or `Collection<String>`.

See Also `DELIVERY_ASYNC_ORDERED` , `DELIVERY_ASYNC_UNORDERED`

Since 1.3

113.12.5.10 public static final String EVENT_FILTER = "event.filter"

Service Registration property specifying a filter to further select `Event` s of interest to an Event Handler service.

Event handlers MAY be registered with this property. The value of this property is a string containing an LDAP-style filter specification. Any of the event's properties may be used in the filter expression. Each event handler is notified for any event which belongs to the topics in which the handler has expressed an interest. If the event handler is also registered with this service property, then the properties of the event must also match the filter for the event to be delivered to the event handler.

If the filter syntax is invalid, then the Event Handler must be ignored and a warning should be logged.

The value of this property must be of type `String`.

See Also `Event` , `Filter`

113.12.5.11 public static final String EVENT_TOPIC = “event.topics”

Service registration property specifying the Event topics of interest to an Event Handler service.

Event handlers SHOULD be registered with this property. Each value of this property is a string that describe the topics in which the handler is interested. An asterisk ('*') may be used as a trailing wildcard. Event Handlers which do not have a value for this property must not receive events. More precisely, the value of each string must conform to the following grammar:

```
topic-description := ‘*’ | topic ( ‘/*’ )?
topic := token ( ‘/’ token )*
```

The value of this property must be of type String, String[], or Collection<String>.

See Also Event

113.12.5.12 public static final String EXCEPTION = “exception”

An exception or error. The type of the value for this event property is Throwable.

113.12.5.13 public static final String EXCEPTION_CLASS = “exception.class”

The name of the exception type. Must be equal to the name of the class of the exception in the event property EXCEPTION. The type of the value for this event property is String.

Since 1.1

113.12.5.14 public static final String EXCEPTION_MESSAGE = “exception.message”

The exception message. Must be equal to the result of calling getMessage() on the exception in the event property EXCEPTION. The type of the value for this event property is String.

113.12.5.15 public static final String EXECPTION_CLASS = “exception.class”

This constant was released with an incorrectly spelled name. It has been replaced by EXCEPTION_CLASS

Deprecated As of 1.1, replaced by EXCEPTION_CLASS

113.12.5.16 public static final String MESSAGE = “message”

A human-readable message that is usually not localized. The type of the value for this event property is String.

113.12.5.17 public static final String SERVICE = “service”

A service reference. The type of the value for this event property is ServiceReference.

113.12.5.18 public static final String SERVICE_ID = “service.id”

A service’s id. The type of the value for this event property is Long.

113.12.5.19 public static final String SERVICE_OBJECTCLASS = “service.objectClass”

A service’s objectClass. The type of the value for this event property is String[].

113.12.5.20 public static final String SERVICE_PID = “service.pid”

A service’s persistent identity. The type of the value for this event property is String or Collection of String.

113.12.5.21 public static final String TIMESTAMP = “timestamp”

The time when the event occurred, as reported by System.currentTimeMillis(). The type of the value for this event property is Long.

113.12.6 public interface EventHandler

Listener for Events.

EventHandler objects are registered with the Framework service registry and are notified with an Event object when an event is sent or posted.

EventHandler objects can inspect the received Event object to determine its topic and properties.

EventHandler objects must be registered with a service property EventConstants.EVENT_TOPIC whose value is the list of topics in which the event handler is interested.

For example:

```
String[] topics = new String[] {"com/isv/*"};
Hashtable ht = new Hashtable();
ht.put(EventConstants.EVENT_TOPIC, topics);
context.registerService(EventHandler.class.getName(), this, ht);
```

Event Handler services can also be registered with an EventConstants.EVENT_FILTER service property to further filter the events. If the syntax of this filter is invalid, then the Event Handler must be ignored by the Event Admin service. The Event Admin service should log a warning.

Security Considerations. Bundles wishing to monitor Event objects will require ServicePermission[EventHandler,REGISTER] to register an EventHandler service. The bundle must also have TopicPermission[topic,SUBSCRIBE] for the topic specified in the event in order to receive the event.

See Also Event

Concurrency Thread-safe

113.12.6.1 public void handleEvent (Event event)

event The event that occurred.

□ Called by the EventAdmin service to notify the listener of an event.

113.12.7 public class EventProperties implements Map<String,Object>

The properties for an Event. An event source can create an EventProperties object if it needs to reuse the same event properties for multiple events.

The keys are all of type String. The values are of type Object. The key "event.topics" is ignored as event topics can only be set when an Event is constructed.

Once constructed, an EventProperties object is unmodifiable. However, the values of the map used to construct an EventProperties object are still subject to modification as they are not deeply copied.

Since 1.3

Concurrency Immutable

113.12.7.1 public EventProperties (Map<String,?> properties)

properties The properties to use for this EventProperties object (may be null).

□ Create an EventProperties from the specified properties.

The specified properties will be copied into this EventProperties. Properties whose key is not of type String will be ignored. A property with the key "event.topics" will be ignored.

113.12.7.2 public void clear ( )

□ This method throws UnsupportedOperationException.

Throws UnsupportedOperationException – if called.

113.12.7.3 public boolean containsKey (Object name)

name The property name.

□ Indicates if the specified property is present.

Returns true If the property is present, false otherwise.

113.12.7.4 public boolean containsValue (Object value)

value The property value.

□ Indicates if the specified value is present.

Returns true If the value is present, false otherwise.

113.12.7.5 public Set<Map.Entry<String,Object>> entrySet ( )

□ Return the property entries.

Returns A set containing the property name/value pairs.

113.12.7.6 public boolean equals (Object object)

object The EventProperties object to be compared.

□ Compares this EventProperties object to another object.

The properties are compared using the java.util.Map.equals() rules which includes identity comparison for array values.

Returns true if object is a EventProperties and is equal to this object; false otherwise.

113.12.7.7 public Object get (Object name)

name The name of the specified property.

□ Return the value of the specified property.

Returns The value of the specified property.

113.12.7.8 public int hashCode ( )

□ Returns a hash code value for this object.

Returns An integer which is a hash code value for this object.

113.12.7.9 public boolean isEmpty ( )

□ Indicate if this properties is empty.

Returns true If this properties is empty, false otherwise.

113.12.7.10 public Set<String> keySet ( )

□ Return the names of the properties.

Returns The names of the properties.

113.12.7.11 public Object put (String key , Object value)

□ This method throws UnsupportedOperationException.

Throws UnsupportedOperationException – if called.

113.12.7.12 public void putAll (Map<? extends String,? extends Object> map)

□ This method throws UnsupportedOperationException.

Throws UnsupportedOperationException – if called.

113.12.7.13 public Object remove (Object key)

- This method throws UnsupportedOperationException.

Throws UnsupportedOperationException – if called.

113.12.7.14 public int size ( )

- Return the number of properties.

Returns The number of properties.

113.12.7.15 public String toString ( )

- Returns the string representation of this object.

Returns The string representation of this object.

113.12.7.16 public Collection<Object> values ( )

- Return the properties values.

Returns The values of the properties.

113.12.8 public final class TopicPermission extends Permission

A bundle's authority to publish or subscribe to event on a topic.

A topic is a slash-separated string that defines a topic.

For example:

```
 org / osgi / service / foo / FooEvent / ACTION
```

TopicPermission has two actions: publish and subscribe.

Concurrency Thread-safe

113.12.8.1 public static final String PUBLISH = "publish"

The action string publish.

113.12.8.2 public static final String SUBSCRIBE = "subscribe"

The action string subscribe.

113.12.8.3 public TopicPermission (String name , String actions)

name Topic name.

actions publish,subscribe (canonical order).

- Defines the authority to publich and/or subscribe to a topic within the EventAdmin service.

The name is specified as a slash-separated string. Wildcards may be used. For example:

```
    org/osgi/service/fooFooEvent/ACTION
    com/isv/*
    *
```

A bundle that needs to publish events on a topic must have the appropriate TopicPermission for that topic; similarly, a bundle that needs to subscribe to events on a topic must have the appropriate TopicPermssion for that topic.

113.12.8.4 public boolean equals (Object obj)

obj The object to test for equality with this TopicPermission object.

□ Determines the equality of two TopicPermission objects. This method checks that specified TopicPermission has the same topic name and actions as this TopicPermission object.

Returns true if obj is a TopicPermission, and has the same topic name and actions as this TopicPermission object; false otherwise.

113.12.8.5 public String getActions ( )

□ Returns the canonical string representation of the TopicPermission actions.

Always returns present TopicPermission actions in the following order: publish,subscribe.

Returns Canonical string representation of the TopicPermission actions.

113.12.8.6 public int hashCode ( )

□ Returns the hash code value for this object.

Returns A hash code value for this object.

113.12.8.7 public boolean implies (Permission p)

p The target permission to interrogate.

□ Determines if the specified permission is implied by this object.

This method checks that the topic name of the target is implied by the topic name of this object. The list of TopicPermission actions must either match or allow for the list of the target object to imply the target TopicPermission action.

```
x/y/*,"publish" -> x/y/z,"publish" is true
*,"subscribe" -> x/y,"subscribe"   is true
*,"publish" -> x/y,"subscribe"     is false
x/y,"publish" -> x/y/z,"publish"   is false
```

Returns true if the specified TopicPermission action is implied by this object; false otherwise.

113.12.8.8 public PermissionCollection newPermissionCollection ( )

□ Returns a new PermissionCollection object suitable for storing TopicPermission objects.

Returns A new PermissionCollection object.

117 Dmt Admin Service Specification

Version 2.0

117.1 Introduction

There are a large number of Device Management standards available today. Starting with the ITU X.700 series in the seventies, SNMP in the eighties and then an explosion of different protocols when the use of the Internet expanded in the nineties. Many device management standards have flourished, and some subsequently withered, over the last decades. Some examples:

- X.700 CMIP
- IETF SNMP
- IETF LDAP
- OMA DM
- Broadband Forum TR-069
- UPnP Forum's Device Management
- IETF NETCONF
- OASIS WS Distributed Management

This heterogeneity of the remote management for OSGi Service Platform based devices is a problem for device manufacturers. Since there is often no dominant protocol these manufacturers have to develop multiple solutions for different remote management protocols. It is also problematic for device operators since they have to choose a specific protocol but by that choice could exclude a class of devices that do not support that protocol. There is therefore a need to allow the use of multiple protocols at minimal costs.

Almost all management standards are based on hierarchical object models and provide *primitives* like:

- Get and replace values
- Add/Remove instances
- Discovery of value names and instance ids
- Provide notifications

A Device Management standard consists of a *protocol stack* and a number of *object models*. The protocol stack is generic and shared for all object types; the object model describes a specific device's properties and methods. For example, the protocol stack can consist of a set of SOAP message formats and an object model is a `Deployment Unit`. An object model consists of a data model and sometimes a set of functions.

The core problem is that the generic Device Management Tree must be mapped to device specific functions. This specification therefore defines an API for managing a device using general device management concepts but providing an effective plugin model to link the generic tree to the specific device functions.

The API is decomposed in the following packages/functionality:

- `org.osgi.service.dmt` – Main package that provides access to the local Device Management Tree. Access is session based.
- `org.osgi.service.dmt.notification` – The notification package provides the capability to send alerts to a management server.

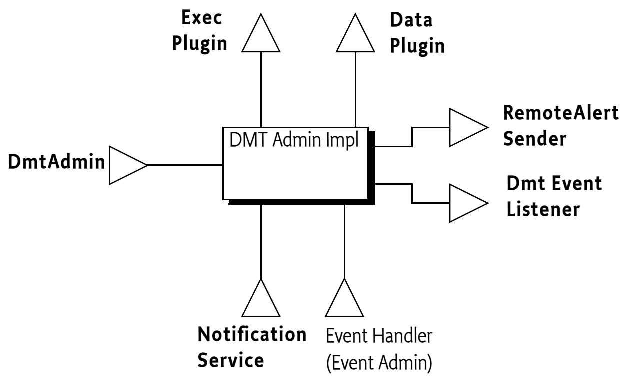
- org.osgi.service.dmt.spi – Provides the capability to register subtree handlers in the Device Management Tree.
- org.osgi.service.dmt.notification.spi – The API to provide the possibilitity to extend the notification system.
- org.osgi.service.dmt.security – Permission classes.

117.1.1 Entities

- *Device Management Tree* – The Device Management Tree (DMT) is the logical view of manageable aspects of an OSGi Environment, implemented by plugins and structured in a tree with named nodes.
- *Dmt Admin* – A service through which the DMT can be manipulated. It is used by *Local Managers* or by *Protocol Adapters* that initiate DMT operations. The Dmt Admin service forwards selected DMT operations to Data Plugins and execute operations to Exec Plugins; in certain cases the Dmt Admin service handles the operations itself. The Dmt Admin service is a singleton.
- *Dmt Session* – A session groups a set of operations on a sub-tree with optional transactionality and locking. Dmt Session objects are created by the Dmt Admin service and are given to a plugin when they first join the session.
- *Local Manager* – A bundle which uses the Dmt Admin service directly to read or manipulate the DMT. Local Managers usually do not have a principal associated with the session.
- *Protocol Adapter* – A bundle that communicates with a management server external to the device and uses the Dmt Admin service to operate on the DMT. Protocol Adapters usually have a principal associated with their sessions.
- *Meta Node* – Information provided by the node implementer about a node for the purpose of performing validation and providing assistance to users when these values are edited.
- *Multi nodes* – Interior nodes that have a homogeneous set of children. All these children share the same meta node.
- *Plugin* – Services which take the responsibility over a given sub-tree of the DMT: Data Plugin services and Exec Plugin services.
- *Data Plugin* – A Plugin that can create a Readable Data Session, Read Write Data Session, or Transactional Data Session for data operations on a sub-tree for a Dmt Session.
- *Exec Plugin* – A Plugin that can handle execute operations.
- *Readable Data Session* – A plugin session that can only read.
- *Read Write Data Session* – A plugin session that can read and write.
- *Transactional Data Session* – A plugin session that is transactional.
- *Principal* – Represents the optional identity of an initiator of a Dmt Session. When a session has a principal, the Dmt Admin must enforce ACLs and must ignore Dmt Permissions.
- *ACL* – An Access Control List is a set of principals that is associated with permitted operations.
- *Dmt Event* – Information about a modification of the DMT.
- *Dmt Event Listener* – Listeners to Dmt Events. These listeners are services according to the white board pattern.
- *Mount Point* – A point in the DMT where a Plugin or the Dmt Admin service allows other Plugins to have their root.

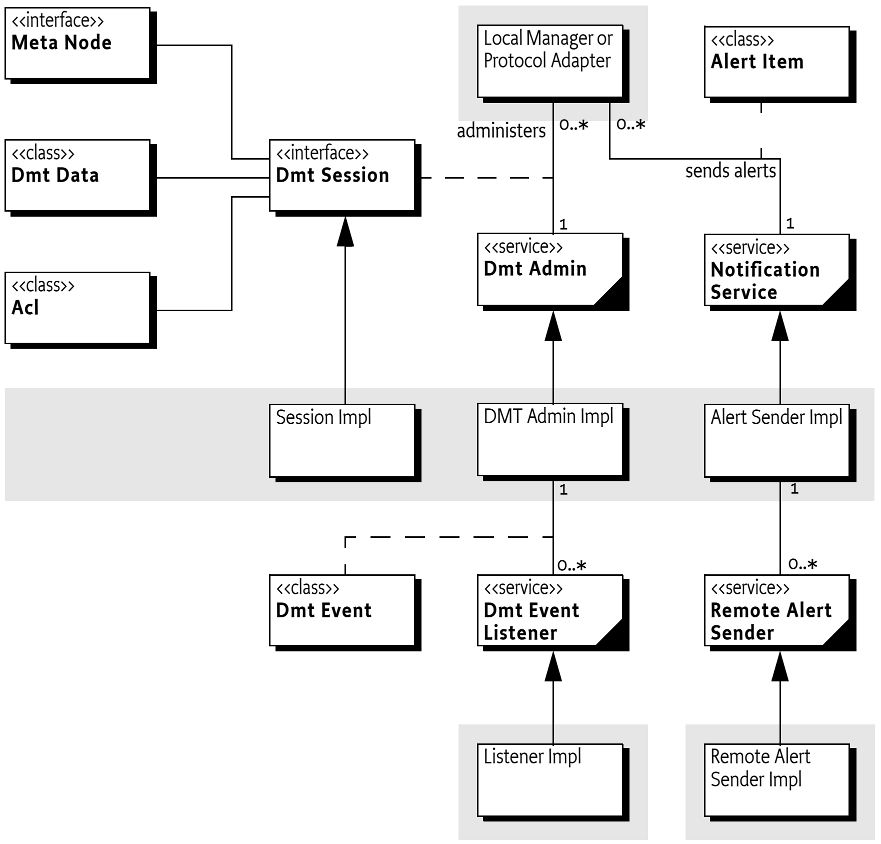
The overall service interaction diagram is depicted in Figure 117.1.

Figure 117.1 Overall Service Diagram



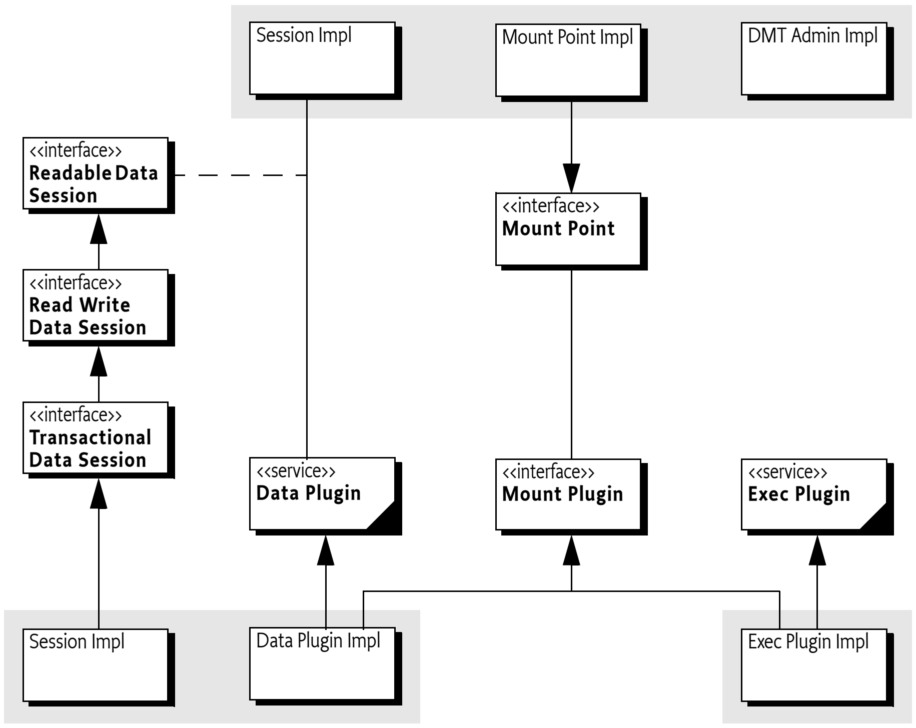
The entities used in the Dmt Admin operations and notifications are depicted in Figure 117.2.

Figure 117.2 Using Dmt Admin service, org.osgi.service.dmt, info, dmt.notification. package*



Extending the Dmt Admin service with Plugins is depicted in Figure 117.3.

Figure 117.3 Extending the Dmt Admin service, org.osgi.service.dmt.spi package


Session Impl
Mount Point Impl
DMT Admin Impl
<<interface>> Readable Data Session
<<interface>> Mount Point
<<interface>> Read Write Data Session
<<interface>> Transactional Data Session
<<service>> Data Plugin
<<interface>> Mount Plugin
<<service>> Exec Plugin
Session Impl
Data Plugin Impl
Exec Plugin Impl

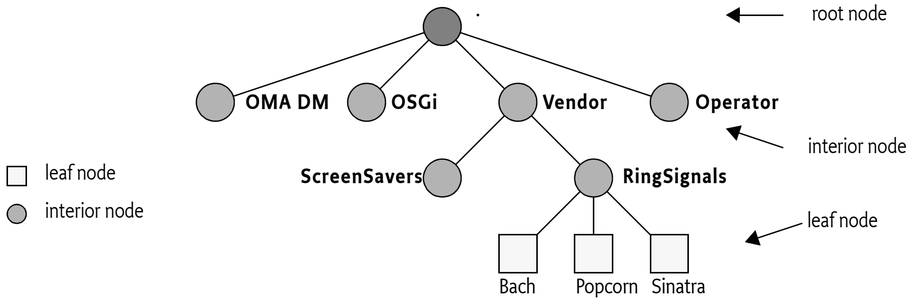
117.2 The Device Management Model

The standard-based features of the DMT model are:

- The Device Management Tree consists of *interior* nodes and *leaf* nodes. Interior nodes can have children and leaf nodes have primitive values.
- All nodes have a set of properties: Name, Title, Format, ACL, Version, Size, Type, Value, and TimeStamp.
- The storage of the nodes is undefined. Nodes typically map to peripheral registers, settings, configuration, databases, etc.
- A node's name must be unique among its siblings.
- Nodes can have Access Control Lists (ACLs), associating operations allowed on those nodes with a particular principal.
- Nodes can have Meta Nodes that describe actual nodes and their siblings.
- Base value types (called *formats* in the standard) are
  - integer
  - long
  - string
  - boolean
  - binary data (multiple types)
  - datetime
  - time
  - float
  - XML fragments
- Leaf nodes in the tree can have default values specified in the meta node.

- Meta Nodes define allowed access operations (Get, Add, Replace, Delete and Exec)

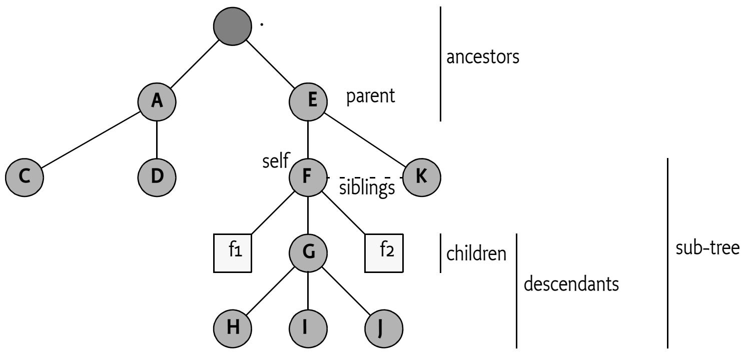
Figure 117.4 Device Management Tree example



117.2.1 Tree Terminology

In the following sections, the DMT is discussed frequently. Thus, well-defined terms for all the concepts that the DMT introduces are needed. The different terms are shown in Figure 117.5.

Figure 117.5 DMT naming, relative to node F



All terms are defined relative to node F. For this node, the terminology is as follows:

- *URI* – The path consisting of node names that uniquely defines a node, see *The DMT Addressing URI* on page 291.
- *ancestors* – All nodes that are above the given node ordered in proximity. The closest node must be first in the list. In the example, this list is [./E, .]
- *parent* – The first ancestor, in this example this is ./E.
- *children* – A list of nodes that are directly beneath the given node without any preferred ordering. For node F this list is { ./E/F/f1, ./E/F/f2, ./E/F/G }.
- *siblings* – An unordered list of nodes that have the same parent. All siblings must have different names. For F, this is { ./E/K}
- *descendants* – A list of all nodes below the given node. For F this is { ./E/F/f1, ./E/F/G, ./E/F/f2, ./E/F/G/H, ./E/F/G/I, ./E/F/G/J }
- *sub-tree* – The given node plus the list of all descendants. For node F this is { ./E/F, ./E/F/f1, ./E/F/G, ./E/F/f2, ./E/F/G/H, ./E/F/G/I, ./E/F/G/J }
- *overlap* – Two given URIs overlap if they share any node in their sub-trees. In the example, the subtree ./E/F and ./E/F/G overlap.
- *data root URI* – A URI which represents the root of a Data Plugin.
- *exec root URI* – A URI which represents the root of an Exec Plugin.
- *Parent Plugin* – A Plugin A is a Parent Plugin of Plugin B if B's root is a in A's sub-tree, this requires a Parent Plugin to at least have one mount point.
- *Child Plugin* – A Plugin A is a Child Plugin of Plugin B if A's root is in B's sub-tree.

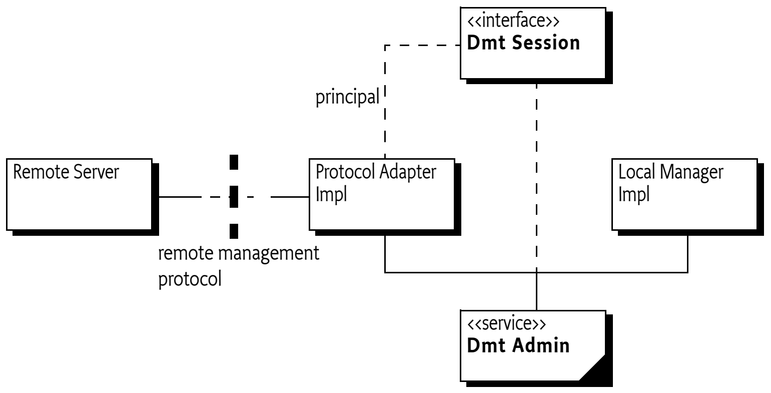
- *Scaffold Node* – An ancestor node of a Plugin that is managed by the Dmt Admin service to ensure that all nodes are discoverable by traversing from the root.

117.2.2 Actors

There are two typical users of the Dmt Admin service:

- *Remote manager* – The typical client of the Dmt Admin service is a *Protocol Adapter.* A management server external to the device can issue DMT operations over some management protocol. The protocol to be used is not specified by this specification. For example, OMA DM, TR-069, or others could be used. The protocol operations reach the service platform through the Protocol Adapter, which forwards the calls to the Dmt Admin service in a session. Protocol Adapters should authenticate the remote manager and set the principal in the session. This association will make the Dmt Admin service enforce the ACLs. This requires that the principal is equal to the server name.
  The Dmt Admin service provides a facility to send notifications to the remote manager with the Notification Service.
- *Local Manager* – A bundle which uses the Dmt Admin service to operate on the DMT: for example, a GUI application that allows the end user to change settings through the DMT.
  Although it is possible to manage some aspects of the system through the DMT, it can be easier for such applications to directly use the services that underlie the DMT; many of the management features available through the DMT are also available as services. These services shield the callers from the underlying details of the abstract, and sometimes hard to use DMT structure. As an example, it is more straightforward to use the Monitor Admin service than to operate upon the monitoring sub-tree. The local management application might listen to Dmt Events if it is interested in updates in the tree made by other entities, however, these events do not necessarily reflect the accurate state of the underlying services.

Figure 117.6 Actors



117.3 The DMT Admin Service

The Dmt Admin service operates on the Device Management Tree of an OSGi-based device. The Dmt Admin API is loosely modelled after the OMA DM protocol: the operations for `Get`, `Replace`, `Add`, `Delete` and `Exec` are directly available. The Dmt Admin is a singleton service.

Access to the DMT is session-based to allow for locking and transactionality. The sessions are, in principle, concurrent, but implementations that queue sessions can be compliant. The client indicates to the Dmt Admin service what kind of session is needed:

- *Exclusive Update Session*– Two or more updating sessions cannot access the same part of the tree simultaneously. An updating session must acquire an exclusive lock on the sub-tree which blocks the creation of other sessions that want to operate on an overlapping sub-tree.
- *Multiple Readers Session* – Any number of read-only sessions can run concurrently, but ongoing read-only sessions must block the creation of an updating session on an overlapping sub-tree.
- *Atomic Session* – An atomic session is the same as an exclusive update session, except that the session can be rolled back at any moment, undoing all changes made so far in the session. The participants must accept the outcome: rollback or commit. There is no prepare phase. The lack of full two phase commit can lead to error situations which are described later in this document; see *Plugins and Transactions* on page 304.

Although the DMT represents a persistent data store with transactional access and without size limitations, the notion of the DMT should not be confused with a general purpose database. The intended purpose of the DMT is to provide a *dynamic view* of the management state of the device; the DMT model and the Dmt Admin service are designed for this purpose.

117.4 Manipulating the DMT

117.4.1 The DMT Addressing URI

The OMA DM limits URIs to the definition of a URI in [8] *RFC 2396 Uniform Resource Identifiers (URI): Generic Syntax*. The Uri utility classes handles nearly all escaping issues with a number of static methods. All URIs in any of the API methods can use the full Unicode character set. For example, the following URIs as used in Java code are valid URIs for the Dmt Admin service.

```
"./ACME © 2000/A/x"
"./ACME/Address/Street/9C, Avenue St. Drézéry"
```

This strategy has a number of consequences.

- A slash ('/' \u002F) collides with the use of the slash as separator of the node names. Slashes must therefore be escaped using a backslash slash ('\/'). The backslash must be escaped with a double backslash sequence. The Dmt Admin service must ignore a backslash when it is not followed by a slash or backslash. The slash and backslash must not be escaped using the %00 like escaping defined for URIs. For example, a node that has the name of a MIME type could look like:

  ```
  ./OSGi/mime/application\/png
  ```

  In Java, a backslash must be escaped as well, therefore requiring double back slashes:

  ```
  String a = "./OSGi/mime/application\\/png";
  ```

  A literal backslash would therefore require 4 backslashes in a Java string.
- The length of a node name is defined to be the length of the byte array that results from UTF-8 encoding a string.

The Uri class provides an encode(String) method to escape a string and a decode(String) method to unescape a string. Though in general the Dmt Admin service implementations should not impose unnecessary constraints on the node name length, it is possible that an implementation runs out of space. In that case it must throw a DmtException URI_TOO_LONG.

Nodes are addressed by presenting a *relative* or *absolute URI* for the requested node. The URI is defined with the following grammar:

```
uri          ::= relative-uri | absolute-uri
absolute-uri ::= './' relative-uri
relative-uri ::= segment ( '/' segment )*
segment      ::= (~['/'])*
```

The Uri isAbsoluteUri(String) method makes it simple to find out if a URI is relative or absolute. Relative URIs require a base URI that is for example provided by the session, see *Locking and Sessions* on page 292.

Each node name is appended to the previous ones using a slash ('/' \u002F) as the separating character. The first node of an absolute URI must be the full stop ('.'\u002E). For example, to access the Bach leaf node in the RingTones interior node from Figure 117.4 on page 289, the URI must be:

```
./Vendor/RingSignals/Bach
```

The URI must be given with the root of the management tree as the starting point. URIs used in the DMT must be treated and interpreted as *case-sensitive*. I.e. ./Vendor and ./vendor designate two different nodes. The following mandatory restrictions on URI syntax are intended to simplify the parsing of URIs.

The full stop has no special meaning in a node name. That is, sequences like .. do not imply parent node. The isValidUri(String) method verifies that a URI fulfills all its obligations and is valid.

117.4.2 Locking and Sessions

The Dmt Admin service is the main entry point into the DMT, its usage is to create sessions. A simple example is getting a session on a specific sub-tree. Such a session can be created with the getSession(String) method. This method creates an updating session with an exclusive lock on the given sub-tree. The given sub-tree can be a single leaf node, if so desired.

Each session has an ID associated with it which is unique to the machine and is never reused. This id is always greater than 0. The value -1 is reserved as place holder to indicate a situation has no session associated with it, for example an event generated from an underlying service. The URI argument addresses the sub-tree root. If null, it addresses the root of the DMT. All nodes can be reached from the root, so specifying a session root node is not strictly necessary but it permits certain optimizations in the implementations.

If the default exclusive locking mode of a session is not adequate, it is possible to specify the locking mode with the getSession(String,int) and getSession(String,String,int) method. These methods supports the following locking modes:

- LOCK_TYPE_SHARED – Creates a *shared session*. It is limited to read-only access to the given sub-tree, which means that multiple sessions are allowed to read the given sub-tree at the same time.
- LOCK_TYPE_EXCLUSIVE – Creates an *exclusive session*. The lock guarantees full read-write access to the tree. Such sessions, however, cannot share their sub-tree with any other session. This type of lock requires that the underlying implementation supports Read Write Data Sessions.
- LOCK_TYPE_ATOMIC – Creates an *atomic session* with an exclusive lock on the sub-tree, but with added transactionality. Operations on such a session must either succeed together or fail together. This type of lock requires that the underlying implementation supports Transactional Data Sessions. If the Dmt Admin service does not support transactions, then it must throw a Dmt Exception with the FEATURE_NOT_SUPPORTED code. If the session accesses data plugins that are not transactional in write mode, then the Dmt Admin service must throw a Dmt Exception with the TRANSACTION_ERROR code. That is, data plugins can participate in a atomic sessions as long as they only perform read operations.

The Dmt Admin service must lock the sub-tree in the requested mode before any operations are performed. If the requested sub-tree is not accessible, the getSession(String,int), getSession(String, String,int), or getSession(String) method must block until the sub-tree becomes available. The implementation can decide after an implementation-dependent period to throw a Dmt Exception with the SESSION_CREATION_TIMEOUT code.

As a simplification, the Dmt Admin service is allowed to lock the entire tree irrespective of the given sub-tree. For performance reasons, implementations should provide more fine-grained locking when possible.

Persisting the changes of a session works differently for exclusive and atomic sessions. Changes to the sub-tree in an atomic session are not persisted until the commit() or close() method of the session is called. Changes since the last transaction point can be rolled back with the rollback() method.

The commit() and rollback() methods can be called multiple times in a session; they do not close the session. The open, commit(), and rollback() methods all establish a *transaction point*. The rollback operation cannot roll back further than the last transaction point.

Once a fatal error is encountered (as defined by the DmtException isFatal() method), all successful changes must be rolled back automatically to the last transaction point. Non-fatal errors do not rollback the session. Any error/exception in the commit or rollback methods invalidates and closes the session. This can happen if, for example, the mapping state of a plugin changes that has its plugin root inside the session's sub-tree.

Changes in an exclusive session are persisted immediately after each separate operation. Errors do not roll back any changes made in such a session.

Due to locking and transactional behavior, a session of any type must be closed once it is no longer used. Locks must always be released, even if the close() method throws an exception.

Once a session is closed no further operations are allowed and manipulation methods must throw a Dmt Illegal State Exception when called. Certain information methods like for example getState() and getRootUri() can still be called for logging or diagnostic purposes. This is documented with the Dmt Session methods.

The close() or commit() method can be expected to fail even if all or some of the individual operations were successful. This failure can occur due to multi-node constraints defined by a specific implementation. The details of how an implementation specifies such constraints is outside the scope of this specification.

Events in an atomic session must only be sent at commit time.

117.4.3 Associating a Principal

Protocol Adapters must use the getSession(String,String,int) method which features the principal as the first parameter. The principal identifies the external entity on whose behalf the session is created. This server identification string is determined during the authentication process in a way specific to the management protocol.

For example, the identity of the OMA DM server can be established during the handshake between the OMA DM agent and the server. In the simpler case of OMA CP protocol, which is a one-way protocol based on WAP Push, the identity of the principal can be a fixed value.

117.4.4 Relative Addressing

All DMT operation methods are found on the session object. Most of these methods accept a *relative* or *absolute* URI as their first parameter: for example, the method isLeafNode(String). This URI is absolute or relative to the sub-tree with which the session is associated. For example, if the session is opened on:

```
./Vendor
```

then the following URIs address the Bach ring tone:

```
RingTones/Bach
./Vendor/RingTones/Bach
```

Opening the session with a null URI is identical to opening the session at the root. But the absolute URI can be used to address the Bach ring tone as well as a relative URI.

```
./Vendor/RingTones/Bach
Vendor/RingTones/Bach
```

If the URI specified does not correspond to a legitimate node in the tree, a Dmt Exception must be thrown. The only exception to this rule is the isNodeUri(String) method that can verify if a node is actually valid. The getMetaNode(String) method must accept URIs to non-existing nodes if an applicable meta node is available; otherwise it must also throw a Dmt Exception.

117.4.5 Creating Nodes

The methods that create interior nodes are:

- createInteriorNode(String) – Create a new interior node using the default meta data. If the principal does not have Replace access rights on the parent of the new node then the session must automatically set the ACL of the new node so that the creating server has Add, Delete and Replace rights on the new node.
- createInteriorNode(String,String) – Create a new interior node. The meta data for this new node is identified by the second argument, which is a URI *identifying* an OMA DM Device Description Framework (DDF) file, this does not have to be a valid location. It uses a format like org.osgi/1.0/LogManagementObject. This meta node must be consistent with any meta information from the parent node.
- createLeafNode(String) – Create a new leaf node with a default value.
- createLeafNode(String,DmtData) – Create a leaf node and assign a value to the leaf-node.
- createLeafNode(String,DmtData,String) – Create a leaf node and assign a value for the node. The last argument is the MIME type, which can be null.

For a node to be created, the following conditions must be fulfilled:

- The URI of the new node has to be a valid URI.
- The principal of the Dmt Session, if present, must have ACL Add permission to add the node to the parent. Otherwise, the caller must have the necessary permission.
- All constraints of the meta node must be verified, including value constraints, name constraints, type constraints, and MIME type constraints. If any of the constraints fail, a Dmt Exception must be thrown with an appropriate code.

117.4.6 Node Properties

A DMT node has a number of runtime properties that can be set through the session object. These properties are:

- *Title* – (String) A human readable title for the object. The title is distinct from the node name. The title can be set with setNodeTitle(String,String) and read with getNodeTitle(String). This specification does not define how this information is localized. This property is optional depending on the implementation that handles the node.
- *Type* –(String) The MIME type, as defined in [9] *MIME Media Types*, of the node's value when it is a leaf node. The type of an interior node is a string identifying a DDF type. These types can be set with setNodeType(String,String) and read with getNodeType(String).
- *Version* – (int) Version number, which must start at 0, incremented after every modification (for both a leaf and an interior node) modulo 0x10000. Changes to the value or any of the properties (including ACLs), or adding/deleting nodes, are considered changes. The getNodeVersion(String) method returns this version; the value is read-only. In certain cases, the underlying data structure does not support change notifications or makes it difficult to support versions. This property is optional depending on the node's implementation.
- *Size* – (int) The size measured in bytes is read-only and can be read with getNodeSize(String). Not all nodes can accurately provide this information.
- *Time Stamp* –(Date) Time of the last change in version. The getNodeTimestamp(String) returns the time stamp. The value is read only. This property is optional depending on the node's implementation.
- *ACL* – The Access Control List for this and descendant nodes. The property can be set with setNodeAcl(String,Acl) and obtained with getNodeAcl(String).

If a plugin that does not implement an optional property is accessed, a Dmt Exception with the code FEATURE_NOT_SUPPORTED must be thrown.

117.4.7 Setting and Getting Data

Values are represented as DmtData objects, which are immutable. The are acquired with the getNodeValue(String) method and set with the setNodeValue(String,DmtData) method.

DmtData objects are dynamically typed by an integer enumeration. In OMA DM, this integer is called the *format* of the data value. The format of the DmtData class is similar to the type of a variable in a programming language, but the word *format* is used here. The available data formats are listed in Table 117.1.

Table 117.1 Data Formats

Format Type	Java Type	Format Name	Constructor	Get	Description
FORMAT_BASE64	byte[]	base64	DmtData(byte[], boolean)	getBase64()	Binary type that must be encoded with base 64, see [10] *RFC 3548 The Base16, Base32, and Base64 Data Encodings.*
FORMAT_BINARY	byte[]	binary	DmtData(byte[]) DmtData(byte[], boolean)	getBinary()	A byte array. The DmtData object is created with the constructor. The byte array can only be acquired with the method.
FORMAT_BOOLEAN	boolean	boolean	DmtData(boolean)	getBoolean()	Boolean. There are two constants for this type: • FALSE_VALUE • TRUE_VALUE
FORMAT_DATE	String	date	DmtData(String,int)	getString() getDate()	A Date (no time). Syntax defined in [13] *XML Schema Part 2: Datatypes Second Edition* as the date type.
FORMAT_DATE_TIME	String	dateTime	DmtData(Date)	getDateTime()	A Date object representing a point in time.
FORMAT_FLOAT	float	float	DmtData(float)	getFloat()	Float
FORMAT_INTEGER	int	integer	DmtData(int)	getInt()	Integer
FORMAT_LONG	long	long	DmtData(long)	getLong()	Long
FORMAT_NODE	Object	NODE	DmtData(Object)	getNode()	A DmtData object can have a format of FORMAT_NODE. This value is returned from a MetaNode getFormat() method if the node is an interior node or for a data value when the Plugin supports complex values.
FORMAT_NULL					No valid data is available. DmtData objects with this format cannot be constructed; the only instance is the DmtData NULL_VALUE constant.

Table 117.1 Data Formats

Format Type	Java Type	Format Name	Constructor	Get	Description
FORMAT_RAW_BINARY	byte[]	<custom>	DmtData(String,byte[])	getRawBinary()	A raw binary format is always created with a format name. This format name allows the creator to define a proprietary format. The format name is available from the getFormatName() method, which has predefined values for the standard formats.
FORMAT_RAW_STRING	String	<custom>	DmtData(String,String)	getRawString()	A raw string format is always created with a format name. This format name allows the creator to define a proprietary format. The format name is available from the getFormatName() method, which has predefined values for the standard formats.
FORMAT_STRING	String	string	DmtData(String)	getString()	String
FORMAT_TIME	String	time	DmtData(String,int)	getString()	Time of Day. Syntax defined in [13] *XML Schema Part 2: Datatypes Second Edition* as the time type.
FORMAT_XML	String	xml	DmtData(String,int)	getXml()	A string containing an XML fragment. It can be obtained with. The validity of the XML must not be verified by the Dmt Admin service.

117.4.8 Complex Values

The OMA DM model prescribes that only leaf nodes have primitive values. This model maps very well to remote managers. However, when a manager is written in Java and uses the Dmt Admin API to access the tree, there are often unnecessary conversions from a complex object, to leaf nodes, and back to a complex object. For example, an interior node could hold the current GPS position as an OSGi Position object, which consists of a longitude, latitude, altitude, speed, and direction. All these objects are Measurement objects which consist of value, error, and unit. Reading such a Position object through its leaf nodes only to make a new Position object is wasting resources. It is therefore that the Dmt Admin service also supports *complex values* as a supplementary facility.

If a complex value is used then the leaves must also be accessible and represent the same semantics as the complex value. A manager unaware of complex values must work correctly by only using the leaf nodes. Setting or getting the complex value of an interior node must be identical to setting or getting the leaf nodes.

Accessing a complex value requires Get access to the node and all its descendants. Setting a complex value requires Replace access to the interior node. Replacing a complex value must only generate a single Replace event.

Trying to set or get a complex value on an interior node that does not support complex values must throw a Dmt Exception with the code COMMAND_NOT_ALLOWED.

117.4.9 Nodes and Types

The node's type can be set with the setNodeType(String,String) method and acquired with getNodeType(String). The namespaces for the types differ for interior and leaf nodes. A leaf node is typed with a MIME type and an interior node is typed with a DDF Document URI. However, in both cases the Dmt Admin service must not verify the syntax of the type name.

The createLeafNode(String,DmtData,String) method takes a MIME type as last argument that will type the leaf node. The MIME type reflects how the data of the node should be *interpreted*. For example, it is possible to store a GIF and a JPEG image in a DmtData object with a FORMAT_BINARY format. Both the GIF and the JPEG object share the same *format*, but will have MIME types of image/jpg and image/gif respectively. The Meta Node provides a list of possible MIME types.

The createInteriorNode(String,String)method takes a DDF Document URI as the last argument that will type the interior node. This specification defines the DDF Document URIs listed in Table 117.2 for interior nodes that have a particular meaning in this specification.

Table 117.2 Standard Interior Node Types

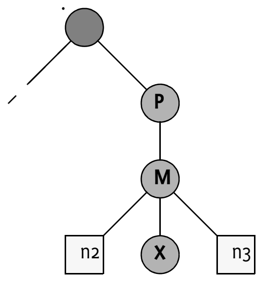
Interior Node Type	Description
DDF_SCAFFOLD	Scaffold nodes are automatically generated nodes by the Dmt Admin service to provide the children node names so that Plugins are reachable from the root. See *Scaffold Nodes* on page 305.
DDF_MAP	MAP nodes define a key -> value mapping construct using the node name (key) and the node value (value). See *MAP Nodes* on page 333.
DDF_LIST	LIST nodes use the node name to maintain an index in a list. See *LIST Nodes* on page 331.

117.4.10 Deleting Nodes

The deleteNode(String) method on the session represents the Delete operation. It deletes the subtree of that node. This method is applicable to both leaf and interior nodes. Nodes can be deleted by the Dmt Admin service in any order. The root node of the session cannot be deleted.

For example, given Figure 117.7, deleting node P must delete the nodes ./P,./P/ M, ./P/M/X, ./P/M/n2 and ./P/M/n3 in any order.

Figure 117.7 DMT node and deletion



117.4.11 Copying Nodes

The copy(String,String,boolean) method on the DmtSession object represents the Copy operation. A node is completely copied to a new URI. It can be specified with a boolean if the whole sub-tree (true) or just the indicated node is copied.

The ACLs must not be copied; the new access rights must be the same as if the caller had created the new nodes individually. This restriction means that the copied nodes inherit the access rights from the parent of the destination node, unless the calling principal does not have Replace rights for the parent. See *Creating Nodes* on page 294 for details.

117.4.12 Renaming Nodes

The renameNode(String,String) method on the DmtSession object represents the Rename operation, which replaces the node name. It requires permission for the Replace operation. The root node for the current session can not be renamed.

117.4.13 Execute

The execute(String,String) and execute(String,String,String) methods can *execute* a node. Executing a node is intended to be used when a problem is hard to model as a set of leaf nodes. This can be related to synchronization issues or data manipulation. The execute methods can provide a correlator for a notification and an opaque string that is forwarded to the implementer of the node.

Execute operations can not take place in a read only session because simultaneous execution could make conflicting changes to the tree.

117.4.14 Closing

When all the changes have been made, the session must be closed by calling the close() method on the session. The Dmt Admin service must then finalize, clean up, and release any locks. For atomic sessions, the Dmt Admin service must automatically commit any changes that were made since the last transaction point.

A session times out and is invalidated after an extended period of inactivity. The exact length of this period is not specified, but is recommended to be at least 1 minute and at most 24 hours. All methods of an invalidated session must throw an Dmt Illegal State Exception after the session is invalidated.

A session's state is one of the following: STATE_CLOSED, STATE_INVALID or STATE_OPEN, as can be queried by the getState() call. The invalid state is reached either after a fatal error case is encountered or after the session is timed out. When an atomic session is invalidated, it is automatically rolled back to the last transaction point of the session.

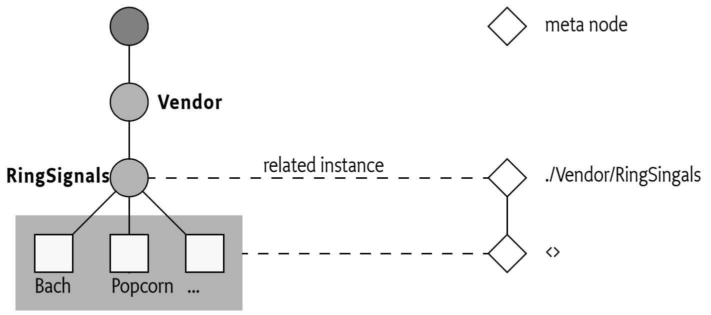
117.5 Meta Data

The getMetaNode(String) method returns a MetaNode object for a given URI. This node is called the *meta node*. A meta node provides information about nodes.

Any node can optionally have a meta node associated with it. The one or more nodes that are described by the meta nodes are called the meta node's *related instances*. A meta node can describe a singleton-related instance, or it can describe all the children of a given parent if it is a *multi-node*. That is to say, meta nodes can exist without an actual instance being present. In order to retrieve the meta node of a multi-node any name can be used.

For example, if a new ring tone, Grieg, was created in Figure 117.8 it would be possible to get the Meta Node for ./Vendor/RingSignals/Grieg before the node was created. This is usually the case for multi nodes. The model is depicted in Figure 117.8.

Figure 117.8 *Nodes and meta nodes*



A URI is generally associated with the same Meta Node. The getMetaNode(String) should return the same meta node for the same URI except in the case of *Scaffold Nodes* on page 305. As the ownership of scaffold nodes can change from the Dmt Admin service to the Parent Plugin service, or from a Parent Plugin to a Child Plugin, the Meta Node can change as well.

The last segment of the URI to get a Meta Node can be any valid node name, for example, instead of Grieg it would have been possible to retrieve the same Meta Node with the name ./Vendor/RingSignals/0, ./Vendor/RingSignals/anyName, ./Vendor/RingSignals/<>, etc.

The actual meta data can come from two sources:

- *Dmt Admin* – Each Dmt Admin service likely has a private meta data repository. This meta data is placed in the device in a proprietary way.
- *Plugins* – Plugins can carry meta nodes and provide these to the Dmt Admin service by implementing the getMetaNode(String[]) method. If a plugin returns a non-null value, the Dmt Admin service must use that value, possibly complemented by its own metadata for elements not provided by the plugin.

The MetaNode interface supports methods to retrieve read-only meta data. The following sections describes this meta-data in more detail.

117.5.1 Operations

The can(int) method provide information as to whether the associated node can perform the given operation. This information is only about the capability; it can still be restricted in runtime by ACLs and permissions.

For example, if the can(MetaNode.CMD_EXECUTE) method returns true, the target object supports the Execute operation. That is, calling the execute(String,String) method with the target URI is possible.

The can(int) method can take the following constants as parameters:

- CMD_ADD
- CMD_DELETE
- CMD_EXECUTE
- CMD_GET
- CMD_REPLACE

For example:

```
void foo( DmtSession session, String nodeUri ) {
   MetaNode meta = session.getMetaNode(nodeUri);
  if ( meta !=null && meta.can(MetaNode.CMD_EXECUTE) )
     session.execute(nodeUri,"foo" );
}
```

117.5.2 Scope

The scope is part of the meta information of a node. It provides information about what the life cycle role is of the node. The getScope() method on the Meta Node provides this information. The value of the scope can be one of the following:

- DYNAMIC – Dynamic nodes are intended to be created and deleted by a management system or an other controlling source. This this not imply that it actually is possible to add new nodes and delete nodes, the actions can still allow or deny this. However, in principle nodes that can be added or deleted have the DYNAMIC scope. The LIST and MAP nodes, see *OSGi Object Modeling* on page 327, always have DYNAMIC scope.
- PERMANENT – Permanent nodes represent an entity in the system. This can be a network interface, de device description, etc. Permanent nodes in general map to an object in an object oriented language. Despite their name, PERMANENT nodes can appear and disappear, for example the plugging in of a USB device might create a new PERMANENT node. Generally, the Plugin roots map to PERMANENT nodes.
- AUTOMATIC – Automatic nodes map in general to nodes that are closely tied to the parent. They are similar to fields of an object in an object oriented language. They cannot be deleted or added.

For example, a node representing the Battery can never be deleted because it is an intrinsic part of the device; it will therefore be PERMANENT. The Level and number of ChargeCycle nodes will be AUTOMATIC. A new ring tone is dynamically created by a manager and is therefore DYNAMIC.

117.5.3 Description and Default

- getDescription() – (String) A description of the node. Descriptions can be used in dialogs with end users: for example, a GUI application that allows the user to set the value of a node. Localization of these values is not defined.
- getDefault() – (DmtData) A default data value.

117.5.4 Validation

The validation information allows the runtime system to verify constraints on the values; it also allows user interfaces to provide guidance.

A node does not have to exist in the DMT in order to have meta data associated with it. Nodes may exist that have only partial meta data, or no metadata, associated with them. For each type of metadata, the default value to assume when it is omitted is described in *MetaNode* on page 379.

117.5.5 Data Types

A leaf node can be constrained to a certain format and one of a set of MIME types.

- getFormat() – (int) The required type. This type is a logical OR of the supported formats.
- getRawFormatNames() – Return an array of possible raw format names. This is only applicable when the getFormat() returns the FORMAT_RAW_BINARY or FORMAT_RAW_STRING formats. The method must return null otherwise.
- getMimeTypes() – (String[]) A list of MIME types for leaf nodes or DDF types for interior nodes. The Dmt Admin service must verify that the actual type of the node is part of this set.

117.5.6 Cardinality

A meta node can constrain the number of *siblings* (i.e., not the number of children) of an interior or leaf node. This constraint can be used to verify that a node must not be deleted, because there should be at least one node left on that level (isZeroOccurrenceAllowed()), or to verify that a node cannot be created, because there are already too many siblings (getMaxOccurrence()).

If the cardinality of a meta node is more than one, all siblings must share the same meta node to prevent an invalid situation. For example, if a node has two children that are described by different meta nodes, and any of the meta nodes has a cardinality >1, that situation is invalid.

For example, the ./Vendor/RingSignals/<> meta node (where <> stands for any name) could specify that there should be between 0 and 12 ring signals.

- getMaxOccurrence() – (int) A value greater than 0 that specifies the maximum number of instances for this node.
- isZeroOccurrenceAllowed() – (boolean) Returns true if zero instances are allowed. If not, the last instance must not be deleted.

117.5.7 Matching

The following methods provide validation capabilities for leaf nodes.

- isValidValue(DmtData) – (DmtData) Verify that the given value is valid for this meta node.
- getValidValues() – (DmtData[]) A set of possible values for a node, or null otherwise. This can for example be used to give a user a set of options to choose from.

117.5.8 Numeric Ranges

Numeric leaf nodes (format must be FORMAT_INTEGER, FORMAT_LONG, or FORMAT_FLOAT) can be checked for a minimum and maximum value.

Minimum and maximum values are inclusive. That is, the range is [getMin(),getMax()]. For example, if the maximum value is 5 and the minimum value is -5, then the range is [-5,5]. This means that valid values are -5,-4,-3,-2... 4, 5.

- getMax() – (double) The value of the node must be less than or equal to this maximum value.
- getMin() – (double) The value of the node must be greater than or equal to this minimum value.

If no meta data is provided for the minimum and maximum values, the meta node must return the Double.MIN_VALUE, and Double.MAX_VALUE respectively.

117.5.9 Name Validation

The meta node provides the following name validation facilities for both leaf and interior nodes:

- isValidName(String) – (String) Verifies that the given name matches the rules for this meta node.
- getValidNames() – (String[]) An array of possible names. A valid name for this node must appear in this list.

117.5.10 User Extensions

The Meta Node provides an extension mechanism; each meta node can be associated with a number of properties. These properties are then interpreted in a proprietary way. The following methods are used for user extensions:

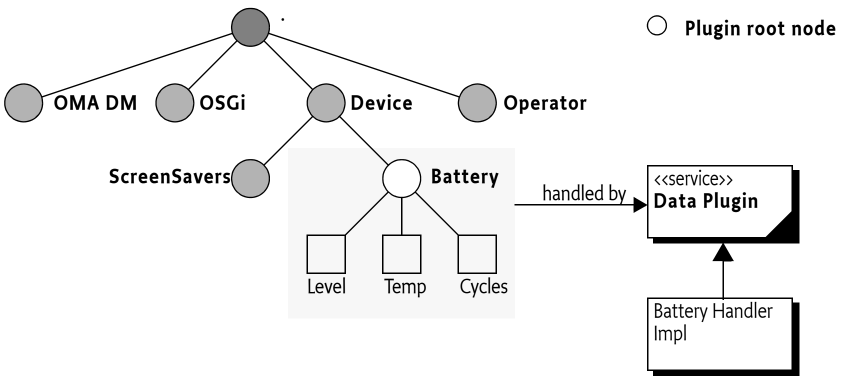
- getExtensionPropertyKeys() – Returns an array of key names that can be provided by this meta node.
- getExtensionProperty(String) – Returns the value of an extension property.

For example, a manufacturer could use a regular expression to validate the node names with the isValidName(String) method. In a web based user interface it is interesting to provide validity checking in the browser, however, in such a case the regular expression string is required. This string could then be provided as a user extension under the key x-acme-regex-javascript.

117.6 Plugins

The Plugins take the responsibility of handling DMT operations within certain sub-trees of the DMT. It is the responsibility of the Dmt Admin service to forward the operation requests to the appropriate plugin. The only exceptions are the ACL manipulation commands. ACLs must be enforced by the Dmt Admin service and never by the plugin. The model is depicted in Figure 117.9.

Figure 117.9 *Device Management Tree example*



Plugins are OSGi services. The Dmt Admin service must dynamically map and unmap the plugins, acting as node handler, as they are registered and unregistered. Service properties are used to specify the sub-tree that the plugin can manage as well as mount points that it provides to *Child Plugins*; plugins that manage part of the Plugin's sub-tree.

For example, a plugin related to Configuration Admin handles the sub-tree which stores configuration data. This sub-tree could start at ./OSGi/Configuration. When the client wants to add a new configuration object to the DMT, it must issue an Add operation to the ./OSGi/Configuration node. The Dmt Admin service then forwards this operation to the configuration plugin. The plugin maps the request to one or more method calls on the Configuration Admin service. Such a plugin can be a simple proxy to the Configuration Admin service, so it can provide a DMT view of the configuration data store.

There are two types of Dmt plugins: *data plugins* and *exec plugins*. A data plugin is responsible for handling the sub-tree retrieval, addition and deletion operations, and handling of meta data, while an exec plugin handles a node execution operation.

117.6.1 Data Sessions

Data Plugins must participate in the Dmt Admin service sessions. A Data Plugin provider must therefore register a Data Plugin service. Such a service can create a session for the Dmt Admin service when the given sub-tree is accessed by a Dmt Session. If the associated Dmt Session is later closed, the Data Session will also be closed. Three types of sessions provide different capabilities. Data Plugins do not have to implement all session types; if they choose not to implement a session type they can return null.

- *Readable Data Session* – Must always be supported. It provides the basic read-only access to the nodes and the close() method. The Dmt Admin service uses this session type when the lock mode is LOCK_TYPE_SHARED for the Dmt Session. Such a session is created with the plugin's openReadOnlySession(String[],DmtSession), method which returns a ReadableDataSession object.
- *Read Write Data Session* – Extends the Readable Data Session with capabilities to modify the DMT. This is used for Dmt Sessions that are opened with LOCK_TYPE_EXCLUSIVE. Such a session is created with the plugin's openReadWriteSession(String[],DmtSession) method, which returns a ReadWriteDataSession object.
- *Transactional Data Session* – Extends the Read Write Data Session with commit and rollback methods so that this session can be used with transactions. It is used when the Dmt Session is opened with lock mode LOCK_TYPE_ATOMIC. Such a session is created with the plugin's openAtomicSession(String[],DmtSession) method, which returns a TransactionalDataSession object.

117.6.2 URIs and Plugins

The plugin Data Sessions do not use a simple string to identify a node as the Dmt Session does. Instead the URI parameter is a String[]. The members of this String[] are the different segments. The first node after the root is the second segment and the node name is the last segment. The different segments require escaping of the slash and backslash ('/' and'\').

The reason to use String[] objects instead of the original string is to reduce the number times that the URI is parsed. The entry String objects, however, are still escaped. For example, the URI ./A/B/image\/jpg gives the following String[]:

```
{ ".", "A", "B", "image\/jpg" }
```

A plugin can assume that the path is validated and can be used directly.

117.6.3 Associating a sub-tree

Each plugin is associated with one ore more DMT sub-trees. The top node of a sub-tree is called the *plugin root*. The plugin root is defined by a service registration property. This property is different for exec plugins and data plugins:

- DATA_ROOT_URIS – (String+) A sequence of *data URI*, defining a plugin root for data plugins.
- EXEC_ROOT_URIS – (String+) A sequence of *exec URI*, defining a plugin root for exec plugins.

If the Plugin modifies these service properties then the Dmt Admin service must reflect these changes as soon as possible. The reason for the different properties is to allow a single service to register both as a Data Plugin service as well as an Exec Plugin service.

Data and Exec Plugins live in independent trees and can fully overlap. However, an Exec Plugin can only execute a node when the there exists a valid node at the corresponding node in the Data tree. that is, to be able to execute a node it is necessary that isNodeUri(String) would return true.

For example, a data plugin can register itself in its activator to handle the sub-tree ./Dev/Battery:

```
public void start(BundleContext context) {
  Hashtable ht = new Hashtable();
  ht.put(Constants.SERVICE_PID, "com.acme.data.plugin");
  ht.put( DataPlugin.DATA_ROOT_URIS, "./Dev/Battery");
  context.registerService(
      DataPlugin.class.getName(),
      new BatteryHandler(context);
      ht );
}
```

If this activator was executed, an access to ./Dev/Battery must be forwarded by the Dmt Admin service to this plugin via one of the data session.

117.6.4 Synchronization with Dmt Admin Service

The Dmt Admin service can, in certain cases, detect that a node was changed without the plugin knowing about this change. For example, if the ACL is changed, the version and timestamp must be updated; these properties are maintained by the plugin. In these cases, the Dmt Admin service must open a ReadableDataSession and call nodeChanged(String[]) method with the changed URI.

117.6.5 Plugin Meta Data

Plugins can provide meta data; meta data from the Plugin must take precedence over the meta data of the Dmt Admin service. If a plugin provides meta information, the Dmt Admin service must verify that an operation is compatible with the meta data of the given node.

For example if the plugin reports in its meta data that the ./A leaf node can only have the text/plain MIME type, the createLeafNode(String) calls must not be forwarded to the Plugin if the third argument specifies any other MIME type. If this contract between the Dmt Admin service and the plugin is violated, the plugin should throw a Dmt Exception METADATA_MISMATCH.

117.6.6 Plugins and Transactions

For the Dmt Admin service to be transactional, transactions must be supported by the data plugins. This support is not mandatory in this specification, and therefore the Dmt Admin service has no transactional guarantees for atomicity, consistency, isolation or durability. The DmtAdmin interface and the DataPlugin (or more specifically the data session) interfaces, however, are designed to support Data Plugin services that are transactional. Exec plugins need not be transaction-aware because the execute method does not provide transactional semantics, although it can be executed in an atomic transaction.

Data Plugins do not have to support atomic sessions. When the Dmt Admin service creates a Transactional Data Session by calling openAtomicSession(String[],DmtSession) the Data Plugin is allowed to return null. In that case, the plugin does not support atomic sessions. The caller receives a Dmt Exception.

Plugins must persist any changes immediately for Read Write Data Sessions. Transactional Data Sessions must delay changes until the commit() method is called, which can happen multiple times during a session. The opening of an atomic session and the commit() and rollback() methods all establish a *transaction point*. Rollback can never go further back than the last transaction point.

- commit() – Commit any changes that were made to the DMT but not yet persisted. This method should not throw an Exception because other Plugins already could have persisted their data and can no longer roll it back. The commit method can be called multiple times in an open session, and if so, the commit must make persistent the changes since the last transaction point.
- rollback() – Undo any changes made to the sub-tree since the last transaction point.
- close() – Clean up and release any locks. The Dmt Admin service must call the commit methods before the close method is called. A Plugin must not perform any persistency operations in the close method.

The commit(), rollback(), and close() plugin data session methods must all be called in reverse order of that in which Plugins joined the session.

If a Plugin throws a fatal exception during an operation, the Dmt Session must be rolled back immediately, automatically rolling back all data plugins, as well as the plugins that threw the fatal Dmt Exception. The fatality of an Exception can be checked with the Dmt Exception isFatal() method.

If a plugin throws a non-fatal exception in any method accessing the DMT, the current operation fails, but the session remains open for further commands. All errors due to invalid parameters (e.g. non-existing nodes, unrecognized values), all temporary errors, etc. should fall into this category.

A rollback of the transaction can take place due to any irregularity during the session. For example:

- A necessary Plugin is unregistered or unmapped
- A fatal exception is thrown while calling a plugin
- Critical data is not available
- An attempt is made to breach the security

Any Exception thrown during the course of a commit() or rollback() method call is considered fatal, because the session can be in a half-committed state and is not safe for further use. The operation in progress should be continued with the remaining Plugins to achieve a *best-effort* solution in this limited transactional model. Once all plugins have been committed or rolled back, the Dmt Admin service must throw an exception, specifying the cause exception(s) thrown by the plugin(s), and should log an error.

117.6.7 Side Effects

Changing a node's value will have a side effect of changing the system. A plugin can also, however, cause state changes with a get operation. Sometimes the pattern to use a get operation to perform a state changing action can be quite convenient. The get operation, however, is defined to have no side effects. This definition is reflected in the session model, which allows the DMT to be shared among readers. Therefore, plugins should refrain from causing side effects for read-only operations.

117.6.8 Copying

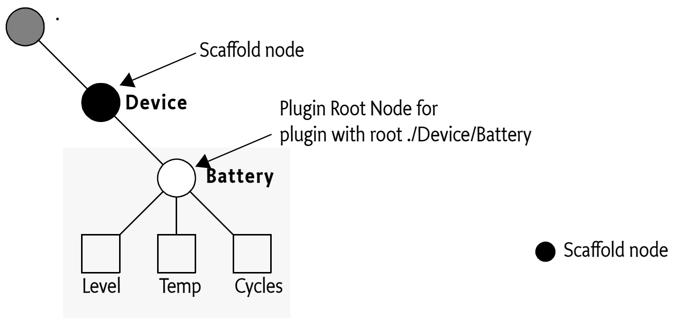
Plugins do not have to support the copy operation. They can throw a Dmt Exception with a code FEATURE_NOT_SUPPORTED. In this case, the Dmt Admin service must do the copying node by node. For the clients of the Dmt Admin service, it therefore appears that the copy method is always supported.

117.6.9 Scaffold Nodes

As Plugins can be mapped anywhere into the DMT it is possible that a part of the URI has no corresponding Plugin, such a plugin would not be *reachable* unless the intermediate nodes were provided. A program that would try to discover the DMT would not be able to find the registered Plugins as the intermediate nodes would not be discoverable.

These intermediate nodes that will make all plugins reachable must therefore be provided by the Dmt Admin service, they are called the *scaffold nodes*. The only purpose of the scaffold nodes is to allow every node to be discovered when the DMT is traversed from the root down. Scaffold nodes are provided both for Data Plugins as well as Exec Plugins as well as for Child Plugins that are mounted inside a Parent Plugin, see *Sharing the DMT* on page 307. In Figure 117.10 the Device node is a scaffold node because there is no plugin associated with it. The Dmt Admin service must, however, provide the Battery node as child node of the Device node.

Figure 117.10 Scaffold Nodes



A scaffold node is always an interior node and has limited functionality, it must have a type of DDF_SCAFFOLD. It has no value, it is impossible to add or delete nodes to it, and the methods that are allowed for a scaffold node are specified in Table 117.3.

Table 117.3 Supported Scaffold Node Methods

Method	Description
getNodeAcl(String)	Must inherit from the root node.
getChildNodeNames(String)	Answer the child node names such that plugin's in the sub-tree are reachable.
getMetaNode(String)	Provides the Meta Node defined in Table 117.4
getNodeSize(String)	Must always return 0.
getNodeTitle(String)	null

Table 117.3 *Supported Scaffold Node Methods*

Method	Description
getNodeTimestamp(String)	Time first created
getNodeType(String)	DDF_SCAFFOLD
isNodeUri(String)	true
isLeafNode(String)	false
getNodeVersion(String)	Away returns 0
copy(String,String,boolean)	Not allowed for a single scaffold node as nodeUri, if the recurse parameter is false the DmtException COMMAND_NOT_ALLOWED

Any other operations must throw a DmtException with error code COMMAND_NOT_ALLOWED. The scope of a scaffold node is always PERMANENT. Scaffold nodes must have a Meta Node provided by the Dmt Admin service. This Meta Node must act as defined in Table 117.4.

Table 117.4 *Scaffold Meta Node Supported Methods*

Method	Description
can(int)	CMD_GET
getDefault()	null
getDescription()	null
getFormat()	FORMAT_NODE
getMax()	Double.MAX_VALUE
getMaxOccurrence()	1
getMimeTypes()	DDF_SCAFFOLD
getMin()	Double.MIN_VALUE
getRawFormatNames()	null
getScope()	PERMANENT
getValidNames()	null
getValidValues()	null
isLeaf()	false
isValidName(String)	true
isValidValue(DmtData)	false
isZeroOccurrenceAllowed()	true

If a Plugin is registered then it is possible that a scaffold node becomes a Data Plugin root node. In that case the node and the Meta Node must subsequently be provided by the Data Plugin and can thus become different. Scaffold nodes are virtual, there are therefore no events associated with the life cycle of a scaffold node.

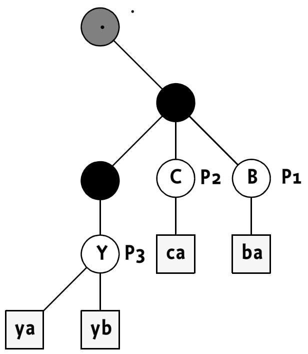
For example, there are three plugins registered:

```
URI                 Plugin     Children
./A/B               P1         ba
./A/C               P2         ca
./A/X/Y             P3         ya,yb
```

In this example, node B, C, and Y are the plugin roots of the different plugins. As there is no plugin the manage node A and X these must be provided by the Dmt Admin service. In this example, the child names returned from each node are summarized as follows:

Node	Children	Provided by
.	{ A }	Dmt Admin (scaffold node)
A	{ X, C, B }	Dmt Admin (scaffold node)
B	{ ba }	P1
C	{ ca }	P2
X	{ Y }	Dmt Admin (scaffold node)
Y	{ ya, yb }	P3

Figure 117.11 Example Scaffold Nodes



117.7 Sharing the DMT

The Dmt Admin service provides a model to integrate the management of the myriad of components that make up an OSGi device. This integration is achieved by sharing a single namespace: the DMT. Sharing a single namespace requires rules to prevent conflicts and to resolve any conflicts when Plugins register with plugin roots that overlap. It also requires rules for the Dmt Admin service when nodes are accessed for which there is no Plugin available.

This section defines the management of overlapping plugins through the *mount points*, places where a Parent Plugin can allow a Child Plugin to take over.

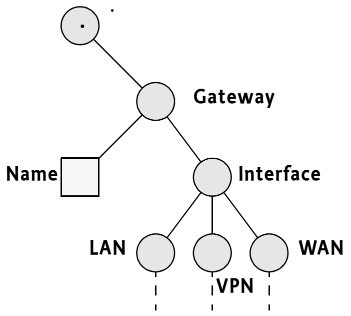
117.7.1 Mount Points

With multiple plugins the DMT is a *shared namespace*. Sharing requires rules to ensure that conflicts are avoided and when they occur, can be resolved in a consistent way. The most powerful and flexible model is to allow general overlapping. However, in practice this flexibility comes at the cost of ordering issues and therefore timing dependent results. A best practice is therefore to strictly control the points where the DMT can be extended for both Data and Exec Plugins.

A *mount point* is such a place. A Dmt Admin service at start up provides virtual mount points anywhere in the DMT and provides scaffold nodes for any intermediate nodes between the root of the DMT and the Plugin's root URI. Once a Plugin is mounted it is free to use its sub-tree (the plugin root and any ancestors) as it sees fit. However, this implies that the Plugin must implement the full subtree. In reality, many object models use a pattern where the different levels in the object model map to different domains.

For example, an Internet Gateway could have an object model where the general information, like the name, vendor, etc. is stored in the first level but any attached interfaces are stored in the sub-tree. However, It is highly unlikely that the code that handles the first level with the general information is actually capable of handling the details of, for example, the different network interfaces. It is actually likely that these network interfaces are dynamic. A Virtual Private Network (VPN) can add virtual network interfaces on demand. Such a could have the object model depicted in Figure 117.12.

Figure 117.12 Data Modeling



Forcing these different levels to be implemented by the same plugin violates one of the primary rules of modularity: *cohesion*. Plugins forced to handle all aspects become complex and hard to maintain. A Plugin like the one managing the Gateway node could provide its own Plugin mechanism but that would force a lot of replication and is error prone. For this reason, the Dmt Admin service allows a Plugin to provide *mount points* inside its sub-tree. A Plugin can specify that it has mount points by registering a MOUNT_POINTS service property (the constant is defined both in DataPlugin and ExecPlugin but have the same constant value). The type of this property must be String+, each string specifies a mount point. Each mount point is specified as a URI that is relative from the plugin root. That is, when the plugin root is ./A/B and the mount point is specified as C then the absolute URI of the mount point is ./A/B/C.

A Plugin that has mount points acts as a *Parent Plugin* to a number of *Child Plugins*. In the previous example, the LAN, VPN, and WAN nodes, can then be provided by separate Child Plugins even though the Gateway/Name node is provided by the Parent Plugin. In this case, the mount points are children of the Interface node.

A mount point can be used by a number of child plugins. In the previous example, there was a Child Plugin for the LAN node, the VPN node, and the WAN node. This model has the implicit problem that it requires coordination to ensure that their names are unique. Such a coordination between independent parties is complicated and error prone. Its is therefore possible to force the Dmt Admin service to provide unique names for these nodes, see *Shared Mount Points* on page 310.

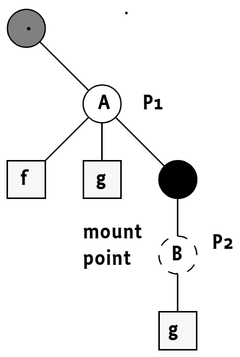
A Parent Plugin is not responsible for any scaffolding nodes to make its Child Plugins reachable. It is the responsibility of the Dmt Admin service to add child node names to any child node names returned from a plugin so that Child Plugins are always reachable.

For example, the following setup of plugins:

Plugin	Plugin Root	Mount Points
P1	./A	X/B
P2	./A/X/B	

This setup is depicted in Figure 117.13.

Figure 117.13 Example Scaffold Nodes For Child Plugin



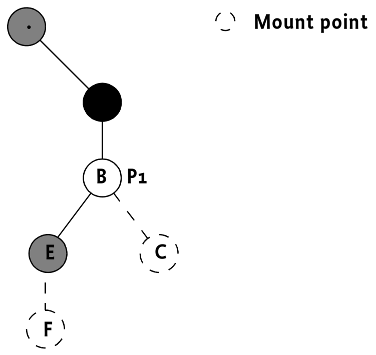
If the child node names are requested for the ./A node then the plugin P1 is asked for the child node names and must return the names [f,g]. However, if plugin P2 is mapped then the Dmt Admin service must add the scaffold node name that makes this plugin reachable from that level, the returned set must therefore be [f, g, X].

117.7.2 Parent Plugin

If a Plugin is registered with mount points then it is a *Parent Plugin*. A Parent Plugin must register with a single plugin root URI, that is the DATA_ROOT_URIS or EXEC_ROOT_URIS service properties must contain only one element. A Parent Plugin is allowed to be a Data and Exec Plugin at the same time. If a Parent Plugin is registered with multiple plugin root URIs then the Dmt Admin service must log an error and ignore the registration of such a Parent Plugin. A Parent Plugin can in itself also be a Child Plugin.

For example, a Plugin P1 that has a plugin root of ./A/B and provides a mount point at ./A/B/C and ./A/B/E/F. as depicted in Figure 117.14.

Figure 117.14 Example Mount Points



Registering such a Plugin would have to register the following service properties to allow the example configuration of the DMT:

```
dataRootUris                 ./A/B
mountPoints                  [ C, E/F ]
```

117.7.3 Shared Mount Points

Mount points can be shared between different Plugins. In the earlier example about the Gateway the Interface node contained a sub-tree of network interfaces. It is very likely in such an example that the Plugins for the VPN interface will be provided by a different organization than the WAN and LAN network interfaces. However, all these network interface plugins must share a single parent node, the Interface node, under which they would have to mount. Sharing therefore requires a prior agreement and a naming scheme.

The naming scheme is defined by using the number sign ('#' \u0023) to specify a *shared mount point*. A plugin root that ends with the number sign, for example ./A/B/#, indicates that it is willing to get any node under node B, leaving the naming of that node up to the Dmt Admin service. Shared mount points cannot overlap with normal mount points, the first one will become mapped and subsequent ones are in error, they are incompatible with each other. A Parent Plugin must specify a mount point explicitly as a shared mount point by using the number sign at the end of the mount point's relative URI.

A plugin is compatible with other plugins if all other plugins specify a shared mount point to the same URI. It is compatible with its Parent Plugin if the child's plugin root and the mount point are either shared or not.

The Dmt Admin service must provide a name for a plugin root that identifies a shared mount point such that every Plugin on that mount point has a unique integer name for that node level. The integer name must be >= 1. The name must be convertible to an int with the static Integer parseInt(String) method.

A management system in general requires permanent links to nodes. It is therefore necessary to choose the same integer every time a plugin is mapped to a shared mount point. A Child Plugin on a shared mount point must therefore get a permanent integer node name when it registers with a Persistent ID (PID). That is, it registers with the service property service.pid. The permanent link is then coupled to the PID and the bundle id since different bundles must be able to use the same PID. If a Plugin is registered with multiple PIDs then the first one must be used. Since permanent links can stay around for a long time implementations must strive to not reuse these integer names.

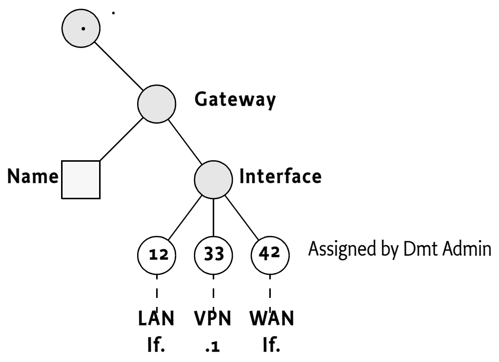
If no PID is provided then the Dmt Admin service must choose a new number that has not been used yet nor matches any persistently stored names that are currently not registered.

The Gateway example would require the following Plugin registrations:

```
Root URI                Mount Points                Plugin      Role
./Gateway               [Interface/#]               Gateway     Parent
./Gateway/Interface/#   []                          WAN If.     Child
./Gateway/Interface/#   []                          LAN If.     Child
./Gateway/Interface/#   []                          VPN.1       Child
```

This setup is depicted in Figure 117.15.

Figure 117.15 Mount Point Sharing



The Meta Node for a Node on the level of the Mount Point can specify either an existing Plugin or it can refer to a non-existing node. If the node exists, the corresponding Plugin must provide the Meta Node. If the node does not exist, the Dmt Admin service must provide the Meta Node. Such a Meta Node must provide the responses as specified in Table 117.4.

Table 117.5 Shared Mount Point Meta Node Supported Methods

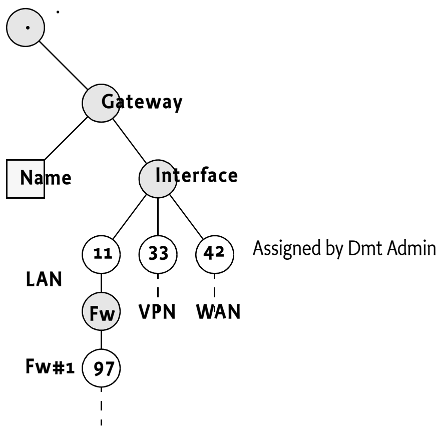
Method	Description
can(int)	CMD_GET
getDefault()	null
getDescription()	null
getFormat()	FORMAT_NODE
getMax()	Double.MAX_VALUE
getMaxOccurrence()	Integer.MAX_VALUE
getMimeTypes()	null
getMin()	Double.MIN_VALUE
getRawFormatNames()	null
getScope()	The scope will depend on the Parent
getValidNames()	null
getValidValues()	null
isLeaf()	false
isValidName(String)	name >=1 && name < Integer.MAX_VALUE
isValidValue(DmtData)	false
isZeroOccurrenceAllowed()	true

A URI can cross multiple mount points, shared and unshared. For example, if a network interface could be associated with a number of firewall rules then it is possible to register a URI on the designated network interface that refers to the Firewall rules. For the previous example, a Plugin could register a Firewall if the following registrations were done:

```
Root URI                    Mount Points    Plugin  Parent  Name
./Gateway                   [Interface/#]   Gw
./Gateway/Interface/#       [Fw/#]          WAN If. Gw      11
./Gateway/Interface/#       []              LAN If. Gw      33
./Gateway/Interface/#       []              VPN.1   Gw      42
./Gateway/Interface/11/Fw/#[]               Fw.1    WAN If. 97
```

This example DMT is depicted in Figure 117.16.

Figure 117.16 Mount Point Multiple Sharing



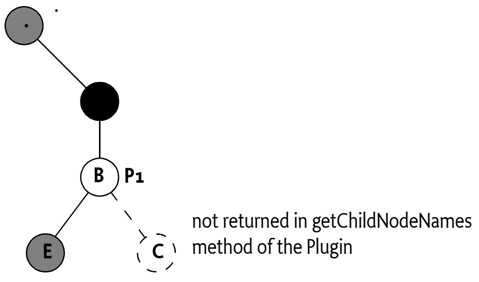
117.7.4 Mount Points are Excluded

Mount nodes are logically not included in the sub-tree of a Plugin. The Dmt Admin service must never ask any information from/about a Mount Point node to its Parent Plugin. A Parent Plugin must also not return the name of a mount point in the list of child node names, the Mount Point and its subtree is logically excluded from the sub-tree. For the Dmt Admin service an unoccupied mount point is a node that does not exist. Its name, must only be discoverable if a Plugin has actually mounted the node. The Dmt Admin service must ensure that the names of the mounted Plugins are included for that level.

In the case of shared mount points the Dmt Admin service must provide the children names of all registered Child Plugins that share that node level.

For example, a Plugin P1 registered with the plugin root of ./A/B, having two leaf nodes E, and a mount point C must not return the name C when the child node names for node B are requested. This is depicted in Figure 117.17. The Dmt Admin service must ensure that C is returned in the list of names when a Plugin is mounted on that node.

Figure 117.17 Example Exclusion



117.7.5 Mapping a Plugin

A Plugin is not stand alone, its validity can depend on other Plugins. Invalid states make it possible that a Plugin is either *mapped* or *unmapped*. When a Plugin is mapped it is available in the DMT and when it is unmapped it is not available. Any registration, unregistration, or modification of its services properties of a Plugin can potentially alter the mapped state of any related Plugin. A plugin becomes *eligible* for mapping when it is registered.

A plugin can have multiple roots. However, the mapping is described as if there is a single plugin root. Plugins with multiple roots must be treated as multiple plugins that can each independently be mapped or unmapped depending on the context.

If no Parent Plugin is available, the Dmt Admin service must act as a virtual Parent Plugin that allows mount points anywhere in the tree where there is no mapped plugin yet.

When a Plugin becomes eligible then the following assertions must be valid for that Plugin to become mapped:

- If it has one or more mount points then
  - It must have at most one Data and/or Exec Root URI.
  - None of its mount points must overlap.
  - Any already mapped Child Plugins must be compatible with its mount points
- If no mount points are specified then there must be no Child Plugins already registered
- The plugin root must be compatible with the corresponding parent's mount point
- The plugin root must be compatible with any other plugins on that mount point

If either of these assertions fail then the Dmt Admin service must log an error and ignore the registered Plugin, it must not become mapped. If, through the unregistration or modification of the service properties, the assertions can become valid then the Dmt Admin service must retry mapping the Plugin so that it can become available in the DMT. Any mappings and unmappings that affect nodes that are in the sub-tree of an active session must abort that session with a CONCURRENT_ACCESS exception.

When there are errors in the configuration then the ordering will define which plugins are mapped or not. Since this is an error situation no ordering is defined between conflicting plugins.

For example, a number of Plugins are registered in the given order:

Plugin Root	Children	Mount Points	Plugin
./A/B	E	C	P1
./A/B/C			P2
./A/B/D			P3

The first Plugin P1 will be registered immediately without problems. It has only a single plugin root as required by the fact that it is a Parent Plugin (it has a mount point). There are no Child Plugins yet so it is impossible to have a violation of the mount points. As there is no Parent Plugin registered, the Dmt Admin service will map plugin P1 and automatically provide the scaffold node A.

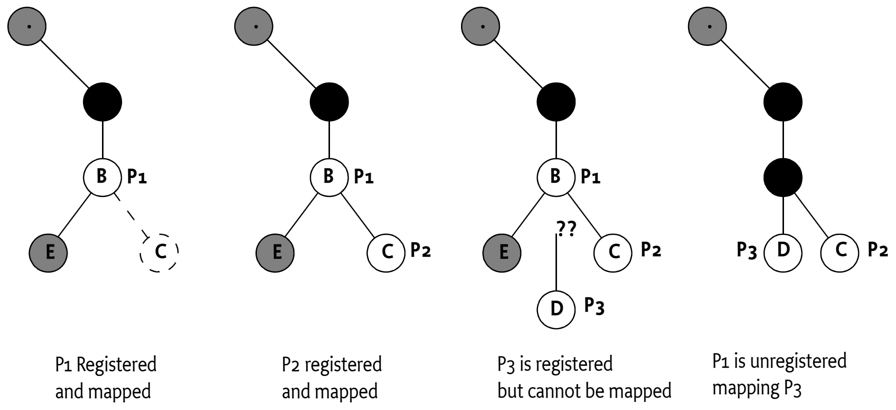
When Plugin P2 is registered its plugin root maps to a mount point in Plugin P1. As P2 is not a Parent Plugin it is only necessary that it has no Child Plugins. As it has no Child Plugins, the mapping will succeed.

Plugin P3 cannot be mapped because the Parent Plugin is P1 but P1 does not provide a mount point for P3's plugin root ./A/B/D.

If, at a later time P1 is unregistered then the Dmt Admin service must map plugin P3 and leave plugin P2 mapped. This sequence of action is depicted in Figure 117.18.

If plugin P1 becomes registered again at a later time it can then in its turn not be mapped as there would be a child plugin (P3) that would not map to its mount point.

Figure 117.18 Plugin Activation



117.7.6 Mount Plugins

In *Mapping a Plugin* on page 313 it is specified that a Plugin can be *mapped* or not. The mapped state of a Plugin can change depending on other plugins that are registered and unregistered. Plugins require in certain cases to know:

- What is the name of their root node if they mount on a shared mount point.
- What is the mapping state of the Plugin.

To find out these details a Plugin can implement the MountPlugin interface; this is a mixin interface, it is not necessary to register it as MountPlugin service. The Dmt Admin service must do an instanceof operation on Data Plugin services and Exec Plugin services to detect if they are interested in the mount point information.

The Mount Point interface is used by the Dmt Admin service to notify the Plugin when it becomes mapped and when it becomes unmapped. The Plugin will be informed about each plugin root separately.

The Mount Plugin specifies the following methods that are called synchronously:

- mountPointAdded(MountPoint) – The Dmt Admin service must call this method after it has mapped a plugin root. From this point on the given mount point provides the actual path until the mountPointRemoved(MountPoint) is called with an equal object. The given Mount Point can be used to post events.
- mountPointRemoved(MountPoint) – The Dmt Admin service must call this method after it has unmapped the given mount point. This method must always be called when a plugin root is unmapped, even if this is caused by the unregistration of the plugin.

As the mapping and unmapping of a plugin root can happen any moment in time a Plugin that implements the Mount Plugin interface must be prepared to handle these events at any time on any thread.

The MountPoint interface has two separate responsibilities:

- *Path* – The path that this Mount Point is associated with. This path is a plugin root of the plugin. This path is identical to the Plugin's root except when it is mounted on a shared mount point; in that case the URI ends in the name chosen by the Dmt Admin service. The getMountPath() method provides the path.
- *Events* – Post events about the given sub-tree that signal internal changes that occur outside a Dmt Session. The Dmt Admin service must treat these events as they were originated from modifications to the DMT. That is, they need to be forwarded to the Event Admin as well as the Dmt Lis-

teners. For this purpose there are the postEvent(String,String[],Dictionary) and postEvent(String,String[],String[],Dictionary) methods.

For example, a Data Plugin monitoring one of the batteries registers with the following service properties:

```
dataRootURIs                "./Device/Battery/#"
```

The Device node is from a Parent Plugin that provided the shared mount point. The Battery Plugin implements the MountPlugin interface so it gets called back when it is mapped. This will cause the Dmt Admin service to call the mountPointAdded(MountPoint) method on the plugin. In this case, it will get just one mount point, the mount point for its plugin root. If the Dmt Admin service would have assigned the Battery Plugin number 101 then the getMountPath() would return:

```
 [ ".", "Device", "Battery", "101" ]
```

As the Plugin monitors the charge state of the battery it can detect a significant change. In that case it must send an event to notify any observers. The following code shows how this could be done:

```
@Component(properties="dataRootURIs=./Device/Battery/#",
        provide=DataPlugin.class)
public class Battery implements DataPlugin, MountPlugin {
   Timer          timer;
   volatile float  charge;
   TimerTask      task;

   public void mountPointsAdded(final MountPoint[] mountPoints) {
      task = new TimerTask() {
         public void run() {
            float next = measure();
            if (Math.abs(charge - next) > 0.2) {
               charge = next;
               mountPoints[0].postEvent(DmtConstants.EVENT_TOPIC_REPLACED,
                     new String[] { "Charge" }, null);
            }
         }
      };
      timer.schedule(task, 1000);
   }

   public void mountPointsRemoved(MountPoint[] mountPoints) {
         task.cancel();
         task = null;
   }
   ... // Other methods
}
```

117.8 Access Control Lists

Each node in the DMT can be protected with an *access control list*, or *ACL*. An ACL is a list of associations between *Principal* and *Operation*:

- *Principal* – The identity that is authorized to use the associated operations. Special principal is the wildcard ('*' \u002A); the operations granted to this principal are called the *global permissions*. The global permissions are available to all principals.
- *Operation* – A list of operations: ADD, DELETE, GET, REPLACE, EXECUTE.

DMT ACLs are defined as strings with an internal syntax in [1] *OMA DM-TND v1.2 draft*. Instances of the ACL class can be created by supplying a valid OMA DM ACL string as its parameter. The syntax of the ACL is presented here in shortened form for convenience:

```
acl        ::= ( acl-entry ( '&' acl-entry )* )
acl-entry  ::= command '=' ( principals | '*' )
principals ::=  principal ( '+' principal )*
principal  ::= ~['=' '&' '*' '+' '\t' '\n' '\r']+
```

The principal name should only use printable characters according to the OMA DM specification.

```
command    ::= 'Add' | 'Delete' | 'Exec' | 'Get' | 'Replace'
```

White space between tokens is not allowed.

Examples:

```
Add=*&Replace=*&Get=*

Add=www.sonera.fi-8765&Delete=www.sonera.fi-8765&Replace=www.sonera.fi-
8765+321_ibm.com&Get=*
```

The Acl(String) constructor can be used to construct an ACL from an ACL string. The toString() method returns a String object that is formatted in the specified form, also called the canonical form. In this form, the principals must be sorted alphabetically and the order of the commands is:

```
 ADD,   DELETE,   EXEC,   GET,   REPLACE
```

The Acl class is immutable, meaning that a Acl object can be treated like a string, and that the object cannot be changed after it has been created.

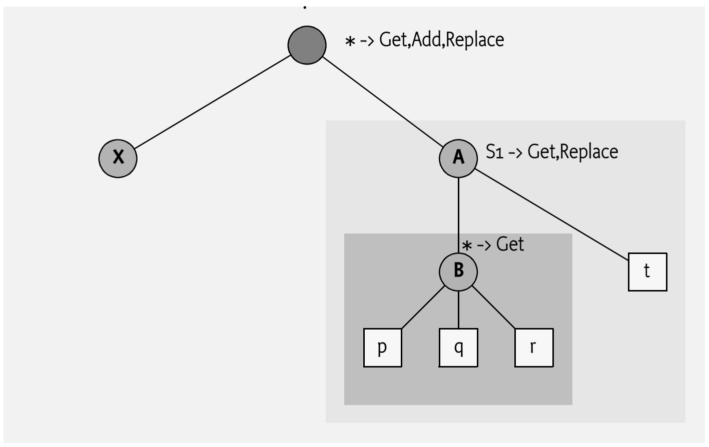
ACLs must only be verified by the Dmt Admin service when the session has an associated principal.

ACLs are properties of nodes. If an ACL is *not set* (i.e. contains no commands nor principals), the *effective* ACL of that node must be the ACL of its first ancestor that has a non-empty ACL. This effective ACL can be acquired with the getEffectiveNodeAcl(String) method. The root node of DMT must always have an ACL associated with it. If this ACL is not explicitly set, it should be set to Add=*&Get=*&Replace=*.

This effect is shown in Figure 117.19. This diagram shows the ACLs set on a node and their effect (which is shown by the shaded rectangles). Any principal can get the value of p, q and r, but they cannot replace, add or delete the node. Node t can only be read and replaced by principal S1.

Node X is fully accessible to any authenticated principal because the root node specifies that all principals have Get, Add and Replace access (*->G,A,R).

Figure 117.19 ACL inheritance



The definition and example demonstrate the access rights to the properties of a node, which includes the value.

Changing the ACL property itself has different rules. If a principal has Replace access to an interior node, the principal is permitted to change its own ACL property *and* the ACL properties of all its child nodes. Replace access on a leaf node does not allow changing the ACL property itself.

In the previous example, only principal S1 is authorized to change the ACL of node B because it has Replace permission on node B's parent node A.

Figure 117.20 ACLs for the ACL property

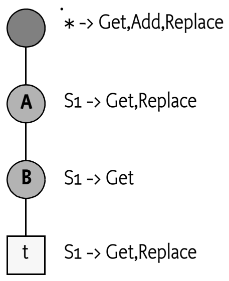


Figure 117.20 demonstrates the effect of this rule with an example. Server S1 can change the ACL properties of all interior nodes. A more detailed analysis:

- *Root* – The root allows all authenticated principals to access it. The root is an interior node so the Replace permission permits the change of the ACL property.
- *Node A* – Server S1 has Replace permission and node A is an interior node so principal S1 can modify the ACL.
- *Node B* – Server S1 has no Replace permission for node B, but the parent node A of node B grants principal S1 Replace permission, and S1 is therefore permitted to change the ACL.
- *Node t* – Server S1 must not be allowed to change the ACL of node t, despite the fact that it has Replace permission on node t. For leaf nodes, permission to change an ACL is defined by the Replace permission in the parent node's ACL. This parent, node B, has no such permission set and thus, access is denied.

The following methods provide access to the ACL property of the node.

- getNodeAcl(String) – Return the ACL for the given node, this method must not take any ACL inheritance into account. The ACL may be null if no ACL is set.

- getEffectiveNodeAcl(String) – Return the effective ACL for the given node, taking any inheritance into account.
- setNodeAcl(String,Acl) – Set the node's ACL. The ACL can be null, in which case the effective permission must be derived from an ancestor. The Dmt Admin service must call nodeChanged(String[]) on the data session with the given plugin to let the plugin update any timestamps and versions.

The Acl class maintains the permissions for a given principal in a bit mask. The following permission masks are defined as constants in the Acl class:

- ADD
- DELETE
- EXEC
- GET
- REPLACE

The class features methods for getting permissions for given principals. A number of methods allow an existing ACL to be modified while creating a new ACL.

- addPermission(String,int) – Return a new Acl object where the given permissions have been added to permissions of the given principal.
- deletePermission(String,int) – Return a new Acl object where the given permissions have been removed from the permissions of the given principal.
- setPermission(String,int) – Return a new Acl object where the permissions of the given principal are overwritten with the given permissions.

Information from a given ACL can be retrieved with:

- getPermissions(String) – (int) Return the combined permission mask for this principal.
- getPrincipals() – (String[]) Return a list of principals (String objects) that have been granted permissions for this node.

Additionally, the isPermitted(String,int) method verifies if the given ACL authorizes the given permission mask. The method returns true if all commands in the mask are allowed by the ACL.

For example:

```
Acl   acl = new Acl("Get=S1&Replace=S1");

if (  acl.isPermitted("S1", Acl.GET+Acl.REPLACE ))
   ... // will execute

if ( acl.isPermitted(
   "S1", Acl.GET+Acl.REPLACE+Acl.ADD ))
   ... // will NOT execute
```

117.8.1 Global Permissions

Global permissions are indicated with the '*' and the given permissions apply to all principals. Processing the global permissions, however, has a number of non-obvious side effects:

- Global permissions can be retrieved and manipulated using the special'*' principal: all methods of the Acl class that have a principal parameter also accept this principal.
- Global permissions are automatically granted to all specific principals. That is, the result of the getPermissions or isPermitted methods will be based on the OR of the global permissions and the principal-specific permissions.
- If a global permission is revoked, it is revoked from all specific principals, even if the specific principals already had that permission before it was made global.

- None of the global permissions can be revoked from a specific principal. The OMA DM ACL format does not handle exceptions, which must be enforced by the deletePermission and setPermission methods.

117.8.2 Ghost ACLs

The ACLs are fully maintained by the Dmt Admin service and enforced when the session has an associated principal. A plugin must be completely unaware of any ACLs. The Dmt Admin service must synchronize the ACLs with any change in the DMT that is made through its service interface. For example, if a node is deleted through the Dmt Admin service, it must also delete an associated ACL.

The DMT nodes, however, are mapped to plugins, and plugins can delete nodes outside the scope of the Dmt Admin service.

As an example, consider a configuration record which is mapped to a DMT node that has an ACL. If the configuration record is deleted using the Configuration Admin service, the data disappears, but the ACL entry in the Dmt Admin service remains. If the configuration dictionary is recreated with the same PID, it will get the old ACL, which is likely not the intended behavior.

This specification does not specify a solution to solve this problem. Suggestions to solve this problem are:

- Use a proprietary callback mechanism from the underlying representation to notify the Dmt Admin service to clean up the related ACLs.
- Implement the services on top of the DMT. For example, the Configuration Admin service could use a plugin that provides general data storage service.

117.9 Notifications

In certain cases it is necessary for some code on the device to alert a remote management server or to initiate a session; this process is called sending a notification or an *alert*. Some examples:

- A Plugin that must send the result of an asynchronous EXEC operation.
- Sending a request to the server to start a management session.
- Notifying the server of completion of a software update operation.

Notifications can be sent to a management server using the sendNotification(String,int,String, AlertItem[]) method on the Notification Service. This method is on the Notification Service and not on the session, because the session can already be closed when the need for an alert arises. If an alert is related to a session, the session can provide the required principal, even after it is closed.

The remote server is alerted with one or more AlertItem objects. The AlertItem class describes details of the alert. An alert code is an alert type identifier, usually requiring specifically formatted AlertItem objects.

The data syntax and semantics vary widely between various alerts, and so does the optionality of particular parameters of an alert item. If an item, such as source or type, is not defined, the corresponding getter method must return null.

The AlertItem class contains the following items. The value of these items must be defined in an alert definition:

- source – (String) The URI of a node that is related to this request. This parameter can be null.
- type – (String) The type of the item. For example, x-oma-application:syncml.samplealert in the Generic Alert example.
- mark – (String) Mark field of an alert. Contents depend on the alert type.
- data – (DmtData) The payload of the alert with its type.

An AlertItem object can be constructed with two different constructors:

- AlertItem(String,String,String,DmtData) – This method takes all the previously defined fields.
- AlertItem(String[],String,String,DmtData) – Same as previous but with a convenience parameter for a segmented URI.

The Notification Service provides the following method to send AlertItem objects to the management server:

- sendNotification(String,int,String,AlertItem[]) – Send the alert to the server that is associated with the session. The first argument is the name of the principal (identifying the remote management system) or null for implementation defined routing. The int argument is the *alert type*. The alert types are defined by *managed object types*. The third argument (String) can be used for the correlation id of a previous execute operation that triggered the alert. The AlertItem objects contain the data of the alert. The method will run asynchronously from the caller. The Notification Service must provide a reliable delivery method for these alerts. Alerts must therefore not be re-transmitted.
  When this method is called with null correlator, null or empty AlertItem array, and a 0 code as values, it should send a protocol specific notification that must initiate a new management session.

Implementers should base the routing on the session or server information provided as a parameter in the sendNotification(String,int,String,AlertItem[]) method. Routing might even be possible without any routing information if there is a well known remote server for the device.

If the request cannot be routed, the Alert Sender service must immediately throw a Dmt Exception with a code of ALERT_NOT_ROUTED. The caller should not attempt to retry the sending of the notification. It is the responsibility of the Notification Service to deliver the notification to the remote management system.

117.9.1 Routing Alerts

The Notification Service allows external parties to route alerts to their destination. This mechanism enables Protocol Adapters to receive any alerts for systems with which they can communicate.

Such a Protocol Adapter should register a Remote Alert Sender service. It should provide the following service property:

- *principals* – (String+) The array of principals to which this Remote Alert Sender service can route alerts. If this property is not registered, the Remote Alert Sender service will be treated as the default sender. The default alert sender is only used when a more specific alert sender cannot be found.

If multiple Remote Alert Sender services register for the same principals highest ranking service is taken as defined in the OSGi Core.

117.10 Exceptions

Most of the methods of this Dmt Admin service API throw Dmt Exceptions whenever an operation fails. The DmtException class contains numeric error codes which describe the cause of the error. Some of the error codes correspond to the codes described by the OMA DM spec, while some are introduced by the OSGi Alliance. The documentation of each method describes what codes could potentially be used for that method.

The fatality of the exception decides if a thrown Exception rolls back an atomic session or not. If the isFatal() method returns true, the Exception is fatal and the session must be rolled back.

All possible error codes are constants in the DmtException class.

117.11 Events

There are the following mechanisms to work with events when using the Dmt Admin service.

- *Event Admin service* – Standard asynchronous notifications
- *Dmt Event Listener service* – A white board model for listener. A registered DmtEventListener service can use service properties to filter the received events

In both cases events are delivered asynchronously and ordered per listener unless otherwise specified. Events to the DMT can occur because of modifications made in a session or they can occur because a Plugin changes its internal state and notifies the Dmt Admin service through the MountPoint interface.

Changes made through a session always start with a SESSION_OPENED event directly after the session is opened. This event must contain the properties defined in *Life Cycle Event Properties* on page 323.

If events originate from an atomic session then these events must be queued until the sessions is successfully committed, which can happen multiple times over the life time of a session. If the session is rolled back or runs into an error then none of the queued events must be sent.

When a session is closed, which can happen automatically when the session fails, then the SESSION_CLOSED event must be sent. This event must happen after any queued events. This closed event must contain the properties defined in *Life Cycle Event Properties* on page 323.

An event must only be sent when that type of event actually occurred.

117.11.1 Event Admin

Event Admin, when present, must be used to deliver the Dmt Admin events asynchronously. The event types are specified in Table 117.7 on page 322, the Topic column defines the Event Admin topic. The Table 117.9 on page 324 and Table 117.10 on page 324 define the Life Cycle and Session properties that must be passed as the event properties of Event Admin.

117.11.2 Dmt Event Listeners

To receive the Dmt Admin events it is necessary to register a Dmt Event Listener service. It is possible to filter the events by registering a combination of the service properties defined in Table 117.6.

Table 117.6 Service Properties for the Dmt Event Listener

Service Property	Data Type	Default	Description
FILTER_EVENT	Integer	All Events	A bitmap of DmtEvent types: SESSION_OPENED, ADDED, COPIED, DELETED, RENAMED, REPLACED, and SESSION_CLOSED. A Dmt Event's type must occur in the bitmap to be delivered.
FILTER_PRINCIPAL	String+	Any node	Only deliver Dmt Events for which at least one of the given principals has the right to Get that node.
FILTER_SUBTREE	String+	Any node	This property defines a number of sub-trees by specifying the URI of the top nodes of these sub-trees. Only events that occur in one of the sub-trees must be delivered.

A Dmt Event must only be delivered to a Dmt Event Listener if the Bundle that registers the Dmt Event Listener service has the GET Dmt Permission for each of the nodes used in the nodes and newNodes properties as tested with the Bundle hasPermission method.

The Dmt Admin service must track Dmt Event Listener services and deliver matching events as long as a Dmt Event Listener service is registered. Any changes in the service properties must be expediently handled.

A Dmt Event Listener must implement the `changeOccurred(DmtEvent)` method. This method is called asynchronously from the actual event occurrence but each listener must receive the events in order.

Events are delivered with a `DmtEvent` object. This object provides access to the properties of the event. Some properties are available as methods others must be retrieved through the `getProperty(String)` method. The methods that provide property information are listed in the property tables, see Table 117.9 on page 324.

117.11.3 Atomic Sessions and Events

The intent of the events is that a listener can follow the modifications to the DMT from the events alone. However, from an efficiency point of view certain events should be coalesced to minimize the number of events that a listener need to handle. For this reason, the Dmt Admin service must coalesce events if possible.

Two consecutive events can be coalesced when they are of the same type. In that case the `nodes` and, if present, the `newNodes` of the second event can be concatenated with the first event and the `timestamp` must be derived from the first event. It is not necessary to remove duplicates from the `nodes` and `newNodes`. This guarantees that the order of the nodes is in the order of the events.

117.11.4 Event Types

This section describes the events that can be generated by the Dmt Admin service. `Event Types`Table 117.7 enumerates all the events and provides the name of the topic of Event Admin and the Dmt Event type for the listener model.

There are two kinds of events:

- *Life Cycle Events* – The events for session open and closed are the session events.
- *Session Events* – `ADDED`, `DELETED`, `REPLACED`, `RENAMED`, and `COPIED`.

Session and life cycle events have different properties.

Table 117.7 Event Types

Event	Topic	Dmt Event Type	Description
SESSION OPENED	org/osgi/service/dmt/DmtEvent/ SESSION_OPENED	SESSION_OPENED	A new session was opened. The event must the properties defined in Table 117.10 on page 324.
ADDED	org/osgi/service/dmt/DmtEvent/ ADDED	ADDED	One or more nodes were added.
DELETED	org/osgi/service/dmt/DmtEvent/ DELETED	DELETED	One or more existing nodes were deleted.
REPLACED	org/osgi/service/dmt/DmtEvent/ REPLACED	REPLACED	Values of nodes were replaced.
RENAMED	org/osgi/service/dmt/DmtEvent/ RENAMED	RENAMED	Existing nodes were renamed.

Table 117.7 Event Types

Event	Topic	Dmt Event Type	Description
COPIED	org/osgi/service/dmt/DmtEvent/COPIED	COPIED	Existing nodes were copied. A copy operation does not trigger an ADDED event (in addition to the COPIED event), even though new node(s) are created. For efficiency reasons, recursive copy and delete operations must only generate a single COPIED and DELETED event for the root of the affected sub-tree.
SESSION CLOSED	org/osgi/service/dmt/DmtEvent/SESSION_CLOSED	SESSION_CLOSED	A session was closed either because it was closed explicitly or because there was an error detected. The event must the properties defined in Table 117.10 on page 324.

117.11.5 General Event Properties

The following properties must be available as the event properties in Event Admin service and the properties in the Dmt Event for Dmt Event Listener services.

Table 117.8 General Event

Property Name	Type	Dmt Event	Description
event.topics	String		Event topic, required by Event Admin but must also be present in the Dmt Events.
session.id	Integer	getSessionId()	A unique identifier for the session that triggered the event. This property has the same value as getSessionId() of the associated DMT session. If this event is generated outside a session then the session id must be -1, otherwise it must be >=1.
timestamp	Long		The time the event was started as defined by System.currentTimeMillis()
bundle	Bundle		The initiating Bundle, this is the bundle that caused the event. This is either the Bundle that opened the associated session or the Plugin's bundle when there is no session (i.e. the session id is -1).
bundle.signer	String+		The signer of the initiating Bundle
bundle.symbolicname	String		The Bundle Symbolic name of the initiating Bundle
bundle.version	Version		The Bundle version of the initiating Bundle.
bundle.id	Long		The Bundle Id of the initiating Bundle.

117.11.6 Session Event Properties

All Life Cycle events must have the properties defined in *Event Properties for Life Cycle Events* on page 324.

117.11.7 Life Cycle Event Properties

All session events must have the properties defined in *Event Properties for Life Cycle Events* on page 324.

Table 117.9 Event Properties for Life Cycle Events

Property Name	Type	Dmt Event	Description
nodes	String[]	getNodes()	The absolute URIs of each affected node. This is the nodeUri parameter of the Dmt API methods. The order of the URIs in the array corresponds to the chronological order of the operations. In case of a recursive delete or copy, only the session root URI is present in the array.
newnodes	String[]	getNewNodes()	The absolute URIs of new renamed or copied nodes. Only the RENAMED and COPIED events have this property. The newnodes array runs parallel to the nodes array. In case of a rename, newnodes[i] must contains the new name of nodes[i], and in case of a copy, newnodes[i] is the URI to which nodes[i] was copied.

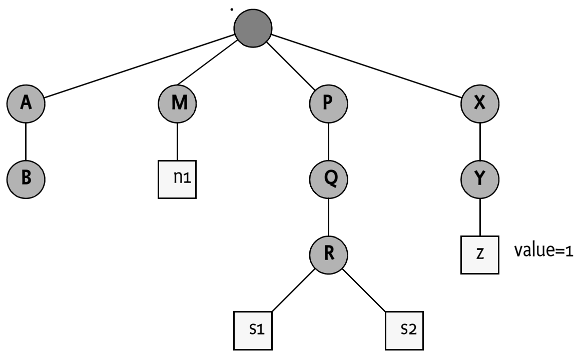
Table 117.10 Event Properties For Session Event

Property Name	Type	Dmt Session	
session.rooturi	String	getRootUri()	The root URI of the session that triggered the event.
session.principal	String	getPrincipal()	The principal of the session, or absent if no principal is associated with this session. In the latter case the method returns null.
session.locktype	Integer	getLockType()	The lock type of the session. The number is mapped as follows: • LOCK_TYPE_SHARED – 0 • LOCK_TYPE_EXCLUSIVE – 1 • LOCK_TYPE_ATOMIC – 2
session.timeout	Boolean		If the session timed out then this property must be set to true. If it did not time out this property must be false.
exception	Throwable		The name of the actual exception class if the session had a fatal exception.
exception.message	String		Must describe the exception if the session had a fatal exception or timed out.
exception.class	String		The name of the actual exception class if the session had a fatal exception or timed out.

117.11.8 Example Event Delivery

The example in this section shows the change of a non-trivial tree and the events that these changes will cause.

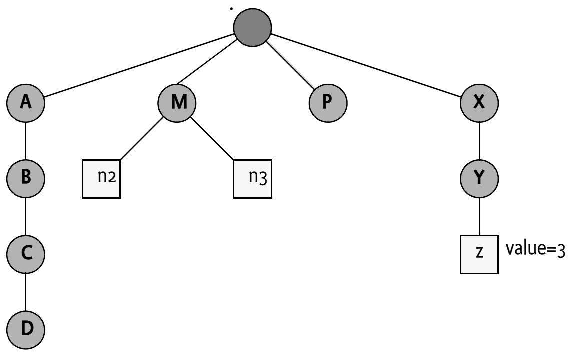
Figure 117.21 Example DMT before



For example, in a given session, when the DMT in Figure 117.21 is modified with the following operations:

- Open atomic session 42 on the root URI
- Add node ./A/B/C
- Add node ./A/B/C/D
- Rename ./M/n1 to./M/n2
- Copy ./M/n2 to ./M/n3
- Delete node ./P/Q
- Add node ./P/Q
- Delete node ./P/Q
- Replace ./X/Y/z with 3
- Commit
- Close

Figure 117.22 Example DMT after



When the Dmt Session is closed (assuming it is atomic), the following events are published:

```
SESSION_OPENED {
      session.id = 42
      session.rooturi=.
      session.principal=null
      session.locktype=2
      timestamp=1313411544752
      bundle=<Bundle>
      bundle.signer=[]
      bundle.symbolicname"com.acme.bundle"
      bundle.version=1.2.4711
      bundle.id=442
```

```
     ...
}
ADDED {
     nodes = [./A/B/C, ./A/B/C/D ]# note the coalescing
     session.id = 42
     ...
}
RENAMED {
     nodes = [ ./M/n1 ]
     newnodes = [ ./M/n2 ]
     session.id = 42
     ...
}
COPIED {
     nodes = [ ./M/n2 ]
     newnodes = [ ./M/n3 ]
     session.id = 42
     ...
}
DELETED {
     nodes = [ ./P/Q ]
     session.id = 42
     ...
}
ADDED {
     nodes = [ ./P/Q ]
     session.id = 42
     ...
}
DELETED {
     nodes = [ ./P/Q ]
     session.id = 42
     ...
}
REPLACED {
     nodes = [ ./X/Y/z ]
     session.id = 42
     ...
}
SESSION_CLOSED {
     session.id = 42
     session.rooturi=.
     session.principal=null
     session.locktype=2
     ...
}
```

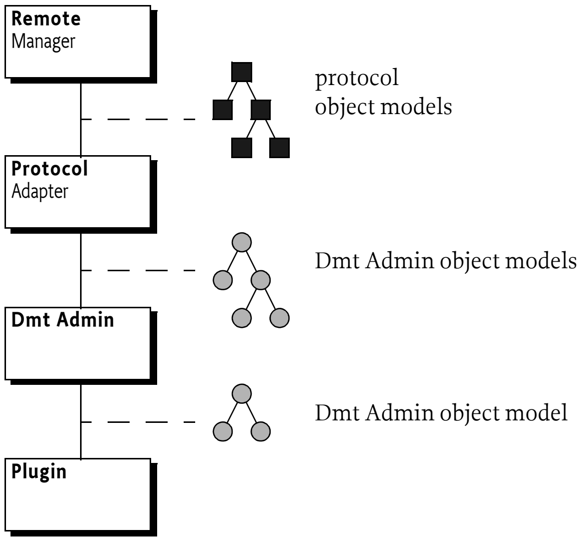
117.12 OSGi Object Modeling

117.12.1 Object Models

Management protocols define only half the picture; the object models associated with a particular protocol are the other half. Object models are always closely associated with a remote management protocol since they are based on the data types and actions that are defined in the protocol. Even small differences between the data types of a protocol and its differences make accurate mapping between protocols virtually impossible. It is therefore necessary to make the distinction between *native* and *foreign* protocols for an object model.

A native protocol for an object model originates from the same specification organization. For example, OMA DM consists of a protocol based on SyncML and a number of object models that define the structure and behavior of the nodes of the DMT. The FOMA specification defines an OMA DM native object model, it defines how firmware management is done. This is depicted in Figure 117.23.

Figure 117.23 Device Management Architecture



If an object implements a standardized data model it must be visible through its *native* Protocol Adapter, that is the Protocol Adapter that belongs to the object model's standard. For example, an `ExecutionUnit` node defined in UPnP Device Management could be implemented as a bundle, exposed through a Data Plugin for the Dmt Admin service, and then translated by its native UPnP Protocol Adapter.

If an object is present in the Dmt Admin service it is also available to *foreign* Protocol Adapters. A foreign Protocol Adapter is any Protocol Adapter except its native Protocol Adapter. For example, the Broadband Forum's `ExecutionUnit` could be browsed on the foreign OMA DM protocol.

In a foreign Protocol Adapter the object model should be *browsable* but it would not map to one of its native object models. Browsable means that the information is available to the Protocol Adapter's remote manager but not recognized as a standard model for the manager. Browse can include, potentially limited, manipulation.

In a native Protocol Adapter it is paramount that the mapping from the DMT to the native object is fully correct. It is the purpose of this part of the Dmt Admin service specification to allow the native Protocol Adapter to map the intentions of the Plugin without requiring knowledge of the specific native object model. That is, a TR-069 Plugin implementing a WAN interface must be available over the TR-069 protocol without the Protocol Adapter having explicit knowledge about the WAN interfaces object models from Broadband Forum.

Therefore, the following use cases are recognized:

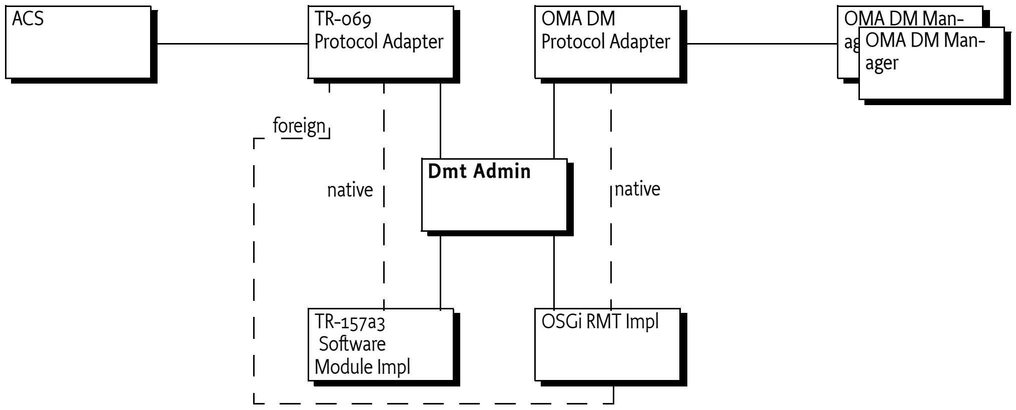
- *Foreign Mapping* – Foreign mapping can is best-effort as there is no object model to follow. Each Protocol Adapter must define how the Dmt Admin model is mapped for this browse mode.
- *Native Mapping* – Native mapping must be 100% correct. As it is impossible automatically map DMTs to arbitrary protocols this specification provides the concept of a mapping model that allows a Plugin to instruct its native Protocol Adapter using Meta Nodes.

117.12.2 Protocol Mapping

The OSGi Alliance specifies an Execution Environment that can be used as a basis for residential gateways, mobiles, or other devices. This raises the issue how to expose the manageability of an OSGi device and the *objects*, the units of manageability, that are implemented through Plugins. Ideally, an object should be able to expose its management interface once and then Protocol Adapters convert the management interface to specific device management stacks. For example, an object can be exposed through the Dmt Admin service where then a TR-069 Protocol Adapter maps the DMT to the TR-069 Remote Procedure Calls (RPC).

Figure 117.24 shows an example of a TR-069 Protocol Adapter and an OMA DM Protocol Adapter. The TR-069 Protocol Adapter should be able to map native TR-069 objects in the DMT (the `Software Modules Impl` in the figure) to Broadband Forum's object models. It should also be able to browse the foreign DMT and other objects that are not defined in Broadband forum but can be accessed with the TR-069 RPCs.

Figure 117.24 Implementing & Browsing



A *Protocol Mapping* is a document that describes the default mapping and the native mechanism for exact mapping.

The following sections specify how Plugins must implement an object model that is exposed through the Dmt Admin service. This model is limited from the full Dmt Admin service capabilities so that for each protocol it is possible to specify a default mapping for browsing as well as a mechanism to ensure that special conversion requirements can be communicated from a Plugin to its native Protocol Adapter.

117.12.3 Hierarchy

The Dmt Admin model provides an hierarchy of *nodes*. Each node has a *type* that is reflected by its Meta Node. A node is addressed with a URI. The flexibility of the Dmt Admin service allows a large number of constructs, for example, the name of the node can be used as a *value*, a feature that some management standards support. To simplify mapping to foreign Protocol Adapters, some of the fundamental constructs have been defined in the following sections.

117.12.4 General Restriction Guidelines

The Dmt Admin service provides a very rich tool to model complex object structures. Many choices can be made that would make it very hard to browse DMTs on non-OMA DM protocols or make the DMT hard to use through the Dmt Admin service. As Plugins can always signal special case handling to their native Protocol Adapter, any object model design should strive to be easy to use for the developers and managers. Therefore, this section provides a number of guidelines for the design of such object models that will improve the browsing experience for many Protocol Adapters.

- *Reading of a node must not change the state of a device* – Management systems must be able to browse a tree without causing any side effects. If reading modified the DMT, a management system would have no way to warn the user that the system is modified. There are a number of technical reasons as well (race conditions, security holes, and eventing) but the most important reason is the browseability of the device.
- *No use of recursive structures* – The Dmt Admin service provides a very rich tree model that has no problem with recursion. However, this does not have to be true for other models. To increase the changes that a model is browsable on another device it is strongly recommended to prevent recursive models. For example, TR-069 cannot handle recursive models.
- *Only a single format per meta node* – Handling different types in different nodes simplifies the data conversion for both foreign and native protocols. Having a single choice from the Meta Node makes the conversion straightforward and does not require guessing.
- *All nodes must provide a Meta Node* – Conversion without a Meta Node makes the conversion very hard since object model schemas are often not available in the Protocol Adapter.
- *Naming* – Structured node members must have names only consisting of [a-zA-Z0-9] and must always start with a character [a-zA-z]. Member names must be different regardless of the case, that is Abc and ABC must not both be members of the same structured node. The reason for this restriction is that it makes it more likely that the chosen names are compatible with the supported protocols and do not require escaping.
- *Typing* – Restrict the used formats to formats that maximize both the interoperability as the ease of use for Java developers. The following type are widely supported and are easy to use from Java:

  - FORMAT_STRING
  - FORMAT_BOOLEAN
  - FORMAT_INTEGER
  - FORMAT_LONG
  - FORMAT_FLOAT
  - FORMAT_DATE_TIME
  - FORMAT_BINARY

117.12.5 DDF

The Data Description Format is part of OMA DM; it provides a description language for the object model. The following table provides an example of the Data Description Format as used in the OSGi specifications.

Name	Actions	Type	Card.	S	Description
FaultType	Get	integer	1	P	...

The columns have the following meanings:

- *Name* – The name of the node
- *Actions* – The set of actions that can be executed on the node, see *Operations* on page 299.
- *Type* – The type of the node. All lower case are primitives, a name starting with an upper case is an interior node type. MAP, LIST, and SCAFFOLD are the special types. The NODE type is like an ANY type. Other type names are then further specified in the document. See *Types* on page 330.
- *Cardinality* – The number of occurrences of the node, see *Cardinality* on page 300.
- *Scope* – The scope of the node, see *Scope* on page 300.
- *Description* – A description of the node.

117.12.6 Types

Each node is considered to have a *type*. The Dmt Admin service has a number of constructs that have typing like behavior. There are therefore the following *kind* of types:

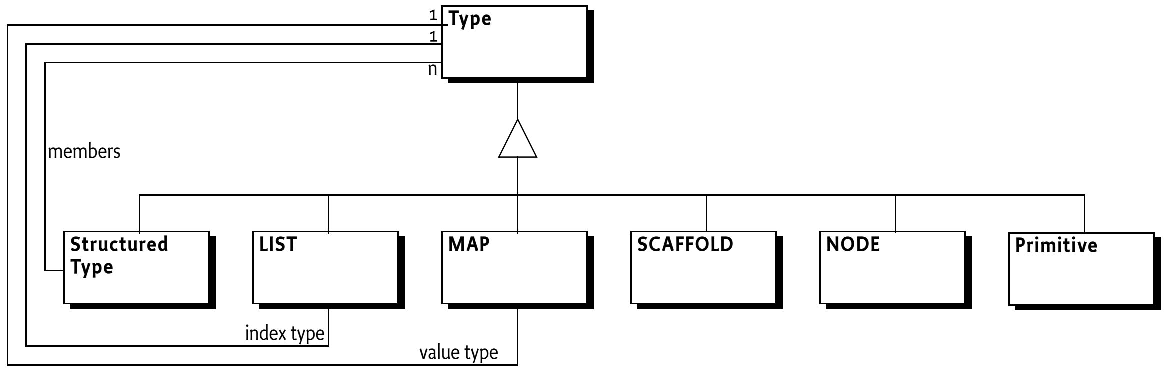
- *Primitives* – Primitives are data types like integers and strings; they include all the Dmt Admin data formats. See *Primitives* on page 331. Primitive type names are always lower case to distinguish them from the interior node type names.
- *Structured Types* – A structured type types a structured node. See *Structured Nodes* on page 331. A structured type has a type name that starts with an uppercase. Object models generally consist of defining these types.
- NODE – A general unqualified Dmt Admin node.
- LIST – A node that represents a homogeneous collection of child nodes; the name of the child nodes is the index in the collection. See *LIST Nodes* on page 331.
- MAP – A node that represents a mapping from a key, the name of the child node, and a value, the value of the child node. All values have the same type. See *MAP Nodes* on page 333.
- SCAFFOLD – A node provided by the Dmt Admin service or a Parent Plugin to make it possible to discover a DMT, see *Scaffold Nodes* on page 305.

Nodes are treated as if there is a single type system. However, the Dmt Admin type system has the following mechanisms to type a node:

- *Format* – The Dmt Admin primitive types used for leaf nodes, as defined on Dmt Data.
- *MIME* – A MIME type on a leaf node which is available through getNodeType(String).
- *DDF Document URI* – A Data Description Format URI that provides a type name for an interior node. The URI provides a similar role as the MIME type for the leaf node and is also available through getNodeType(String).

The Dmt Admin service provides the MIME type for leaf nodes and the DDF Document URI for interior nodes through the getNodeType(String) method. As both are strings they can both be used as type identifiers. The different types are depicted in Figure 117.25.

Figure 117.25 Type inheritance and structure



117.12.7 Primitives

A primitive is a value stored in a leaf node. In the Dmt Admin service, the type of the primitive is called the *format*. The Dmt Admin service supports a large number of types that have semantic overlap. A Protocol Mapping must provide a unique mapping from each Dmt Admin format to the corresponding protocol type and provide conversion from a protocol type to the corresponding Dmt Admin types defined in a Meta Node.

Primitives are documented in OSGi object models with a lower case name that is the last part of their format definition. For example, for FORMAT_STRING the DDF type name is string. A primitive DDF for an integer leaf node therefore looks like:

Name	Act	Type	Card.	S	Description
FaultType	Get	integer	1	P	...

117.12.8 Structured Nodes

A *structured node* is like a struct in C or a class in an object oriented languages. A structured node is an interior node with a set of members (child nodes) with fixed names, it is never possible to add or remove such members dynamically. The meaning of each named node and its type is usually defined in a management specification. For example, a node representing the OSGi Bundle could have a BundleId child-node that maps to the getBundleId() method on the Bundle interface.

It is an error to add or delete members to a Structured node, this must be reflected in the corresponding Meta Node, that is, Structured nodes must never have the Add or Delete action.

A structured node is defined in a *structured type* to allow the reuse of the same information in different places in an object model. A structured type defines the members and their behaviors. A structured type can be referred by its name. The name of the type is often, but not required, the name of the member.

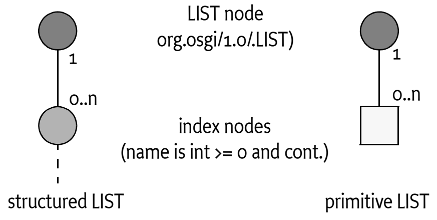
For example, a Unit structured type could look like:

Name	Act	Type	Card.	S	Description
Id	Get	long	1	P	...
URL	Get Set	string	1	P	...
Name	Get	string	1	P	...
Certificate	Get	LIST	1	P	
[index]	Get	Certificate	1	D	Note the use of a structured type.

117.12.9 LIST Nodes

A LIST node is an interior node representing a *collection* of elements. The elements are stored in the child nodes of the LIST node, they are called the *index nodes*. All index nodes must have the same type. The names of the index nodes are synthesized and represent the index of the index node. The first node is always named 0 and the sibling is 1, 2, etc. The sequence must be continuous and must have no missing indexes. A node name is always a string, it is therefore the responsibility of the plugin to provide the proper names. The index is assumed to be a signed positive integer limiting the LIST nodes size to Integer.MAX_VALUE elements.

Figure 117.26 *LIST Nodes*



Index nodes should only be used for types where the value of the index node is the identity. For example, a network interface has an identity; a manager will expect that a node representing such as a network interface node will always have the same URI even if other interfaces are added and deleted. Since LIST nodes renumber the index node names when an element is deleted or added, the URI would fail if a network interface was added or removed. If such a case, a MAP node should be used, see *MAP Nodes* on page 333, as they allow the key to be managed by the remote manager.

LIST nodes can be mutable if the Meta Node of its index nodes support the Add or Delete action. A LIST node is modeled after a java.util.List that can automatically accomodate new elements. Get and Replace operations use the node name to index in this list.

To rearrange the list the local manager can Add and Delete nodes or rename them as it sees fit. At any moment in time the underlying implementation must maintain a list that runs from 0 to max(index) (inclusive), where index is the name of the LIST child nodes. Inserting a node requires renaming all subsequent nodes. Any missing indexes must automatically be provided by the plugin when the child node names are retrieved.

For example, a LIST node named L contains the following nodes:

```
L/0    A
L/1    B
L/2    C
```

To insert a node after B, L/2 must be renamed to L/3. This will automatically extend the LIST node to 4 elements. That is, even though L/2 is renamed, the implementation must automatically provide a new L/2 node. The value of this node depends on the underlying implementation. The value of the list will therefore then be: [A,B,?,C]. If node 1 is deleted, then the list will be [A,?,C]. If a node L/5 is added then the list will be [A,?,C,?,?,?]. It is usually easiest to use the LIST node as a complex value, this is discussed in the next section.

117.12.9.1 Complex Collections

An implementation of a LIST node must support a complex node value if its members are primitive; the interior node must then have a value of a Java object implementing the Collection interface from java.util. The elements in this map must be converted according to Table 117.11.

Table 117.11 *Conversion for Collections*

Format	Associated Java Type
FORMAT_STRING	String
FORMAT_BOOLEAN	Boolean
FORMAT_INTEGER	Integer
FORMAT_LONG	Long
FORMAT_FLOAT	Float
FORMAT_DATE_TIME	Date
FORMAT_BINARY	byte[]

Alternatively, the Collection may contain Dmt Data objects but the collection must be homogeneous. The collection must always be a copy and changes made to the collection must not affect the DMT.

For example, a LIST type for a list of URIs could look like:

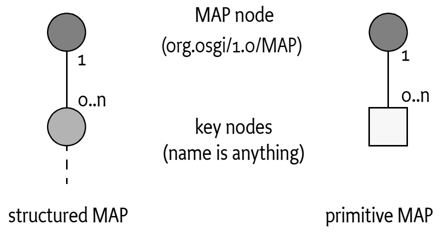
Name	Act	Type	Card.	S	Description
URIs	Get	LIST	1	P	A List of URIs
[index]	Get Set Add Del	string	0..n	D	A primitive index node

Replacing a complex value will generate a single EVENT_TOPIC_REPLACED event for the LIST node.

117.12.10 MAP Nodes

A MAP node represents a mapping from a *key* to a *value*. The key is the name of the node and the value is the node's value. A MAP node performs the same functions as a Java Map. See Figure 117.27.

Figure 117.27 MAP Nodes



A MAP node has *key nodes* as children. A key node is an association between the name of the key node (which is the key) and the value of the key node. Key nodes are depicted with [<type>], where the <type> indicates the type used for the string name. For example, a long type will have node names that can be converted to a long. A key type must always be one of the primitive types. For example, a list of Bundle locations can be handled with a MAP with [string] key nodes that have a value type of string. Since the key is used in URIs it must always be escaped, see *The DMT Addressing URI* on page 291.

For example:

Name	Act	Type	Card.	S	Description
Location	Get	MAP	1	P	A MAP of location where the index node is the Bundle Id.
[long]	Get Set Add Del	string	0..n	D	Name is the Bundle Id and the value is the location.

117.12.10.1 Complex Value

An implementation of a MAP node must support an interior node value if its child nodes are primitive; the interior node must then be associated with a Java object implementing the Map interface from java.util. The values in this Map must homogeneous and be converted according to Table 117.11 or the given values must of type DmtData. The Map object must a copy and does not track changes in the DMT or vice-versa.

Replacing a complex value will generate a single EVENT_TOPIC_REPLACED event for that node.

117.12.11 Instance Id

Some protocols cannot handle arbitrary names in the access URI, they need a well defined *instance id* to index in a table or put severe restrictions on the node name's character set, length, or other aspects. For example, TR-069 can access an object with the following URI:

```
Device.VOIP.12.Name
```

The more natural model for the DMT is to use:

```
Device.VOIP.<Name>...
```

To provide assistance to these protocols this section defines a mechanism that can be used by Protocol Adapters to simplify access.

An Object Model can define a child node `InstanceId`. The `InstanceId` node, if present, holds a long value that has the following qualities:

- Its value must be between 1 and `Long.MAX_VALUE`.
- No other index/key node on the same level must have the same value for the `InstanceId` node
- The value must be persistent between sessions and restarts of the plugin
- A value must not be reused when a node is deleted until the number space is exhausted

Protocol Adapters can use this information to provide alternative access paths for the DMT.

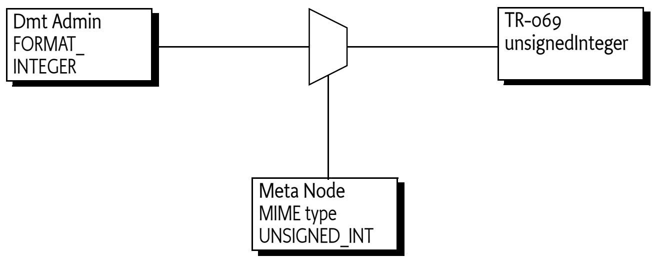
117.12.12 Conversions

Each Protocol Mapping document should define a default conversion from the Dmt Admin data formats to the protocol types and vice versa, including the `LIST` and `MAP` nodes. However, this default mapping is likely to be too constraining in real world models since different protocols support different data types and a 1:1 mapping is likely to be impossible.

For this reason, the Protocol Mapping document should define a number of protocol specific MIME types for each unique data type that they support. A Data Plugin can associate such a MIME type with a node. The Protocol Adapter can then look for this MIME type. If none of the Protocol Adapter specific MIME types are available in a node the default conversion is used.

For example, in the TR-069 Protocol Adapter specification there is a MIME type for each TR-069 data type. If for a given leaf node the Meta Node's type specifies `TR069_MIME_UNSIGNED_INT`and the node specifies the format `FORMAT_INTEGER` then the Protocol Adapter must convert the integer to an unsigned integer and encode the value as such in the response message. The Protocol Adapter there does not have to have specific knowledge of the object model, the Plugin drives the Protocol Adapter by providing the protocol specific MIME types on the leaf node Meta Nodes. This model is depicted in Figure 117.28.

Figure 117.28 Conversions



Since a Meta Node can contain multiple MIME types, there is no restrictions on the number of Protocol Adapters; a Plugin can specify the MIME types of multiple Protocol Adapters.

117.12.13 Extensions

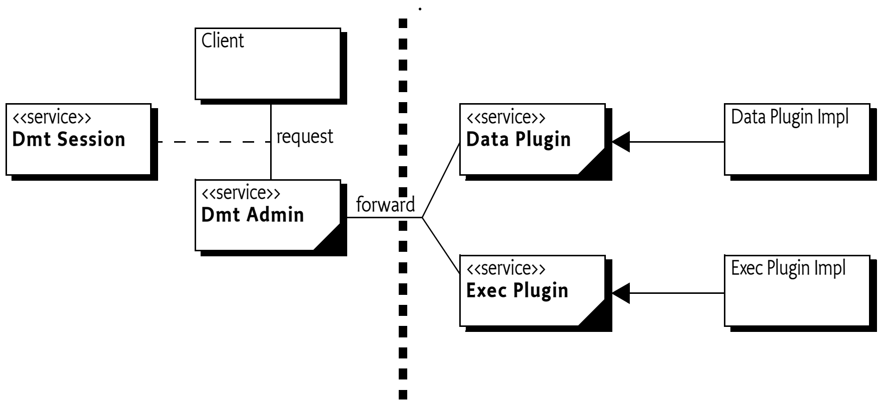
All interior nodes in this specification can have a node named Ext. These nodes are the *extension* nodes. If an implementation needs to expose additional details about an interior node then they should expose these extensions under the corresponding Ext node. To reduce name conflicts, it is recommended to group together implementation specific extensions under a unique name, recommended is to use the reverse domain name. For example, the following DDF defines an Ext node with extensions for the ACME provider.

Name	Act	Type	Card.	S	Description
Framework	Get	Framework	1	P	...
Ext	Get		1	P	Extension node
com.acme	Get	AcmeFrameworkExt	1	P	The node for the ACME extensions
Transactional	Get	boolean	1	P	...

117.13 Security

A key aspect of the Dmt Admin service model is the separation from DMT clients and plugins. The Dmt Admin service receives all the operation requests and, after verification of authority, forwards the requests to the plugins.

Figure 117.29 Separation of clients and plugins



This architecture makes it straightforward to use the OSGi security architecture to protect the different actors.

117.13.1 Principals

The caller of the getSession(String,String,int) method must have the Dmt Principal Permission with a target that matches the given principal. This Dmt Principal Permission is used to enforce that only trusted entities can act on behalf of remote managers.

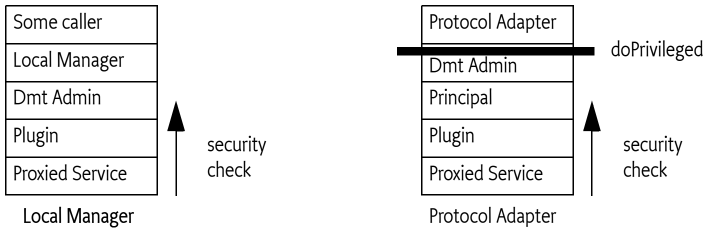
The Dmt Admin service must verify that all operations from a session with a principal can be executed on the given nodes using the available ACLs.

The other two forms of the getSession method are meant for local management applications where no principal is available. No special permission is defined to restrict the usage of these methods. The callers that want to execute device management commands, however, need to have the appropriate Dmt Permissions.

117.13.2 Operational Permissions

The operational security of a Local Manager and a remote manager is distinctly different. The distinction is made on the principal. Protocol Adapters should use the getSession method that takes an authenticated principal. Local Managers should not specify a principal.

Figure 117.30 Access control context, for Local Manager and Protocol Adapter operation



117.13.3 Protocol Adapters

A Protocol Adapter must provide a principal to the Dmt Admin service when it gets a session. It must use the getSession(String,String,int) method. The Protocol Adapter must have Dmt Principal Permission for the given principal. The Dmt Admin service must then use this principal to determine the *security scope* of the given principal. This security scope is a set of permissions. How these permissions are found is not defined in this specification; they are usually in the management tree of a device. For example, the Mobile Specification stores these under the $/Policy/Java/DmtPrincipalPermission sub-tree.

Additionally, a Dmt Session with a principal implies that the Dmt Admin service must verify the ACLs on the node for all operations.

Any operation that is requested by a Protocol Adapter must be executed in a doPrivileged block that takes the principal's security scope. The doPrivileged block effectively hides the permissions of the Protocol Adapter; all operations must be performed under the security scope of the principal.

The security check for a Protocol Adapter is therefore as follows:

- The operation method calls doPrivileged with the security scope of the principal.
- The operation is forwarded to the appropriate plugin. The underlying service must perform its normal security checks. For example, the Configuration Admin service must check for the appropriate Configuration Permission.

The Access Control context is shown in Figure 117.30 within the Protocol Adapter column.

This principal-based security model allows for minimal permissions on the Protocol Adapter, because the Dmt Admin service performs a doPrivileged on behalf of the principal, inserting the permissions for the principal on the call stack. This model does not guard against malicious Protocol Adapters, though the Protocol Adapter must have the appropriate Dmt Principal Permission.

The Protocol Adapter is responsible for the authentication of the principal. The Dmt Admin service must trust that the Protocol Adapter has correctly verified the identity of the other party. This specification does not address the type of authentication mechanisms that can be used. Once it has permission to use that principal, it can use any DMT command that is permitted for that principal at any time.

117.13.4 Local Manager

A Local Manager does not specify a principal. Security checks are therefore performed against the security scope of the Local Manager bundle, as shown in Figure 117.30 with the Local Manager stack. An operation is checked only with a Dmt Permission for the given node URI and operation. A thrown Security Exception must be passed unmodified to the caller of the operation method. The Dmt Admin service must not check the ACLs when no principal is set.

A Local Manager, and all its callers, must therefore have sufficient permission to handle the DMT operations as well as the permissions required by the plugins when they proxy other services (which is likely an extensive set of Permissions).

117.13.5 Plugin Security

Plugins are required to hold the maximum security scope for any services they proxy. For example, the plugin that manages the Configuration Admin service must have ConfigurationPermission("*", "*") to be effective.

Plugins should not make doPrivileged calls, but should use the caller's context on the stack for permission checks.

117.13.6 Events and Permissions

Dmt Event Listener services must have the appropriate Dmt Permission to receive the event since this must be verified with the hasPermission() method on Bundle.

The Dmt Event Listener services registered with a FILTER_PRINCIPAL service property requires Dmt Principal Permission for the given principal. In this case, the principal must have Get access to see the nodes for the event. Any nodes that the listener does not have access to must be removed from the event.

Plugins are not required to have access to the Event Admin service. If they send an event through the MountPointinterface then the Dmt Admin service must use a doPrivileged block to send the event to the Event Admin service.

117.13.7 Dmt Principal Permission

Execution of the getSession methods of the Dmt Admin service featuring an explicit principal name is guarded by the Dmt Principal Permission. This permission must be granted only to Protocol Adapters that open Dmt Sessions on behalf of remote management servers.

The DmtPrincipalPermission class does not have defined actions; it must always be created with a * to allow future extensions. The target is the principal name. A wildcard character is allowed at the end of the string to match a prefix.

Example:

```
new DmtPrincipalPermission("com.acme.dep*", "*" )
```

117.13.8 Dmt Permission

The Dmt Permission controls access to management objects in the DMT. It is intended to control only the *local* access to the DMT. The Dmt Permission target string identifies the target node's URI (absolute path is required, starting with the './' prefix) and the action field lists the management commands that are permitted on the node.

The URI can end in a wildcard character * to indicate it is a prefix that must be matched. This comparison is string based so that node boundaries can be ignored.

The following actions are defined:

- ADD
- DELETE

- EXEC
- GET
- REPLACE

For example, the following code creates a Dmt Permission for a bundle to add and replace nodes in any URI that starts with ./D.

```
new DmtPermission("./D*", "Add,Replace")
```

This permission must imply the following permission:

```
new DmtPermission("./Dev/Operator/Name", "Replace")
```

117.13.9 Alert Permission

The Alert Permission permits the holder of this permission to send a notification to a specific *target principal*. The target is identical to *Dmt Principal Permission* on page 337. No actions are defined for Alert Permission.

117.13.10 Security Summary

117.13.10.1 Dmt Admin Service and Notification Service

The Dmt Admin service is likely to require All Permission. This requirement is caused by the plugin model. Any permission required by any of the plugins must be granted to the Dmt Admin service. This set of permissions is large and hard to define. The following list shows the minimum permissions required if the plugin permissions are left out.

```
ServicePermission    ..DmtAdmin               REGISTER
ServicePermission    ..NotificationService    REGISTER
ServicePermission    ..DataPlugin             GET
ServicePermission    ..ExecPlugin             GET
ServicePermission    ..EventAdmin             GET
ServicePermission    ..RemoteAlertSender      GET
ServicePermission    ..DmtEventListener       GET
DmtPermission        *                        *
DmtPrincipal
  Permission         *                        *
PackagePermission    org.osgi.service.dmt     EXPORTONLY
PackagePermission    org.osgi.service.dmt.spi EXPORTONLY
PackagePermission    org.osgi.service.dmt.notificationEXPORTONLY
PackagePermission    org.osgi.service.dmt.notification.spiEXPORTONLY
PackagePermission    org.osgi.service.dmt.registryEXPORTONLY
PackagePermission    org.osgi.service.dmt.securityEXPORTONLY
```

117.13.10.2 Dmt Event Listener Service

```
ServicePermission    ..DmtEventListener       REGISTER
PackagePermission    org.osgi.service.dmt     IMPORT
```

Dmt Event Listeners must have the appropriate DmtPermission to see the nodes in the events. If they are registered with a principal then they also need DmtPrincipalPermission for the given principals.

117.13.10.3 Data and Exec Plugin

```
ServicePermission    ..NotificationService    GET
ServicePermission    ..DataPlugin             REGISTER
ServicePermission    ..ExecPlugin             REGISTER
PackagePermission    org.osgi.service.dmt     IMPORT
PackagePermission    org.osgi.service.dmt.notificationIMPORT
PackagePermission    org.osgi.service.dmt.spi IMPORT
PackagePermission     org.osgi.service.dmt.securityIMPORT
```

The plugin is also required to have any permissions to call its underlying services.

117.13.10.4 Local Manager

```
ServicePermission     ..DmtAdmin                GET
PackagePermission     org.osgi.service.dmt      IMPORT
PackagePermission     org.osgi.service.dmt.securityIMPORT
DmtPermission         <scope>                   ...
```

Additionally, the Local Manager requires all permissions that are needed by the plugins it addresses.

117.13.10.5 Protocol Adapter

The Protocol Adapter only requires Dmt Principal Permission for the instances that it is permitted to manage. The other permissions are taken from the security scope of the principal.

```
ServicePermission     ..DmtAdmin                GET
ServicePermission     ..RemoteAlertSender       REGISTER
PackagePermission     org.osgi.service.dmt      IMPORT
PackagePermission     org.osgi.service.dmt.notification.spiIMPORT
PackagePermission     org.osgi.service.dmt.notificationIMPORT
DmtPrincipalPermission<scope>
```

117.14 Changes

The changes to this document are quite large, even the package has been renamed. Despite the rename, package version 2.0 is taken to indicate that this is a major update. The following items provide a general overview of the changes. However, this update is major and no backward compatibility should be expected.

- Renamed the info.dmtree package to org.osgi.service.dmt.
- Removed the static factory methods needed to provide a service registry in non-OSGi environments.
- Overlapping sub-trees. It is now possible to create parent-child relations between plugins. API was added to manage the sharing, see *Sharing the DMT* on page 307.
- A format FORMAT_DATE_TIME was added to support actual time of day, see *Data Types* on page 300.
- A general assumption was added that the Dmt Admin nodes had not artificial limits on their URI length nor their segment length.
- Introduced so called scaffold nodes that provide intermediate nodes to allow discovery of plugins, *Scaffold Nodes* on page 305.
- Allow Plugins to send events about internal changes, see *Mount Points* on page 307.
- Changed the event model of atomic session to allow accurately track changes to the DMT, see *Notifications* on page 319.
- Introduced a section about modeling with the Dmt Admin service, see *OSGi Object Modeling* on page 327.
- Clarifications throughout the specification.

117.15 org.osgi.service.dmt

Device Management Tree Package Version 2.0.

This package contains the public API for the Device Management Tree manipulations. Permission classes are provided by the org.osgi.service.dmt.security package, and DMT plugin interfaces can be found in the org.osgi.service.dmt.spi package. Asynchronous notifications to remote management servers can be sent using the interfaces in the org.osgi.service.dmt.notification package.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.dmt; version="[2.0,3.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.dmt; version="[2.0,2.1)"
```

117.15.1 Summary

- Acl – Acl is an immutable class representing structured access to DMT ACLs.
- DmtAdmin – An interface providing methods to open sessions and register listeners.
- DmtConstants – Defines standard names for DmtAdmin.
- DmtData – An immutable data structure representing the contents of a leaf or interior node.
- DmtEvent – Event class storing the details of a change in the tree.
- DmtEventListener – Registered implementations of this class are notified via DmtEvent objects about important changes in the tree.
- DmtException – Checked exception received when a DMT operation fails.
- DmtIllegalStateException – Unchecked illegal state exception.
- DmtSession – DmtSession provides concurrent access to the DMT.
- MetaNode – The MetaNode contains meta data as standardized by OMA DM but extends it (without breaking the compatibility) to provide for better DMT data quality in an environment where many software components manipulate this data.
- Uri – This class contains static utility methods to manipulate DMT URIs.

117.15.2 Permissions

117.15.3 public final class Acl

Acl is an immutable class representing structured access to DMT ACLs. Under OMA DM the ACLs are defined as strings with an internal syntax.

The methods of this class taking a principal as parameter accept remote server IDs (as passed to DmtAdmin.getSession), as well as "* "indicating any principal.

The syntax for valid remote server IDs:
<*server-identifier*> ::= All printable characters except '=', '&', '*', '+' or white-space characters.

117.15.3.1 public static final int ADD = 2

Principals holding this permission can issue ADD commands on the node having this ACL.

117.15.3.2 public static final int ALL_PERMISSION = 31

Principals holding this permission can issue any command on the node having this ACL. This permission is the logical OR of ADD, DELETE, EXEC, GET and REPLACE permissions.

117.15.3.3 public static final int DELETE = 8

Principals holding this permission can issue DELETE commands on the node having this ACL.

117.15.3.4 public static final int EXEC = 16

Principals holding this permission can issue EXEC commands on the node having this ACL.

117.15.3.5 public static final int GET = 1

Principals holding this permission can issue GET command on the node having this ACL.

117.15.3.6 public static final int REPLACE = 4

Principals holding this permission can issue REPLACE commands on the node having this ACL.

117.15.3.7 public Acl (String acl)

acl The string representation of the ACL as defined in OMA DM. If null or empty then it represents an empty list of principals with no permissions.

□ Create an instance of the ACL from its canonical string representation.

Throws IllegalArgumentException – if acl is not a valid OMA DM ACL string

117.15.3.8 public Acl (String[] principals , int[] permissions)

principals The array of principals

permissions The array of permissions

□ Creates an instance with a specified list of principals and the permissions they hold. The two arrays run in parallel, that is principals[i] will hold permissions[i] in the ACL.

A principal name may not appear multiple times in the 'principals' argument. If the "*" principal appears in the array, the corresponding permissions will be granted to all principals (regardless of whether they appear in the array or not).

Throws IllegalArgumentException – if the length of the two arrays are not the same, if any array element is invalid, or if a principal appears multiple times in the principals array

117.15.3.9 public synchronized Acl addPermission (String principal , int permissions)

principal The entity to which permissions should be granted, or "*" to grant permissions to all principals.

permissions The permissions to be given. The parameter can be a logical or of more permission constants defined in this class.

□ Create a new Acl instance from this Acl with the given permission added for the given principal. The already existing permissions of the principal are not affected.

Returns a new Acl instance

Throws IllegalArgumentException – if principal is not a valid principal name or if permissions is not a valid combination of the permission constants defined in this class

117.15.3.10 public synchronized Acl deletePermission (String principal , int permissions)

principal The entity from which permissions should be revoked, or "*" to revoke permissions from all principals.

permissions The permissions to be revoked. The parameter can be a logical or of more permission constants defined in this class.

□ Create a new Acl instance from this Acl with the given permission revoked from the given principal. Other permissions of the principal are not affected.

Note, that it is not valid to revoke a permission from a specific principal if that permission is granted globally to all principals.

Returns a new Acl instance

Throws IllegalArgumentException – if principal is not a valid principal name, if permissions is not a valid combination of the permission constants defined in this class, or if a globally granted permission would have been revoked from a specific principal

117.15.3.11 public boolean equals (Object obj)

obj the object to compare with this Acl instance

□ Checks whether the given object is equal to this Acl instance. Two Acl instances are equal if they allow the same set of permissions for the same set of principals.

Returns true if the parameter represents the same ACL as this instance

117.15.3.12 public synchronized int getPermissions (String principal)

principal The entity whose permissions to query, or "*" to query the permissions that are granted globally, to all principals

□ Get the permissions associated to a given principal.

Returns The permissions of the given principal. The returned int is a bitmask of the permission constants defined in this class

Throws IllegalArgumentException – if principal is not a valid principal name

117.15.3.13 public String[] getPrincipals ( )

□ Get the list of principals who have any kind of permissions on this node. The list only includes those principals that have been explicitly assigned permissions (so "*" is never returned), globally set permissions naturally apply to all other principals as well.

Returns The array of principals having permissions on this node.

117.15.3.14 public int hashCode ( )

□ Returns the hash code for this ACL instance. If two Acl instances are equal according to the equals(Object) method, then calling this method on each of them must produce the same integer result.

Returns hash code for this ACL

117.15.3.15 public synchronized boolean isPermitted (String principal , int permissions)

principal The entity to check, or "*" to check whether the given permissions are granted to all principals globally

permissions The permissions to check

□ Check whether the given permissions are granted to a certain principal. The requested permissions are specified as a bitfield, for example (Acl.ADD | Acl.DELETE | Acl.GET).

Returns true if the principal holds all the given permissions

Throws IllegalArgumentException – if principal is not a valid principal name or if permissions is not a valid combination of the permission constants defined in this class

117.15.3.16 public synchronized Acl setPermission (String principal , int permissions)

principal The entity to which permissions should be granted, or "*" to globally grant permissions to all principals.

permissions The set of permissions to be given. The parameter is a bitmask of the permission constants defined in this class.

□ Create a new Acl instance from this Acl where all permissions for the given principal are overwritten with the given permissions.

Note, that when changing the permissions of a specific principal, it is not allowed to specify a set of permissions stricter than the global set of permissions (that apply to all principals).

Returns a new Acl instance

Throws IllegalArgumentException – if principal is not a valid principal name, if permissions is not a valid combination of the permission constants defined in this class, or if a globally granted permission would have been revoked from a specific principal

117.15.3.17 public synchronized String toString ( )

□ Give the canonical string representation of this ACL. The operations are in the following order: {Add, Delete, Exec, Get, Replace}, principal names are sorted alphabetically.

Returns The string representation as defined in OMA DM.

117.15.4 public interface DmtAdmin

An interface providing methods to open sessions and register listeners. The implementation of DmtAdmin should register itself in the OSGi service registry as a service. DmtAdmin is the entry point for applications to use the DMT API.

The getSession methods are used to open a session on a specified subtree of the DMT. A typical way of usage:

```
serviceRef = context.getServiceReference(DmtAdmin.class.getName());
DmtAdmin admin = (DmtAdmin) context.getService(serviceRef);
DmtSession session = admin.getSession(”./OSGi/Configuration”);
session.createInteriorNode(”./OSGi/Configuration/my.table”);
```

The methods for opening a session take a node URI (the session root) as a parameter. All segments of the given URI must be within the segment length limit of the implementation, and the special characters '/' and '\'must be escaped (preceded by a '\').

See the Uri.encode(String) method for support on escaping invalid characters in a URI.

It is possible to specify a lock mode when opening the session (see lock type constants in DmtSession). This determines whether the session can run in parallel with other sessions, and the kinds of operations that can be performed in the session. All Management Objects constituting the device management tree must support read operations on their nodes, while support for write operations depends on the Management Object. Management Objects supporting write access may support transactional write, non-transactional write or both. Users of DmtAdmin should consult the Management Object specification and implementation for the supported update modes. If Management Object definition permits, implementations are encouraged to support both update modes.

117.15.4.1 public DmtSession getSession (String subtreeUri) throws DmtException

subtreeUri the subtree on which DMT manipulations can be performed within the returned session

□ Opens a DmtSession for local usage on a given subtree of the DMT with non transactional write lock. This call is equivalent to the following: getSession(null, subtreeUri, DmtSession.LOCK_TYPE_EXCLUSIVE)

The subtreeUri parameter must contain an absolute URI. It can also be null, in this case the session is opened with the default session root, “.”, that gives access to the whole tree.

To perform this operation the caller must have DmtPermission for the subtreeUri node with the Get action present.

Returns a DmtSession object for the requested subtree

Throws DmtException – with the following possible error codes:
INVALID_URI if subtreeUri is syntactically invalid
URI_TOO_LONG if subtreeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if subtreeUri specifies a non-existing node
SESSION_CREATION_TIMEOUT if the operation timed out because of another ongoing session
COMMAND_FAILED if subtreeUri specifies a relative URI, or some unspecified error is encountered while attempting to complete the command

SecurityException – if the caller does not have DmtPermission for the given root node with the Get action present

117.15.4.2 public DmtSession getSession (String subtreeUri , int lockMode) throws DmtException

subtreeUri the subtree on which DMT manipulations can be performed within the returned session

lockMode one of the lock modes specified in `DmtSession`

□ Opens a `DmtSession` for local usage on a specific DMT subtree with a given lock mode. This call is equivalent to the following: `getSession(null, subtreeUri, lockMode)`

The `subtreeUri` parameter must contain an absolute URI. It can also be `null`, in this case the session is opened with the default session root, ".", that gives access to the whole tree.

To perform this operation the caller must have `DmtPermission` for the `subtreeUri` node with the Get action present.

Returns a `DmtSession` object for the requested subtree

Throws `DmtException` – with the following possible error codes:
INVALID_URI if subtreeUri is syntactically invalid
URI_TOO_LONG if subtreeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if subtreeUri specifies a non-existing node
FEATURE_NOT_SUPPORTED if atomic sessions are not supported by the implementation and lock-Mode requests an atomic session
SESSION_CREATION_TIMEOUT if the operation timed out because of another ongoing session
COMMAND_FAILED if subtreeUri specifies a relative URI, if lockMode is unknown, or some unspecified error is encountered while attempting to complete the command

`SecurityException` – if the caller does not have `DmtPermission` for the given root node with the Get action present

117.15.4.3 public DmtSession getSession (String principal , String subtreeUri , int lockMode) throws DmtException

principal the identifier of the remote server on whose behalf the data manipulation is performed, or `null` for local sessions

subtreeUri the subtree on which DMT manipulations can be performed within the returned session

lockMode one of the lock modes specified in `DmtSession`

□ Opens a `DmtSession` on a specific DMT subtree using a specific lock mode on behalf of a remote principal. If local management applications are using this method then they should provide `null` as the first parameter. Alternatively they can use other forms of this method without providing a principal string.

The `subtreeUri` parameter must contain an absolute URI. It can also be `null`, in this case the session is opened with the default session root, ".", that gives access to the whole tree.

This method is guarded by `DmtPrincipalPermission` in case of remote sessions. In addition, the caller must have Get access rights (ACL in case of remote sessions, `DmtPermission` in case of local sessions) on the `subtreeUri` node to perform this operation.

Returns a `DmtSession` object for the requested subtree

Throws `DmtException` – with the following possible error codes:
INVALID_URI if subtreeUri is syntactically invalid
URI_TOO_LONG if subtreeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if subtreeUri specifies a non-existing node
PERMISSION_DENIED if principal is not null and the ACL of the node does not allow the Get operation for the principal on the given root node
FEATURE_NOT_SUPPORTED if atomic sessions are not supported by the implementation and lock-Mode requests an atomic session

SESSION_CREATION_TIMEOUT if the operation timed out because of another ongoing session COMMAND_FAILED if subtreeUri specifies a relative URI, if lockMode is unknown, or some unspecified error is encountered while attempting to complete the command

SecurityException – in case of remote sessions, if the caller does not have the required DmtPrincipalPermission with a target matching the principal parameter, or in case of local sessions, if the caller does not have DmtPermission for the given root node with the Get action present

117.15.5 public class DmtConstants

Defines standard names for DmtAdmin.

Since 2.0

117.15.5.1 public static final String DDF_LIST = "org.osgi/1.0/LIST"

A string defining a DDF URI, indicating that the node is a LIST node.

117.15.5.2 public static final String DDF_MAP = "org.osgi/1.0/MAP"

A string defining a DDF URI, indicating that the node is a MAP node node.

117.15.5.3 public static final String DDF_SCAFFOLD = "org.osgi/1.0/SCAFFOLD"

A string defining a DDF URI, indicating that the node is a SCAFFOLD node.

117.15.5.4 public static final String EVENT_PROPERTY_NEW_NODES = "newnodes"

A string defining the property key for the newnodes property in node related events.

117.15.5.5 public static final String EVENT_PROPERTY_NODES = "nodes"

A string defining the property key for the @{code nodes} property in node related events.

117.15.5.6 public static final String EVENT_PROPERTY_SESSION_ID = "session.id"

A string defining the property key for the session.id property in node related events.

117.15.5.7 public static final String EVENT_TOPIC_ADDED = "org/osgi/service/dmt/DmtEvent/ADDED"

A string defining the topic for the event that is sent for added nodes.

117.15.5.8 public static final String EVENT_TOPIC_COPIED = "org/osgi/service/dmt/DmtEvent/COPIED"

A string defining the topic for the event that is sent for copied nodes.

117.15.5.9 public static final String EVENT_TOPIC_DELETED = "org/osgi/service/dmt/DmtEvent/DELETED"

A string defining the topic for the event that is sent for deleted nodes.

117.15.5.10 public static final String EVENT_TOPIC_RENAMED = "org/osgi/service/dmt/DmtEvent/RENAMED"

A string defining the topic for the event that is sent for renamed nodes.

117.15.5.11 public static final String EVENT_TOPIC_REPLACED = "org/osgi/service/dmt/DmtEvent/REPLACED"

A string defining the topic for the event that is sent for replaced nodes.

117.15.5.12 public static final String EVENT_TOPIC_SESSION_CLOSED = "org/osgi/service/dmt/DmtEvent/SESSION_CLOSED"

A string defining the topic for the event that is sent for a closed session.

117.15.5.13 public static final String EVENT_TOPIC_SESSION_OPENED = "org/osgi/service/dmt/DmtEvent/

SESSION_OPENED"

A string defining the topic for the event that is sent for a newly opened session.

117.15.6 public final class DmtData

An immutable data structure representing the contents of a leaf or interior node. This structure represents only the value and the format property of the node, all other properties (like MIME type) can be set and read using the DmtSession interface.

Different constructors are available to create nodes with different formats. Nodes of null format can be created using the static NULL_VALUE constant instance of this class.

FORMAT_RAW_BINARY and FORMAT_RAW_STRING enable the support of future data formats. When using these formats, the actual format name is specified as a String. The application is responsible for the proper encoding of the data according to the specified format.

Concurrency Immutable

117.15.6.1 public static final DmtData FALSE_VALUE

Constant instance representing a boolean false value.

Since 2.0

117.15.6.2 public static final int FORMAT_BASE64 = 128

The node holds an OMA DM b64 value. Like FORMAT_BINARY, this format is also represented by the Java byte[] type, the difference is only in the corresponding OMA DM format. This format does not affect the internal storage format of the data as byte[]. It is intended as a hint for the external representation of this data. Protocol Adapters can use this hint for their further processing.

117.15.6.3 public static final int FORMAT_BINARY = 64

The node holds an OMA DM bin value. The value of the node corresponds to the Java byte[] type.

117.15.6.4 public static final int FORMAT_BOOLEAN = 8

The node holds an OMA DM bool value.

117.15.6.5 public static final int FORMAT_DATE = 16

The node holds an OMA DM date value.

117.15.6.6 public static final int FORMAT_DATE_TIME = 16384

The node holds a Date object. If the getTime() equals zero then the date time is not known. If the getTime() is negative it must be interpreted as a relative number of milliseconds.

Since 2.0

117.15.6.7 public static final int FORMAT_FLOAT = 2

The node holds an OMA DM float value.

117.15.6.8 public static final int FORMAT_INTEGER = 1

The node holds an OMA DM int value.

117.15.6.9 public static final int FORMAT_LONG = 8192

The node holds a long value. The getFormatName() method can be used to get the actual format name.

Since 2.0

117.15.6.10 public static final int FORMAT_NODE = 1024

Format specifier of an internal node. An interior node can hold a Java object as value (see DmtData.DmtData(Object) and DmtData.getNode()). This value can be used by Java programs that know a specific URI understands the associated Java type. This type is further used as a return value of the MetaNode.getFormat() method for interior nodes.

117.15.6.11 public static final int FORMAT_NULL = 512

The node holds an OMA DM null value. This corresponds to the Java null type.

117.15.6.12 public static final int FORMAT_RAW_BINARY = 4096

The node holds raw protocol data encoded in binary format. The getFormatName() method can be used to get the actual format name.

117.15.6.13 public static final int FORMAT_RAW_STRING = 2048

The node holds raw protocol data encoded as String. The getFormatName() method can be used to get the actual format name.

117.15.6.14 public static final int FORMAT_STRING = 4

The node holds an OMA DM chr value.

117.15.6.15 public static final int FORMAT_TIME = 32

The node holds an OMA DM time value.

117.15.6.16 public static final int FORMAT_XML = 256

The node holds an OMA DM xml value.

117.15.6.17 public static final DmtData NULL_VALUE

Constant instance representing a leaf node of null format.

117.15.6.18 public static final DmtData TRUE_VALUE

Constant instance representing a boolean true value.

Since 2.0

117.15.6.19 public DmtData (String string)

string the string value to set

□ Create a DmtData instance of chr format with the given string value. The null string argument is valid.

117.15.6.20 public DmtData (Date date)

date the Date object to set

□ Create a DmtData instance of dateTime format with the given Date value. The given Date value must be a non-null Date object.

117.15.6.21 public DmtData (Object complex)

complex the complex data object to set

□ Create a DmtData instance of node format with the given object value. The value represents complex data associated with an interior node.

Certain interior nodes can support access to their subtrees through such complex values, making it simpler to retrieve or update all leaf nodes in a subtree.

The given value must be a non-null immutable object.

117.15.6.22 public DmtData (String value , int format)

value the string, XML, date, or time value to set

format the format of the DmtData instance to be created, must be one of the formats specified above

□ Create a DmtData instance of the specified format and set its value based on the given string. Only the following string-based formats can be created using this constructor:

- FORMAT_STRING - value can be any string
- FORMAT_XML - value must contain an XML fragment (the validity is not checked by this constructor)
- FORMAT_DATE - value must be parsable to an ISO 8601 calendar date in complete representation, basic format (pattern CCYYMMDD)
- FORMAT_TIME - value must be parsable to an ISO 8601 time of day in either local time, complete representation, basic format (pattern hhmmss) or Coordinated Universal Time, basic format (pattern hhmmssZ)

null string argument is only valid if the format is string or XML.

Throws IllegalArgumentException – if format is not one of the allowed formats, or value is not a valid string for the given format

NullPointerException – if a string, XML, date, or time is constructed and value is null

117.15.6.23 public DmtData (int integer)

integer the integer value to set

□ Create a DmtData instance of int format and set its value.

117.15.6.24 public DmtData (float flt)

flt the float value to set

□ Create a DmtData instance of float format and set its value.

117.15.6.25 public DmtData (long lng)

lng the long value to set

□ Create a DmtData instance of long format and set its value.

Since 2.0

117.15.6.26 public DmtData (boolean bool)

bool the boolean value to set

□ Create a DmtData instance of bool format and set its value.

117.15.6.27 public DmtData (byte[] bytes)

bytes the byte array to set, must not be null

□ Create a DmtData instance of bin format and set its value.

Throws NullPointerException – if bytes is null

117.15.6.28 public DmtData (byte[] bytes , boolean base64)

bytes the byte array to set, must not be null

base64 if true, the new instance will have b64 format, if false, it will have bin format

□ Create a DmtData instance of bin or b64 format and set its value. The chosen format is specified by the base64 parameter.

Throws NullPointerException – if bytes is null

117.15.6.29 public DmtData (byte[] bytes , int format)

bytes the byte array to set, must not be null

format the format of the DmtData instance to be created, must be one of the formats specified above

□ Create a DmtData instance of the specified format and set its value based on the given byte[]. Only the following byte[] based formats can be created using this constructor:

- FORMAT_BINARY
- FORMAT_BASE64

Throws IllegalArgumentException – if format is not one of the allowed formats

NullPointerException – if bytes is null

117.15.6.30 public DmtData (String formatName , String data)

formatName the name of the format, must not be null

data the data encoded according to the specified format, must not be null

□ Create a DmtData instance in FORMAT_RAW_STRING format. The data is provided encoded as a String. The actual data format is specified in formatName. The encoding used in data must conform to this format.

Throws NullPointerException – if formatName or data is null

117.15.6.31 public DmtData (String formatName , byte[] data)

formatName the name of the format, must not be null

data the data encoded according to the specified format, must not be null

□ Create a DmtData instance in FORMAT_RAW_BINARY format. The data is provided encoded as binary. The actual data format is specified in formatName. The encoding used in data must conform to this format.

Throws NullPointerException – if formatName or data is null

117.15.6.32 public boolean equals (Object obj)

obj the object to compare with this DmtData

□ Compares the specified object with this DmtData instance. Two DmtData objects are considered equal if their format is the same, and their data (selected by the format) is equal.

In case of FORMAT_RAW_BINARY and FORMAT_RAW_STRING the textual name of the data format - as returned by getFormatName() - must be equal as well.

Returns true if the argument represents the same DmtData as this object

117.15.6.33 public byte[] getBase64 ( )

□ Gets the value of a node with base 64 (b64) format.

Returns the binary value

Throws DmtIllegalStateException – if the format of the node is not base 64.

117.15.6.34 public byte[] getBinary ( )

□ Gets the value of a node with binary (bin) format.

Returns the binary value

Throws DmtIllegalStateException – if the format of the node is not binary

117.15.6.35 public boolean getBoolean ( )

- Gets the value of a node with boolean (bool) format.

Returns the boolean value

Throws DmtIllegalStateException – if the format of the node is not boolean

117.15.6.36 public String getDate ( )

- Gets the value of a node with date format. The returned date string is formatted according to the ISO 8601 definition of a calendar date in complete representation, basic format (pattern CCYYMMDD).

Returns the date value

Throws DmtIllegalStateException – if the format of the node is not date

117.15.6.37 public Date getDateTime ( )

- Gets the value of a node with dateTime format.

Returns the Date value

Throws DmtIllegalStateException – if the format of the node is not time

Since 2.0

117.15.6.38 public float getFloat ( )

- Gets the value of a node with float format.

Returns the float value

Throws DmtIllegalStateException – if the format of the node is not float

117.15.6.39 public int getFormat ( )

- Get the node's format, expressed in terms of type constants defined in this class. Note that the 'format' term is a legacy from OMA DM, it is more customary to think of this as 'type'.

Returns the format of the node

117.15.6.40 public String getFormatName ( )

- Returns the format of this DmtData as String. For the predefined data formats this is the OMA DM defined name of the format. For FORMAT_RAW_STRING and FORMAT_RAW_BINARY this is the format specified when the object was created.

Returns the format name as String

117.15.6.41 public int getInt ( )

- Gets the value of a node with integer (int) format.

Returns the integer value

Throws DmtIllegalStateException – if the format of the node is not integer

117.15.6.42 public long getLong ( )

- Gets the value of a node with long format.

Returns the long value

Throws DmtIllegalStateException – if the format of the node is not long

Since 2.0

117.15.6.43 public Object getNode ( )

- Gets the complex data associated with an interior node (node format).

Certain interior nodes can support access to their subtrees through complex values, making it simpler to retrieve or update all leaf nodes in the subtree.

Returns the data object associated with an interior node

Throws `DmtIllegalStateException` – if the format of the data is not `node`

117.15.6.44 public byte[] getRawBinary ( )

☐ Gets the value of a node in raw binary (`FORMAT_RAW_BINARY`) format.

Returns the data value in raw binary format

Throws `DmtIllegalStateException` – if the format of the node is not raw binary

117.15.6.45 public String getRawString ( )

☐ Gets the value of a node in raw `String` (`FORMAT_RAW_STRING`) format.

Returns the data value in raw `String` format

Throws `DmtIllegalStateException` – if the format of the node is not raw `String`

117.15.6.46 public int getSize ( )

☐ Get the size of the data. The returned value depends on the format of data in the node:

- `FORMAT_STRING`, `FORMAT_XML`, `FORMAT_BINARY`, `FORMAT_BASE64`, `FORMAT_RAW_STRING`, and `FORMAT_RAW_BINARY`: the length of the stored data, or 0 if the data is `null`
- `FORMAT_INTEGER` and `FORMAT_FLOAT`: 4
- `FORMAT_LONG` and `FORMAT_DATE_TIME`: 8
- `FORMAT_DATE` and `FORMAT_TIME`: the length of the date or time in its string representation
- `FORMAT_BOOLEAN`: 1
- `FORMAT_NODE`: -1 (unknown)
- `FORMAT_NULL`: 0

Returns the size of the data stored by this object

117.15.6.47 public String getString ( )

☐ Gets the value of a node with string (`chr`) format.

Returns the string value

Throws `DmtIllegalStateException` – if the format of the node is not string

117.15.6.48 public String getTime ( )

☐ Gets the value of a node with time format. The returned time string is formatted according to the ISO 8601 definition of the time of day. The exact format depends on the value the object was initialized with: either local time, complete representation, basic format (pattern `hhmmss`) or Coordinated Universal Time, basic format (pattern `hhmmssZ`).

Returns the time value

Throws `DmtIllegalStateException` – if the format of the node is not time

117.15.6.49 public String getXml ( )

☐ Gets the value of a node with `xml` format.

Returns the XML value

Throws `DmtIllegalStateException` – if the format of the node is not `xml`

117.15.6.50 public int hashCode ( )

☐ Returns the hash code value for this `DmtData` instance. The hash code is calculated based on the data (selected by the format) of this object.

Returns the hash code value for this object

117.15.6.51 public String toString ( )

- Gets the string representation of the DmtData. This method works for all formats.

  For string format data - including FORMAT_RAW_STRING - the string value itself is returned, while for XML, date, time, integer, float, boolean, long and node formats the string form of the value is returned. Binary - including FORMAT_RAW_BINARY - base64 data is represented by two-digit hexadecimal numbers for each byte separated by spaces. The NULL_VALUE data has the string form of "null". Data of string or XML format containing the Java null value is represented by an empty string. DateTime data is formatted as yyyy-MM-dd'T'HH:mm:SS'Z').

Returns the string representation of this DmtData instance

117.15.7 public interface DmtEvent

Event class storing the details of a change in the tree. DmtEvent is used by DmtAdmin to notify registered EventListeners services about important changes. Events are generated after every successful DMT change, and also when sessions are opened or closed. If a DmtSession is opened in atomic mode, DMT events are only sent when the session is committed, when the changes are actually performed.

The type of the event describes the change that triggered the event delivery. Each event carries the unique identifier of the session in which the described change happened or -1 when the change originated outside a session. The events describing changes in the DMT carry the list of affected nodes. In case of COPIED or RENAMED events, the event carries the list of new nodes as well.

117.15.7.1 public static final int ADDED = 1

Event type indicating nodes that were added.

117.15.7.2 public static final int COPIED = 2

Event type indicating nodes that were copied.

117.15.7.3 public static final int DELETED = 4

Event type indicating nodes that were deleted.

117.15.7.4 public static final int RENAMED = 8

Event type indicating nodes that were renamed.

117.15.7.5 public static final int REPLACED = 16

Event type indicating nodes that were replaced.

117.15.7.6 public static final int SESSION_CLOSED = 64

Event type indicating that a session was closed. This type of event is sent when the session is closed by the client or becomes inactive for any other reason (session timeout, fatal errors in business methods, etc.).

117.15.7.7 public static final int SESSION_OPENED = 32

Event type indicating that a new session was opened.

117.15.7.8 public String[] getNewNodes ( )

- This method can be used to query the new nodes, when the type of the event is COPIED or RENAMED. For all other event types this method returns null.

The array returned by this method runs parallel to the array returned by getNodes(), the elements in the two arrays contain the source and destination URIs for the renamed or copied nodes in the same order. All returned URIs are absolute.

This method returns only those nodes where the caller has the GET permission for the source or destination node of the operation. Therefore, it is possible that the method returns an empty array.

Returns the array of newly created nodes

117.15.7.9 public String[] getNodes ( )

□ This method can be used to query the subject nodes of this event. The method returns null for SESSION_OPENED and SESSION_CLOSED.

The method returns only those affected nodes that the caller has the GET permission for (or in case of COPIED or RENAMED events, where the caller has GET permissions for either the source or the destination nodes). Therefore, it is possible that the method returns an empty array. All returned URIs are absolute.

Returns the array of affected nodes

See Also getNewNodes()

117.15.7.10 public Object getProperty (String key)

key the name of the requested property

□ This method can be used to get the value of a single event property.

Returns the requested property value or null, if the key is not contained in the properties

See Also getPropertyNames()

Since 2.0

117.15.7.11 public String[] getPropertyNames ( )

□ This method can be used to query the names of all properties of this event.

The returned names can be used as key value in subsequent calls to getProperty(String).

Returns the array of property names

See Also getProperty(String)

Since 2.0

117.15.7.12 public int getSessionId ( )

□ This method returns the identifier of the session in which this event took place. The ID is guaranteed to be unique on a machine.

For events that do not result from a session, the session id is -1.

The availability of a session.id can also be check by using getProperty() with “session.id” as key.

Returns the unique identifier of the session that triggered the event or -1 if there is no session associated

117.15.7.13 public int getType ( )

□ This method returns the type of this event.

Returns the type of this event.

117.15.8 public interface DmtEventListener

Registered implementations of this class are notified via DmtEvent objects about important changes in the tree. Events are generated after every successful DMT change, and also when sessions are opened or closed. If a DmtSession is opened in atomic mode, DMT events are only sent when the session is committed, when the changes are actually performed.

Dmt Event Listener services must have permission DmtPermission.GET for the nodes in the nodes and newNodes property in the Dmt Event.

117.15.8.1 public static final String FILTER_EVENT = "osgi.filter.event"

A number of event types packed in a bitmap. If this service property is provided with a Dmt Event Listener service registration than that listener must only receive events where one of the Dmt Event types occur in the bitmap. The type of this service property must be Integer.

117.15.8.2 public static final String FILTER_PRINCIPAL = "osgi.filter.principal"

A number of names of principals. If this service property is provided with a Dmt Event Listener service registration than that listener must only receive events for which at least one of the given principals has Get rights. The type of this service property is String+.

117.15.8.3 public static final String FILTER_SUBTREE = "osgi.filter.subtree"

A number of sub-tree top nodes that define the scope of the Dmt Event Listener. If this service property is registered then the service must only receive events for nodes that are part of one of the subtrees. The type of this service property is String+.

117.15.8.4 public void changeOccurred (DmtEvent event)

event the DmtEvent describing the change in detail

□ DmtAdmin uses this method to notify the registered listeners about the change. This method is called asynchronously from the actual event occurrence.

117.15.9 public class DmtException extends Exception

Checked exception received when a DMT operation fails. Beside the exception message, a DmtException always contains an error code (one of the constants specified in this class), and may optionally contain the URI of the related node, and information about the cause of the exception.

Some of the error codes defined in this class have a corresponding error code defined in OMA DM, in these cases the name and numerical value from OMA DM is used. Error codes without counterparts in OMA DM were given numbers from a different range, starting from 1.

The cause of the exception (if specified) can either be a single Throwable instance, or a list of such instances if several problems occurred during the execution of a method. An example for the latter is the close method of DmtSession that tries to close multiple plugins, and has to report the exceptions of all failures.

Each constructor has two variants, one accepts a String node URI, the other accepts a String[] node path. The former is used by the DmtAdmin implementation, the latter by the plugins, who receive the node URI as an array of segment names. The constructors are otherwise identical.

Getter methods are provided to retrieve the values of the additional parameters, and the printStackTrace(PrintWriter) method is extended to print the stack trace of all causing throwables as well.

117.15.9.1 public static final int ALERT_NOT_ROUTED = 5

An alert can not be sent from the device to the given principal. This can happen if there is no Remote Alert Sender willing to forward the alert to the given principal, or if no principal was given and the DmtAdmin did not find an appropriate default destination.

This error code does not correspond to any OMA DM response status code. It should be translated to the code 500 "Command Failed" when transferring over OMA DM.

117.15.9.2 public static final int COMMAND_FAILED = 500

The recipient encountered an error which prevented it from fulfilling the request.

This error code is only used in situations not covered by any of the other error codes that a method may use. Some methods specify more specific error situations for this code, but it can generally be used for any unexpected condition that causes the command to fail.

This error code corresponds to the OMA DM response status code 500 “Command Failed”.

117.15.9.3 public static final int COMMAND_NOT_ALLOWED = 405

The requested command is not allowed on the target node. This includes the following situations:

- an interior node operation is requested for a leaf node, or vice versa (e.g. trying to retrieve the children of a leaf node)
- an attempt is made to create a node where the parent is a leaf node
- an attempt is made to rename or delete the root node of the tree
- an attempt is made to rename or delete the root node of the session
- a write operation (other than setting the ACL) is performed in a non-atomic write session on a node provided by a plugin that is read-only or does not support non-atomic writing
- a node is copied to its descendant
- the ACL of the root node is changed not to include Add rights for all principals

This error code corresponds to the OMA DM response status code 405 “Command not allowed”.

117.15.9.4 public static final int CONCURRENT_ACCESS = 4

An error occurred related to concurrent access of nodes. This can happen for example if a configuration node was deleted directly through the Configuration Admin service, while the node was manipulated via the tree.

This error code does not correspond to any OMA DM response status code. It should be translated to the code 500 “Command Failed” when transferring over OMA DM.

117.15.9.5 public static final int DATA_STORE_FAILURE = 510

An error related to the recipient data store occurred while processing the request. This error code may be thrown by any of the methods accessing the tree, but whether it is really used depends on the implementation, and the data store it uses.

This error code corresponds to the OMA DM response status code 510 “Data store failure”.

117.15.9.6 public static final int FEATURE_NOT_SUPPORTED = 406

The requested command failed because an optional feature required by the command is not supported. For example, opening an atomic session might return this error code if the DmtAdmin implementation does not support transactions. Similarly, accessing the optional node properties (Title, Timestamp, Version, Size) might not succeed if either the DmtAdmin implementation or the underlying plugin does not support the property.

When getting or setting values for interior nodes (an optional optimization feature), a plugin can use this error code to indicate that the given interior node does not support values.

This error code corresponds to the OMA DM response status code 406 “Optional feature not supported”.

117.15.9.7 public static final int INVALID_URI = 3

The requested command failed because the target URI or node name is null or syntactically invalid. This covers the following cases:

- the URI or node name ends with the ‘\’or ‘/’ character

- the URI is an empty string (only invalid if the method does not accept relative URIs)
- the URI contains the segment "." at a position other than the beginning of the URI
- the node name is ".." or the URI contains such a segment
- the node name contains an unescaped '/' character

See the `Uri.encode(String)` method for support on escaping invalid characters in a URI.

This code is only used if the URI or node name does not match any of the criteria for `URI_TOO_LONG`. This error code does not correspond to any OMA DM response status code. It should be translated to the code 404 "Not Found" when transferring over OMA DM.

117.15.9.8 public static final int LIMIT_EXCEEDED = 413

The requested operation failed because a specific limit was exceeded, e.g. if a requested resource exceeds a size limit.

This error code corresponds to the OMA DM response status code 413 "Request entity too large".

Since 2.0

117.15.9.9 public static final int METADATA_MISMATCH = 2

Operation failed because of meta data restrictions. This covers any attempted deviation from the parameters defined by the `MetaNode` objects of the affected nodes, for example in the following situations:

- creating, deleting or renaming a permanent node, or modifying its type
- creating an interior node where the meta-node defines it as a leaf, or vice versa
- any operation on a node which does not have the required access type (e.g. executing a node that lacks the `MetaNode.CMD_EXECUTE` access type)
- any node creation or deletion that would violate the cardinality constraints
- any leaf node value setting that would violate the allowed formats, values, mime types, etc.
- any node creation that would violate the allowed node names

This error code can also be used to indicate any other meta data violation, even if it cannot be described by the `MetaNode` class. For example, detecting a multi-node constraint violation while committing an atomic session should result in this error.

This error code does not correspond to any OMA DM response status code. It should be translated to the code 405 "Command not allowed" when transferring over OMA DM.

117.15.9.10 public static final int NODE_ALREADY_EXISTS = 418

The requested node creation operation failed because the target already exists. This can occur if the node is created directly (with one of the `create...` methods), or indirectly (during a `copy` operation).

This error code corresponds to the OMA DM response status code 418 "Already exists".

117.15.9.11 public static final int NODE_NOT_FOUND = 404

The requested target node was not found. No indication is given as to whether this is a temporary or permanent condition, unless otherwise noted.

This is only used when the requested node name is valid, otherwise the more specific error codes `URI_TOO_LONG` or `INVALID_URI` are used. This error code corresponds to the OMA DM response status code 404 "Not Found".

117.15.9.12 public static final int PERMISSION_DENIED = 425

The requested command failed because the principal associated with the session does not have adequate access control permissions (ACL) on the target. This can only appear in case of remote sessions, i.e. if the session is associated with an authenticated principal.

This error code corresponds to the OMA DM response status code 425 "Permission denied".

117.15.9.13 public static final int REMOTE_ERROR = 1

A device initiated remote operation failed. This is used when the protocol adapter fails to send an alert for any reason.

Alert routing errors (that occur while looking for the proper protocol adapter to use) are indicated by ALERT_NOT_ROUTED, this code is only for errors encountered while sending the routed alert. This error code does not correspond to any OMA DM response status code. It should be translated to the code 500 “Command Failed” when transferring over OMA DM.

117.15.9.14 public static final int ROLLBACK_FAILED = 516

The rollback command was not completed successfully. The tree might be in an inconsistent state after this error.

This error code corresponds to the OMA DM response status code 516 “Atomic roll back failed”.

117.15.9.15 public static final int SESSION_CREATION_TIMEOUT = 7

Creation of a session timed out because of another ongoing session. The length of time while the DmtAdmin waits for the blocking session(s) to finish is implementation dependent.

This error code does not correspond to any OMA DM response status code. OMA has several status codes related to timeout, but these are meant to be used when a request times out, not if a session can not be established. This error code should be translated to the code 500 “Command Failed” when transferring over OMA DM.

117.15.9.16 public static final int TRANSACTION_ERROR = 6

A transaction-related error occurred in an atomic session. This error is caused by one of the following situations:

- an updating method within an atomic session can not be executed because the underlying plugin is read-only or does not support atomic writing
- a commit operation at the end of an atomic session failed because one of the underlying plugins failed to close

The latter case may leave the tree in an inconsistent state due to the lack of a two-phase commit system, see DmtSession.commit() for details.

This error code does not correspond to any OMA DM response status code. It should be translated to the code 500 “Command Failed” when transferring over OMA DM.

117.15.9.17 public static final int UNAUTHORIZED = 401

The originator’s authentication credentials specify a principal with insufficient rights to complete the command.

This status code is used as response to device originated sessions if the remote management server cannot authorize the device to perform the requested operation.

This error code corresponds to the OMA DM response status code 401 “Unauthorized”.

117.15.9.18 public static final int URI_TOO_LONG = 414

The requested command failed because the target URI is too long for what the recipient is able or willing to process.

This error code corresponds to the OMA DM response status code 414 “URI too long”.

See Also OSGi Service Platform, Mobile Specification Release 4

117.15.9.19 public DmtException (String uri , int code , String message)

uri	the node on which the failed DMT operation was issued, or null if the operation is not associated with a node
code	the error code of the failure
message	the message associated with the exception, or null if there is no error message

□ Create an instance of the exception. The uri and message parameters are optional. No originating exception is specified.

117.15.9.20 public DmtException (String uri , int code , String message , Throwable cause)

uri	the node on which the failed DMT operation was issued, or null if the operation is not associated with a node
code	the error code of the failure
message	the message associated with the exception, or null if there is no error message
cause	the originating exception, or null if there is no originating exception

□ Create an instance of the exception, specifying the cause exception. The uri, message and cause parameters are optional.

117.15.9.21 public DmtException (String uri , int code , String message , Vector causes , boolean fatal)

uri	the node on which the failed DMT operation was issued, or null if the operation is not associated with a node
code	the error code of the failure
message	the message associated with the exception, or null if there is no error message
causes	the list of originating exceptions, or empty list or null if there are no originating exceptions
fatal	whether the exception is fatal

□ Create an instance of the exception, specifying the list of cause exceptions and whether the exception is a fatal one. This constructor is meant to be used by plugins wishing to indicate that a serious error occurred which should invalidate the ongoing atomic session. The uri, message and causes parameters are optional.

If a fatal exception is thrown, no further business methods will be called on the originator plugin. In case of atomic sessions, all other open plugins will be rolled back automatically, except if the fatal exception was thrown during commit.

117.15.9.22 public DmtException (String[] path , int code , String message)

path	the path of the node on which the failed DMT operation was issued, or null if the operation is not associated with a node
code	the error code of the failure
message	the message associated with the exception, or null if there is no error message

□ Create an instance of the exception, specifying the target node as an array of path segments. This method behaves in exactly the same way as if the path was given as a URI string.

See Also `DmtException(String, int, String)`

117.15.9.23 public DmtException (String[] path , int code , String message , Throwable cause)

path	the path of the node on which the failed DMT operation was issued, or null if the operation is not associated with a node
code	the error code of the failure

message the message associated with the exception, or `null` if there is no error message

cause the originating exception, or `null` if there is no originating exception

□ Create an instance of the exception, specifying the target node as an array of path segments, and specifying the cause exception. This method behaves in exactly the same way as if the path was given as a URI string.

See Also `DmtException(String, int, String, Throwable)`

117.15.9.24 public DmtException (String[] path , int code , String message , Vector causes , boolean fatal)

path the path of the node on which the failed DMT operation was issued, or `null` if the operation is not associated with a node

code the error code of the failure

message the message associated with the exception, or `null` if there is no error message

causes the list of originating exceptions, or empty list or `null` if there are no originating exceptions

fatal whether the exception is fatal

□ Create an instance of the exception, specifying the target node as an array of path segments, the list of cause exceptions, and whether the exception is a fatal one. This method behaves in exactly the same way as if the path was given as a URI string.

See Also `DmtException(String, int, String, Vector, boolean)`

117.15.9.25 public Throwable getCause ( )

□ Get the cause of this exception. Returns non-`null`, if this exception is caused by one or more other exceptions (like a `NullPointerException` in a DmtPlugin). If there are more than one cause exceptions, the first one is returned.

Returns the cause of this exception, or `null` if no cause was given

117.15.9.26 public Throwable[] getCauses ( )

□ Get all causes of this exception. Returns the causing exceptions in an array. If no cause was specified, an empty array is returned.

Returns the list of causes of this exception

117.15.9.27 public int getCode ( )

□ Get the error code associated with this exception. Most of the error codes within this exception correspond to OMA DM error codes.

Returns the error code

117.15.9.28 public String getMessage ( )

□ Get the message associated with this exception. The returned string also contains the associated URI (if any) and the exception code. The resulting message has the following format (parts in square brackets are only included if the field inside them is not `null`):

```
<exception_code>[: ‘<uri>‘][: <error_message>]
```

Returns the error message in the format described above

117.15.9.29 public String getURI ( )

□ Get the node on which the failed DMT operation was issued. Some operations like `DmtSession.close()` don’t require an URI, in this case this method returns `null`.

Returns the URI of the node, or `null`

117.15.9.30 public boolean isFatal ( )

- Check whether this exception is marked as fatal in the session. Fatal exceptions trigger an automatic rollback of atomic sessions.

Returns whether the exception is marked as fatal

117.15.9.31 public void printStackTrace (PrintStream s)

s PrintStream to use for output

- Prints the exception and its backtrace to the specified print stream. Any causes that were specified for this exception are also printed, together with their backtraces.

117.15.10 public class DmtIllegalStateException extends RuntimeException

Unchecked illegal state exception. This class is used in DMT because java.lang.IllegalStateException does not exist in CLDC.

117.15.10.1 public DmtIllegalStateException ( )

- Create an instance of the exception with no message.

117.15.10.2 public DmtIllegalStateException (String message)

message the reason for the exception

- Create an instance of the exception with the specified message.

117.15.10.3 public DmtIllegalStateException (Throwable cause)

cause the cause of the exception

- Create an instance of the exception with the specified cause exception and no message.

117.15.10.4 public DmtIllegalStateException (String message , Throwable cause)

message the reason for the exception

cause the cause of the exception

- Create an instance of the exception with the specified message and cause exception.

117.15.11 public interface DmtSession

DmtSession provides concurrent access to the DMT. All DMT manipulation commands for management applications are available on the DmtSession interface. The session is associated with a root node which limits the subtree in which the operations can be executed within this session.

Most of the operations take a node URI as parameter, which can be either an absolute URI (starting with "./") or a URI relative to the root node of the session. The empty string as relative URI means the root URI the session was opened with. All segments of a URI must be within the segment length limit of the implementation, and the special characters '/' and '\'must be escaped (preceded by a '\').

See the Uri.encode(String) method for support on escaping invalid characters in a URI.

If the URI specified does not correspond to a legitimate node in the tree an exception is thrown. The only exception is the isNodeUri(String) method which returns false in case of an invalid URI.

Each method of DmtSession that accesses the tree in any way can throw DmtIllegalStateException if the session has been closed or invalidated (due to timeout, fatal exceptions, or unexpectedly unregistered plugins).

117.15.11.1 public static final int LOCK_TYPE_ATOMIC = 2

LOCK_TYPE_ATOMIC is an exclusive lock with transactional functionality. Commands of an atomic session will either fail or succeed together, if a single command fails then the whole session will be rolled back.

117.15.11.2 public static final int LOCK_TYPE_EXCLUSIVE = 1

LOCK_TYPE_EXCLUSIVE lock guarantees full access to the tree, but can not be shared with any other locks.

117.15.11.3 public static final int LOCK_TYPE_SHARED = 0

Sessions created with LOCK_TYPE_SHARED lock allows read-only access to the tree, but can be shared between multiple readers.

117.15.11.4 public static final int STATE_CLOSED = 1

The session is closed, DMT manipulation operations are not available, they throw DmtIllegalStateException if tried.

117.15.11.5 public static final int STATE_INVALID = 2

The session is invalid because a fatal error happened. Fatal errors include the timeout of the session, any DmtException with the 'fatal' flag set, or the case when a plugin service is unregistered while in use by the session. DMT manipulation operations are not available, they throw DmtIllegalStateException if tried.

117.15.11.6 public static final int STATE_OPEN = 0

The session is open, all session operations are available.

117.15.11.7 public void close ( ) throws DmtException

□ Closes a session. If the session was opened with atomic lock mode, the DmtSession must first persist the changes made to the DMT by calling commit() on all (transactional) plugins participating in the session. See the documentation of the commit() method for details and possible errors during this operation.

The state of the session changes to DmtSession.STATE_CLOSED if the close operation completed successfully, otherwise it becomes DmtSession.STATE_INVALID.

Throws DmtException – with the following possible error codes:
METADATA_MISMATCH in case of atomic sessions, if the commit operation failed because of meta-data restrictions
CONCURRENT_ACCESS in case of atomic sessions, if the commit operation failed because of some modification outside the scope of the DMT to the nodes affected in the session
TRANSACTION_ERROR in case of atomic sessions, if an underlying plugin failed to commit
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if an underlying plugin failed to close, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException – if the session is already closed or invalidated

SecurityException – if the caller does not have the necessary permissions to execute the underlying management operation

117.15.11.8 public void commit ( ) throws DmtException

□ Commits a series of DMT operations issued in the current atomic session since the last transaction boundary. Transaction boundaries are the creation of this object that starts the session, and all subsequent commit() and rollback() calls.

This method can fail even if all operations were successful. This can happen due to some multi-node semantic constraints defined by a specific implementation. For example, node A can be required to always have children A/B, A/C and A/D. If this condition is broken when `commit()` is executed, the method will fail, and throw a `METADATA_MISMATCH` exception.

An error situation can arise due to the lack of a two phase commit mechanism in the underlying plugins. As an example, if plugin A has committed successfully but plugin B failed, the whole session must fail, but there is no way to undo the commit performed by A. To provide predictable behaviour, the commit operation should continue with the remaining plugins even after detecting a failure. All exceptions received from failed commits are aggregated into one `TRANSACTION_ERROR` exception thrown by this method.

In many cases the tree is not the only way to manage a given part of the system. It may happen that while modifying some nodes in an atomic session, the underlying settings are modified in parallel outside the scope of the DMT. If this is detected during commit, an exception with the code `CONCURRENT_ACCESS` is thrown.

Throws `DmtException` – with the following possible error codes:
METADATA_MISMATCH if the operation failed because of meta-data restrictions
CONCURRENT_ACCESS if it is detected that some modification has been made outside the scope of the DMT to the nodes affected in the session's operations
TRANSACTION_ERROR if an error occurred during the commit of any of the underlying plugins
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session was not opened using the `LOCK_TYPE_ATOMIC` lock type, or if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

117.15.11.9 public void copy (String nodeUri , String newNodeUri , boolean recursive) throws DmtException

nodeUri the node or root of a subtree to be copied

newNodeUri the URI of the new node or root of a subtree

recursive `false` if only a single node is copied, `true` if the whole subtree is copied

□ Create a copy of a node or a whole subtree. Beside the structure and values of the nodes, most properties are also copied, with the exception of the ACL (Access Control List), Timestamp and Version properties.

The copy method is essentially a convenience method that could be substituted with a sequence of retrieval and update operations. This determines the permissions required for copying. However, some optimization can be possible if the source and target nodes are all handled by DmtAdmin or by the same plugin. In this case, the handler might be able to perform the underlying management operation more efficiently: for example, a configuration table can be copied at once instead of reading each node for each entry and creating it in the new tree.

This method may result in any of the errors possible for the contributing operations. Most of these are collected in the exception descriptions below, but for the full list also consult the documentation of `getChildNodeNames(String)`, `isLeafNode(String)`, `getNodeValue(String)`, `getNodeType(String)`, `getNodeTitle(String)`, `setNodeTitle(String, String)`, `createLeafNode(String, DmtData, String)` and `createInteriorNode(String, String)`.

Throws `DmtException` – with the following possible error codes:
INVALID_URI if nodeUri or newNodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a non-existing node, or if newNodeUri points to a node that

cannot exist in the tree according to the meta-data (see getMetaNode(String))
NODE_ALREADY_EXISTS if newNodeUri points to a node that already exists
PERMISSION_DENIED if the session is associated with a principal and the ACL of the copied node(s) does not allow the Get operation, or the ACL of the parent of the target node does not allow the Add operation for the associated principal
COMMAND_NOT_ALLOWED if nodeUri is an ancestor of newNodeUri, or if any of the implied retrieval or update operations are not allowed
METADATA_MISMATCH if any of the meta-data constraints of the implied retrieval or update operations are violated
TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if either URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session was opened using the `LOCK_TYPE_SHARED` lock type, or if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have `DmtPermission` for the copied node(s) with the Get action present, or for the parent of the target node with the Add action

117.15.11.10 public void createInteriorNode (String nodeUri) throws DmtException

nodeUri the URI of the node to create

□ Create an interior node. If the parent node does not exist, it is created automatically, as if this method were called for the parent URI. This way all missing ancestor nodes leading to the specified node are created. Any exceptions encountered while creating the ancestors are propagated to the caller of this method, these are not explicitly listed in the error descriptions below.

If meta-data is available for the node, several checks are made before creating it. The node must have `MetaNode.CMD_ADD` access type, it must be defined as a non-permanent interior node, the node name must conform to the valid names, and the creation of the new node must not cause the maximum occurrence number to be exceeded.

If the meta-data cannot be retrieved because the given node cannot possibly exist in the tree (it is not defined in the specification), the `NODE_NOT_FOUND` error code is returned (see `getMetaNode(String)`).

Throws `DmtException` – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a node that cannot exist in the tree (see above)
NODE_ALREADY_EXISTS if nodeUri points to a node that already exists
PERMISSION_DENIED if the session is associated with a principal and the ACL of the parent node does not allow the Add operation for the associated principal
COMMAND_NOT_ALLOWED if the parent node is not an interior node, or in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
METADATA_MISMATCH if the node could not be created because of meta-data restrictions (see above)
TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session was opened using the `LOCK_TYPE_SHARED` lock type, or if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have `DmtPermission` for the parent node with the Add action present

117.15.11.11 public void createInteriorNode (String nodeUri , String type) throws DmtException

nodeUri the URI of the node to create

type the type URI of the interior node, can be `null` if no node type is defined

□ Create an interior node with a given type. The type of interior node, if specified, is a URI identifying a DDF document. If the parent node does not exist, it is created automatically, as if `createInteriorNode(String)` were called for the parent URI. This way all missing ancestor nodes leading to the specified node are created. Any exceptions encountered while creating the ancestors are propagated to the caller of this method, these are not explicitly listed in the error descriptions below.

If meta-data is available for the node, several checks are made before creating it. The node must have `MetaNode.CMD_ADD` access type, it must be defined as a non-permanent interior node, the node name must conform to the valid names, and the creation of the new node must not cause the maximum occurrence number to be exceeded.

If the meta-data cannot be retrieved because the given node cannot possibly exist in the tree (it is not defined in the specification), the `NODE_NOT_FOUND` error code is returned (see `getMetaNode(String)`).

Interior node type identifiers must follow the format defined in section 7.7.7.2 of the OMA Device Management Tree and Description document. Checking the validity of the type string does not have to be done by the DmtAdmin, this can be left to the plugin handling the node (if any), to avoid unnecessary double-checks.

Throws `DmtException` – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a node that cannot exist in the tree (see above)
NODE_ALREADY_EXISTS if nodeUri points to a node that already exists
PERMISSION_DENIED if the session is associated with a principal and the ACL of the parent node does not allow the Add operation for the associated principal
COMMAND_NOT_ALLOWED if the parent node is not an interior node, or in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
METADATA_MISMATCH if the node could not be created because of meta-data restrictions (see above)
TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, if the type string is invalid (see above), or if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session was opened using the `LOCK_TYPE_SHARED` lock type, or if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have `DmtPermission` for the parent node with the Add action present

See Also `createInteriorNode(String)` , OMA Device Management Tree and Description v1.2 draft (http://member.openmobilealliance.org/ftp/public_documents/dm/Permanent_documents/OMA-TS-DM-TND-V1_2-20050615-C.zip)

117.15.11.12 public void createLeafNode (String nodeUri) throws DmtException

nodeUri the URI of the node to create

□ Create a leaf node with default value and MIME type. If a node does not have a default value or MIME type, this method will throw a `DmtException` with error code `METADATA_MISMATCH`. Note that a node might have a default value or MIME type even if there is no meta-data for the node or its meta-data does not specify the default.

If the parent node does not exist, it is created automatically, as if `createInteriorNode(String)` were called for the parent URI. This way all missing ancestor nodes leading to the specified node are created. Any exceptions encountered while creating the ancestors are propagated to the caller of this method, these are not explicitly listed in the error descriptions below.

If meta-data is available for a node, several checks are made before creating it. The node must have `MetaNode.CMD_ADD` access type, it must be defined as a non-permanent leaf node, the node name must conform to the valid names, and the creation of the new node must not cause the maximum occurrence number to be exceeded.

If the meta-data cannot be retrieved because the given node cannot possibly exist in the tree (it is not defined in the specification), the `NODE_NOT_FOUND` error code is returned (see `getMetaNode(String)`).

Throws `DmtException` – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a node that cannot exist in the tree (see above)
NODE_ALREADY_EXISTS if nodeUri points to a node that already exists
PERMISSION_DENIED if the session is associated with a principal and the ACL of the parent node does not allow the Add operation for the associated principal
COMMAND_NOT_ALLOWED if the parent node is not an interior node, or in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
METADATA_MISMATCH if the node could not be created because of meta-data restrictions (see above)
TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session was opened using the `LOCK_TYPE_SHARED` lock type, or if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have `DmtPermission` for the parent node with the Add action present

See Also `createLeafNode(String, DmtData)`

117.15.11.13 public void createLeafNode (String nodeUri , DmtData value) throws DmtException

nodeUri the URI of the node to create

value the value to be given to the new node, can be `null`

□ Create a leaf node with a given value and the default MIME type. If the specified value is `null`, the default value is taken. If the node does not have a default MIME type or value (if needed), this method will throw a `DmtException` with error code `METADATA_MISMATCH`. Note that a node might have a default value or MIME type even if there is no meta-data for the node or its meta-data does not specify the default.

If the parent node does not exist, it is created automatically, as if `createInteriorNode(String)` were called for the parent URI. This way all missing ancestor nodes leading to the specified node are created. Any exceptions encountered while creating the ancestors are propagated to the caller of this method, these are not explicitly listed in the error descriptions below.

If meta-data is available for a node, several checks are made before creating it. The node must have `MetaNode.CMD_ADD` access type, it must be defined as a non-permanent leaf node, the node name must conform to the valid names, the node value must conform to the value constraints, and the creation of the new node must not cause the maximum occurrence number to be exceeded.

If the meta-data cannot be retrieved because the given node cannot possibly exist in the tree (it is not defined in the specification), the `NODE_NOT_FOUND` error code is returned (see `getMetaNode(String)`).

Nodes of `null` format can be created by using `DmtData.NULL_VALUE` as second argument.

Throws `DmtException` – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a node that cannot exist in the tree (see above)
NODE_ALREADY_EXISTS if nodeUri points to a node that already exists
PERMISSION_DENIED if the session is associated with a principal and the ACL of the parent node does not allow the Add operation for the associated principal
COMMAND_NOT_ALLOWED if the parent node is not an interior node, or in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
METADATA_MISMATCH if the node could not be created because of meta-data restrictions (see above)
TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session was opened using the `LOCK_TYPE_SHARED` lock type, or if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have `DmtPermission` for the parent node with the Add action present

117.15.11.14 public void createLeafNode (String nodeUri , DmtData value , String mimeType) throws DmtException

nodeUri the URI of the node to create

value the value to be given to the new node, can be `null`

mimeType the MIME type to be given to the new node, can be `null`

□ Create a leaf node with a given value and MIME type. If the specified value or MIME type is `null`, their default values are taken. If the node does not have the necessary defaults, this method will throw a `DmtException` with error code `METADATA_MISMATCH`. Note that a node might have a default value or MIME type even if there is no meta-data for the node or its meta-data does not specify the default.

If the parent node does not exist, it is created automatically, as if `createInteriorNode(String)` were called for the parent URI. This way all missing ancestor nodes leading to the specified node are created. Any exceptions encountered while creating the ancestors are propagated to the caller of this method, these are not explicitly listed in the error descriptions below.

If meta-data is available for a node, several checks are made before creating it. The node must have MetaNode.CMD_ADD access type, it must be defined as a non-permanent leaf node, the node name must conform to the valid names, the node value must conform to the value constraints, the MIME type must be among the listed types, and the creation of the new node must not cause the maximum occurrence number to be exceeded.

If the meta-data cannot be retrieved because the given node cannot possibly exist in the tree (it is not defined in the specification), the NODE_NOT_FOUND error code is returned (see getMetaNode(String)).

Nodes of null format can be created by using DmtData.NULL_VALUE as second argument.

The MIME type string must conform to the definition in RFC 2045. Checking its validity does not have to be done by the DmtAdmin, this can be left to the plugin handling the node (if any), to avoid unnecessary double-checks.

Throws DmtException – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a node that cannot exist in the tree (see above)
NODE_ALREADY_EXISTS if nodeUri points to a node that already exists
PERMISSION_DENIED if the session is associated with a principal and the ACL of the parent node does not allow the Add operation for the associated principal
COMMAND_NOT_ALLOWED if the parent node is not an interior node, or in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
METADATA_MISMATCH if the node could not be created because of meta-data restrictions (see above)
TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, if mimeType is not a proper MIME type string (see above), or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException – if the session was opened using the LOCK_TYPE_SHARED lock type, or if the session is already closed or invalidated

SecurityException – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the parent node with the Add action present

See Also createLeafNode(String, DmtData) , RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt)

117.15.11.15 public void deleteNode (String nodeUri) throws DmtException

nodeUri the URI of the node

□ Delete the given node. Deleting interior nodes is recursive, the whole subtree under the given node is deleted. It is not allowed to delete the root node of the session.

If meta-data is available for a node, several checks are made before deleting it. The node must be non-permanent, it must have the MetaNode.CMD_DELETE access type, and if zero occurrences of the node are not allowed, it must not be the last one.

Throws DmtException – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a non-existing node
PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not

allow the Delete operation for the associated principal
COMMAND_NOT_ALLOWED if the target node is the root of the session, or in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
METADATA_MISMATCH if the node could not be deleted because of meta-data restrictions (see above)
TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session was opened using the `LOCK_TYPE_SHARED` lock type, or if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have `DmtPermission` for the node with the Delete action present

117.15.11.16 public void execute (String nodeUri , String data) throws DmtException

nodeUri the node on which the execute operation is issued

data the parameter of the execute operation, can be `null`

☐ Executes a node. This corresponds to the EXEC operation in OMA DM. This method cannot be called in a read-only session.

The semantics of an execute operation and the data parameter it takes depends on the definition of the managed object on which the command is issued.

Throws `DmtException` – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if the node does not exist and the plugin does not allow executing unexisting nodes
PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Execute operation for the associated principal
METADATA_MISMATCH if the node cannot be executed according to the meta-data (does not have MetaNode.CMD_EXECUTE access type)
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, if no DmtExecPlugin is associated with the node and the DmtAdmin can not execute the node, or if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session was opened using the `LOCK_TYPE_SHARED` lock type, or if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have `DmtPermission` for the node with the Exec action present

See Also `execute(String, String, String)`

117.15.11.17 public void execute (String nodeUri , String correlator , String data) throws DmtException

nodeUri the node on which the execute operation is issued

correlator an identifier to associate this operation with any notifications sent in response to it, can be `null` if not needed

data the parameter of the execute operation, can be `null`

□ Executes a node, also specifying a correlation ID for use in response notifications. This operation corresponds to the EXEC command in OMA DM. This method cannot be called in a read-only session.

The semantics of an execute operation and the data parameter it takes depends on the definition of the managed object on which the command is issued. If a correlation ID is specified, it should be used as the `correlator` parameter for notifications sent in response to this execute operation.

Throws `DmtException` – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if the node does not exist and the plugin does not allow executing unexisting nodes
PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Execute operation for the associated principal
METADATA_MISMATCH if the node cannot be executed according to the meta-data (does not have MetaNode.CMD_EXECUTE access type)
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, if no DmtExecPlugin is associated with the node, or if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session was opened using the `LOCK_TYPE_SHARED` lock type, or if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have `DmtPermission` for the node with the Exec action present

See Also `execute(String, String)`

117.15.11.18 public String[] getChildNodeNames (String nodeUri) throws DmtException

nodeUri the URI of the node

□ Get the list of children names of a node. The returned array contains the names - not the URIs - of the immediate children nodes of the given node. The elements are in no particular order. The returned array must not contain `null` entries.

Returns the list of child node names as a string array or an empty string array if the node has no children

Throws `DmtException` – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a non-existing node
PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Get operation for the associated principal
COMMAND_NOT_ALLOWED if the specified node is not an interior node
METADATA_MISMATCH if node information cannot be retrieved according to the meta-data (it does not have MetaNode.CMD_GET access type)
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have `DmtPermission` for the node with the Get action present

117.15.11.19 public Acl getEffectiveNodeAcl (String nodeUri) throws DmtException

nodeUri the URI of the node

□ Gives the Access Control List in effect for a given node. The returned Acl takes inheritance into account, that is if there is no ACL defined for the node, it will be derived from the closest ancestor having an ACL defined.

Returns the Access Control List belonging to the node

Throws DmtException – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a non-existing node
PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Get operation for the associated principal
METADATA_MISMATCH if node information cannot be retrieved according to the meta-data (the node does not have MetaNode.CMD_GET access type)
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException – if the session is already closed or invalidated

SecurityException – in case of local sessions, if the caller does not have DmtPermission for the node with the Get action present

See Also getNodeAcl(String)

117.15.11.20 public int getLockType ( )

□ Gives the type of lock the session has.

Returns the lock type of the session, one of LOCK_TYPE_SHARED, LOCK_TYPE_EXCLUSIVE and LOCK_TYPE_ATOMIC

117.15.11.21 public MetaNode getMetaNode (String nodeUri) throws DmtException

nodeUri the URI of the node

□ Get the meta data which describes a given node. Meta data can only be inspected, it can not be changed.

The MetaNode object returned to the client is the combination of the meta data returned by the data plugin (if any) plus the meta data returned by the DmtAdmin. If there are differences in the meta data elements known by the plugin and the DmtAdmin then the plugin specific elements take precedence.

Note, that a node does not have to exist for having meta-data associated with it. This method may provide meta-data for any node that can possibly exist in the tree (any node defined in the specification). For nodes that are not defined, it may throw DmtException with the error code NODE_NOT_FOUND. To allow easier implementation of plugins that do not provide meta-data, it is allowed to return null for any node, regardless of whether it is defined or not.

Returns a MetaNode which describes meta data information, can be null if there is no meta data available for the given node

Throws DmtException – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a node that is not defined in the tree (see above)
PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not

allow the Get operation for the associated principal
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException – if the session is already closed or invalidated

SecurityException – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Get action present

117.15.11.22 public Acl getNodeAcl (String nodeUri) throws DmtException

nodeUri the URI of the node

☐ Get the Access Control List associated with a given node. The returned Acl object does not take inheritance into account, it gives the ACL specifically given to the node.

Returns the Access Control List belonging to the node or null if none defined

Throws DmtException – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a non-existing node
PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Get operation for the associated principal
METADATA_MISMATCH if node information cannot be retrieved according to the meta-data (the node does not have MetaNode.CMD_GET access type)
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException – if the session is already closed or invalidated

SecurityException – in case of local sessions, if the caller does not have DmtPermission for the node with the Get action present

See Also getEffectiveNodeAcl(String)

117.15.11.23 public int getNodeSize (String nodeUri) throws DmtException

nodeUri the URI of the leaf node

☐ Get the size of the data in a leaf node. The returned value depends on the format of the data in the node, see the description of the DmtData.getSize() method for the definition of node size for each format.

Returns the size of the data in the node

Throws DmtException – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a non-existing node
PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Get operation for the associated principal
COMMAND_NOT_ALLOWED if the specified node is not a leaf node
METADATA_MISMATCH if node information cannot be retrieved according to the meta-data (it does not have MetaNode.CMD_GET access type)
FEATURE_NOT_SUPPORTED if the Size property is not supported by the DmtAdmin implementation or the underlying plugin
DATA_STORE_FAILURE if an error occurred while accessing the data store

COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have `DmtPermission` for the node with the Get action present

See Also `DmtData.getSize()`

117.15.11.24 public Date getNodeTimestamp (String nodeUri) throws DmtException

nodeUri the URI of the node

□ Get the timestamp when the node was created or last modified.

Returns the timestamp of the last modification

Throws `DmtException` – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a non-existing node
PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Get operation for the associated principal
METADATA_MISMATCH if node information cannot be retrieved according to the meta-data (it does not have MetaNode.CMD_GET access type)
FEATURE_NOT_SUPPORTED if the Timestamp property is not supported by the DmtAdmin implementation or the underlying plugin
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have `DmtPermission` for the node with the Get action present

117.15.11.25 public String getNodeTitle (String nodeUri) throws DmtException

nodeUri the URI of the node

□ Get the title of a node. There might be no title property set for a node.

Returns the title of the node, or `null` if the node has no title

Throws `DmtException` – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a non-existing node
PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Get operation for the associated principal
METADATA_MISMATCH if node information cannot be retrieved according to the meta-data (it does not have MetaNode.CMD_GET access type)
FEATURE_NOT_SUPPORTED if the Title property is not supported by the DmtAdmin implementation or the underlying plugin
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have `DmtPermission` for the node with the Get action present

117.15.11.26 public String getNodeType (String nodeUri) throws DmtException

nodeUri the URI of the node

□ Get the type of a node. The type of leaf node is the MIME type of the data it contains. The type of an interior node is a URI identifying a DDF document; a `null` type means that there is no DDF document overriding the tree structure defined by the ancestors.

Returns the type of the node, can be `null`

Throws `DmtException` – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a non-existing node
PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Get operation for the associated principal
METADATA_MISMATCH if node information cannot be retrieved according to the meta-data (it does not have MetaNode.CMD_GET access type)
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have `DmtPermission` for the node with the Get action present

117.15.11.27 public DmtData getNodeValue (String nodeUri) throws DmtException

nodeUri the URI of the node to retrieve

□ Get the data contained in a leaf or interior node. When retrieving the value associated with an interior node, the caller must have rights to read all nodes in the subtree under the given node.

Returns the data of the node, can not be `null`

Throws `DmtException` – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a non-existing node
PERMISSION_DENIED if the session is associated with a principal and the ACL of the node (and the ACLs of all its descendants in case of interior nodes) do not allow the Get operation for the associated principal
METADATA_MISMATCH if the node value cannot be retrieved according to the meta-data (it does not have MetaNode.CMD_GET access type)
FEATURE_NOT_SUPPORTED if the specified node is an interior node and does not support Java object values
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have `DmtPermission` for the node (and all its descendants in case of interior nodes) with the Get action present

117.15.11.28 public int getNodeVersion (String nodeUri) throws DmtException

nodeUri the URI of the node

□ Get the version of a node. The version can not be set, it is calculated automatically by the device. It is incremented modulo 0x10000 at every modification of the value or any other property of the node, for both leaf and interior nodes. When a node is created the initial value is 0.

Returns the version of the node

Throws `DmtException` – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a non-existing node
PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Get operation for the associated principal
METADATA_MISMATCH if node information cannot be retrieved according to the meta-data (it does not have MetaNode.CMD_GET access type)
FEATURE_NOT_SUPPORTED if the Version property is not supported by the DmtAdmin implementation or the underlying plugin
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have `DmtPermission` for the node with the Get action present

117.15.11.29 public String getPrincipal ( )

□ Gives the name of the principal on whose behalf the session was created. Local sessions do not have an associated principal, in this case `null` is returned.

Returns the identifier of the remote server that initiated the session, or `null` for local sessions

117.15.11.30 public String getRootUri ( )

□ Get the root URI associated with this session. Gives "." if the session was created without specifying a root, which means that the target of this session is the whole DMT.

Returns the root URI

117.15.11.31 public int getSessionId ( )

□ The unique identifier of the session. The ID is generated automatically, and it is guaranteed to be unique on a machine for a specific Dmt Admin. A session id must be larger than 0.

Returns the session identification number

117.15.11.32 public int getState ( )

□ Get the current state of this session.

Returns the state of the session, one of `STATE_OPEN`, `STATE_CLOSED` and `STATE_INVALID`

117.15.11.33 public boolean isLeafNode (String nodeUri) throws DmtException

nodeUri the URI of the node

□ Tells whether a node is a leaf or an interior node of the DMT.

Returns true if the given node is a leaf node

Throws `DmtException` – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a non-existing node
PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Get operation for the associated principal
METADATA_MISMATCH if node information cannot be retrieved according to the meta-data (it does not have MetaNode.CMD_GET access type)
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have `DmtPermission` for the node with the Get action present

117.15.11.34 public boolean isNodeUri (String nodeUri)

nodeUri the URI to check

□ Check whether the specified URI corresponds to a valid node in the DMT.

Returns true if the given node exists in the DMT

Throws `DmtIllegalStateException` – if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have `DmtPermission` for the node with the Get action present

117.15.11.35 public void renameNode (String nodeUri , String newName) throws DmtException

nodeUri the URI of the node to rename

newName the new name property of the node

□ Rename a node. This operation only changes the name of the node (updating the timestamp and version properties if they are supported), the value and the other properties are not changed. The new name of the node must be provided, the new URI is constructed from the base of the old URI and the given name. It is not allowed to rename the root node of the session.

If available, the meta-data of the original and the new nodes are checked before performing the rename operation. Neither node can be permanent, their leaf/interior property must match, and the name change must not violate any of the cardinality constraints. The original node must have the `MetaNode.CMD_REPLACE` access type, and the name of the new node must conform to the valid names.

Throws `DmtException` – with the following possible error codes:
INVALID_URI if nodeUri or newName is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a non-existing node, or if the new node is not defined in the tree according to the meta-data (see getMetaNode(String))
NODE_ALREADY_EXISTS if there already exists a sibling of nodeUri with the name newName
PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Replace operation for the associated principal
COMMAND_NOT_ALLOWED if the target node is the root of the session, or in non-atomic sessions

if the underlying plugin is read-only or does not support non-atomic writing
METADATA_MISMATCH if the node could not be renamed because of meta-data restrictions (see above)
TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session was opened using the `LOCK_TYPE_SHARED` lock type, or if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have `DmtPermission` for the node with the Replace action present

117.15.11.36 public void rollback ( ) throws DmtException

□ Rolls back a series of DMT operations issued in the current atomic session since the last transaction boundary. Transaction boundaries are the creation of this object that starts the session, and all subsequent `commit()` and `rollback()` calls.

Throws `DmtException` – with the error code `ROLLBACK_FAILED` in case the rollback did not succeed

`DmtIllegalStateException` – if the session was not opened using the `LOCK_TYPE_ATOMIC` lock type, or if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

117.15.11.37 public void setDefaultNodeValue (String nodeUri) throws DmtException

nodeUri the URI of the node

□ Set the value of a leaf or interior node to its default. The default can be defined by the node's `MetaNode`. The method throws a `METADATA_MISMATCH` exception if the node does not have a default value.

Throws `DmtException` – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a non-existing node
PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Replace operation for the associated principal
COMMAND_NOT_ALLOWED in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
METADATA_MISMATCH if the node is permanent or cannot be modified according to the meta-data (does not have the MetaNode.CMD_REPLACE access type), or if there is no default value defined for this node
FEATURE_NOT_SUPPORTED if the specified node is an interior node and does not support Java object values
TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session was opened using the `LOCK_TYPE_SHARED` lock type, or if the session is already closed or invalidated

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have `DmtPermission` for the node with the Replace action present

See Also `setNodeValue(String, DmtData)`

117.15.11.38 public void setNodeAcl (String nodeUri , Acl acl) throws DmtException

nodeUri the URI of the node

acl the Access Control List to be set on the node, can be `null`

□ Set the Access Control List associated with a given node. To perform this operation, the caller needs to have replace rights (`Acl.REPLACE` or the corresponding Java permission depending on the session type) as described below:

- if `nodeUri` specifies a leaf node, replace rights are needed on the parent of the node
- if `nodeUri` specifies an interior node, replace rights on either the node or its parent are sufficient

If the given `acl` is `null` or an empty ACL (not specifying any permissions for any principals), then the ACL of the node is deleted, and the node will inherit the ACL from its parent node.

Throws `DmtException` – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a non-existing node
PERMISSION_DENIED if the session is associated with a principal and the ACL of the node or its parent (see above) does not allow the Replace operation for the associated principal
COMMAND_NOT_ALLOWED if the command attempts to set the ACL of the root node not to include Add rights for all principals
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

`DmtIllegalStateException` – if the session was opened using the `LOCK_TYPE_SHARED` lock type, or if the session is already closed or invalidated

`SecurityException` – in case of local sessions, if the caller does not have `DmtPermission` for the node or its parent (see above) with the Replace action present

117.15.11.39 public void setNodeTitle (String nodeUri , String title) throws DmtException

nodeUri the URI of the node

title the title text of the node, can be `null`

□ Set the title property of a node. The length of the title string in UTF-8 encoding must not exceed 255 bytes.

Throws `DmtException` – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a non-existing node
PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Replace operation for the associated principal
COMMAND_NOT_ALLOWED in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
METADATA_MISMATCH if the node cannot be modified according to the meta-data (does not have the MetaNode.CMD_REPLACE access type)
FEATURE_NOT_SUPPORTED if the Title property is not supported by the DmtAdmin implementation or the underlying plugin

TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the title string is too long, if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException – if the session was opened using the LOCK_TYPE_SHARED lock type, or if the session is already closed or invalidated

SecurityException – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Replace action present

117.15.11.40 public void setNodeType (String nodeUri , String type) throws DmtException

nodeUri the URI of the node

type the type of the node, can be null

□ Set the type of a node. The type of leaf node is the MIME type of the data it contains. The type of an interior node is a URI identifying a DDF document.

For interior nodes, a null type string means that there is no DDF document overriding the tree structure defined by the ancestors. For leaf nodes, it requests that the default MIME type is used for the given node. If the node does not have a default MIME type this method will throw a DmtException with error code METADATA_MISMATCH. Note that a node might have a default MIME type even if there is no meta-data for the node or its meta-data does not specify the default.

MIME types must conform to the definition in RFC 2045. Interior node type identifiers must follow the format defined in section 7.7.7.2 of the OMA Device Management Tree and Description document. Checking the validity of the type string does not have to be done by the DmtAdmin, this can be left to the plugin handling the node (if any), to avoid unnecessary double-checks.

Throws DmtException – with the following possible error codes:
INVALID_URI if nodeUri is null or syntactically invalid
URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources)
NODE_NOT_FOUND if nodeUri points to a non-existing node
PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Replace operation for the associated principal
COMMAND_NOT_ALLOWED in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing
METADATA_MISMATCH if the node is permanent or cannot be modified according to the meta-data (does not have the MetaNode.CMD_REPLACE access type), and in case of leaf nodes, if null is given and there is no default MIME type, or the given MIME type is not allowed
TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if the URI is not within the current session's subtree, if the type string is invalid (see above), or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException – if the session was opened using the LOCK_TYPE_SHARED lock type, or if the session is already closed or invalidated

SecurityException – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Replace action present

See Also RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt) , OMA Device Management Tree and Description v1.2 draft (http://member.openmobilealliance.org/ftp/public_documents/dm/Permanent_documents/OMA-TS-DM-TND-V1_2-20050615-C.zip)

117.15.11.41 public void setNodeValue (String nodeUri , DmtData data) throws DmtException

nodeUri the URI of the node

data the data to be set, can be null

□ Set the value of a leaf or interior node. The format of the node is contained in the DmtData object. For interior nodes, the format must be FORMAT_NODE, while for leaf nodes this format must not be used.

If the specified value is null, the default value is taken. In this case, if the node does not have a default value, this method will throw a DmtException with error code METADATA_MISMATCH. Nodes of null format can be set by using DmtData.NULL_VALUE as second argument.

An Event of type REPLACE is sent out for a leaf node. A replaced interior node sends out events for each of its children in depth first order and node names sorted with Arrays.sort(String[]). When setting a value on an interior node, the values of the leaf nodes under it can change, but the structure of the subtree is not modified by the operation.

Throws DmtException – with the following possible error codes: INVALID_URI if nodeUri is null or syntactically invalid URI_TOO_LONG if nodeUri is longer than accepted by the DmtAdmin implementation (especially on systems with limited resources) NODE_NOT_FOUND if nodeUri points to a non-existing node PERMISSION_DENIED if the session is associated with a principal and the ACL of the node does not allow the Replace operation for the associated principal COMMAND_NOT_ALLOWED if the given data has FORMAT_NODE format but the node is a leaf node (or vice versa), or in non-atomic sessions if the underlying plugin is read-only or does not support non-atomic writing METADATA_MISMATCH if the node is permanent or cannot be modified according to the meta-data (does not have the MetaNode.CMD_REPLACE access type), or if the given value does not conform to the meta-data value constraints FEATURE_NOT_SUPPORTED if the specified node is an interior node and does not support Java object values TRANSACTION_ERROR in an atomic session if the underlying plugin is read-only or does not support atomic writing DATA_STORE_FAILURE if an error occurred while accessing the data store COMMAND_FAILED if the URI is not within the current session's subtree, or if some unspecified error is encountered while attempting to complete the command

DmtIllegalStateException – if the session was opened using the LOCK_TYPE_SHARED lock type, or if the session is already closed or invalidated

SecurityException – if the caller does not have the necessary permissions to execute the underlying management operation, or, in case of local sessions, if the caller does not have DmtPermission for the node with the Replace action present

117.15.12 public interface MetaNode

The MetaNode contains meta data as standardized by OMA DM but extends it (without breaking the compatibility) to provide for better DMT data quality in an environment where many software components manipulate this data.

The interface has several types of functions to describe the nodes in the DMT. Some methods can be used to retrieve standard OMA DM metadata such as access type, cardinality, default, etc., others are for data extensions such as valid names and values. In some cases the standard behaviour has been extended, for example it is possible to provide several valid MIME types, or to differentiate between normal and automatic dynamic nodes.

Most methods in this interface receive no input, just return information about some aspect of the node. However, there are two methods that behave differently, isValidName(String) and isValidValue(DmtData). These validation methods are given a potential node name or value (respectively), and can decide whether it is valid for the given node. Passing the validation methods is a necessary condition for a name or value to be used, but it is not necessarily sufficient: the plugin may carry out more thorough (more expensive) checks when the node is actually created or set.

If a MetaNode is available for a node, the DmtAdmin must use the information provided by it to filter out invalid requests on that node. However, not all methods on this interface are actually used for this purpose, as many of them (e.g. getFormat() or getValidNames()) can be substituted with the validating methods. For example, isValidValue(DmtData) can be expected to check the format, minimum, maximum, etc. of a given value, making it unnecessary for the DmtAdmin to call getFormat(), getMin(), getMax() etc. separately. It is indicated in the description of each method if the DmtAdmin does not enforce the constraints defined by it - such methods are only for external use, for example in user interfaces.

Most of the methods of this class return null if a certain piece of meta information is not defined for the node or providing this information is not supported. Methods of this class do not throw exceptions.

117.15.12.1 public static final int AUTOMATIC = 2

Constant for representing an automatic node in the tree. This must be returned by getScope(). AUTOMATIC nodes are part of the life cycle of their parent node, they usually describe attributes/ properties of the parent.

117.15.12.2 public static final int CMD_ADD = 0

Constant for the ADD access type. If can(int) returns true for this operation, this node can potentially be added to its parent. Nodes with PERMANENT or AUTOMATIC scope typically do not have this access type.

117.15.12.3 public static final int CMD_DELETE = 1

Constant for the DELETE access type. If can(int) returns true for this operation, the node can potentially be deleted.

117.15.12.4 public static final int CMD_EXECUTE = 2

Constant for the EXECUTE access type. If can(int) returns true for this operation, the node can potentially be executed.

117.15.12.5 public static final int CMD_GET = 4

Constant for the GET access type. If can(int) returns true for this operation, the value, the list of child nodes (in case of interior nodes) and the properties of the node can potentially be retrieved.

117.15.12.6 public static final int CMD_REPLACE = 3

Constant for the REPLACE access type. If can(int) returns true for this operation, the value and other properties of the node can potentially be modified.

117.15.12.7 public static final int DYNAMIC = 1

Constant for representing a dynamic node in the tree. This must be returned by getScope(). Dynamic nodes can be added and deleted.

117.15.12.8 public static final int PERMANENT = 0

Constant for representing a PERMANENT node in the tree. This must be returned by getScope() if the node cannot be added, deleted or modified in any way through tree operations. PERMANENT nodes in general map to the roots of Plugins.

117.15.12.9 public boolean can (int operation)

operation One of the MetaNode.CMD_... constants.

□ Check whether the given operation is valid for this node. If no meta-data is provided for a node, all operations are valid.

Returns false if the operation is not valid for this node or the operation code is not one of the allowed constants

117.15.12.10 public DmtData getDefault ( )

□ Get the default value of this node if any.

Returns The default value or null if not defined

117.15.12.11 public String getDescription ( )

□ Get the explanation string associated with this node. Can be null if no description is provided for this node.

Returns node description string or null for no description

117.15.12.12 public Object getExtensionProperty (String key)

key the key for the extension property

□ Returns the value for the specified extension property key. This method only works if the provider of this MetaNode provides proprietary extensions to node meta data.

Returns the value of the requested property, cannot be null

Throws IllegalArgumentException – if the specified key is not supported by this MetaNode

117.15.12.13 public String[] getExtensionPropertyKeys ( )

□ Returns the list of extension property keys, if the provider of this MetaNode provides proprietary extensions to node meta data. The method returns null if the node doesn't provide such extensions.

Returns the array of supported extension property keys

117.15.12.14 public int getFormat ( )

□ Get the node's format, expressed in terms of type constants defined in DmtData. If there are multiple formats allowed for the node then the format constants are OR-ed. Interior nodes must have DmtData.FORMAT_NODE format, and this code must not be returned for leaf nodes. If no meta-data is provided for a node, all applicable formats are considered valid (with the above constraints regarding interior and leaf nodes).

Note that the 'format' term is a legacy from OMA DM, it is more customary to think of this as 'type'.

The formats returned by this method are not checked by DmtAdmin, they are only for external use, for example in user interfaces. DmtAdmin only calls isValidValue(DmtData) for checking the value, its behavior should be consistent with this method.

Returns the allowed format(s) of the node

117.15.12.15 public double getMax ( )

□ Get the maximum allowed value associated with a node of numeric format. If no meta-data is provided for a node, there is no upper limit to its value. This method is only meaningful if the node has one of the numeric formats: integer, float, or long. format. The returned limit has double type, as this can be used to denote all numeric limits with full precision. The actual maximum should be the largest integer, float or long number that does not exceed the returned value.

The information returned by this method is not checked by DmtAdmin, it is only for external use, for example in user interfaces. DmtAdmin only calls isValidValue(DmtData) for checking the value, its behavior should be consistent with this method.

Returns the allowed maximum, or Double.MAX_VALUE if there is no upper limit defined or the node's format is not one of the numeric formats integer, float, or long

117.15.12.16 public int getMaxOccurrence ( )

- Get the number of maximum occurrences of this type of nodes on the same level in the DMT. Returns Integer.MAX_VALUE if there is no upper limit. Note that if the occurrence is greater than 1 then this node can not have siblings with different metadata. In other words, if different types of nodes coexist on the same level, their occurrence can not be greater than 1. If no meta-data is provided for a node, there is no upper limit on the number of occurrences.

Returns The maximum allowed occurrence of this node type

117.15.12.17 public String[] getMimeTypes ( )

- Get the list of MIME types this node can hold. The first element of the returned list must be the default MIME type.

  All MIME types are considered valid if no meta-data is provided for a node or if null is returned by this method. In this case the default MIME type cannot be retrieved from the meta-data, but the node may still have a default. This hidden default (if it exists) can be utilized by passing null as the type parameter of DmtSession.setNodeType(String, String) or DmtSession.createLeafNode(String, DmtData, String).

Returns the list of allowed MIME types for this node, starting with the default MIME type, or null if all types are allowed

117.15.12.18 public double getMin ( )

- Get the minimum allowed value associated with a node of numeric format. If no meta-data is provided for a node, there is no lower limit to its value. This method is only meaningful if the node has one of the numeric formats: integer, float, or long format. The returned limit has double type, as this can be used to denote both integer and float limits with full precision. The actual minimum should be the smallest integer, float or long value that is equal or larger than the returned value.

  The information returned by this method is not checked by DmtAdmin, it is only for external use, for example in user interfaces. DmtAdmin only calls isValidValue(DmtData) for checking the value, its behavior should be consistent with this method.

Returns the allowed minimum, or Double.MIN_VALUE if there is no lower limit defined or the node's format is not one of the numeric formats integer, float, or long

117.15.12.19 public String[] getRawFormatNames ( )

- Get the format names for any raw formats supported by the node. This method is only meaningful if the list of supported formats returned by getFormat() contains DmtData.FORMAT_RAW_STRING or DmtData.FORMAT_RAW_BINARY: it specifies precisely which raw format(s) are actually supported. If the node cannot contain data in one of the raw types, this method must return null.

  The format names returned by this method are not checked by DmtAdmin, they are only for external use, for example in user interfaces. DmtAdmin only calls isValidValue(DmtData) for checking the value, its behavior should be consistent with this method.

Returns the allowed format name(s) of raw data stored by the node, or null if raw formats are not supported

117.15.12.20 public int getScope ( )

- Return the scope of the node. Valid values are MetaNode.PERMANENT, MetaNode.DYNAMIC and MetaNode.AUTOMATIC. Note that a permanent node is not the same as a node where the DELETE operation is not allowed. Permanent nodes never can be deleted, whereas a non-deletable node can disappear in a recursive DELETE operation issued on one of its parents. If no meta-data is provided for a node, it can be assumed to be a dynamic node.

Returns PERMANENT for permanent nodes, AUTOMATIC for nodes that are automatically created, and DYNAMIC otherwise

117.15.12.21 **public String[] getValidNames ( )**

□ Return an array of Strings if valid names are defined for the node, or null if no valid name list is defined or if this piece of meta info is not supported. If no meta-data is provided for a node, all names are considered valid.

The information returned by this method is not checked by DmtAdmin, it is only for external use, for example in user interfaces. DmtAdmin only calls isValidName(String) for checking the name, its behavior should be consistent with this method.

Returns the valid values for this node name, or null if not defined

117.15.12.22 **public DmtData[] getValidValues ( )**

□ Return an array of DmtData objects if valid values are defined for the node, or null otherwise. If no meta-data is provided for a node, all values are considered valid.

The information returned by this method is not checked by DmtAdmin, it is only for external use, for example in user interfaces. DmtAdmin only calls isValidValue(DmtData) for checking the value, its behavior should be consistent with this method.

Returns the valid values for this node, or null if not defined

117.15.12.23 **public boolean isLeaf ( )**

□ Check whether the node is a leaf node or an internal one.

Returns true if the node is a leaf node

117.15.12.24 **public boolean isValidName (String name)**

name the node name to check for validity

□ Checks whether the given name is a valid name for this node. This method can be used for example to ensure that the node name is always one of a predefined set of valid names, or that it matches a specific pattern. This method should be consistent with the values returned by getValidNames() (if any), the DmtAdmin only calls this method for name validation.

This method may return true even if not all aspects of the name have been checked, expensive operations (for example those that require external resources) need not be performed here. The actual node creation may still indicate that the node name is invalid.

Returns false if the specified name is found to be invalid for the node described by this meta-node, true otherwise

117.15.12.25 **public boolean isValidValue (DmtData value)**

value the value to check for validity

□ Checks whether the given value is valid for this node. This method can be used to ensure that the value has the correct format and range, that it is well formed, etc. This method should be consistent with the constraints defined by the getFormat(), getValidValues(), getMin() and getMax() methods (if applicable), as the Dmt Admin only calls this method for value validation.

This method may return true even if not all aspects of the value have been checked, expensive operations (for example those that require external resources) need not be performed here. The actual value setting method may still indicate that the value is invalid.

Returns false if the specified value is found to be invalid for the node described by this meta-node, true otherwise

117.15.12.26 **public boolean isZeroOccurrenceAllowed ( )**

□ Check whether zero occurrence of this node is valid. If no meta-data is returned for a node, zero occurrences are allowed.

Returns true if zero occurrence of this node is valid

117.15.13 public final class Uri

This class contains static utility methods to manipulate DMT URIs.

Syntax of valid DMT URIs:

- A slash ('/' \u002F) is the separator of the node names. Slashes used in node name must therefore be escaped using a backslash slash ("\/"). The backslash must be escaped with a double backslash sequence. A backslash found must be ignored when it is not followed by a slash or backslash.
- The node name can be constructed using full Unicode character set (except the Supplementary code, not being supported by CLDC/CDC). However, using the full Unicode character set for node names is discouraged because the encoding in the underlying storage as well as the encoding needed in communications can create significant performance and memory usage overhead. Names that are restricted to the URI set [-a-zA-Z0-9_.!~*'()] are most efficient.
- URIs used in the DMT must be treated and interpreted as case sensitive.
- No End Slash: URI must not end with the delimiter slash ('/' \u002F). This implies that the root node must be denoted as "." and not "./".
- No parent denotation: URI must not be constructed using the character sequence "../" to traverse the tree upwards.
- Single Root: The character sequence "./" must not be used anywhere else but in the beginning of a URI.

117.15.13.1 public static final String PATH_SEPARATOR = "/"

This constant stands for a string identifying the path separator in the DmTree ("/").

Since 2.0

117.15.13.2 public static final char PATH_SEPARATOR_CHAR = 47

This constant stands for a char identifying the path separator in the DmTree ('/').

Since 2.0

117.15.13.3 public static final String ROOT_NODE = "."

This constant stands for a string identifying the root of the DmTree (".").

Since 2.0

117.15.13.4 public static final char ROOT_NODE_CHAR = 46

This constant stands for a char identifying the root of the DmTree ('.').

Since 2.0

117.15.13.5 public static String decode (String nodeName)

nodeName the node name to be decoded

□ Decode the node name so that back slash and forward slash are un-escaped from a back slash.

Returns the decoded node name

Since 2.0

117.15.13.6 public static String encode (String nodeName)

nodeName the node name to be encoded

□ Encode the node name so that back slash and forward slash are escaped with a back slash. This method is the reverse of decode(String).

Returns the encoded node name

Since 2.0

117.15.13.7 public static boolean isAbsoluteUri (String uri)

uri the URI to be checked, must not be null and must contain a valid URI

□ Checks whether the specified URI is an absolute URI. An absolute URI contains the complete path to a node in the DMT starting from the DMT root (".").

Returns whether the specified URI is absolute

Throws `NullPointerException` – if the specified URI is null

`IllegalArgumentException` – if the specified URI is malformed

117.15.13.8 public static boolean isValidUri (String uri)

uri the URI to be validated

□ Checks whether the specified URI is valid. A URI is considered valid if it meets the following constraints:

- the URI is not null;
- the URI follows the syntax defined for valid DMT URIs;

`getMaxUriLength()` and `getMaxSegmentNameLength()` methods.

Returns whether the specified URI is valid

117.15.13.9 public static String mangle (String nodeName)

nodeName the node name to be mangled (if necessary), must not be null or empty

□ Returns a node name that is valid for the tree operation methods, based on the given node name. This transformation is not idempotent, so it must not be called with a parameter that is the result of a previous mangle method call.

Node name mangling is needed in the following cases:

- if the name contains '/' or '\'characters

A node name that does not suffer from either of these problems is guaranteed to remain unchanged by this method. Therefore the client may skip the mangling if the node name is known to be valid (though it is always safe to call this method).

The method returns the normalized nodeName as described below. Invalid node names are normalized in different ways, depending on the cause. If the name contains '/' or '\'characters, then these are simply escaped by inserting an additional '\'before each occurrence. If the length of the name does exceed the limit, the following mechanism is used to normalize it:

- the SHA 1 digest of the name is calculated
- the digest is encoded with the base 64 algorithm
- all '/' characters in the encoded digest are replaced with '_'
- trailing '=' signs are removed

Returns the normalized node name that is valid for tree operations

Throws `NullPointerException` – if nodeName is null

`IllegalArgumentException` – if nodeName is empty

117.15.13.10 public static String[] toPath (String uri)

uri the URI to be split, must not be null

□ Split the specified URI along the path separator '/' characters and return an array of URI segments. Special characters in the returned segments are escaped. The returned array may be empty if the specified URI was empty.

Returns an array of URI segments created by splitting the specified URI

Throws `NullPointerException` – if the specified URI is null

`IllegalArgumentException` – if the specified URI is malformed

117.15.13.11 public static String toUri (String[] path)

path a possibly empty array of URI segments, must not be `null`

□ Construct a URI from the specified URI segments. The segments must already be mangled.

If the specified path is an empty array then an empty URI (“”) is returned.

Returns the URI created from the specified segments

Throws `NullPointerException` – if the specified path or any of its segments are `null`

`IllegalArgumentException` – if the specified path contains too many or malformed segments or the resulting URI is too long

117.16 org.osgi.service.dmt.spi

Device Management Tree SPI Package Version 2.0.

This package contains the interface classes that compose the Device Management SPI (Service Provider Interface). These interfaces are implemented by DMT plugins; users of the `DmtAdmin` interface do not interact directly with these.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle’s manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.dmt.spi; version=”[2.0,3.0)”`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.dmt.spi; version=”[2.0,2.1)”`

117.16.1 Summary

- `DataPlugin` – An implementation of this interface takes the responsibility of handling data requests in a subtree of the DMT.
- `ExecPlugin` – An implementation of this interface takes the responsibility of handling node execute requests requests in a subtree of the DMT.
- `MountPlugin` – This interface can be optionally implemented by a `DataPlugin` or `ExecPlugin` in order to get information about its absolute mount points in the overall DMT.
- `MountPoint` – This interface can be implemented to represent a single mount point.
- `ReadableDataSession` – Provides read-only access to the part of the tree handled by the plugin that created this session.
- `ReadWriteDataSession` – Provides non-atomic read-write access to the part of the tree handled by the plugin that created this session.
- `TransactionalDataSession` – Provides atomic read-write access to the part of the tree handled by the plugin that created this session.

117.16.2 Permissions

117.16.3 public interface DataPlugin

An implementation of this interface takes the responsibility of handling data requests in a subtree of the DMT.

In an OSGi environment such implementations should be registered at the OSGi service registry specifying the list of root node URIs in a `String` array or in case of a single value as `String` in the `dataRootURIs` registration parameter.

When the first reference in a session is made to a node handled by this plugin, the DmtAdmin calls one of the `open...` methods to retrieve a plugin session object for processing the request. The called method depends on the lock type of the current session. In case of `openReadWriteSession(String[], DmtSession)` and `openAtomicSession(String[], DmtSession)`, the plugin may return `null` to indicate that the specified lock type is not supported. In this case the DmtAdmin may call `openReadOnlySession(String[], DmtSession)` to start a read-only plugin session, which can be used as long as there are no write operations on the nodes handled by this plugin.

The `sessionRoot` parameter of each method is a String array containing the segments of the URI pointing to the root of the session. This is an absolute path, so the first segment is always ".". Special characters appear escaped in the segments.

117.16.3.1 public static final String DATA_ROOT_URIS = "dataRootURIs"

The string to be used as key for the " dataRootURIs" property when an DataPlugin is registered.

Since 2.0

117.16.3.2 public static final String MOUNT_POINTS = "mountPoints"

The string to be used as key for the mount points property when a DataPlugin is registered with mount points.

117.16.3.3 public TransactionalDataSession openAtomicSession (String[] sessionRoot , DmtSession session) throws DmtException

sessionRoot the path to the subtree which is locked in the current session, must not be `null`

session the session from which this plugin instance is accessed, must not be `null`

□ This method is called to signal the start of an atomic read-write session when the first reference is made within a `DmtSession` to a node which is handled by this plugin. Session information is given as it is needed for sending alerts back from the plugin.

The plugin can assume that there are no other sessions open on any subtree that has any overlap with the subtree of this session.

Returns a plugin session capable of executing read-write operations in an atomic block, or `null` if the plugin does not support atomic read-write sessions

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if sessionRoot points to a non-existing node
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if some underlying operation failed because of lack of permissions

117.16.3.4 public ReadableDataSession openReadOnlySession (String[] sessionRoot , DmtSession session) throws DmtException

sessionRoot the path to the subtree which is accessed in the current session, must not be `null`

session the session from which this plugin instance is accessed, must not be `null`

□ This method is called to signal the start of a read-only session when the first reference is made within a `DmtSession` to a node which is handled by this plugin. Session information is given as it is needed for sending alerts back from the plugin.

The plugin can assume that there are no writing sessions open on any subtree that has any overlap with the subtree of this session.

Returns a plugin session capable of executing read operations

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if sessionRoot points to a non-existing node
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if some underlying operation failed because of lack of permissions

117.16.3.5 public ReadWriteDataSession openReadWriteSession (String[] sessionRoot , DmtSession session) throws DmtException

sessionRoot the path to the subtree which is locked in the current session, must not be `null`

session the session from which this plugin instance is accessed, must not be `null`

□ This method is called to signal the start of a non-atomic read-write session when the first reference is made within a `DmtSession` to a node which is handled by this plugin. Session information is given as it is needed for sending alerts back from the plugin.

The plugin can assume that there are no other sessions open on any subtree that has any overlap with the subtree of this session.

Returns a plugin session capable of executing read-write operations, or `null` if the plugin does not support non-atomic read-write sessions

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if sessionRoot points to a non-existing node
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if some underlying operation failed because of lack of permissions

117.16.4 public interface ExecPlugin

An implementation of this interface takes the responsibility of handling node execute requests requests in a subtree of the DMT.

In an OSGi environment such implementations should be registered at the OSGi service registry specifying the list of root node URIs in a `String` array or in case of a single value as `String` in the `execRootURIs` registration parameter.

117.16.4.1 public static final String EXEC_ROOT_URIS = "execRootURIs"

The string to be used as key for the " execRootURIs" property when an ExecPlugin is registered.

Since 2.0

117.16.4.2 public static final String MOUNT_POINTS = "mountPoints"

The string to be used as key for the mount points property when an Exec Plugin is registered with mount points.

117.16.4.3 public void execute (DmtSession session , String[] nodePath , String correlator , String data) throws DmtException

session a reference to the session in which the operation was issued, must not be `null`

nodePath the absolute path of the node to be executed, must not be `null`

correlator an identifier to associate this operation with any alerts sent in response to it, can be `null`

data the parameter of the execute operation, can be `null`

□ Execute the given node with the given data. This operation corresponds to the EXEC command in OMA DM.

The semantics of an execute operation and the data parameter it takes depends on the definition of the managed object on which the command is issued. Session information is given as it is needed for sending alerts back from the plugin. If a correlation ID is specified, it should be used as the `correlator` parameter for alerts sent in response to this execute operation.

The `nodePath` parameter contains an array of path segments identifying the node to be executed in the subtree of this plugin. This is an absolute path, so the first segment is always ".". Special characters appear escaped in the segments.

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if the node does not exist
METADATA_MISMATCH if the command failed because of meta-data restrictions
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

See Also `DmtSession.execute(String, String)`, `DmtSession.execute(String, String, String)`

117.16.5 public interface MountPlugin

This interface can be optionally implemented by a `DataPlugin` or `ExecPlugin` in order to get information about its absolute mount points in the overall DMT.

This is especially interesting, if the plugin is mapped to the tree as part of a list. In such a case the id for this particular data plugin is determined by the DmtAdmin after the registration of the plugin and therefore unknown to the plugin in advance.

This is not a service interface, the Data or Exec Plugin does not also have to register this interface as a service, the Dmt Admin should use an `instanceof` to detect that a Plugin is also a Mount Plugin.

Since 2.0

117.16.5.1 public void mountPointAdded (MountPoint mountPoint)

mountPoint the newly mapped mount point

□ Provides the `MountPoint` describing the path where the plugin is mapped in the overall DMT. The given mountPoint is withdrawn with the `mountPointRemoved(MountPoint)` method. Corresponding mount points must compare equal and have an appropriate hash code.

117.16.5.2 public void mountPointRemoved (MountPoint mountPoint)

mountPoint The unmapped mount point array of `MountPoint` objects that have been removed from the mapping

□ Informs the plugin that the provided `MountPoint` objects have been removed from the mapping. The given mountPoint is withdrawn method. Mount points must compare equal and have an appropriate hash code with the given Mount Point in `mountPointAdded(MountPoint)`.

NOTE: attempts to invoke the `postEvent` method on the provided `MountPoint` must be ignored.

117.16.6 public interface MountPoint

This interface can be implemented to represent a single mount point.

It provides function to get the absolute mounted uri and a shortcut method to post events via the DmtAdmin.

Since 2.0

117.16.6.1 public boolean equals (Object other)

□ This object must provide a suitable hash function such that a Mount Point given in `MountPlugin.mountPointAdded(MountPoint)` is equal to the corresponding Mount Point in `MountPlugin.mountPointRemoved(MountPoint)`. `Object.equals(Object)`

117.16.6.2 public String[] getMountPath ( )

□ Provides the absolute mount path of this MountPoint

Returns the absolute mount path of this MountPoint

117.16.6.3 public int hashCode ( )

□ This object must provide a suitable hash function such that a Mount Point given in MountPlugin.mountPointAdded(MountPoint) has the same hashCode as the corresponding Mount Point in MountPlugin.mountPointRemoved(MountPoint). Object.hashCode()

117.16.6.4 public void postEvent (String topic , String[] relativeURIs , Dictionary properties)

topic the topic of the event to send. Valid values are:
org/osgi/service/dmt/DmtEvent/ADDED if the change was caused by a rename action org/osgi/service/dmt/DmtEvent/DELETED if the change was caused by a copy action org/osgi/service/dmt/DmtEvent/REPLACED if the change was caused by a copy action Must not be null.

relativeURIs an array of affected node URI's. All URI's specified here are relative to the current MountPoint's mountPath. The value of this parameter determines the value of the event property EVENT_PROPERTY_NODES. An empty array or null is permitted. In both cases the value of the events EVENT_PROPERTY_NODES property will be set to an empty array.

properties an optional parameter that can be provided to add properties to the Event that is going to be send by the DMTAdmin. If the properties contain a key EVENT_PROPERTY_NODES, then the value of this property is ignored and will be overwritten by relativeURIs.

□ Posts an event via the DmtAdmin about changes in the current plugins subtree.

This method distributes Events asynchronously to the EventAdmin as well as to matching local DmtEventListeners.

Throws IllegalArgumentException – if the topic has not one of the defined values

117.16.6.5 public void postEvent (String topic , String[] relativeURIs , String[] newRelativeURIs , Dictionary properties)

topic the topic of the event to send. Valid values are:
org/osgi/service/dmt/DmtEvent/RENAMED if the change was caused by a rename action org/osgi/service/dmt/DmtEvent/COPIED if the change was caused by a copy action Must not be null.

relativeURIs an array of affected node URI's. All URI's specified here are relative to the current MountPoint's mountPath. The value of this parameter determines the value of the event property EVENT_PROPERTY_NODES. An empty array or null is permitted. In both cases the value of the events EVENT_PROPERTY_NODES property will be set to an empty array.

newRelativeURIs an array of affected node URI's.The value of this parameter determines the value of the event property EVENT_PROPERTY_NEW_NODES. An empty array or null is permitted. In both cases the value of the events EVENT_PROPERTY_NEW_NODES property will be set to an empty array.

properties an optional parameter that can be provided to add properties to the Event that is going to be send by the DMTAdmin. If the properties contain the keys EVENT_PROPERTY_NODES or EVENT_PROPERTY_NEW_NODES, then the values of these properties are ignored and will be overwritten by relativeURIs and newRelativeURIs.

□ Posts an event via the DmtAdmin about changes in the current plugins subtree.

This method distributes Events asynchronously to the EventAdmin as well as to matching local DmtEventListeners.

Throws IllegalArgumentException – if the topic has not one of the defined values

117.16.7 public interface ReadableDataSession

Provides read-only access to the part of the tree handled by the plugin that created this session.

Since the ReadWriteDataSession and TransactionalDataSession interfaces inherit from this interface, some of the method descriptions do not apply for an instance that is only a ReadableDataSession. For example, the close() method description also contains information about its behaviour when invoked as part of a transactional session.

The nodePath parameters appearing in this interface always contain an array of path segments identifying a node in the subtree of this plugin. This parameter contains an absolute path, so the first segment is always ".". Special characters appear escaped in the segments.

Error handling

When a tree access command is called on the DmtAdmin service, it must perform an extensive set of checks on the parameters and the authority of the caller before delegating the call to a plugin. Therefore plugins can take certain circumstances for granted: that the path is valid and is within the subtree of the plugin and the session, the command can be applied to the given node (e.g. the target of getChildNodeNames is an interior node), etc. All errors described by the error codes DmtException.INVALID_URI, DmtException.URI_TOO_LONG, DmtException.PERMISSION_DENIED, DmtException.COMMAND_NOT_ALLOWED and DmtException.TRANSACTION_ERROR are fully filtered out before control reaches the plugin.

If the plugin provides meta-data for a node, the DmtAdmin service must also check the constraints specified by it, as described in MetaNode. If the plugin does not provide meta-data, it must perform the necessary checks for itself and use the DmtException.METADATA_MISMATCH error code to indicate such discrepancies.

The DmtAdmin does not check that the targeted node exists before calling the plugin. It is the responsibility of the plugin to perform this check and to throw a DmtException.NODE_NOT_FOUND if needed. In this case the DmtAdmin must pass through this exception to the caller of the corresponding DmtSession method.

The plugin can use the remaining error codes as needed. If an error does not fit into any other category, the DmtException.COMMAND_FAILED code should be used.

117.16.7.1 public void close ( ) throws DmtException

□ Closes a session. This method is always called when the session ends for any reason: if the session is closed, if a fatal error occurs in any method, or if any error occurs during commit or rollback. In case the session was invalidated due to an exception during commit or rollback, it is guaranteed that no methods are called on the plugin until it is closed. In case the session was invalidated due to a fatal exception in one of the tree manipulation methods, only the rollback method is called before this (and only in atomic sessions).

This method should not perform any data manipulation, only cleanup operations. In non-atomic read-write sessions the data manipulation should be done instantly during each tree operation, while in atomic sessions the DmtAdmin always calls TransactionalDataSession.commit() automatically before the session is actually closed.

Throws DmtException – with the error code COMMAND_FAILED if the plugin failed to close for any reason

117.16.7.2 public String[] getChildNodeNames (String[] nodePath) throws DmtException

nodePath the absolute path of the node

□ Get the list of children names of a node. The returned array contains the names - not the URIs - of the immediate children nodes of the given node. The returned array may contain null entries, but these are removed by the DmtAdmin before returning it to the client.

Returns the list of child node names as a string array or an empty string array if the node has no children

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if nodePath points to a non-existing node
METADATA_MISMATCH if the information could not be retrieved because of meta-data restrictions
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

117.16.7.3 public MetaNode getMetaNode (String[] nodePath) throws DmtException

nodePath the absolute path of the node

□ Get the meta data which describes a given node. Meta data can be only inspected, it can not be changed.

Meta data support by plugins is an optional feature. It can be used, for example, when a data plugin is implemented on top of a data store or another API that has their own metadata, such as a relational database, in order to avoid metadata duplication and inconsistency. The meta data specific to the plugin returned by this method is complemented by meta data from the DmtAdmin before returning it to the client. If there are differences in the meta data elements known by the plugin and the `DmtAdmin` then the plugin specific elements take precedence.

Note, that a node does not have to exist for having meta-data associated with it. This method may provide meta-data for any node that can possibly exist in the tree (any node defined by the Management Object provided by the plugin). For nodes that are not defined, a `DmtException` may be thrown with the `NODE_NOT_FOUND` error code. To allow easier implementation of plugins that do not provide meta-data, it is allowed to return `null` for any node, regardless of whether it is defined or not.

Returns a MetaNode which describes meta data information, can be `null` if there is no meta data available for the given node

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if nodeUri points to a node that is not defined in the tree (see above)
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

117.16.7.4 public int getNodeSize (String[] nodePath) throws DmtException

nodePath the absolute path of the leaf node

□ Get the size of the data in a leaf node. The value to return depends on the format of the data in the node, see the description of the `DmtData.getSize()` method for the definition of node size for each format.

Returns the size of the data in the node

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if nodePath points to a non-existing node
METADATA_MISMATCH if the information could not be retrieved because of meta-data restrictions
FEATURE_NOT_SUPPORTED if the Size property is not supported by the plugin
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

See Also `DmtData.getSize()`

117.16.7.5 public Date getNodeTimestamp (String[] nodePath) throws DmtException

nodePath the absolute path of the node

□ Get the timestamp when the node was last modified.

Returns the timestamp of the last modification

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if nodePath points to a non-existing node
METADATA_MISMATCH if the information could not be retrieved because of meta-data restrictions
FEATURE_NOT_SUPPORTED if the Timestamp property is not supported by the plugin
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

117.16.7.6 public String getNodeTitle (String[] nodePath) throws DmtException

nodePath the absolute path of the node

□ Get the title of a node. There might be no title property set for a node.

Returns the title of the node, or `null` if the node has no title

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if nodePath points to a non-existing node
METADATA_MISMATCH if the information could not be retrieved because of meta-data restrictions
FEATURE_NOT_SUPPORTED if the Title property is not supported by the plugin
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

117.16.7.7 public String getNodeType (String[] nodePath) throws DmtException

nodePath the absolute path of the node

□ Get the type of a node. The type of leaf node is the MIME type of the data it contains. The type of an interior node is a URI identifying a DDF document; a `null` type means that there is no DDF document overriding the tree structure defined by the ancestors.

Returns the type of the node, can be `null`

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if nodePath points to a non-existing node
METADATA_MISMATCH if the information could not be retrieved because of meta-data restrictions
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

117.16.7.8 public DmtData getNodeValue (String[] nodePath) throws DmtException

nodePath the absolute path of the node to retrieve

□ Get the data contained in a leaf or interior node.

Returns the data of the leaf node, must not be `null`

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if nodePath points to a non-existing node
METADATA_MISMATCH if the information could not be retrieved because of meta-data restrictions
FEATURE_NOT_SUPPORTED if the specified node is an interior node and does not support Java object values
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

117.16.7.9 public int getNodeVersion (String[] nodePath) throws DmtException

nodePath the absolute path of the node

□ Get the version of a node. The version can not be set, it is calculated automatically by the device. It is incremented modulo 0x10000 at every modification of the value or any other property of the node, for both leaf and interior nodes. When a node is created the initial value is 0.

Returns the version of the node

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if nodePath points to a non-existing node
METADATA_MISMATCH if the information could not be retrieved because of meta-data restrictions
FEATURE_NOT_SUPPORTED if the Version property is not supported by the plugin
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

117.16.7.10 public boolean isLeafNode (String[] nodePath) throws DmtException

nodePath the absolute path of the node

□ Tells whether a node is a leaf or an interior node of the DMT.

Returns true if the given node is a leaf node

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if nodePath points to a non-existing node
METADATA_MISMATCH if the information could not be retrieved because of meta-data restrictions
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

117.16.7.11 public boolean isNodeUri (String[] nodePath)

nodePath the absolute path to check

□ Check whether the specified path corresponds to a valid node in the DMT.

Returns true if the given node exists in the DMT

117.16.7.12 public void nodeChanged (String[] nodePath) throws DmtException

nodePath the absolute path of the node that has changed

□ Notifies the plugin that the given node has changed outside the scope of the plugin, therefore the Version and Timestamp properties must be updated (if supported). This method is needed because the ACL property of a node is managed by the DmtAdmin instead of the plugin. The DmtAdmin must call this method whenever the ACL property of a node changes.

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if nodePath points to a non-existing node
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

117.16.8 public interface ReadWriteDataSession extends ReadableDataSession

Provides non-atomic read-write access to the part of the tree handled by the plugin that created this session.

The `nodePath` parameters appearing in this interface always contain an array of path segments identifying a node in the subtree of this plugin. This parameter contains an absolute path, so the first segment is always ".". Special characters appear escaped in the segments.

Error handling

When a tree manipulation command is called on the DmtAdmin service, it must perform an extensive set of checks on the parameters and the authority of the caller before delegating the call to a plugin. Therefore plugins can take certain circumstances for granted: that the path is valid and is within the subtree of the plugin and the session, the command can be applied to the given node (e.g. the target of `setNodeValue` is a leaf node), etc. All errors described by the error codes `DmtException.INVALID_URI`, `DmtException.URI_TOO_LONG`, `DmtException.PERMISSION_DENIED`, `DmtException.COMMAND_NOT_ALLOWED` and `DmtException.TRANSACTION_ERROR` are fully filtered out before control reaches the plugin.

If the plugin provides meta-data for a node, the DmtAdmin service must also check the constraints specified by it, as described in `MetaNode`. If the plugin does not provide meta-data, it must perform the necessary checks for itself and use the `DmtException.METADATA_MISMATCH` error code to indicate such discrepancies.

The DmtAdmin does not check that the targeted node exists (or that it does not exist, in case of a node creation) before calling the plugin. It is the responsibility of the plugin to perform this check and to throw a `DmtException.NODE_NOT_FOUND` or `DmtException.NODE_ALREADY_EXISTS` if needed. In this case the DmtAdmin must pass through this exception to the caller of the corresponding DmtSession method.

The plugin can use the remaining error codes as needed. If an error does not fit into any other category, the `DmtException.COMMAND_FAILED` code should be used.

117.16.8.1 public void copy (String[] nodePath , String[] newNodePath , boolean recursive) throws DmtException

nodePath an absolute path specifying the node or the root of a subtree to be copied

newNodePath the absolute path of the new node or root of a subtree

recursive `false` if only a single node is copied, `true` if the whole subtree is copied

□ Create a copy of a node or a whole subtree. Beside the structure and values of the nodes, most properties managed by the plugin must also be copied, with the exception of the Timestamp and Version properties.

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if nodePath points to a non-existing node, or if newNodePath points to a node that cannot exist in the tree

NODE_ALREADY_EXISTS if newNodePath points to a node that already exists
METADATA_MISMATCH if the node could not be copied because of meta-data restrictions
FEATURE_NOT_SUPPORTED if the copy operation is not supported by the plugin
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

See Also `DmtSession.copy(String, String, boolean)`

117.16.8.2 public void createInteriorNode (String[] nodePath , String type) throws DmtException

nodePath the absolute path of the node to create

type the type URI of the interior node, can be `null` if no node type is defined

□ Create an interior node with a given type. The type of interior node, if specified, is a URI identifying a DDF document.

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if nodePath points to a node that cannot exist in the tree
NODE_ALREADY_EXISTS if nodeUri points to a node that already exists
METADATA_MISMATCH if the node could not be created because of meta-data restrictions
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

See Also `DmtSession.createInteriorNode(String)`, `DmtSession.createInteriorNode(String, String)`

117.16.8.3 public void createLeafNode (String[] nodePath , DmtData value , String mimeType) throws DmtException

nodePath the absolute path of the node to create

value the value to be given to the new node, can be `null`

mimeType the MIME type to be given to the new node, can be `null`

□ Create a leaf node with a given value and MIME type. If the specified value or MIME type is `null`, their default values must be taken.

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if nodePath points to a node that cannot exist in the tree
NODE_ALREADY_EXISTS if nodePath points to a node that already exists
METADATA_MISMATCH if the node could not be created because of meta-data restrictions
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

See Also `DmtSession.createLeafNode(String)`, `DmtSession.createLeafNode(String, DmtData)`, `DmtSession.createLeafNode(String, DmtData, String)`

117.16.8.4 public void deleteNode (String[] nodePath) throws DmtException

nodePath the absolute path of the node to delete

□ Delete the given node. Deleting interior nodes is recursive, the whole subtree under the given node is deleted.

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if nodePath points to a non-existing node
METADATA_MISMATCH if the node could not be deleted because of meta-data restrictions
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

See Also `DmtSession.deleteNode(String)`

117.16.8.5 public void renameNode (String[] nodePath , String newName) throws DmtException

nodePath the absolute path of the node to rename

newName the new name property of the node

□ Rename a node. This operation only changes the name of the node (updating the timestamp and version properties if they are supported), the value and the other properties are not changed. The new name of the node must be provided, the new path is constructed from the base of the old path and the given name.

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if nodePath points to a non-existing node, or if the new node is not defined in the tree
NODE_ALREADY_EXISTS if there already exists a sibling of nodePath with the name newName
METADATA_MISMATCH if the node could not be renamed because of meta-data restrictions
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

See Also `DmtSession.renameNode(String, String)`

117.16.8.6 public void setNodeTitle (String[] nodePath , String title) throws DmtException

nodePath the absolute path of the node

title the title text of the node, can be `null`

□ Set the title property of a node. The length of the title is guaranteed not to exceed the limit of 255 bytes in UTF-8 encoding.

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if nodePath points to a non-existing node
METADATA_MISMATCH if the title could not be set because of meta-data restrictions
FEATURE_NOT_SUPPORTED if the Title property is not supported by the plugin
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

See Also `DmtSession.setNodeTitle(String, String)`

117.16.8.7 public void setNodeType (String[] nodePath , String type) throws DmtException

nodePath the absolute path of the node *type* the type of the node, can be `null`

□ Set the type of a node. The type of leaf node is the MIME type of the data it contains. The type of an interior node is a URI identifying a DDF document.

For interior nodes, the `null` type should remove the reference (if any) to a DDF document overriding the tree structure defined by the ancestors. For leaf nodes, it requests that the default MIME type is used for the given node.

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if nodePath points to a non-existing node
METADATA_MISMATCH if the type could not be set because of meta-data restrictions
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

See Also `DmtSession.setNodeType(String, String)`

117.16.8.8 public void setNodeValue (String[] nodePath , DmtData data) throws DmtException

nodePath the absolute path of the node

data the data to be set, can be `null`

□ Set the value of a leaf or interior node. The format of the node is contained in the `DmtData` object. For interior nodes, the format is `FORMAT_NODE`, while for leaf nodes this format is never used.

If the specified value is `null`, the default value must be taken; if there is no default value, a `DmtException` with error code `METADATA_MISMATCH` must be thrown.

Throws `DmtException` – with the following possible error codes:
NODE_NOT_FOUND if nodePath points to a non-existing node
METADATA_MISMATCH if the value could not be set because of meta-data restrictions
FEATURE_NOT_SUPPORTED if the specified node is an interior node and does not support Java object values
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

See Also `DmtSession.setNodeValue(String, DmtData)`

117.16.9 public interface TransactionalDataSession extends ReadWriteDataSession

Provides atomic read-write access to the part of the tree handled by the plugin that created this session.

117.16.9.1 public void commit ( ) throws DmtException

□ Commits a series of DMT operations issued in the current atomic session since the last transaction boundary. Transaction boundaries are the creation of this object that starts the session, and all subsequent `commit()` and `rollback()` calls.

This method can fail even if all operations were successful. This can happen due to some multi-node semantic constraints defined by a specific implementation. For example, node A can be required to always have children A/B, A/C and A/D. If this condition is broken when `commit()` is executed, the method will fail, and throw a `METADATA_MISMATCH` exception.

In many cases the tree is not the only way to manage a given part of the system. It may happen that while modifying some nodes in an atomic session, the underlying settings are modified parallelly outside the scope of the DMT. If this is detected during commit, an exception with the code `CONCURRENT_ACCESS` is thrown.

Throws `DmtException` – with the following possible error codes
METADATA_MISMATCH if the operation failed because of meta-data restrictions
CONCURRENT_ACCESS if it is detected that some modification has been made outside the scope of the DMT to the nodes affected in the session's operations
DATA_STORE_FAILURE if an error occurred while accessing the data store
COMMAND_FAILED if some unspecified error is encountered while attempting to complete the command

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

117.16.9.2 public void rollback ( ) throws DmtException

□ Rolls back a series of DMT operations issued in the current atomic session since the last transaction boundary. Transaction boundaries are the creation of this object that starts the session, and all subsequent `commit` and `rollback` calls.

Throws `DmtException` – with the error code `ROLLBACK_FAILED` in case the rollback did not succeed

`SecurityException` – if the caller does not have the necessary permissions to execute the underlying management operation

117.17 org.osgi.service.dmt.notification

Device Management Tree Notification Package Version 2.0.

This package contains the public API of the Notification service. This service enables the sending of asynchronous notifications to management servers. Permission classes are provided by the `org.osgi.service.dmt.security` package.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.dmt.notification; version="[2.0,3.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.dmt.notification; version="[2.0,2.1)"
```

117.17.1 Summary

- `AlertItem` – Immutable data structure carried in an alert (client initiated notification).
- `NotificationService` – NotificationService enables sending aynchronous notifications to a management server.

117.17.2 Permissions

117.17.3 public class AlertItem

Immutable data structure carried in an alert (client initiated notification). The AlertItem describes details of various notifications that can be sent by the client, for example as alerts in the OMA DM protocol. The use cases include the client sending a session request to the server (alert 1201), the client notifying the server of completion of a software update operation (alert 1226) or sending back results in response to an asynchronous EXEC command.

The data syntax and semantics varies widely between various alerts, so does the optionality of particular parameters of an alert item. If an item, such as source or type, is not defined, the corresponding getter method returns null. For example, for alert 1201 (client-initiated session) all elements will be null.

The syntax used in AlertItem class corresponds to the OMA DM alert format. NotificationService implementations on other management protocols should map these constructs to the underlying protocol.

117.17.3.1 public AlertItem (String source , String type , String mark , DmtData data)

source the URI of the node which is the source of the alert item

type a MIME type or a URN that identifies the type of the data in the alert item

data a DmtData object that contains the format and value of the data in the alert item

mark the mark parameter of the alert item

□ Create an instance of the alert item. The constructor takes all possible data entries as parameters. Any of these parameters can be null. The semantics of the parameters may be refined by the definition of a specific alert, identified by its alert code (see NotificationService.sendNotification(String, int, String, AlertItem[])). In case of Generic Alerts for example (code 1226), the mark parameter contains a severity string.

117.17.3.2 public AlertItem (String[] source , String type , String mark , DmtData data)

source the path of the node which is the source of the alert item

type a MIME type or a URN that identifies the type of the data in the alert item

data a DmtData object that contains the format and value of the data in the alert item

mark the mark parameter of the alert item

□ Create an instance of the alert item, specifying the source node URI as an array of path segments. The constructor takes all possible data entries as parameters. Any of these parameters can be null. The semantics of the parameters may be refined by the definition of a specific alert, identified by its alert code (see NotificationService.sendNotification(String, int, String, AlertItem[])). In case of Generic Alerts for example (code 1226), the mark parameter contains a severity string.

117.17.3.3 public DmtData getData ( )

□ Get the data associated with the alert item. The returned DmtData object contains the format and the value of the data in the alert item. There might be no data associated with the alert item.

Returns the data associated with the alert item, or null if there is no data

117.17.3.4 public String getMark ( )

□ Get the mark parameter associated with the alert item. The interpretation of the mark parameter depends on the alert being sent, as identified by the alert code in NotificationService.sendNotification(String, int, String, AlertItem[]) . There might be no mark associated with the alert item.

Returns the mark associated with the alert item, or null if there is no mark

117.17.3.5 public String getSource ( )

- Get the node which is the source of the alert. There might be no source associated with the alert item.

Returns the URI of the node which is the source of this alert, or null if there is no source

117.17.3.6 public String getType ( )

- Get the type associated with the alert item. The type string is a MIME type or a URN that identifies the type of the data in the alert item (returned by getData()). There might be no type associated with the alert item.

Returns the type type associated with the alert item, or null if there is no type

117.17.3.7 public String toString ( )

- Returns the string representation of this alert item. The returned string includes all parameters of the alert item, and has the following format:

```
AlertItem(<source>, <type>, <mark>, <data>)
```

The last parameter is the string representation of the data value. The format of the data is not explicitly included.

Returns the string representation of this alert item

117.17.4 public interface NotificationService

NotificationService enables sending aynchronous notifications to a management server. The implementation of NotificationService should register itself in the OSGi service registry as a service.

117.17.4.1 public void sendNotification (String principal , int code , String correlator , AlertItem[] items) throws DmtException

principal the principal name which is the recipient of this notification, can be null

code the alert code, can be 0 if not needed

correlator optional field that contains the correlation identifier of an associated exec command, can be null if not needed

items the data of the alert items carried in this alert, can be null or empty if not needed

- Sends a notification to a named principal. It is the responsibility of the NotificationService to route the notification to the given principal using the registered org.osgi.service.dmt.notification.spi.RemoteAlertSender services.

In remotely initiated sessions the principal name identifies the remote server that created the session, this can be obtained using the session's getPrincipal call.

The principal name may be omitted if the client does not know the principal name. Even in this case the routing might be possible if the Notification Service finds an appropriate default destination (for example if it is only connected to one protocol adapter, which is only connected to one management server).

Since sending the notification and receiving acknowledgment for it is potentially a very time-consuming operation, notifications are sent asynchronously. This method should attempt to ensure that the notification can be sent successfully, and should throw an exception if it detects any problems. If the method returns without error, the notification is accepted for sending and the implementation must make a best-effort attempt to deliver it.

In case the notification is an asynchronous response to a previous execute command, a correlation identifier can be specified to provide the association between the execute and the notification.

In order to send a notification using this method, the caller must have an AlertPermission with a target string matching the specified principal name. If the principal parameter is null (the principal name is not known), the target of the AlertPermission must be "*".

When this method is called with null correlator, null or empty AlertItem array, and a 0 code as values, it should send a protocol specific default notification to initiate a management session. For example, in case of OMA DM this is alert 1201 "Client Initiated Session". The principal parameter can be used to determine the recipient of the session initiation request.

Throws DmtException – with the following possible error codes:
UNAUTHORIZED when the remote server rejected the request due to insufficient authorization
ALERT_NOT_ROUTED when the alert can not be routed to the given principal
REMOTE_ERROR in case of communication problems between the device and the destination
COMMAND_FAILED for unspecified errors encountered while attempting to complete the command
FEATURE_NOT_SUPPORTED if the underlying management protocol doesn't support asynchronous notifications

SecurityException – if the caller does not have the required AlertPermission with a target matching the principal parameter, as described above

117.18 org.osgi.service.dmt.notification.spi

Device Management Tree Notification SPI Package Version 2.0.

This package contains the SPI (Service Provider Interface) of the Notification service. These interfaces are implemented by Protocol Adapters capable of delivering notifications to management servers on a specific protocol. Users of the NotificationService interface do not interact directly with this package.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.dmt.notification.spi; version="[2.0,3.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.dmt.notification.spi; version="[2.0,2.1)"
```

117.18.1 public interface RemoteAlertSender

The RemoteAlertSender can be used to send notifications to (remote) entities identified by principal names. This service is provided by Protocol Adapters, and is used by the org.osgi.service.dmt.notification.NotificationService when sending alerts. Implementations of this interface have to be able to connect and send alerts to one or more management servers in a protocol specific way.

The properties of the service registration should specify a list of destinations (principals) where the service is capable of sending alerts. This can be done by providing a String array of principal names in the principals registration property. If this property is not registered, the service will be treated as the default sender. The default alert sender is only used when a more specific alert sender cannot be found.

The principals registration property is used when the org.osgi.service.dmt.notification.NotificationService.sendNotification(String, int, String, AlertItem[]) method is called, to find the proper RemoteAlertSender for the given destination. If the caller does not specify a principal, the alert is only sent if the Notification Sender finds a default alert sender, or if the choice is unambiguous for some other reason (for example if only one alert sender is registered).

117.18.1.1 public void sendAlert (String principal , int code , String correlator , AlertItem[] items) throws Exception

principal the name identifying the server where the alert should be sent, can be null

code the alert code, can be 0 if not needed

correlator the correlation identifier of an associated EXEC command, or null if there is no associated EXEC

items the data of the alert items carried in this alert, can be empty or null if no alert items are needed

□ Sends an alert to a server identified by its principal name. In case the alert is sent in response to a previous execute command, a correlation identifier can be specified to provide the association between the execute and the alert.

The principal parameter specifies which server the alert should be sent to. This parameter can be null if the client does not know the name of the destination. The alert should still be delivered if possible; for example if the alert sender is only connected to one destination.

Any exception thrown on this method will be propagated to the original sender of the event, wrapped in a DmtException with the code REMOTE_ERROR.

Since sending the alert and receiving acknowledgment for it is potentially a very time-consuming operation, alerts are sent asynchronously. This method should attempt to ensure that the alert can be sent successfully, and should throw an exception if it detects any problems. If the method returns without error, the alert is accepted for sending and the implementation must make a best-effort attempt to deliver it.

Throws Exception – if the alert can not be sent to the server

117.19 org.osgi.service.dmt.security

Device Management Tree Security Package Version 2.0.

This package contains the permission classes used by the Device Management API in environments that support the Java 2 security model.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.dmt.security; version="[2.0,3.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.dmt.security; version="[2.0,2.1)"

117.19.1 Summary

- AlertPermission – Indicates the callers authority to send alerts to management servers, identified by their principal names.
- DmtPermission – Controls access to management objects in the Device Management Tree (DMT).

- DmtPrincipalPermission – Indicates the callers authority to create DMT sessions on behalf of a remote management server.

117.19.2 Permissions

117.19.3 public class AlertPermission extends Permission

Indicates the callers authority to send alerts to management servers, identified by their principal names.

AlertPermission has a target string which controls the principal names where alerts can be sent. A wildcard is allowed at the end of the target string, to allow sending alerts to any principal with a name matching the given prefix. The "*" target means that alerts can be sent to any destination.

117.19.3.1 public AlertPermission (String target)

target the name of a principal, can end with * to match any principal identifier with the given prefix

□ Creates a new AlertPermission object with its name set to the target string. Name must be non-null and non-empty.

Throws NullPointerException – if name is null

IllegalArgumentException – if name is empty

117.19.3.2 public AlertPermission (String target , String actions)

target the name of the server, can end with * to match any server identifier with the given prefix

actions no actions defined, must be "*" for forward compatibility

□ Creates a new AlertPermission object using the 'canonical' two argument constructor. In this version this class does not define any actions, the second argument of this constructor must be "*" so that this class can later be extended in a backward compatible way.

Throws NullPointerException – if name or actions is null

IllegalArgumentException – if name is empty or actions is not "*"

117.19.3.3 public boolean equals (Object obj)

obj the object to compare to this AlertPermission instance

□ Checks whether the given object is equal to this AlertPermission instance. Two AlertPermission instances are equal if they have the same target string.

Returns true if the parameter represents the same permissions as this instance

117.19.3.4 public String getActions ( )

□ Returns the action list (always * in the current version).

Returns the action string "*"

117.19.3.5 public int hashCode ( )

□ Returns the hash code for this permission object. If two AlertPermission objects are equal according to the equals(Object) method, then calling this method on each of the two AlertPermission objects must produce the same integer result.

Returns hash code for this permission object

117.19.3.6 public boolean implies (Permission p)

p the permission to check for implication

- Checks if this AlertPermission object implies the specified permission. Another AlertPermission instance is implied by this permission either if the target strings are identical, or if this target can be made identical to the other target by replacing a trailing "*" with any string.

Returns true if this AlertPermission instance implies the specified permission

117.19.3.7 public PermissionCollection newPermissionCollection ( )

- Returns a new PermissionCollection object for storing AlertPermission objects.

Returns the new PermissionCollection

117.19.4 public class DmtPermission extends Permission

Controls access to management objects in the Device Management Tree (DMT). It is intended to control local access to the DMT. DmtPermission target string identifies the management object URI and the action field lists the OMA DM commands that are permitted on the management object. Example:

```
DmtPermission("./OSGi/bundles", "Add,Replace,Get");
```

This means that owner of this permission can execute Add, Replace and Get commands on the ./OSGi/bundles management object. It is possible to use wildcards in both the target and the actions field. Wildcard in the target field means that the owner of the permission can access children nodes of the target node. Example:

```
DmtPermission("./OSGi/bundles/*", "Get");
```

This means that owner of this permission has Get access on every child node of ./OSGi/bundles. The asterix does not necessarily have to follow a '/' character. For example the "./OSGi/a*" target matches the ./OSGi/applications subtree.

If wildcard is present in the actions field, all legal OMA DM commands are allowed on the designated nodes(s) by the owner of the permission. Action names are interpreted case-insensitively, but the canonical action string returned by getActions() uses the forms defined by the action constants.

117.19.4.1 public static final String ADD = "Add"

Holders of DmtPermission with the Add action present can create new nodes in the DMT, that is they are authorized to execute the createInteriorNode() and createLeafNode() methods of the DmtSession. This action is also required for the copy() command, which needs to perform node creation operations (among others).

117.19.4.2 public static final String DELETE = "Delete"

Holders of DmtPermission with the Delete action present can delete nodes from the DMT, that is they are authorized to execute the deleteNode() method of the DmtSession.

117.19.4.3 public static final String EXEC = "Exec"

Holders of DmtPermission with the Exec action present can execute nodes in the DMT, that is they are authorized to call the execute() method of the DmtSession.

117.19.4.4 public static final String GET = "Get"

Holders of DmtPermission with the Get action present can query DMT node value or properties, that is they are authorized to execute the isLeafNode(), getNodeAcl(), getEffectiveNodeAcl(), getMetaNode(), getNodeValue(), getChildNodeNames(), getNodeTitle(), getNodeVersion(), getNodeTimeStamp(), getNodeSize() and getNodeType() methods of the DmtSession. This action is also required for the copy() command, which needs to perform node query operations (among others).

117.19.4.5 public static final String REPLACE = "Replace"

Holders of DmtPermission with the Replace action present can update DMT node value or properties, that is they are authorized to execute the setNodeAcl(), setNodeTitle(), setNodeValue(), setNodeType() and renameNode() methods of the DmtSession. This action is also be required for the copy() command if the original node had a title property (which must be set in the new node).

117.19.4.6 public DmtPermission (String dmtUri , String actions)

dmtUri URI of the management object (or subtree)

actions OMA DM actions allowed

□ Creates a new DmtPermission object for the specified DMT URI with the specified actions. The given URI can be:

- "`*`", which matches all valid (see `Uri.isValidUri(String)`) absolute URIs;
- the prefix of an absolute URI followed by the * character (for example "`./OSGi/L*`"), which matches all valid absolute URIs beginning with the given prefix;
- a valid absolute URI, which matches itself.

Since the * character is itself a valid URI character, it can appear as the last character of a valid absolute URI. To distinguish this case from using * as a wildcard, the * character at the end of the URI must be escaped with the \ charater. For example the URI "`./a*`" matches "`./a`", "`./aa`", "`./a/b`" etc. while "`./a\*`" matches "`./a*`" only.

The actions string must either be "*" to allow all actions, or it must contain a non-empty subset of the valid actions, defined as constants in this class.

Throws `NullPointerException` – if any of the parameters are `null`

`IllegalArgumentException` – if any of the parameters are invalid

117.19.4.7 public boolean equals (Object obj)

obj the object to compare to this DmtPermission instance

□ Checks whether the given object is equal to this DmtPermission instance. Two DmtPermission instances are equal if they have the same target string and the same action mask. The "*" action mask is considered equal to a mask containing all actions.

Returns `true` if the parameter represents the same permissions as this instance

117.19.4.8 public String getActions ( )

□ Returns the String representation of the action list. The allowed actions are listed in the following order: Add, Delete, Exec, Get, Replace. The wildcard character is not used in the returned string, even if the class was created using the "*" wildcard.

Returns canonical action list for this permission object

117.19.4.9 public int hashCode ( )

□ Returns the hash code for this permission object. If two DmtPermission objects are equal according to the `equals(Object)` method, then calling this method on each of the two DmtPermission objects must produce the same integer result.

Returns hash code for this permission object

117.19.4.10 public boolean implies (Permission p)

p the permission to check for implication

□ Checks if this DmtPermission object "implies" the specified permission. This method returns `false` if and only if at least one of the following conditions are fulfilled for the specified permission:

- it is not a DmtPermission

- its set of actions contains an action not allowed by this permission
- the set of nodes defined by its path contains a node not defined by the path of this permission

Returns true if this DmtPermission instance implies the specified permission

117.19.4.11 public PermissionCollection newPermissionCollection ( )

□ Returns a new PermissionCollection object for storing DmtPermission objects.

Returns the new PermissionCollection

117.19.5 public class DmtPrincipalPermission extends Permission

Indicates the callers authority to create DMT sessions on behalf of a remote management server. Only protocol adapters communicating with management servers should be granted this permission.

`DmtPrincipalPermission` has a target string which controls the name of the principal on whose behalf the protocol adapter can act. A wildcard is allowed at the end of the target string, to allow using any principal name with the given prefix. The "*" target means the adapter can create a session in the name of any principal.

117.19.5.1 public DmtPrincipalPermission (String target)

target the name of the principal, can end with * to match any principal with the given prefix

□ Creates a new `DmtPrincipalPermission` object with its name set to the target string. Name must be non-null and non-empty.

Throws `NullPointerException` – if `name` is `null`

`IllegalArgumentException` – if `name` is empty

117.19.5.2 public DmtPrincipalPermission (String target , String actions)

target the name of the principal, can end with * to match any principal with the given prefix

actions no actions defined, must be "*" for forward compatibility

□ Creates a new `DmtPrincipalPermission` object using the 'canonical' two argument constructor. In this version this class does not define any actions, the second argument of this constructor must be "*" so that this class can later be extended in a backward compatible way.

Throws `NullPointerException` – if `name` or `actions` is `null`

`IllegalArgumentException` – if `name` is empty or `actions` is not "*"

117.19.5.3 public boolean equals (Object obj)

obj the object to compare to this DmtPrincipalPermission instance

□ Checks whether the given object is equal to this DmtPrincipalPermission instance. Two DmtPrincipalPermission instances are equal if they have the same target string.

Returns `true` if the parameter represents the same permissions as this instance

117.19.5.4 public String getActions ( )

□ Returns the action list (always * in the current version).

Returns the action string "*"

117.19.5.5 public int hashCode ( )

□ Returns the hash code for this permission object. If two DmtPrincipalPermission objects are equal according to the `equals(Object)` method, then calling this method on each of the two DmtPrincipalPermission objects must produce the same integer result.

Returns hash code for this permission object

117.19.5.6 public boolean implies (Permission p)

p the permission to check for implication

□ Checks if this DmtPrincipalPermission object implies the specified permission. Another DmtPrincipalPermission instance is implied by this permission either if the target strings are identical, or if this target can be made identical to the other target by replacing a trailing "*" with any string.

Returns true if this DmtPrincipalPermission instance implies the specified permission

117.19.5.7 public PermissionCollection newPermissionCollection ( )

□ Returns a new PermissionCollection object for storing DmtPrincipalPermission objects.

Returns the new PermissionCollection

117.20 References

[1] *OMA DM-TND v1.2 draft*
http://member.openmobilealliance.org/ftp/public_documents/dm/Permanent_documents/OMA-TS-DM-TND-V1_2-20050615-C.zip

[2] *OMA DM-RepPro v1.2 draft:*
http://member.openmobilealliance.org/ftp/public_documents/dm/Permanent_documents/OMA-DM-RepPro-V1_2_0-20050131-D.zip

[3] *IETF RFC2578. Structure of Management Information*
Version 2 (SMIv2), http://www.ietf.org/rfc/rfc2578.txt

[4] *Java™ Management Extensions Instrumentation and Agent Specification v1.2, October 2002,*
http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html

[5] *JSR 9 - Federated Management Architecture (FMA) Specification*
Version 1.0, January 2000, http://www.jcp.org/en/jsr/detailid=9

[6] *WBEM Profile Template, DSP1000*
Status: Draft, Version 1.0 Preliminary, March 11, 2004
http://www.dmtf.org/standards/wbem

[7] *SNMP*
http://www.wtcs.org/snmp4tpc/snmp_rfc.htm#rfc

[8] *RFC 2396 Uniform Resource Identifiers (URI): Generic Syntax*
http://www.ietf.org/rfc/rfc2396.txt

[9] *MIME Media Types*
http://www.iana.org/assignments/media-types/

[10] *RFC 3548 The Base16, Base32, and Base64 Data Encodings*
http://www.ietf.org/rfc/rfc3548.txt

[11] *Secure Hash Algorithm 1*
http://www.itl.nist.gov/fipspubs/fip180-1.htm

[12] *TR-069 CPE WAN Management Protocol (CWMP)*
Customer Premises Equipment Wide Area Network Management Protocol (CWMP)
http://en.wikipedia.org/wiki/TR-069

[13] *XML Schema Part 2: Datatypes Second Edition*
http://www.w3.org/TR/2004/PER-xmlschema-2-20040318/

131 TR069 Connector Service Specification

Version 1.0

131.1 Introduction

This chapter provides a specification for the TR069 Connector, an assistant to a Protocol Adapter based on [1] *TR-069 Amendment 3*. A TR069 Connector provides a mapping of TR-069 concepts to/from the *Dmt Admin Service Specification* on page 285. It mainly handles the low level details of Object/ Parameter Name to Dmt Admin URI mapping, and vice versa. TR-069 Protocol Adapter developers can use this service to simplify the use the Dmt Admin service. The TR069 Connector service is based on the definition of a Protocol Mapping in *Protocol Mapping* on page 328. It is assumed that the reader understands TR-069 and has a basic understanding of the Dmt Admin service.

The examples in this specification are not from a Broadband Forum Technical Report and are purely fictional.

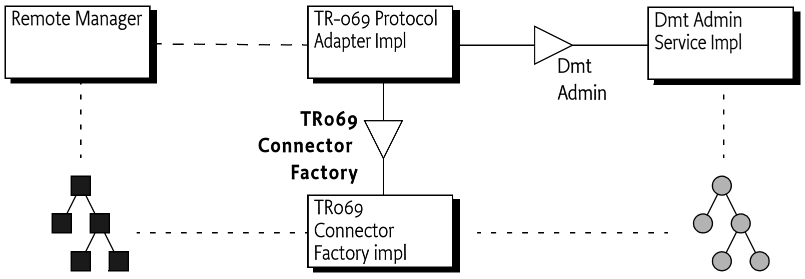
131.1.1 Essentials

- *Connector* – Provide a TR-069 view on top of the Dmt Admin service.
- *Simplify* – Simplify the handling of data models implemented through the DMT through the TR-069 protocol.
- *Browse* – Implement the constructs for MAP and LIST handling.
- *Native* – Provide a mechanism for Data Plugins to convey conversion information to the Protocol Adapter so that native TR-069 object models can be implemented as Data Plugins.

131.1.2 Entities

- TR069ConnectorFactory – Provides a way to create a TR069Connector that is bound to an active Dmt Session.
- TR069Connector – Created by TR069ConnectorFactory on a Dmt Session; provides methods that helps in using the TR-069 namespace and RPCs on a Dmt Admin DMT.
- ParameterValue – The value of a parameter, maps to the TR-069 ParameterValueStruct.
- ParameterInfo – Information about the parameter, maps to the TR-069 ParameterInfoStruct.
- DMT – The Device Management Tree as available through the Dmt Admin service.

Figure 131.1 *TR-069 Entities*



131.1.3 Synopsis

A TR-069 Protocol Adapter first creates a Dmt Session on the node in the DMT that maps to an object model that should be visible to the TR-069 Management Server. A Protocol Adapter can choose to map a whole sub-tree or it can create a virtual object model based on different nodes, this depends on the implementation of the Protocol Adapter.

When a TR-069 RPC arrives, the Protocol Adapter must parse the SOAP message and analyze the request. In general, an RPC can request the update or retrieval of multiple values. The Protocol Adapter must decompose these separate requests into single requests and execute them as a single unit. If the request is a retrieval or update of a data model maintained in the Dmt Admin service then the Protocol Adapter can use a TR069 Connector to simplify implementing this request. The TR069 Connector Factory service can be used to create an instance of a TR069 Connector that is based on a specific Dmt Session.

The TR069 Connector maps the Object or Parameter Name to a URI and perform the requested operation on the corresponding node. The name-to-URI conversion supports the LIST and MAP concepts as defined in *OSGi Object Modeling* on page 327.

The TR069 Connector handles conversion from the Dmt Admin data types to the TR-069 data types. There is a default mapping for the standard Dmt Admin formats including the comma separated list supported by TR-069. However, Data Plugins that implement TR-069 aware object models can instruct the TR069 Connector by providing specific MIME types on the Meta Node.

Objects can be added and deleted but are, in general, not added immediately. These objects are lazily created when they are accessed. The reason is that TR-069 does not support the concept of a session with atomic semantics, a fact leveraged by certain object models in the DMT. Therefore, adding an object will assign a instance id to an object but the creation of the object is delayed until the object is used.

After all the requests in an RPC are properly handled the TR069 Connector must be closed, the Dmt Session must be closed separately.

Errors are reported to the caller as they happen, if a Dmt Admin service error is fatal then the Dmt Session will be closed and it will be necessary to create a new TR069 Connector.

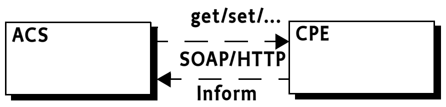
131.2 TR-069 Protocol Primer

The [6] *Broadband Forum* is an organization for broadband wire-line solutions. They develop multi-service broadband packet networking specifications addressing inter-operability, architecture, and management. Their specifications enable home, business and converged broadband services, encompassing customer, access and backbone networks. One of the specifications of the Broadband Forum is the *Technical Report No 69*, also called TR-069, a specification of a management model.

131.2.1 Architecture

[1] *TR-069 Amendment 3* is a technical report (Broadband Forum's specification model) that specifies a management protocol based on [4] *SOAP 1.1* over HTTP. The TR-069 technical report defines a number of mandatory Remote Procedure Calls (RPCs) that allow a management system, the Auto-Configuration Server (ACS), to discover the capabilities of the Consumer Premises Equipment (CPE) and do basic management. This model is depicted in Figure 131.2.

Figure 131.2 TR-069 Reference Architecture



In TR-069, the CPE is always initiating the conversation with the ACS though the ACS can request a session.

Inside the CPE there is a Protocol Adapter that implements the TR-069 RPCs. These RPCs read and modify the objects models present in the CPE. There is usually a mechanism that allows the different modules in the CPE to contribute a management object to the Protocol Adapter so that the Protocol Adapter does not require knowledge about highly specialized domains.

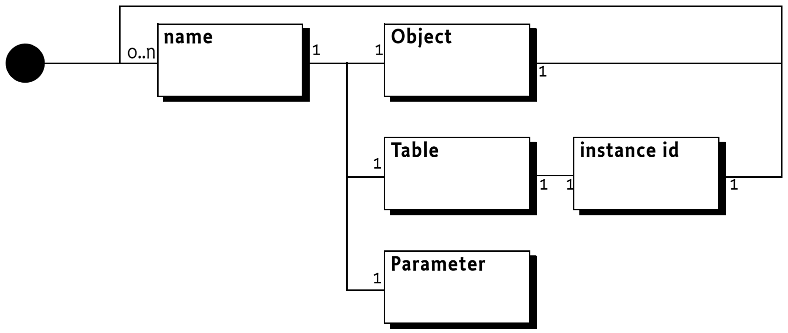
[2] *TR-106 Amendment 3* specifies object model guidelines to be followed by all TR-069-enabled devices as well as a formal model to document these object models.

131.2.2 Object Model

The object model of TR-069 consists of *objects* that contain *parameters* as well as *tables* that contain objects. TR-106 says:

- *Object* – A named collection of parameters and/or other objects.
- *Parameter* – A name-value pair.
- *Table* – An enumeration of objects identified by an instance id.

Figure 131.3 Type Model TR-069



Objects can also occur in tables, in that case the object name is suffixed with an *instance id*. An object that has no instance id is a singleton, with an instance id they are referred to as *tables*. In the Broadband Forum technical reports tables end in the special suffix {i}, the instance id.

This provides the following structural definition for this specification:

```
named-value     ::= NAME ( object | table | parameter )
object          ::= named-value +
table           ::= ( instance object )*
parameter       ::=
instance        ::= INTEGER > 0
```

TR-069 talks about partial paths and parameter names. In this specification, a *name* is reserved for the short relative name used inside an object, also called the *local name*. The term *path* is reserved for the combination of object names, table names, and instance ids that are separated by a full stop ('.'\u022D) and used to traverse an instance model.

```
path            ::= parameter-path | object-path | table-path
segment         ::= NAME '.' ( instance '.' )?
object-path      ::= segment+
table-path      ::= segment* NAME '.'   // expect INTEGER next
parameter-path  ::= object-path NAME
instance-path   ::= table-path instance '.'
```

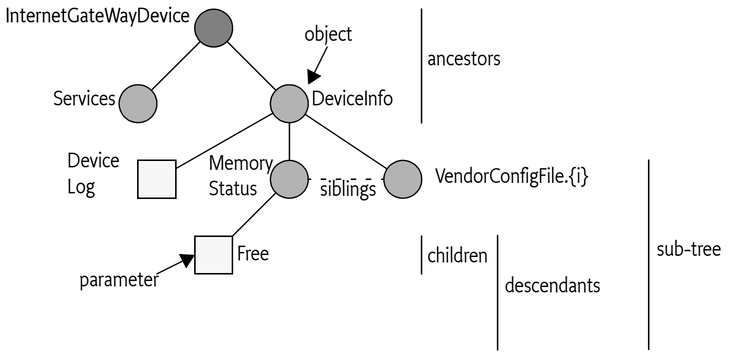
In this specification the following terms are used consistently:

- *Object* – Refers to a named type defining a certain set of parameters, objects, and tables.

- *Table* – A list of instances for a given object.
- *Instance* – An object element in a table at a certain id.
- *Instance Id* – The integer id used to identify an instance in a table.
- *Alias* – A name chosen by the ACS that uniquely identifies an instance.
- *Singleton* – An object that is not in a table.
- *Name* – The name of an object, table, or parameter refers to the local name only and not the path.
- *Segment* – A component in a path that always ends in a full stop. A segment can contain instance ids to identify an instance.
- *Path* – A string uniquely identifying a path in the tree to either a parameter, an object, or a table.
- *Object Path* – A path that uniquely identifies an instance or a singleton. An object path must always ends in a full stop. This maps to the TR-069 concept of an `ObjectName`.
- *Parameter Path* – The name of the parameter preceded by the owning object. A path that does not end in a full stop is always a parameter path.
- *Table Path* – An object path that lacks the last instance id. In TR-069 this is also sometimes called a partial path. The last segment is an object path that must be followed by an instance id to address an instance.
- *Instance Path* – A path to an instance in a table

This provides a hierarchy as depicted in Figure 131.4.

Figure 131.4 TR-069 Object and Parameter naming relative to the parameter MemoryStatus



131.2.3 Parameter Names

The grammars for parameter names and object names are as follows:

```
NAME          ::= (Letter | '_' )
                       ( Letter | Digit | '-' | '_' | CombiningChar | Extender)*
```

The productions `Letter`, `Digit`, `CombiningChar`, and `Extender` are defined in [5] *Extensible Markup Language (XML) 1.0 (Second Edition)*. The name basically supports the full unicode character set for letters and digits (including digits for other languages), including sets for languages like Hebrew and Chinese. Examples of different parameter names are:

```
name             // simple name
Name             // case sensitive

_
_-_-_
ångstrom
þingsten
ΨΣΩΠ
```

131.2.4 Parameter Type

A parameter value can have one of the data types defined in[2] *TR-106 Amendment 3*, they are summarized in Table 131.1

Table 131.1 TR-106 Data types

TR-106 Type	Description
object	Represents a structured type
string	A Unicode string, optionally restricted in length
int	32 bit integer
long	64 bit integer
unsignedInt	32 bit unsigned integer
unsignedLong	64 bit unsigned integer
boolean	Can have values 0 or false (false) or 1 or true (true)
dateTime	TR-069 recognizes three different date times. These three cases are differentiated in the following way: • *Unknown time* – If the time is not known. • *Relative time* – Relative time is the time since boot time. • *Absolute time* – Normal date and time.
base64	An array of bytes
hexBinary	An array of bytes

SOAP messages always provide a type for the parameter value. For example:

```
<ParameterValueStruct>
  <name>Parameter1</name>
  <value xsi:type="long">1234</value>
</ParameterValueStruct>
```

The xsi prefix refers to the http://www.w3.org/2001/XMLSchema-instance namespace. However, this makes not all TR-106 types well defined, for example in XML Schema base64 is called base64Binary. This specification assumes that the names and definitions in Table 131.1 and provides appropriate constants for the Protocol Adapter.

Parameters can be read-only or read-write. All writable Parameters must also be readable although security can cause certain parameters to be read as an empty string, for example passwords. Parameters can reflect configuration as well as status of the device. External causes can cause parameters to change at any time. The TR-069 protocol has the facility to call an Inform RPC to provide the ACS with a notification of changed parameters.

131.2.5 Parameter Attributes

Parameter attributes provide the meta data for a parameter. In TR-069, the attributes are used to manage notifications and access control. Each parameter in TR-069 can be watched by the ACS by setting the corresponding parameter attribute to *active* or *passive notifications*. Passive notifications are passed whenever the CPE communicates with the ACS and active notifications initiate a session. Parameters that have a notification are said to be *watched*.

Access to the parameters can be managed by setting Access Control Lists via the corresponding parameter attribute.

131.2.6 Objects and Tables

TR-106 has the concept of an *object* stored in a *table* to allow multiple instances of the same type. It is part of the object definition if it is stored in a table or not. An object cannot both appear as a table instance and as a singleton.

Each instance in the table is addressed with an integer >= 1. This *instance id* is not chosen by the ACS since it can be required to create a new instance due to an external event. For example the user plugging in a USB device or starting a new VOIP session. The ACS must discover these instance ids by asking the device for the instance ids in a table.

For example, the parameter path `Device.LAN.DHCPOption.4.Request` refers to a parameter on a `DHCPOption` object that has the instance id 4. Instance ids are not sequential nor predictable. It is the responsibility of the device to choose an instance id when an object is created. Instance ids are assumed to be persistent so that the ACS can cache results from a discovery process.

Newer TR-069 objects have been given an `Alias` parameter. This alias uniquely identifies the table instance.

TR-069 defines a convention for a parameter that contains the number of entries in a table. Any parameter name that ends with `NumberOfEntries` contains the number of entries in a table with the name of the prefix in the same object. For example `A.B.CNumberOfEntries` provides the number of entries in the table:

```
A.B.C.
```

131.2.7 RPCs

The object model implemented in a device is accessed and modified with *RPCs*. RPCs are remote procedure calls; a way to invoke a function remotely. TR-069 defines a number of mandatory RPCs and provides a mechanism to extend and discover the set of RPCs implemented by a CPE. The mandatory RPCs are listed in Table 131.2.

Table 131.2 TR-069 RPCs

RPC	Description
`GetRPCMethods`	Return a list of RPC methods
`SetParameterValues`	Set one or more parameter values
`GetParameterValues`	Get one or more parameter values
`GetParameterNames`	Get the parameter information for a parameter, object, or table.
`SetParameterAttributes`	Set parameter attributes
`GetParameterAttributes`	Get parameter attributes
`AddObject`	Add a new object to a table
`DeleteObject`	Delete an object from a table
`Download`	Download software/firmware
`Reboot`	Reboot the device

131.2.8 Authentication

The security model of TR-069 is based around the authentication taking place during the setup of a TLS (formerly SSL) connection. This authentication is then used to manage the access control lists via the parameter attributes.

131.2.9 Sessions and Transactions

A *session* with the ACS is always initiated by the CPE. The ACS can request a session, but it is always the CPE that starts a session by opening the connection to the ACS and then sending an `Inform` RPC. The session ends when the connection is closed, which happens after the ACS has informed the CPE it has no more requests.

During a session, a CPE has the requirements that parameters must not change due to other sources than the session and that the parameters are consistent with the changes. However, there is no transactionality over the session, atomicity is only guaranteed for one RPC. An RPC can consist of multiple parameter modifications that should therefore be atomically applied.

131.2.10 Events and Notifications

TR-069 sessions always start with an `Inform` RPC from the CPE to the ACS. This RPC contains any events and notifications for parameters that were watched. Events signal crucial state changes from the CPE to the ACS. For example, if a device has rebooted it will inform the ACS. Notifications are caused by parameter changes, the `Inform` RPC contains a list of events and parameters with changed values.

131.2.11 Errors

Invoked RPCs can return a fault status if errors occur during the execution of the RPC. For ACS to CPE RPCs these fault codes start at 9000, for the reverse direction they start at 8000. Each RPC defines the fault codes that can occur and their semantics in that context.

131.3 TR069 Connector

A TR-069 Protocol Adapter must be able to browse foreign Data Plugins on the device and support native TR069 objects models implemented by a Data Plugin. As Data Plugins are available through the Dmt Admin service, the Protocol Adapter must provide a bi-directional mapping between Dmt Admin nodes and TR-069 parameters, notifications, and error codes.The mapping must enable a Data Plugin to provide a native Broadband Forum object model that limits itself to the required RPCs.

131.3.1 Role

Developers implementing the TR-069 protocol are not likely to be also experts in the Dmt Admin service. This specification therefore provides a TR069 Connector Factory service that provides an object that can map from the TR-069 concepts to the Dmt Admin concepts, supporting all the constructs defined in the *OSGi Object Modeling* on page 327.

The TR069 Connector only specifies a number of primitive functions to manage the DMT. Parsing the SOAP messages, handling the notifications, and splitting the requests for TR069 Connector is the responsibility of the Protocol Adapter. The reason that the TR069 Connector does not work on a higher level is that a Protocol Adapter for TR-069 will likely communicate with other subsystems in the CPE than the OSGi framework alone. Though the Dmt Plugin model is an attractive approach to implement object models, there is history. Existing code will likely not be rewritten just because it can be done better as a Data Plugin.

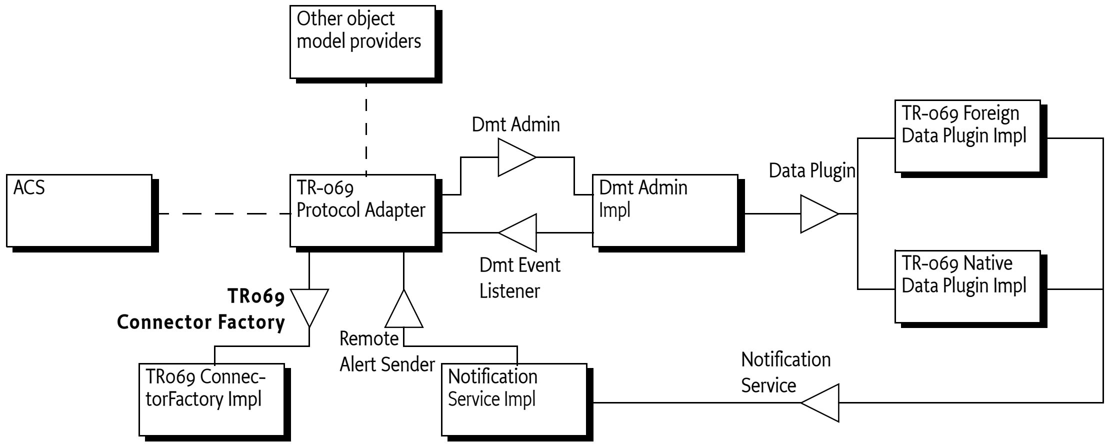
For example, a Data Plugin could implement the `Device.DeviceInfo.` object. However, this object actually resides in the DMT at a node:

```
./TR-069/Device/DeviceInfo
```

A TR-069 Protocol Adapter will therefore be confronted with a number of data models that reside in different places. Each place provides one or more consistent data models but it is the responsibility of the TR-069 Protocol Adapter to ensure the ACS gets a consistent and standardized view of the whole. To create this consistent view it will be necessary to adapt the paths given in the RPCs. It is expected that a Protocol Adapter is required to have a certain amount of domain knowledge, for example a table, that maps TR-069 paths to their actual providers.

The basic model is depicted in Figure 131.5.

Figure 131.5 TR-069 Connector Context


Other object model providers
Dmt Admin
TR-069 Foreign Data Plugin Impl
Data Plugin
ACS
TR-069 Protocol Adapter
Dmt Admin Impl
Dmt Event Listener
TR-069 Native Data Plugin Impl
TR069 Connector Factory
Remote Alert Sender
Notification Service
TR069 ConnectorFactory Impl
Notification Service Impl

The Protocol Adapter can be implemented as an OSGi Bundle or it can be implemented in native code in the device. Both architectures are viable. For certain aspects like the TR-157a3 Software Modules a certain amount of native code will be required to manage the OSGi Framework as an Execution Environment.

In an environment where the Protocol Adapter is implemented outside an OSGi Framework it will be necessary to create a link to the Dmt Admin service. This can be achieved with a proxy bundle inside the OSGi framework that dispatches any requests from the native Protocol Adapter to the functionality present in the OSGi Framework. In this specification, it is assumed that such proxies can be present. However, the examples are all assuming that the Protocol Adapter is running as a Bundle.

131.3.2 Obtaining a TR069 Connector

A TR069 Connector is associated with a Dmt Session, the TR069ConnectorFactoryprovides the create(DmtSession) method that will return a TR069Connector object. This object remains associated with the Dmt Session until the Dmt Session is closed, which can happen because of a fatal error or when the TR069 Connector Factory is unregistered or un-gotten/released. Creating a TR069 Connector must not be expensive, Protocol Adapters should create and close them at will. Closing the connector must not close the corresponding Dmt Session.

The TR069 Connector must use the root of the session as its *base*. That is, their URI mapping all parameters must start from the base. For example, if the session is opened at ./TR-069 then the parameter IGD/DeviceInfo/Manufacturer must map to URI ./TR-069/IGD/DeviceInfo/Manufacturer.

If a Protocol Adapter will modify the tree then it should use an atomic session for all RPCs even if the RPC indicates read-only. The reason for the atomicity is that in certain cases the lazy behavior of the TR069 Connector requires the creation of objects during a read operation. If a non-atomic session is used then the TR069 Connector must not attempt to lazily create objects and reject any addObject(String) and deleteObject(String) methods. See also *Lazy and Sessions* on page 420.

131.3.3 Supported RPCs

The TR069 Connector supports a limited number of RPCs, and for those RPCs it only supports the singleton case. The TR069 Connector provides support for the RPCs primitives listed in Table 131.2.

Table 131.3 Supported TR-069 RPCs

RPC	Related Method	Description
SetParameterValues	setParameterValue(String,String,int)	Set one or more parameter values. The connector supports setting a single value, ensuring the proper path traversal and data type conversion
GetParameterValues	getParameterValue(String)	Get one or more parameter values. The connector supports getting a single value, converting it to a ParameterValue object, which contains the value and the type.
GetParameterNames	getParameterNames(String,boolean)	Get the paths of objects and parameters from the sub-tree or children that begins at the parameter path. The TR-069 Connector supports the full traversal of the given path and the next level option.
AddObject	addObject(String)	Add a new object to a table. The fully supports the semantics, taking the MAP and LIST nodes into account. Node creation can be delayed until a node is really needed.
DeleteObject	deleteObject(String)	Delete an object from a table.

131.3.4 Name Escaping

An object or parameter path describes a traversal through a set of objects, this is almost the same model that Dmt Admin provides. The difference is that the characters allowed in a TR-069 parameter name are different from the Dmt Admin node names and that TR-069 does not support application specific parameter/object names like the Dmt Admin service does.

A path consist of a number segments, where each segment identifies a name or instance id. TR-069 names can always be mapped to Dmt Admin node names as the character set of TR-069 parameter names is restricted and falls within the character set of the Dmt Admin node names. The length of a segment could be a problem but TR-069 paths are generally limited to have a length of less than 256 bytes. This specification therefore assumes that a segment of a TR-069 path is never too long to fit in a Dmt Admin node name.

Mapping a Dmt Admin node name to a parameter name, needed for browsing, is more complicated as Dmt Admin node names allow virtually every Unicode character except the forward slash ('/' \u002F). It is therefore necessary to escape Dmt Admin URIs into a path that is acceptable for the TR-069 protocol. It is assumed that escaping is only used in a browsing mode since native object models will never require escaping. The TR069 Connector must return names from the getParameterNames(String,boolean) call that the ACS can handle, optionally show to the user, and then use to construct new paths for subsequent RPCs.

There is no obvious escape character defined in TR-069, like for example the backward slash that the Dmt Admin uses for escaping. The character for escaping is the latin small letter thorn ('þ' \u00FE) because his character is highly unlikely to ever be used in a TR-069 path for a native object model, however, even if it is then it would be no problem for the escaping algorithm. The thorn is a letter, allowing it to be used as the first character in a parameter name, this allows escaping the first character.

A character in a segment that is not allowed must be escaped into the following sequence:

```
þ[0-9A-Z][0-9A-Z][0-9A-Z][0-9A-Z]
```

The 4 hexadecimal upper case digits form a hexadecimal number that is the Unicode for that character. Each character that does not conform to the syntax specified in *Parameter Names* on page 412 or the thorn character itself must be replaced with the escape sequence. For example, the name 3ABCþ must be translated to:

```
þ0033ABCþ00FE
```

If the segment is an instance id then the segment must not be escaped. Otherwise, if the segment does not start with a `Letter` or underscore, then the first character must be escaped with the thorn.

Unescaping must undo the escaping. Any sequence of `þ[0-9A-Z][0-9A-Z][0-9A-Z][0-9A-Z]` must be replaced with the character with the corresponding Unicode. A thorn found without the subsequent 4 hexadecimal upper case digits must be treated as a single thorn. For readability it is best to minimize the escaping. However, any name given to the TR069 Connector that is escaped must be properly interpreted even if the unescaped string did not require escaping. For example, þ0031þ0032þ0033 must be usable as an object instance id as the unescaped form is 123, which is a number.

A number of examples of the escaping are shown in Table 131.4.

Table 131.4 Escaping Parameter Names

Segment	Dmt Admin Escaped	TR-069 Escaped	Notes
DeviceInfo	DeviceInfo	DeviceInfo	Most common case.
3x Hello World	3x Hello World	þ0033xþ0020Helloþ0020World	The initial digit and the spaces must be escaped in TR-069.
þorn	þorn	þorn þ00FEorn	A single thorn does not require escaping as it is not followed by 4 hexadecimal digits. So both forms are valid for unescaping although escaping must deliver the þ00FE form.
application/bin	application\/bin	applicationþ002Fbin	The solidus must be escaped in both.
234	234	234	A numeral does not require escaping, it is assumed to be an instance id.
234x	234x	þ003234x	A name that starts with a digit requires the first digit to be escaped.
þ00FEorn	þ00FEorn	þ00FE00FEorn	It is possible to encode even already escaped names.

The TR069 Connector only accepts escaped paths and returns escaped paths. When a method returns a path it must be properly escaped and suitable as a TR-069 path.

131.3.5 Root

In general, the TR-069 Protocol Adapter is free to choose what parts of the DMT it wants to expose. A simple mapping table containing path prefixes can be used to define the handler for the given data model. However, since the intention is to allow TR-069 object models to be implemented in Dmt Admin Data Plugins there is a need to know where those plugins should reside in the DMT. This root is defined as:

```
./TR-069
```

Any Data Plugin that wants to provide an object model in the TR-069 family of object models should provide a Data Plugin rooted at the TR-069 root. For example, a Data Plugin implementing the `InternetGatewayDevice.DeviceInfo.` object should register its Data Plugin under the data Root URI `./TR-069/ InternetGatewayDevice/DeviceInfo`

131.3.6 DMT Traversal

A path must be mapped from the TR-069 hierarchy to the Dmt Admin nodes URI. The Protocol Adapter decides the *base* in the DMT by opening the Dmt Session with a session root parameter. The TR-069 Connector must then traverse the tree from this base based on the TR-069 path. The Protocol Adapter must use the *Instance Id* on page 334 for MAP and LIST nodes to traverse the DMT.

Assume that the URI of a node is requested for a given path P. The path P must be traversed from the root node. The root node can find the child, the first segment in P, and then use the same routine recursively for the remainder. This recursive routine must perform the following actions on each current node:

- If path P is empty, then this is the requested node.
- S = first segment of path P up to the first full stop.
- R = remainder of path P after the first full stop or empty if no full stop.
- If S is an alias (surrounded by '[' and ']'), replace S with the alias inside the brackets. For Dmt Admin nodes aliases are identical to normal node names.
- unescape S (replace the thorns)
- If the current node is a MAP or a LIST and S is an integer
  - Get the list L of children of the current nodes
  - If the nodes in L have an InstanceId node find the node where the InstanceId matches the segment S as integer, this becomes then the next level node N and the algorithm is repeated with path R.
- If no next node N was found then make it the child node of the current node with the name S.
- Repeat the algorithm with N with path R

Since each node that is traversed this way knows the node name it corresponds to it is easy to create an encoded URI for Dmt Admin.

For example, the TR-069 path:

```
Device.DeviceInfo.Interface.14.Connections.3.BytesSent
```

Assuming that Interface node is a MAP node and its children have an InstanceId node, where the WAN_1 node has an InstanceId of 14.

The Connections node is a LIST and the children have no InstanceId, therefore the name is the index. The translated URI then looks like:

```
Device/DeviceInfo/Interface/WAN_1/Connections/3/BytesSent
```

The toURI(String,boolean) method can take a TR-069 path and perform the substitutions. If the create parameter is true then the TR069 Connector will create *missing nodes* if possible. Missing nodes can only be created under a LIST or MAP node.

A missing node is a node that is addressed by a path but not present in the DMT. For example, the root of the session is ./TR-069 and the parameter path is A.B.C. If the DMT contains ./TR-069/A but not ./TR-069/A/B then node B is a missing node.

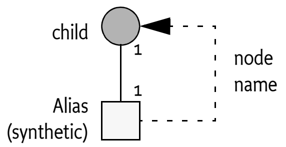
131.3.7 Synthetic Nodes

The Protocol Adapter must synthesize an Alias parameter and for any MAP or LIST node called X it must provide a sibling XNumberOfEntries parameter that provides the number of entries in table X.

131.3.7.1 Alias

The Alias node is a read-write parameter that must map to the actual node name of its parent. For example, ./A/B/C/Alias must map to C. Reading it must provide the this parent's node name and writing it must rename this parent's node name. The Alias must be automatically provided on any child of a MAP node. The Alias parameter must also be returned in the result of getParameterNames(String,boolean) if its parent's children are included. It is not possible to convert an Alias parameter name to a URI as the Alias node is synthetic and does not exist in the DMT. The model of aliases are depicted in Figure 131.6.

Figure 131.6 Aliases



Aliases can be used by the ACS to set the key of a MAP. For example, if a set of properties is defined as a MAP:

Name	Act	Type	Card.	S	Description
Properties	Get	MAP	1	P	A Properties map
[string]	Get Set Add Del	string	0..n	A	Key/Value

An ACS can first add an object to the table. This will create an entry with a calculated instance id. However, the ACS can then rename the node with the Alias node. In pseudo code:

```
AddObject          ..Properties.                  (returns node name = 3421)
SetParameterValue  ..Properties.3421.Alias = MyKey
```

Alternatively, addressing with an alias in the parameter name would be simpler:

```
AddObject          ..Properties.[MyKey]
```

131.3.7.2 Number Of Entries

TR-069 has the convention of parameters that end with NumberOfEntries. For example, the parameter UserNumberOfEntries in the object InternetGatewayDevice object contains the number of entries of the InternetGatewayDevice.User table.

The Protocol Adapter must synthesize these NumberOfEntries parameters for each MAP or LIST node. The NumberOfEntries parameter must be a sibling of the MAP or LIST node. Any such parameter must also be returned in the result of the getParameterNames(String,boolean) method.

131.3.8 Lazy and Sessions

In the Dmt Admin service the session plays an important role in how the object model operates. Especially atomic sessions have a clear point to commit any changes so that many actions can be deferred until all the information is available. In TR-069 there is no real session concept although one RPC must be executed atomically even if it changes multiple parameters. As there are different RPCs to create objects and set their parameters it is impossible to create and parameterize an object in a single session. This creates problems with general DMT models.

It is recommended to operate all RPCs in an atomic session to allow these DMT models to leverage the session commit phase. However, a TR-069 Connector must also accept a read only or exclusive session. The session can then of course cause exceptions to be thrown at certain operations.

The connector must *lazily* create instances. An addObject(String) method must not actually create the object, it only has to create an instance id and ensure the uniqueness of this id over time. The id must follow the rules from TR-069, it must not clash with an existing id even after such an id has been used in the past.

This id is then returned to the ACS who will then use it in subsequent RPCs. When one of the subsequent RPCs tries to access this not-yet existent node, for example a get or set, then the TR069 Connector must create it before it sets or gets the value of this node. This lazy strategy allows the node creation and the parameterization of that node to happen in a single session/RPC.

For example, in session 100 the addObject(String) creates a new node. This node is not really created but the unique instance id 4311 is assigned to it. After this RPC, the session is closed. The ACS receives this instance and then prepares a GetParameterValues RPC to get the ../4311/Foo parameter. The management agent receives the RPC and opens a new session 200, it then calls getParameterValue(String). The TR069 Connector will not find the appropriate entry 4311 in the table. Instead of raising an error it creates this node and then gets the value for the ../4311/Foo parameter.

A Data Plugin implementing a native TR-069 object model can override the lazy behavior by adding a application/x-tr-069-eager MIME type to the list of MIME types in the Meta Node. If this MIME type is present then the node must be eagerly created during the addObject(String) method.

The TR069 Connector must assign the unique id according to the TR-069 rules for instance ids.

131.3.9 Data Types

This specifications assume the [2] *TR-106 Amendment 3* defined data types. TR-106 defines a number of data types, derived from XML Schema and creates a number of sub-types to discriminate between different use cases. A Protocol Adapter must be able to understand the types defined in Table 131.5 to be able to faithfully define a data model based on [2] *TR-106 Amendment 3*. Discriminating between some of the sub-types requires inspection of the data. Each sub-type requires mapping rules that are defined later. Each mapping is assigned a unique MIME sub-type in the application media type. That is, the TR-069 int type has a MIME type of application/x-tr-069-int.

Table 131.5 *TR-069 Types, MIME types*

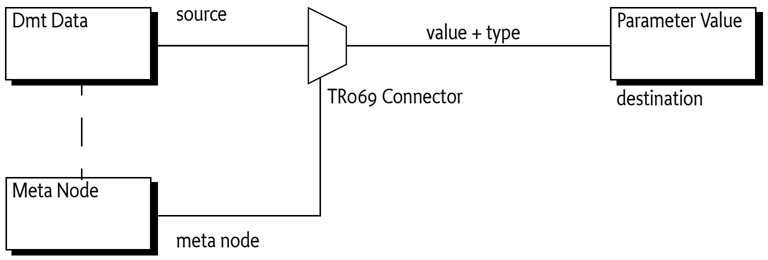
TR-069 Type	MIME Type	Notes
base64	x-tr-069-base64	Base 64 encoded
hexBinary	x-tr-069-hexBinary	Hex encoded
boolean	x-tr-069-boolean	
string	x-tr-069-string	General string type.
string (list)	x-tr-069-list	A comma separated string that acts as a list.
int	x-tr-069-int	Signed integer
unsignedInt	x-tr-069-unsignedInt	Unsigned integer
long	x-tr-069-long	Signed long
unsignedLong	x-tr-069-unsignedLong	Unsigned long
dateTime	x-tr-069-dateTime	Absolute UTC time, relative boot time, or unknown time
	x-tr-069-eager	Eager creation (not a data type, see *Lazy and Sessions* on page 420).

It is the responsibility of the Protocol Adapter to properly clean up the parameter values, that is, remove any unnecessary white space, etc. The TR069 Connector must accept any lexically correct form of the value of a parameter. However, the connector must always return the value according to the format of the data types specified by TR-069.

131.3.10 DMT to TR-069 Conversion

This section describes the conversion from a DMT node (a Dmt Data) to a TR-069 Parameter value. The *source* is the DMT node retrieved from the DMT. The *destination* is the value and its type that must be encoded in the TR-069 response. The *meta node* is the Meta Node associated with the source. This model is depicted in Figure 131.7.

Figure 131.7 DMT to TR-069



The different conversions possible for the Dmt Data to the TR-069 Parameter value are shown in Table 131.7. This table shows vertically the Dmt Admin formats and horizontally the TR-106 types defined in Table 131.5. Each row has a default conversion type, indicated with a bold entry. For example, the default conversion of a FORMAT_BOOLEAN to the boolean type is the default conversion.

This default conversion can be overridden by the Data Plugin by specifying an alternative MIME type in the list of allowed MIME types in the Meta Node getMimeTypes(). If this list contains a MIME type that has the prefix application/x-tr-069- then the first entry in this list must be chosen as the destination type instead of the default type. This way, a TR-069 Data Plugin can indicate the exact type to a TR-069 Protocol Adapter.

For example, a Dmt Data has the format FORMAT_BASE64. However, the Data Plugin for this node has a Meta Node that contains

```
String[] { "application/x-tr-069-hexBinary" }
```

The resulting type must therefore be hexBinary in this example.

The Dmt Data nodes are leaf nodes, however, there is a special case for interior LIST nodes marked with a application/x-tr-069-list type in the Meta Node. These nodes must be converted to a comma separated string as described in *List* on page 424.

Cells that are empty in the table indicate an impossible conversion that must be reported. Cells with a name refer to one of the subsequent sections.

Table 131.6 Dmt Data Format to TR-069 Data

	base64	boolean	dateTime	hexBinary	int	long	string	unsignedInt	unsignedLong
FORMAT_BASE64	binary			binary					
FORMAT_BINARY	**binary**			binary					
FORMAT_BOOLEAN		**=**					true | false		
FORMAT_DATE			**date**				=		
FORMAT_DATE_TIME			**date**				date		

Table 131.6 *Dmt Data Format to TR-069 Data*

	base64	boolean	dateTime	hexBinary	int	long	string	unsignedInt	unsignedLong
FORMAT_FLOAT					number	**number**	number	number	number
FORMAT_INTEGER					**number**	number	number	number	number
FORMAT_LONG					number	**number**	number	number	number
LIST							**list**		
FORMAT_NULL		false	date		0	0	"null"	0	0
FORMAT_RAW_BINARY	**binary**			binary					
FORMAT_RAW_STRING							=		
FORMAT_STRING							=		
FORMAT_TIME			date				=		
FORMAT_XML							=		

131.3.10.1 Date

If the destination type is string then a date must be formatted according to the TR-069 dateTime format. FORMAT_DATE and FORMAT_TIME must be set to a TR069_DATETIME typed destination with just the day or just the time respectively. That is, the FORMAT_TIME must be treated as a relative time for TR-069.

The Date object of the Dmt Data object represents the three different TR069_DATETIME types with the getTime() method. The value of getTime() indicates what type of date time it is:

- *Unknown* – The getTime() method must be 0
- *Relative* - The getTime() method must return a negative number
- *Absolute* – The getTime() method must return a positive number

If a FORMAT_DATE, FORMAT_TIME, or FORMAT_DATE_TIME is converted to a string the string representation of TR069_DATETIME must be used, including the form of unknown, relative, or absolute. A FORMAT_NULL stands for an unknown time.

131.3.10.2 Binary

The Dmt Admin service has several binary formats (FORMAT_BASE64, FORMAT_BINARY, and FORMAT_RAW_BINARY) that can be converted to TR069_HEXBINARY and TR069_BASE64. All binary formats maintain their data as a byte[]. Conversion is therefore straightforward encoding of the byte[] into the proper encoding: hex or base 64.

131.3.10.3 Number

The TR-069 Connector must convert numeric values (FORMAT_INTEGER, FORMAT_LONG, and FORMAT_FLOAT) to TR069_INT, TR069_LONG, TR069_UNSIGNED_INT, and TR069_UNSIGNED_LONG values. Float values must be rounded according to the standard Java rounding rules when converted to an integer or long.

A conversion must not exceed the range of the destination type. That is, if an integer is converted to an unsigned int then negative values must be treated as an error. If the destination type is string then the numeric value must be calculated with the Dmt Data toString method.

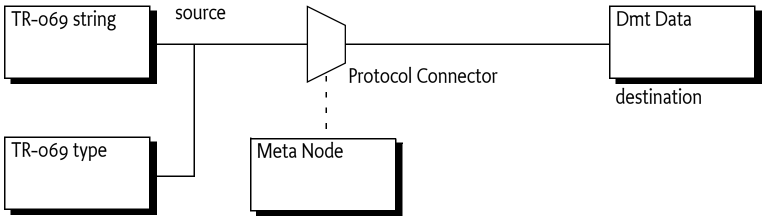
131.3.10.4 List

LIST nodes with primitive children must be converted to a comma separated list. If the children nodes are interior nodes then an error must be raised. The values of the comma separated list must come from the children of the value node. Each of these children must be converted to a string type according to Table 131.6 on page 422. These children must then be escaped and concatenated with a comma as separator according to the rules of TR-106 comma separated lists. Nested lists are not allowed.

131.3.11 TR-069 to Dmt Data Conversion

A TR-069 Parameter value consists of a string and a type identifier from the set of TR-069 types, see *Data Types* on page 421. The conversion is depicted in Figure 131.7.

Figure 131.8 *DMT to TR-069*



The destination type is obtained from the corresponding Meta Node. If multiple formats are specified in the result of the getFormat() method then the most applicable type must be used. The following table lists the applicability for each TR-106 data type.

base64	FORMAT_BASE64, FORMAT_BINARY, FORMAT_RAW_BINARY
boolean	FORMAT_BOOLEAN, FORMAT_STRING
dateTime	FORMAT_DATE_TIME, FORMAT_DATE, FORMAT_TIME
hexBinary	FORMAT_BASE64, FORMAT_BINARY, FORMAT_RAW_BINARY
int	FORMAT_INTEGER, FORMAT_LONG, FORMAT_FLOAT, FORMAT_STRING
long	FORMAT_LONG, FORMAT_FLOAT, FORMAT_INTEGER, FORMAT_STRING
string	FORMAT_STRING, FORMAT_BOOLEAN, FORMAT_INTEGER, FORMAT_LONG, FORMAT_FLOAT, FORMAT_RAW_STRING, FORMAT_XML
unsignedInt	FORMAT_INTEGER, FORMAT_LONG, FORMAT_FLOAT, FORMAT_STRING
unsignedLong	FORMAT_LONG, FORMAT_FLOAT, FORMAT_INTEGER, FORMAT_STRING

If the conversion fails and there are untried formats left then the other formats must be used.

There is a special case when the destination node is a LIST node with primitive children and the source is a string type. In that case the string must be parsed according to TR-106 comma separated lists and each element must be stored as a child node.

The conversion matrix is in Table 131.7. The equal sign indicates identity taking into account any encoding. It is not necessary that the source type corresponds to a MIME type in the meta node.

131.3.11.1 Date

A TR069_DATETIME can be converted to a FORMAT_DATE, FORMAT_TIME, and FORMAT_DATE_TIME. A FORMAT_DATE must take the day part and a FORMAT_TIME must take the time part.

131.3.11.2 Num

Source numbers must be converted to their destination counterpart. The conversion result must fail if the result falls outside the range of the destination.

Table 131.7 TR-069 Value to Dmt Data

	FORMAT_BASE64	FORMAT_BINARY	FORMAT_BOOLEAN	FORMAT_DATE	FORMAT_DATE_TIME	FORMAT_FLOAT	FORMAT_INTEGER	FORMAT_LONG	FORMAT_RAW_BINARY	FORMAT_RAW_STRING	FORMAT_STRING	FORMAT_TIME	FORMAT_XML	LIST
base64	binary	binary							binary					
boolean			bool								true|false			
dateTime				date	date						=	date		
hexBinary	binary	binary							binary					
int						num	num	num			=			
long						num	num	num			=			
string			bool			num	num	num		=	=		=	list
unsignedInt						num	num	num			=			
unsignedLong						num	num	num			=			

131.3.11.3 Bool

If the source is a string or boolean type and the destination FORMAT_BOOLEAN then the conversion must parse the string ignoring the case. The strings true and false map to their corresponding value. The strings 0 must map to false and 1 to true.

131.3.11.4 Binary

The source must be decoded according to its TR-069 type (TR069_BASE64 or TR069_HEXBINARY). The resulting byte array can then be set with the DmtData(byte[],int) with the destination format: FORMAT_BINARY or FORMAT_BASE64.

131.3.11.5 List

The source is a comma separated list and must be stored as children of the destination node.

131.4 RPCs

The following sections explain in more detail how the different RPCs are supported by the TR069 Connector operate.

131.4.1 Get Parameter Values

The GetParameterValues RPC retrieves the value from one or more parameters. Each request in the RPC can request one parameter value or provides an object or table path, requesting multiple values with one path.

The getParameterValue(String) method retrieves the value of one parameter in the DMT. The getParameterNames(String,boolean) method can be used to retrieve the values of a table or object.

For the getParameterValue(String) method the TR069 Connector must first check for synthesized parameters, see 131.3.7 *Synthetic Nodes* (Alias and NumberOfEntries). Otherwise, the parameter name must be converted to a URI, this must be done according to the toURI(String,boolean) method with the boolean set to true, creating any missing nodes if possible. The Dmt Data for this node must be converted according to *DMT to TR-069 Conversion* on page 422. The returned ParameterValue contains the type and value of the parameter.

For example:

```
ParameterValue v = connector.getParameterValue(
          "Device.DeviceInfo.Manufacturer");
String value = v.getValue();
int type = v.getType();
```

131.4.2 Set Parameter Values

The SetParameterValues RPC sets a number of values in one RPC. The setParameterValue(String, String,int) method corresponds to setting a single parameter in the DMT. It takes a parameter path, a value, and the type of this parameter.

The TR069 Connector must first check if the requested destination is the Alias node of a MAP child. If the Alias node is set, the name of the parent node must be renamed to the given value. The value of the Alias node must be a TR-069 string type, the Connector must ensure the value is escaped when necessary. See *Synthetic Nodes* on page 419 for further information about aliases.

Otherwise, the parameter name must be converted to a URI, this must be done according to the toURI(String,boolean) method with the boolean set to true.

The given value must be converted to a Dmt Data according to the *TR-069 to Dmt Data Conversion* on page 424. For example:

```
connector.setParameterValue("Starwars.R2D.2.Start",
                              "20110805T10:15:20Z", TR069_DATETIME );
```

131.4.3 Get Parameter Names

The GetParameterNames RPC allows an ACS to discover the parameters accessible on a particular CPE as well as verifying the existence of a parameter. There are modes for this RPC depending on the path and next level arguments, See Table 131.8.

Table 131.8 Modes based on type of path and NextLevel arguments

NextLevel	Parameter Path	Table or Object Path
true	Invalid Argument Fault code 9003 since this field must always be false for a parameter path.	Include only the children of the object or table.
false	A single ParameterInfo object is returned that provides information about the given parameter.	The whole sub-tree rooted at the given object or table path, this includes the object at the path itself. All objects must be included even if they are empty

The result must include only parameters, objects, and tables that are actually implemented by the CPE. If a parameter is listed then a getParameterValue(String) method called with this parameter's path should succeed. As a convenience, the ParameterInfo class provides a getParameterValue() method as a short cut to the value.

For example, assume the following instances:

```
IGD.LAN.1.Hosts.
IGD.LAN.1.Hosts.HostNumberOfEntries
IGD.LAN.1.Hosts.Host.
IGD.LAN.1.Hosts.Host.1.
IGD.LAN.1.Hosts.Host.1.Active
IGD.LAN.1.Hosts.Host.2.
IGD.LAN.1.Hosts.Host.2.Active
IGD.LAN.2.Hosts.
IGD.LAN.2.Hosts.HostNumberOfEntries
```

Table 131.9 demonstrates some of the different results based on these example instances.

Table 131.9 *Example Get Parameter Names*

Parameter Name	Next level	Results	Comments
IGD.LAN.1.	false	IGD.LAN.1. IGD.LAN.1.Hosts. IGD.LAN.1.Hosts.HostNumberOfEntries IGD.LAN.1.Hosts.Host. IGD.LAN.1.Hosts.Host.1. IGD.LAN.1.Hosts.Host.1.Active IGD.LAN.1.Hosts.Host.2. IGD.LAN.1.Hosts.Host.2.Active	The path specifies an instance in at table and since the Next Level is false the whole sub-tree must be returned, including the root of the sub-tree.
	true	IGD.LAN.1.Hosts.	The path is the same, an instance in a table, but now only the children must be returned for the source. There is only one child, Hosts. This must be returned as an object path.
IGD.LAN.1.Hosts.« 1.Active	false	IGD.LAN.1.Hosts.Host. 1.Active	The path is a parameter path, therefore only the source is returned.
	true	Fault 9003 Invalid Arguments, next level must be false for a parameter path.	Next Level must not be set to true for a parameter path
IGD.LAN.1	false or true	Fault 9003 Invalid Arguments, it is not a parameter path but an instance id	It is not allowed to specify a parameter path that is actual pointing to an instance.

For example:

```
Collection<ParameterInfo> pinfos = connector.getParameterNames("Device.");
for ( ParameterInfo info : pinfos ) {
  if ( info.isParameter() ) {
     System.out.println(
         connector.getParameterValue(info.getName()).getValue() );
  }
}
```

131.4.4 Add Object

The AddObject RPC creates a new instance in a table. There basic form for this RPC is to create an object and return the name of this object. It is also possible to specify an alias (a name specified in square brackets) after the table path. In that case, the alias is used as the node name. In either case, the path must be a valid table path pointing to a an existing MAP or LIST node.

When an object is added without an alias then the TR069 Connector must assign a unique id. TR-069 mandates that this id is unique for the table. The TR069 Connector must be able to create and maintain such a persistent id range. The Connector must ensure that any id chosen is not actually already in use or has been handed out recently. How such an id is calculated and maintained is implementation dependent.

If alias based addressing is used, a name between square brackets, then the alias is retrieved from the square brackets. The DMT must then be verified that no node exists in the corresponding table. If it does already exist, an INVALID_PARAMETER_NAME exception is thrown. Otherwise the alias is returned as the selected name.

If the corresponding MAP or LIST node has a Meta Node with a MIME type of application/x-tr-68-eager then the alias or instance id must be used to create the node. Otherwise the alias or instance id must be returned without creating the node. The purpose of this lazy creation is to allow a single Set Parameter Values RPC to atomically create a number of nodes and set their values.

For example:

```
String id = connector.addObject( "Starwars.CP.3.Obiwan.");
connector.setParameterValue( "Starwars.CP.3.Obiwan." + id + ".Name",
                             "cp30", TR069_STRING );
```

The previous code gets an assigned id with the addObject(String) method. The setParameterValue(String,String,int) then assigns the string cp30 to the Name node. This will first create the actual node since it was not created in the addObject(String) method and then sets the value of the DMT Starwars/CP/3/Obiwan/<id>/Name node.

The addObject(String) method does not require an atomic session.

131.4.5 Delete Object

The DeleteObject RPC deletes an object from the tree, it takes the instance path as argument. This behavior is implemented in the deleteObject(String) method. The corresponding node must be deleted if it exists. No error must be raised if the node does not exist in the DMT.

For example, deleting the object created in *Add Object* on page 427:

```
connector.deleteObject("Starwars.CP.3.Obiwan.cp30.");
```

131.5 Error and Fault Codes

The TR069 Connector must translate any Dmt Admin codes into a TR-069 fault code. Since the methods in the TR069Connector only relate to a single value it is possible to provide a mapping from Dmt Exception codes to TR-069 fault codes. It is the responsibility of the Protocol Adapter to aggregate these errors in the response to a SetParameterValues RPCs.

A TR069 Connector must prevent exceptions from happening and ensure that the different applicable error cases defined in the TR-069 RPCs are properly reported as a TR069 Exception with the intended fault code. However, this section defines a list of default translations between Dmt Exceptions and TR-069 fault codes.

Table 131.10 contains the exceptions and the resulting fault codes. Any obligations that are mandated by the TR-069 protocol are the responsibility of the TR-069 Protocol Adapter. The Dmt Exception is available from the TR-069 Exception for further inspection.

Table 131.10 Exceptions to TR-069 Fault code.

Exception	Fault code	Comments
ALERT_NOT_ROUTED	INTERNAL_ERROR	
COMMAND_FAILED	INTERNAL_ERROR	
COMMAND_NOT_ALLOWED	REQUEST_DENIED	
CONCURRENT_ACCESS	INTERNAL_ERROR	
DATA_STORE_FAILURE	INTERNAL_ERROR	
FEATURE_NOT_SUPPORTED	REQUEST_DENIED	
INVALID_URI	INVALID_PARAMETER_NAME	
LIMIT_EXCEEDED	RESOURCES_EXCEEDED	
METADATA_MISMATCH	INVALID_PARAMETER_TYPE	
NODE_ALREADY_EXISTS	INTERNAL_ERROR	
NODE_NOT_FOUND	INVALID_PARAMETER_NAME	
PERMISSION_DENIED	NON_WRITABLE_PARAMETER	
REMOTE_ERROR	INTERNAL_ERROR	
ROLLBACK_FAILED	INTERNAL_ERROR	
SESSION_CREATION_TIMEOUT	REQUEST_DENIED	
TRANSACTION_ERROR	REQUEST_DENIED	
UNAUTHORIZED	REQUEST_DENIED	
URI_TOO_LONG	INVALID_PARAMETER_NAME	
Dmt Illegal State Exception	INTERNAL_ERROR	
Security Exception	REQUEST_DENIED	
Other Exceptions	REQUEST_DENIED	

131.6 Managing the RMT

The RMT is not a native TR-069 model as it is not defined by BBF and it takes advantage of the Dmt Admin features. This section therefore shows a number of examples how the RMT can be managed from an ACS.

For example, on a specific CPE the following bundles are installed, the given name is the location

```
System Bundle
org-apache-felix-webconsole
org-apache-felix-configadmin
org-eclipse-equinox-scr
jp-co-ntt-admin
de-telekom-shell
```

The intention is to:

- Uninstall `org-apache-felix-configadmin`,
- Install and start `org-eclipse-equinox-cm`,
- Update `jp-co-ntt-admin`.

After the successful reconfiguration, the framework must restart. As framework changes must happen in a atomic session, the following parameters must be set in a single RPC:

```
SetParameterValues {
  Framework.Bundle.org-apache-felix-configadmin.RequestedState  = UNINSTALLED
  Framework.Bundle.jp-co-ntt-admin.URL                          = http://....
  Framework.Bundle.org-eclipse-equinox-cm.URL                   = http://....
  Framework.Bundle.org-eclipse-equinox-cm.RequestedState        = ACTIVE
  Framework.Bundle.org-eclipse-equinox-cm.AutoStart             = true
  Framework.Bundle.Systemþ0020Bundle.URL                        = ""
}
```

The Protocol Adapter must open an atomic session on the $ node as defined in the RMT. It will then set all the parameters in the previous list. As the Framework/Bundle/org-eclipse-equinox-cm node does not exist, the TR069 Connector will create it because it is below a writable MAP node. The System Bundle is updated with an empty string, signalling an update. A System Bundle update is a framework restart.

Once the session is committed after all the SetParameterValues elements are executed the Data Plugin will perform the actions and report success or failure. The handler must then restart the framework after the commit has returned.

131.7 Native TR-069 Object Models

This section provides an example of a Data Plugin that provides a native TR-069 Object Model. As example is chosen a naive implementation of the Configuration Admin service. The object model implemented has the following definition:

Table 131.11

Path	Type	Write	Read	Description
CM.{i}.	Object			
CM.{i}.Pid	string	x	x	The PID
CM.{i}.Properties.{i}.	Object			Property nodes
CM.{i}.Properties.{i}.Key	string	x	x	The key
CM.{i}.Properties.{i}.Value	string	x	x	Comma separated values

The corresponding DMT sub-tree is defined like:

Name	Act	Type	Card.	S	Description
CM	Get	MAP	1	P	Base node for the CM model
[string]	Get Set Add Del	Configuration	0..n	D	A MAP of the PID
InstanceId	Get	int	1	P	The persistent instance Id
Pid	Get	string	1	P	The PID of the configuration
Properties	Get	MAP	1	P	The properties
[string]	Get Set Add Del	LIST	0..n	D	A property definitions; a property consists of a list of strings. Single values are just a list with one element.
[index]	Get Set Add Del	string	0..n	D	An element in the list

The Protocol Adapter allows an ACS to access the data model implemented in the Dmt Plugin. It also allows the creation of new configuration objects.

131.8 org.osgi.service.tr069todmt

TR069 Connector Service Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.tr069todmt; version="[1.0,2.0)"`

Example import for providers of the API in this package:

`Import-Package: org.osgi.service.tr069todmt; version="[1.0,1.1)"`

131.8.1 Summary

- `ParameterInfo` – Maps to the TR-069 `ParameterInfoStruct` that is returned from the `TR069Connector.getParameterNames(String, boolean)` method.
- `ParameterValue` – Maps to the TR-069 `ParameterValueStruct`
- `TR069Connector` – A TR-069 Connector is an assistant to a TR-069 Protocol Adapter developer.
- `TR069ConnectorFactory` – A service that can create TR069 Connector
- `TR069Exception` – This exception is defined in terms of applicable TR-069 fault codes.

131.8.2 Permissions

131.8.3 public interface ParameterInfo

Maps to the TR-069 `ParameterInfoStruct` that is returned from the `TR069Connector.getParameterNames(String, boolean)` method.

131.8.3.1 public ParameterValue getParameterValue ( ) throws TR069Exception

□ Provide the value of the node. This method throws an exception if it is called for anything but a parameter

Returns The Parameter Value of the corresponding object

Throws `TR069Exception` – If there is a problem

131.8.3.2 public String getPath ( )

□ The path of the parameter, either a parameter path, an instance path, a table path, or an object path.

Returns The name of the parameter

131.8.3.3 public boolean isParameter ( )

□ Returns `true` of this is a parameter, if it returns `false` it is an object or table.

Returns `true` for a parameter, `false` otherwise

131.8.3.4 public boolean isWriteable ( )

□ Return `true` if this parameter is writeable, otherwise `false`. A parameter is writeable if the SetParamaterValue with the given name would be successful if an appropriate value was given.

Returns If this parameter is writeable

131.8.4 public interface ParameterValue

Maps to the TR-069 ParameterValueStruct

131.8.4.1 public String getPath ( )

- This is the path of a Parameter. In TR-069 this is called the Parameter Name.

Returns The path of the parameter

131.8.4.2 public int getType ( )

- The type of the parameter. One of TR069Connector.TR069_INT, TR069Connector.TR069_UNSIGNED_INT, TR069Connector.TR069_LONG, TR069Connector.TR069_UNSIGNED_LONG, TR069Connector.TR069_STRING, TR069Connector.TR069_DATETIME, TR069Connector.TR069_BASE64, TR069Connector.TR069_HEXBINARY. This method is not part of the ParameterValueStruct but is necessary to encode the type in the XML.

Returns The parameter type

131.8.4.3 public String getValue ( )

- This is the value of the parameter. The returned value must be in a representation defined by the TR-069 protocol.

Returns The value of the parameter

131.8.5 public interface TR069Connector

A TR-069 Connector is an assistant to a TR-069 Protocol Adapter developer. The connector manages the low level details of converting the different TR-069 RPCs to a Device Management Tree managed by Dmt Admin. The connector manages the conversions from the TR-069 Object Names to a node in the DMT and vice versa.

The connector uses a Dmt Session from the caller, which is given when the connector is created. The connector does not implement the exact RPCs but only provides the basic functions to set and get the parameters of an object as well as adding and deleting an object in a table. A TR-069 developer must still parse the XML, handle the relative and absolute path issues, open a Dmt Session etc.

The connector assumes that each parameter or object path is relative to the root of the Dmt Session.

This connector must convert the TR-069 paths to Dmt Admin URIs. This conversion must take into account the LIST and MAP concepts defined in the specifications as well as the synthetic parameters NumberOfEntries and Alias. These concepts define the use of an InstanceId node that must be used by the connector to provide a TR-069 table view on the LIST and MAP nodes.

131.8.5.1 public static final String PREFIX = “application/x-tr-069-”

The MIME type prefix.

131.8.5.2 public static final int TR069_BASE64 = 64

Constant representing the TR-069 base64 type.

131.8.5.3 public static final int TR069_BOOLEAN = 32

Constant representing the TR-069 boolean type.

131.8.5.4 public static final int TR069_DATETIME = 256

Constant representing the TR-069 date time type.

131.8.5.5 public static final int TR069_DEFAULT = 0

Constant representing the default or unknown type. If this type is used a default conversion will take place

131.8.5.6 public static final int TR069_HEXBINARY = 128

Constant representing the TR-069 hex binary type.

131.8.5.7 public static final int TR069_INT = 1

Constant representing the TR-069 integer type.

131.8.5.8 public static final int TR069_LONG = 4

Constant representing the TR-069 long type.

131.8.5.9 public static final String TR069_MIME_BASE64 = "application/x-tr-069-base64"

Constant representing the TR-069 base64 type.

131.8.5.10 public static final String TR069_MIME_BOOLEAN = "application/x-tr-069-boolean"

Constant representing the TR-069 boolean type.

131.8.5.11 public static final String TR069_MIME_DATETIME = "application/x-tr-069-dateTime"

Constant representing the TR-069 date time type.

131.8.5.12 public static final String TR069_MIME_DEFAULT = "application/x-tr-069-default"

Constant representing the default or unknown type. If this type is used a default conversion will take place

131.8.5.13 public static final String TR069_MIME_EAGER = "application/x-tr-069-eager"

Constant representing the TR-069 eager type.

131.8.5.14 public static final String TR069_MIME_HEXBINARY = "application/x-tr-069-hexBinary"

Constant representing the TR-069 hex binary type.

131.8.5.15 public static final String TR069_MIME_INT = "application/x-tr-069-int"

Constant representing the TR-069 integer type.

131.8.5.16 public static final String TR069_MIME_LONG = "application/x-tr-069-long"

Constant representing the TR-069 long type.

131.8.5.17 public static final String TR069_MIME_STRING = "application/x-tr-069-string"

Constant representing the TR-069 string type.

131.8.5.18 public static final String TR069_MIME_STRING_LIST = "application/x-tr-069-string-list"

Constant representing the TR-069 string list type.

131.8.5.19 public static final String TR069_MIME_UNSIGNED_INT = "application/x-tr-069-unsignedInt"

Constant representing the TR-069 unsigned integer type.

131.8.5.20 public static final String TR069_MIME_UNSIGNED_LONG = "application/x-tr-069-unsignedLong"

Constant representing the TR-069 unsigned long type.

131.8.5.21 public static final int TR069_STRING = 16

Constant representing the TR-069 string type.

131.8.5.22 public static final int TR069_UNSIGNED_INT = 2

Constant representing the TR-069 unsigned integer type.

131.8.5.23 public static final int TR069_UNSIGNED_LONG = 8

Constant representing the TR-069 unsigned long type.

131.8.5.24 public String addObject (String path) throws TR069Exception

path A table path with an optional alias at the end

□ Add a new node to the Dmt Admin as defined by the AddObject RPC. The path must map to either a `LIST` or `MAP` node as no other nodes can accept new children.

If the path ends in an alias ([`ALIAS`]) then the node name must be the alias, however, no new node must be created. Otherwise, the Connector must calculate a unique instance id for the new node name that follows the TR-069 rules for instance ids. That is, this id must not be reused and must not be in use. That is, the id must be reserved persistently.

If the `LIST` or `MAP` node has a Meta Node with a MIME type application/x-tr-069-eager then the node must be immediately created. Otherwise no new node must be created, this node must be created when the node is accessed in a subsequent RPC.

The alias name or instance id must be returned as identifier for the ACS.

Returns The name of the new node.

Throws `TR069Exception` – The following fault codes are defined for this method: 9001, 9002, 9003, 9004, 9005. If an AddObject request would result in exceeding the maximum number of such objects supported by the CPE, the CPE MUST return a fault response with the Resources Exceeded (9004) fault code.

131.8.5.25 public void close ( )

□ Close this connector. This will **not** close the corresponding session.

131.8.5.26 public void deleteObject (String objectPath) throws TR069Exception

objectPath The path to an object in a table to be deleted.

□ Delete an object from a table. A missing node must be ignored.

Throws `TR069Exception` – The following fault codes are defined for this method: 9001, 9002, 9003, 9005. If the fault is caused by an invalid objectPath value, the Invalid Parameter Name fault code (9005) must be used instead of the more general Invalid Arguments fault code (9003). A missing node for `objectPath` must be ignored.

131.8.5.27 public Collection<ParameterInfo> getParameterNames (String objectOrTablePath , boolean nextLevel) throws TR069Exception

objectOrTablePath A path to an object or table.

nextLevel If `true` consider only the children of the object or table addressed by `path`, otherwise include the whole sub-tree, including the addressed object or table.

□ Getting the `ParameterInfo` objects addressed by path. This method is intended to be used to implement the GetParameterNames RPC.

The connector must attempt to create any missing nodes that are needed for the `objectOrTablePath` by using the `toURI(String, boolean)` method with `true`.

This method must traverse the sub-tree addressed by the path and return the paths to all the objects, tables, and parameters in that tree. If the nextLevel argument is true then only the children object, table, and parameter information must be returned.

The returned ParameterInfo objects must be usable to discover the sub-tree.

If the child nodes have an InstanceId node then the returned names must include the InstanceId values instead of the node names.

If the parent node is a MAP, then the synthetic Alias parameter must be included.

Any MAP and LIST node must include a ParameterInfo for the corresponding NumberOfEntries parameter.

Returns A collection of ParameterInfo objects representing the resulting child parameter, objects, and tables as defined by the TR-069 ParameterInfoStruct.

Throws TR069Exception – If the fault is caused by an invalid ParameterPath value, the Invalid Parameter Name fault code (9005) MUST be used instead of the more general Invalid Arguments fault code (9003). A ParameterPath value must be considered invalid if it is not an empty string and does not exactly match a parameter or object name currently present in the data model. If nextLevel is true and objectOrTablePath is a parameter path rather than an object/table path, the method must return a fault response with the Invalid Arguments fault code (9003). If the value cannot be gotten for some reason, this method can generate the following fault codes::
9001 TR069Exception.REQUEST_DENIED
9002 TR069Exception.INTERNAL_ERROR
9003 TR069Exception.INVALID_ARGUMENTS
9005 TR069Exception.INVALID_PARAMETER_NAME

131.8.5.28 public ParameterValue getParameterValue (String parameterPath) throws TR069Exception

parameterPath A parameter path (must refer to a valid parameter, not an object or table).

□ Getting a parameter value. This method should be used to implement the GetParameterValues RPC. This method does **not** handle retrieving multiple values as the corresponding RPC can request with an object or table path, this method only accepts a parameter path. Retrieving multiple values can be achieved with the getParameterNames(String, boolean).

If the parameterPath ends in NumberOfEntries then the method must synthesize the value. The parameterPath then has a pattern like (object-path)(table-name)NumberOfEntries. The returned value must be an TR069_UNSIGNED_INT that contains the number of child nodes in the table (object-path)(table-name). For example, if A.B.CNumberOfEntries is requested the return value must be the number of child nodes under A/B/C.

If the value of a an Alias node is requested then the name of the parent node must be returned. For example, if the path is M.X.Alias then the returned value must be X.

The connector must attempt to create any missing nodes along the way, creating parent nodes on demand.

Returns The name, value, and type triad of the requested parameter as defined by the TR-069 ParameterValueStruct.

Throws TR069Exception – The following fault codes are defined for this method: 9001, 9002, 9003, 9004, 9005.
9001 TR069Exception.REQUEST_DENIED
9002 TR069Exception.INTERNAL_ERROR
9003 TR069Exception.INVALID_ARGUMENTS
9004 TR069Exception.RESOURCES_EXCEEDED
9005 TR069Exception.INVALID_PARAMETER_NAME

131.8.5.29 public void setParameterValue (String parameterPath , String value , int type) throws TR069Exception

parameterPath The parameter path

value A trimmed string value that has the given type. The value can be in either canonical or lexical representation by TR069.

type The type of the parameter (TR069_INT, TR069_UNSIGNED_INT, TR069_LONG, TR069_UNSIGNED_LONG, TR069_STRING, TR069_DATETIME, TR069_BASE64, TR069_HEXBINARY)

□ Setting a parameter. This method should be used to provide the SetParameterValues RPC. This method must convert the parameter Name to a URI and replace the DMT node at that place. It must follow the type conversions as described in the specification.

The connector must attempt to create any missing nodes along the way, creating parent nodes on demand.

If the value of a an Alias node is set then the parent node must be renamed. For example, if the value of M/X/Alias is set to Y then the node will have a URI of M/Y/Alias. The value must not be escaped as the connector will escape it.

Throws TR069Exception – The following fault codes are defined for this method: 9001, 9002, 9003, 9004, 9005, 9006, 9007, 9008.
9001 TR069Exception.REQUEST_DENIED
9002 TR069Exception.INTERNAL_ERROR
9003 TR069Exception.INVALID_ARGUMENTS
9004 TR069Exception.RESOURCES_EXCEEDED
9005 TR069Exception.INVALID_PARAMETER_NAME
9006 TR069Exception.INVALID_PARAMETER_TYPE
9007 TR069Exception.INVALID_PARAMETER_VALUE
9008 TR069Exception.NON_WRITABLE_PARAMETER

131.8.5.30 public String toPath (String uri) throws TR069Exception

uri A Dmt Session relative URI

□ Convert a Dmt Session relative Dmt Admin URI to a valid TR-069 path, either a table, object, or parameter path depending on the structure of the DMT. The translation takes into account the special meaning LIST, MAP , Alias, and InstanceId nodes.

Returns An object, table, or parameter path

Throws TR069Exception – If there is an error

131.8.5.31 public String toURI (String name , boolean create) throws TR069Exception

name A TR-069 path

create If true, create missing nodes when they reside under a MAP or LIST

□ Convert a TR-069 path to a Dmt Session relative Dmt Admin URI. The translation takes into account the special meaning LIST, MAP, InstanceId node semantics.

The synthetic Alias or NumberOfEntries parameter cannot be mapped and must throw an TR069Exception.INVALID_PARAMETER_NAME.

The returned path is properly escaped for TR-069.

The mapping from the path to a URI requires support from the meta data in the DMT, it is not possible to use a mapping solely based on string replacements. The translation takes into account the semantics of the MAP and LIST nodes. If at a certain point a node under a MAP node does not exist then the Connector can create it if the create flag is set to true. Otherwise a non-existent node will terminate the mapping.

Returns A relative Dmt Admin URI

Throws TR069Exception – If there is an error

131.8.6 public interface TR069ConnectorFactory

A service that can create TR069 Connector

131.8.6.1 public TR069Connector create (DmtSession session)

session The session to use for the adaption. This session must not be closed before the TR069 Connector is closed.

□ Create a TR069 connector based on the given session .

The session must be an atomic session when objects are added and/or parameters are going to be set, otherwise it can be a read only or exclusive session. Due to the lazy creation nature of the TR069 Connector it is possible that a node must be created in a read method after a node has been added, it is therefore necessary to always provide an atomic session when an ACS session requires modifying parameters.

Returns A new TR069 Connector bound to the given session

131.8.7 public class TR069Exception extends RuntimeException

This exception is defined in terms of applicable TR-069 fault codes. The TR-069 specification defines the fault codes that can occur in different situations.

131.8.7.1 public static final int INTERNAL_ERROR = 9002

9002 Internal error

131.8.7.2 public static final int INVALID_ARGUMENTS = 9003

9003 Invalid Arguments

131.8.7.3 public static final int INVALID_PARAMETER_NAME = 9005

9005 Invalid parameter name (associated with Set/GetParameterValues, GetParameterNames, Set/GetParameterAttributes, AddObject, and DeleteObject)

131.8.7.4 public static final int INVALID_PARAMETER_TYPE = 9006

9006 Invalid parameter type (associated with SetParameterValues)

131.8.7.5 public static final int INVALID_PARAMETER_VALUE = 9007

9007 Invalid parameter value (associated with SetParameterValues)

131.8.7.6 public static final int METHOD_NOT_SUPPORTED = 9000

9000 Method not supported

131.8.7.7 public static final int NON_WRITABLE_PARAMETER = 9008

9008 Attempt to set a non-writable parameter (associated with SetParameterValues)

131.8.7.8 public static final int NOTIFICATION_REJECTED = 9009

9009 Notification request rejected (associated with SetParameterAttributes method).

131.8.7.9 public static final int REQUEST_DENIED = 9001

9001 Request denied (no reason specified

131.8.7.10 **public static final int RESOURCES_EXCEEDED = 9004**

9004 Resources exceeded (when used in association with SetParameterValues, this MUST NOT be used to indicate parameters in error)

131.8.7.11 **public TR069Exception (String message)**

message The message

□ A default constructor when only a message is known. This will generate a INTERNAL_ERROR fault.

131.8.7.12 **public TR069Exception (String message , int faultCode , DmtException e)**

message The message

faultCode The TR-069 defined fault code

e

□ A Constructor with a message and a fault code.

131.8.7.13 **public TR069Exception (String message , int faultCode)**

message The message

faultCode The TR-069 defined fault code

□ A Constructor with a message and a fault code.

131.8.7.14 **public TR069Exception (DmtException e)**

e The Dmt Exception

□ Create a TR069Exception from a Dmt Exception.

131.8.7.15 **public DmtException getDmtException ( )**

Returns the corresponding Dmt Exception

131.8.7.16 **public int getFaultCode ( )**

□ Answer the associated TR-069 fault code.

Returns Answer the associated TR-069 fault code.

131.9 References

[1] *TR-069 Amendment 3*
http://www.broadband-forum.org/technical/download/TR-069_Amendment-3.pdf

[2] *TR-106 Amendment 3*
http://www.broadband-forum.org/technical/download/TR-106_Amendment-3.pdf

[3] *XML Schema Part 2: Datatypes Second Edition*
http://www.w3.org/TR/xmlschema-2/

[4] *SOAP 1.1*
http://www.w3.org/TR/2000/NOTE- SOAP-20000508

[5] *Extensible Markup Language (XML) 1.0 (Second Edition)*
http://www.w3.org/TR/2000/WD-xml-2e-20000814#NT-Letter

[6] *Broadband Forum*
http://www.broadband-forum.org/

701 Tracker Specification

Version 1.5

701.1 Introduction

The Framework provides a powerful and very dynamic programming environment: Bundles are installed, started, stopped, updated, and uninstalled without shutting down the Framework. Dependencies between bundles are monitored by the Framework, but bundles *must* cooperate in handling these dependencies correctly. Two important *dynamic* aspects of the Framework are the service registry and the set of installed bundles.

Bundle developers must be careful not to use service objects that have been unregistered and are therefore stale. The dynamic nature of the Framework service registry makes it necessary to track the service objects as they are registered and unregistered to prevent problems. It is easy to overlook race conditions or boundary conditions that will lead to random errors. Similar problems exist when tracking the set of installed bundles and their state.

This specification defines two utility classes, `ServiceTracker` and `BundleTracker`, that make tracking services and bundles easier. A `ServiceTracker` class can be customized by implementing the `ServiceTrackerCustomizer` interface or by sub-classing the `ServiceTracker` class. Similarly, a `BundleTracker` class can be customized by sub-classing or implementing the `BundleTrackerCustomizer` interface.

These utility classes significantly reduce the complexity of tracking services in the service registry and the set of installed bundles.

701.1.1 Essentials

- *Simplify* – Simplify the tracking of services or bundles.
- *Customizable* – Allow a default implementation to be customized so that bundle developers can start simply and later extend the implementation to meet their needs.
- *Small* – Every Framework implementation should have this utility implemented. It should therefore be very small because some Framework implementations target minimal OSGi Service Platforms.
- *Services* – Track a set of services, optionally filtered, or track a single service.
- *Bundles* – Track bundles based on their state.
- *Cleanup* – Properly clean up when tracking is no longer necessary
- *Generified* – Generics are used to promote type safety.

701.1.2 Operation

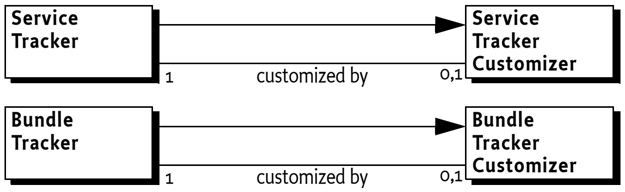
The fundamental tasks of a tracker are:

- To create an initial list of *targets* (service or bundle).
- To listen to the appropriate events so that the targets are properly tracked.
- To allow the client to customize the tracking process through programmatic selection of the services/bundles to be tracked, as well as to perform client code when a service/bundle is added or removed.

A `ServiceTracker` object is populated with a set of services that match given search criteria, and then listens to `ServiceEvent` objects which correspond to those services. A Bundle Tracker is populated with the set of installed bundles and then listens to `BundleEvent` objects to notify the customizer of changes in the state of the bundles.

701.1.3 Entities

Figure 701.1 Class diagram of org.osgi.util.tracker



701.2 Tracking

The OSGi Framework is a dynamic multi-threaded environment. In such an environments callbacks can occur on different threads at the same time. This dynamism causes many complexities. One of the surprisingly hard aspects of this environment is to reliably track services and bundles (called *targets* from now on).

The complexity is caused by the fact that the BundleListener and ServiceListener interfaces are only providing access to the *changed* state, not to the existing state when the listener is registered. This leaves the programmer with the problem to merge the set of existing targets with the changes to the state as signified by the events, without unwantedly duplicating a target or missing a remove event that would leave a target in the tracked map while it is in reality gone. These problems are caused by the multi-threaded nature of an OSGi service platform.

The problem is illustrated with the following (quite popular) code:

```
// Bad Example! Do not do this!
Bundle[] bundles = context.getBundles();
for ( Bundle bundle : bundles ) {
   map.put(bundle.getLocation(), bundle );
}

context.addBundleListener( new BundleListener() {
   public void bundleChanged(BundleEvent event) {
      Bundle bundle = event.getBundle();
      switch(event.getType()) {
      case BundleEvent.INSTALLED:
         map.put(bundle.getLocation(), bundle );
         break;

      case BundleEvent.UNINSTALLED:
         map.remove(bundle.getLocation());
         break;

      default:
         // ignore
      }
   }
});
```

Assume the code runs the first part, getting the existing targets. If during this time a targets state changes, for example bundle is installed or uninstalled, then the event is missed and the map will miss a bundle or it will contain a bundle that is already gone. An easy solution seems to be to first register the listener and then get the existing targets. This solves the earlier problem but will be introduce other problems. In this case, an uninstall event can occur before the bundle has been discovered.

Proper locking can alleviate the problem but it turns out that this easily create solutions that are very prone to deadlocks. Solving this tracking problem is surprisingly hard. For this reason, the OSGi specifications contain a *bundle tracker* and a *service tracker* that are properly implemented. These classes significantly reduce the complexity of the dynamics in an OSGi Service Platform.

701.2.1 Usage

Trackers can be used with the following patterns:

- *As-is* – Each tracker can be used without further customizing. A tracker actively tracks a map of targets and this map can be consulted with a number of methods when the information is needed. This is especially useful for the Service Tracker because it provides convenience methods to wait for services to arrive.
- *Callback object* – Each tracker provides a call back interface that can be implemented by the client code.
- *Sub-classing* – The trackers are designed to be sub-classed. Sub-classes have access to the bundle context and only have to override the callback methods they need.

701.2.2 General API

A tracker hides the mechanisms in the way the targets are stored and evented. From a high level, a tracker maintains a *map* of targets to *wrapper* objects. The wrapper object can be defined by the client, though the Bundle Tracker uses the `Bundle` object and the Service Tracker uses the service object as default wrapper. The tracker notifies the client of any changes in the state of the target.

A tracker must be constructed with a Bundle Context. This context is used to register listeners and obtain the initial list of targets during the call to the `open` method. At the end of the life of a tracker it must be closed to release any remaining objects. It is advised to properly close all trackers in the bundle activator's stop method.

A tracker provides a uniform callback interface, which has 3 different methods.

- *Adding* – Provide a new object, obtained from the store or from an event and return the wrapper or a related object. The adding method can decide not to track the target by returning a `null` object. When null is returned, no modified or remove methods are further called. However, it is possible that the adding method is called again for the same target.
- *Modified* –The target is modified. For example, the service properties have changed or the bundle has changed state. This callback provides a mechanism for the client to update its internal structures. The callback provides the wrapper object.
- *Removing* – The target is no longer tracked. This callback is provided the wrapper object returned from the adding method. This allows for simplified cleanup if the client maintains state about the target.

Each tracker is associated with a callback interface, which it implements itself. That is, a Service Tracker implements the `ServiceTrackerCustomizer` interface. By implementing this customizer, the tracker can also be sub-classed, this can be quite useful in many cases. Sub-classing can override only one or two of the methods instead of having to implement all methods. When overriding the callback methods, it must be ensured that the wrapper object is treated accordingly to the base implementation in all methods. For example, the Service Tracker's default implementation for the adding method checks out the service and therefore the remove method must unget this same service. Changing the wrapper object type to something else can therefore clash with the default implementations.

Trackers can provide all the objects that are tracked, return the mapped wrapper from the target, and deliver the number of tracked targets.

701.2.3 Tracking Count

The tracker also maintains a count that is updated each time that an object is added, modified, or removed, that is any change to the implied map. This tracking count makes it straightforward to verify that a tracker has changed; just store the tracking count and compare it later to see if it has changed.

701.2.4 Multi Threading

The dynamic environment of OSGi requires that tracker are thread safe. However, the tracker closely interacts with the client through a callback interface. The tracker implementation must provide the following guarantees:

- The tracker code calling a callback must not hold any locks

Clients must be aware that their callbacks are reentrant though the tracker implementations guarantee that the add/modified/remove methods can only called in this order for a specific target. A tracker must not call these methods out of order.

701.2.5 Synchronous

Trackers use *synchronous* listeners; the callbacks are called on the same thread as that of the initiating event. Care should be taken to not linger in the callback and perform non-trivial work. Callbacks should return immediately and move substantial work to other threads.

701.3 Service Tracker

The purpose of a Service Tracker is to track *service references*, that is, the target is the ServiceReference object. The Service Tracker uses generics to provide a type safe interface. It has two type arguments:

- S – The service type.
- T – The type used by the program. T can differ from S if the program creates a wrapper around the service object, a common pattern.

The ServiceTracker interface defines three constructors to create ServiceTracker objects, each providing different search criteria:

- ServiceTracker(BundleContext,String,ServiceTrackerCustomizer) – This constructor takes a service interface name as the search criterion. The ServiceTracker object must then track all services that are registered under the specified service interface name.
- ServiceTracker(BundleContext,Filter,ServiceTrackerCustomizer) – This constructor uses a Filter object to specify the services to be tracked. The ServiceTracker must then track all services that match the specified filter.
- ServiceTracker(BundleContext,ServiceReference,ServiceTrackerCustomizer) – This constructor takes a ServiceReference object as the search criterion. The ServiceTracker must then track only the service that corresponds to the specified ServiceReference. Using this constructor, no more than one service must ever be tracked, because a ServiceReference refers to a specific service.
- ServiceTracker(BundleContext,Class,ServiceTrackerCustomizer) – This constructor takes a class as argument. The tracker must only track services registered with this name. This is in general the most convenient way to use the Service Tracker.

Each of the ServiceTracker constructors takes a BundleContext object as a parameter. This BundleContext object must be used by a ServiceTracker object to track, get, and unget services.

A new ServiceTracker object must not begin tracking services until its open method is called. There are 2 versions of the open method:

- open() – This method is identical to open(false). It is provided for backward compatibility reasons.
- open(boolean) – The tracker must start tracking the services as were specified in its constructor. If the boolean parameter is true, it must track all services, regardless if they are compatible with the bundle that created the Service Tracker or not. See Section 5.9 “Multiple Version Export Considerations” for a description of the compatibility issues when multiple variations of the same package can exist. If the parameter is false, the Service Tracker must only track compatible versions.

701.3.1 Using a Service Tracker

Once a ServiceTracker object is opened, it begins tracking services immediately. A number of methods are available to the bundle developer to monitor the services that are being tracked, including the ones that are in the service registry at that time. The ServiceTracker class defines these methods:

- getService() – Returns one of the services being tracked or null if there are no active services being tracked.
- getServices() – Returns an array of all the tracked services. The number of tracked services is returned by the size method.
- getServices(T[]) – Like getServices() but provides a convenient way to get these services into a correctly typed array.
- getServiceReference() – Returns a ServiceReference object for one of the services being tracked. The service object for this service may be returned by calling the ServiceTracker object’s getService() method.
- getServiceReferences() – Returns a list of the ServiceReference objects for services being tracked. The service object for a specific tracked service may be returned by calling the ServiceTracker object’s getService(ServiceReference) method.
- waitForService(long) – Allows the caller to wait until at least one instance of a service is tracked or until the time-out expires. If the time-out is zero, the caller must wait until at least one instance of a service is tracked. waitForService must not used within the BundleActivator methods, as these methods are expected to complete in a short period of time. A Framework could wait for the start method to complete before starting the bundle that registers the service for which the caller is waiting, creating a deadlock situation.
- remove(ServiceReference) – This method may be used to remove a specific service from being tracked by the ServiceTracker object, causing removedService to be called for that service.
- close() – This method must remove all services being tracked by the ServiceTracker object, causing removedService to be called for all tracked services.
- getTrackingCount() – A Service Tracker can have services added, modified, or removed at any moment in time. The getTrackingCount method is intended to efficiently detect changes in a Service Tracker. Every time the Service Tracker is changed, it must increase the tracking count.
- isEmpty() – To detect that the tracker has no tracked services.
- getTracked() – Return the tracked objects.

701.3.2 Customizing the Service Tracker class

The behavior of the ServiceTracker class can be customized either by providing a ServiceTrackerCustomizer object, implementing the desired behavior when the ServiceTracker object is constructed, or by sub-classing the ServiceTracker class and overriding the ServiceTrackerCustomizer methods.

The ServiceTrackerCustomizer interface defines these methods:

- addingService(ServiceReference) – Called whenever a service is being added to the ServiceTracker object.
- modifiedService(ServiceReference,T) – Called whenever a tracked service is modified.

- removedService(ServiceReference,T) – Called whenever a tracked service is removed from the ServiceTracker object.

When a service is being added to the ServiceTracker object or when a tracked service is modified or removed from the ServiceTracker object, it must call addingService, modifiedService, or removedService, respectively, on the ServiceTrackerCustomizer object (if specified when the ServiceTracker object was created); otherwise it must call these methods on itself.

A bundle developer may customize the action when a service is tracked. Another reason for customizing the ServiceTracker class is to programmatically select which services are tracked. A filter may not sufficiently specify the services that the bundle developer is interested in tracking. By implementing addingService, the bundle developer can use additional runtime information to determine if the service should be tracked. If null is returned by the addingService method, the service must not be tracked.

Finally, the bundle developer can return a specialized object from addingService that differs from the service object. This specialized object could contain the service object and any associated information. This returned object is then tracked instead of the service object. When the removedService method is called, the object that is passed along with the ServiceReference object is the one that was returned from the earlier call to the addingService method.

701.3.3 Customizing Example

An example of customizing the action taken when a service is tracked might be registering a MyServlet object with each Http Service that is tracked. This customization could be done by sub-classing the ServiceTracker class and overriding the addingService and removedService methods as follows:

```
new ServiceTracker<HttpService,MyServlet>(context,HttpService.class,null) {
  public MyServlet addingService( ServiceReference<HttpService> reference) {
     HttpService svc = context.getService(reference);
     MyServlet ms = new MyServlet(scv);
     return ms;
  }
  public void removedService( ServiceReference<HttpService> reference,
     MyServlet ms){
     ms.close();
     context.ungetService(reference);
  }
}
```

In this example, the service type is the HttpService class and the wrapper type is the servlet.

701.4 Bundle Tracker

The purpose of the Bundle Tracker is to simplify tracking bundles. A popular example where bundles need to be tracked is the *extender* pattern. An extender uses information in other bundles to provide its function. For example, a Declarative Services implementation reads the component XML file from the bundle to learn of the presence of any components in that bundle.

There are, however, other places where it is necessary to track bundles. The Bundle Tracker significantly simplifies this task.

701.4.1 Bundle States

The state diagram of a Bundle is significantly more complex than that of a service. However, the interface is simpler because there is only a need to specify for which states the bundle tracker should track a service.

Bundle states are defined as a bit in an integer, allowing the specifications of multiple states by setting multiple bits. The Bundle Tracker therefore uses a *bit mask* to specify which states are of interest. For example, if a client is interested in active and resolved bundles, it is possible to specify the Bundle ACTIVE | RESOLVED | STARTING states in the mask.

The Bundle Tracker tracks bundles whose state matches the mask. That is, when a bundle is not tracked it adds that bundle to the tracked map when its state matches the mask. If the bundle reaches a new state that is not listed in the mask, the bundle will be removed from the tracked map. If the state changes but the bundle should still be tracked, then the bundle is considered to be modified.

701.4.2 Constructor

The BundleTracker interface defines the following constructors to create BundleTracker objects:

- BundleTracker(BundleContext,int,BundleTrackerCustomizer) – Create a Bundle Tracker that tracks the bundles which state is listed in the mask. The customizer may be null, in that case the callbacks can be implemented in a subclass.

A new BundleTracker object must not begin tracking services until its open method is called.

- open() – Start tracking the bundles, callbacks can occur before this method is called.

701.4.3 Using a Bundle Tracker

Once a BundleTracker object is opened, it begins tracking bundles immediately. A number of methods are available to the bundle developer to monitor the bundles that are being tracked. The BundleTracker class defines the following methods:

- getBundles() – Returns an array of all the tracked bundles.
- getObject(Bundle) – Returns the wrapper object that was returned from the addingBundle method.
- remove(Bundle) – Removes the bundle from the tracked bundles. The removedBundle method is called when the bundle is not in the tracked map.
- size() – Returns the number of bundles being tracked.
- getTrackingCount() – A Bundle Tracker can have bundles added, modified, or removed at any moment in time. The getTrackingCount method is intended to efficiently detect changes in a Bundle Tracker. Every time the Bundle Tracker is changed, it must increase the tracking count.
- isEmpty() – To detect that the tracker has no tracked bundles.
- getTracked() – Return the tracked objects.

701.4.4 Customizing the Bundle Tracker class

The behavior of the BundleTracker class can be customized either by providing a BundleTrackerCustomizer object when the BundleTracker object is constructed, or by sub-classing the BundleTracker class and overriding the BundleTrackerCustomizer methods on the BundleTracker class.

The BundleTrackerCustomizer interface defines these methods:

- addingBundle(Bundle,BundleEvent) – Called whenever a bundle is being added to the BundleTracker object. This method should return a wrapper object, which can be the Bundle object itself. If null is returned, the Bundle must not be further tracked.
- modifiedBundle(Bundle,BundleEvent,T) – Called whenever a tracked bundle is modified. The object that is passed is the object returned from the addingBundle method, the wrapper object.
- removedBundle(Bundle,BundleEvent,T) – Called whenever a tracked bundle is removed from the BundleTracker object. The passed object is the wrapper returned from the addingBundle method.

The BundleEvent object in the previous methods can be null.

When a bundle is being added the OSGi Framework, or when a tracked bundle is modified or uninstalled from the OSGi Framework, the Bundle Tracker must call addingBundle, modifiedBundle, or removedBundle, respectively, on the BundleTrackerCustomizer object (if specified when the BundleTracker object was created); otherwise it must call these methods on itself, allowing them to be overridden in a subclass.

The bundle developer can return a specialized object from addingBundle that differs from the Bundle object. This wrapper object could contain the Bundle object and any associated client specific information. This returned object is then used as the wrapper instead of the Bundle object. When the removedBundle method is called, the wrapper is passed as an argument.

701.4.5 Extender Model

The Bundle Tracker allows the implementation of extenders with surprisingly little effort. The following example checks a manifest header (Http-Mapper) in all active bundles to see if the bundle has resources that need to be mapped to the HTTP service. This extender enables bundles that have no code, just content.

This example is implemented with a BundleTrackerCustomizer implementation, though sub-classing the BundleTracker class is slightly simpler because the open/close methods would be inherited, the tracker field is not necessary and it is not necessary to provide a dummy implementation of modifiedBundle method. However, the Service Tracker example already showed how to use inheritance.

The Extender class must implement the customizer and declare fields for the Http Service and a Bundle Tracker.

```
public class Extender implements BundleTrackerCustomizer<ExtenderContext> {
  final HttpService                    http;
  final BundleTracker<ExtenderContext> tracker;
```

It is necessary to parse the Http-Mapper header. Regular expression allow this to be done very concise.

```
final static Pattern HTTPMAPPER=
  Pattern.compile(
    "\\s*([-/\\w.]+)\\s*=\\s*([-/\\w.]+)\\s*");
```

The Bundle Tracker requires a specialized constructor. This example only works for *active* bundles. This implies that a bundle only provides contents when it is started, enabling an administrator to control the availability.

```
Extender(BundleContext context, HttpService http) {
  tracker = new BundleTracker<ExtenderContext>(
    context,Bundle.ACTIVE, this );
  this.http = http;
}
```

The following method implements the callback from the Bundle Tracker when a new bundle is discovered. In this method a specialized HttpContext object is created that knows how to retrieve its resources from the bundle that was just discovered. This context is registered with the Http Service. If no header is found null is returned so that non-participating bundles are no longer tracked.

```
public ExtenderContext addingBundle(Bundle bundle,
  BundleEvent event) {
  String header = bundle.getHeaders()
    .get("Http-Mapper") + "";
  Matcher match = HTTPMAPPER.matcher(header);
  if (match.matches()) {
```

```
            try {
               ExtenderContext wrapper =
                  new ExtenderContext(bundle, match.group(1));
               http.registerResources(
                  match.group(1), // alias
                  match.group(2), // resource path
                  wrapper         // the http context
               );
               return wrapper;
            } catch (NamespaceException nspe) {
               // error is handled in the fall through
            }
         }
         System.err.println(
           "Invalid header for Http-Mapper: " + header);
         return null;
      }
```

The modifiedBundle method does not have to be implemented because this example is not interested in state changes because the only state of interest is the ACTIVE state. Therefore, the remaining method left to implement is the removedBundle method. If the wrapper object is non-null then we need to unregister the alias to prevent collisions in the http namespace when the bundle is reinstalled or updated.

```
   public void removedBundle(
      Bundle bundle, BundleEvent event,
      ExtenderContext wrapper) {
         http.unregister(wrapper.alias);
   }
```

The remaining methods would be unnecessary if the Extender class had extended the BundleTracker class. The BundleTrackerCustomizer interface requires a dummy implementation of the modifiedBundle method:

```
   public void modifiedBundle(
      Bundle bundle, BundleEvent event, ExtenderContext object) {
      // Nothing to do
   }
```

It is usually not a good idea to start a tracker in a constructor because opening a service tracker will immediately cause a number of callbacks for the existing bundles. If the Extender class was subclassed, then this could call back the uninitialized sub class methods. It is therefore better to separate the initialization from the opening. There is therefore a need for an open and close method.

```
      public void close() {
         tracker.close();
      }
      public void open() {
         tracker.open();
      }
   }
```

The previous example uses an HttpContext subclass that can retrieve resources from the target bundle:

```
   public class ExtenderContext implements HttpContext {
      final Bundle    bundle;
      final String    alias;
      ExtenderContext(Bundle bundle, String alias) {
         this.bundle = bundle;
         this.alias = alias;
      }
      public boolean handleSecurity(
         HttpServletRequest rq, HttpServletResponse rsp) {
         return true;
      }
      public String getMimeType(String name) {
         return null;
      }
      public URL getResource(String name) {
         return bundle.getResource(name);
      }
   }
```

701.5 Security

A tracker contains a BundleContext instance variable that is accessible to the methods in a subclass. A BundleContext object should never be given to other bundles because it is a *capability*. The framework makes allocations based on the bundle context with respect to security and resource management.

The tracker implementations do not have a method to get the BundleContext object, however, subclasses should be careful not to provide such a method if the tracker is given to other bundles.

The services that are being tracked are available via a ServiceTracker. These services are dependent on the BundleContext as well. It is therefore necessary to do a careful security analysis when ServiceTracker objects are given to other bundles. The same counts for the Bundle Tracker. It is strongly advised to not pass trackers to other bundles.

701.5.1 Synchronous Bundle Listener

The Bundle Tracker uses the synchronous bundle listener because it is impossible to provide some of the guarantees the Bundle Tracker provides without handling the events synchronously. Synchronous events can block the complete system, therefore Synchronous Bundle Listeners require AdminPermission[*,LISTENER]. The wildcard * can be replaced with a specifier for the bundles that should be visible to the Bundle Tracker. See *Admin Permission* on page 107 for more information.

Code that calls the open and close methods of Bundle Trackers must therefore have the appropriate Admin Permission.

701.6 Changes

- The Service Tracker is generified
- A new constructor was added that takes a Class object as criterion
- Added BundleTracker.isEmpty and getTracked methods
- Added ServiceTracker.isEmpty, getTracked, and getServices(T[]) methods.

701.7 org.osgi.util.tracker

Tracker Package Version 1.5.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest.

Example import for consumers using the API in this package:

Import-Package: org.osgi.util.tracker; version="[1.5,2.0)"

701.7.1 Summary

- BundleTracker – The BundleTracker class simplifies tracking bundles much like the ServiceTracker simplifies tracking services.
- BundleTrackerCustomizer – The BundleTrackerCustomizer interface allows a BundleTracker to customize the Bundles that are tracked.
- ServiceTracker – The ServiceTracker class simplifies using services from the Framework's service registry.
- ServiceTrackerCustomizer – The ServiceTrackerCustomizer interface allows a ServiceTracker to customize the service objects that are tracked.

701.7.2 Permissions

701.7.3 public class BundleTracker<T> implements BundleTrackerCustomizer<T>

<T> The type of the tracked object.

The BundleTracker class simplifies tracking bundles much like the ServiceTracker simplifies tracking services.

A BundleTracker is constructed with state criteria and a BundleTrackerCustomizer object. A BundleTracker can use the BundleTrackerCustomizer to select which bundles are tracked and to create a customized object to be tracked with the bundle. The BundleTracker can then be opened to begin tracking all bundles whose state matches the specified state criteria.

The getBundles method can be called to get the Bundle objects of the bundles being tracked. The getObject method can be called to get the customized object for a tracked bundle.

The BundleTracker class is thread-safe. It does not call a BundleTrackerCustomizer while holding any locks. BundleTrackerCustomizer implementations must also be thread-safe.

Since 1.4

Concurrency Thread-safe

701.7.3.1 protected final BundleContext context

The Bundle Context used by this BundleTracker.

701.7.3.2 public BundleTracker (BundleContext context , int stateMask , BundleTrackerCustomizer<T> customizer)

context The BundleContext against which the tracking is done.

stateMask The bit mask of the ORing of the bundle states to be tracked.

customizer The customizer object to call when bundles are added, modified, or removed in this BundleTracker. If customizer is null, then this BundleTracker will be used as the BundleTrackerCustomizer and this BundleTracker will call the BundleTrackerCustomizer methods on itself.

□ Create a BundleTracker for bundles whose state is present in the specified state mask.

Bundles whose state is present on the specified state mask will be tracked by this BundleTracker.

See Also Bundle.getState()

701.7.3.3 public T addingBundle (Bundle bundle , BundleEvent event)

bundle The Bundle being added to this BundleTracker object.

event The bundle event which caused this customizer method to be called or null if there is no bundle event associated with the call to this method.

□ Default implementation of the BundleTrackerCustomizer.addingBundle method.

This method is only called when this BundleTracker has been constructed with a null BundleTrackerCustomizer argument.

This implementation simply returns the specified Bundle.

This method can be overridden in a subclass to customize the object to be tracked for the bundle being added.

Returns The specified bundle.

See Also BundleTrackerCustomizer.addingBundle(Bundle, BundleEvent)

701.7.3.4 public void close ( )

□ Close this BundleTracker.

This method should be called when this BundleTracker should end the tracking of bundles.

This implementation calls getBundles() to get the list of tracked bundles to remove.

701.7.3.5 public Bundle[] getBundles ( )

□ Return an array of Bundles for all bundles being tracked by this BundleTracker.

Returns An array of Bundles or null if no bundles are being tracked.

701.7.3.6 public T getObject (Bundle bundle)

bundle The Bundle being tracked.

□ Returns the customized object for the specified Bundle if the specified bundle is being tracked by this BundleTracker.

Returns The customized object for the specified Bundle or null if the specified Bundle is not being tracked.

701.7.3.7 public Map<Bundle,T> getTracked ( )

□ Return a Map with the Bundles and customized objects for all bundles being tracked by this BundleTracker.

Returns A Map with the Bundles and customized objects for all services being tracked by this BundleTracker. If no bundles are being tracked, then the returned map is empty.

Since 1.5

701.7.3.8 public int getTrackingCount ( )

□ Returns the tracking count for this BundleTracker. The tracking count is initialized to 0 when this BundleTracker is opened. Every time a bundle is added, modified or removed from this BundleTracker the tracking count is incremented.

The tracking count can be used to determine if this BundleTracker has added, modified or removed a bundle by comparing a tracking count value previously collected with the current tracking count value. If the value has not changed, then no bundle has been added, modified or removed from this BundleTracker since the previous tracking count was collected.

Returns The tracking count for this BundleTracker or -1 if this BundleTracker is not open.

701.7.3.9 public boolean isEmpty ( )

□ Return if this BundleTracker is empty.

Returns true if this BundleTracker is not tracking any bundles.

Since 1.5

701.7.3.10 public void modifiedBundle (Bundle bundle , BundleEvent event , T object)

bundle The Bundle whose state has been modified.

event The bundle event which caused this customizer method to be called or null if there is no bundle event associated with the call to this method.

object The customized object for the specified Bundle.

□ Default implementation of the BundleTrackerCustomizer.modifiedBundle method.

This method is only called when this BundleTracker has been constructed with a null BundleTrackerCustomizer argument.

This implementation does nothing.

See Also `BundleTrackerCustomizer.modifiedBundle(Bundle, BundleEvent, Object)`

701.7.3.11 public void open ( )

□ Open this BundleTracker and begin tracking bundles.

Bundle which match the state criteria specified when this BundleTracker was created are now tracked by this BundleTracker.

Throws IllegalStateException – If the BundleContext with which this BundleTracker was created is no longer valid.

SecurityException – If the caller and this class do not have the appropriate AdminPermission[context bundle,LISTENER], and the Java Runtime Environment supports permissions.

701.7.3.12 public void remove (Bundle bundle)

bundle The Bundle to be removed.

□ Remove a bundle from this BundleTracker. The specified bundle will be removed from this BundleTracker . If the specified bundle was being tracked then the BundleTrackerCustomizer.removedBundle method will be called for that bundle.

701.7.3.13 public void removedBundle (Bundle bundle , BundleEvent event , T object)

bundle The Bundle being removed.

event The bundle event which caused this customizer method to be called or null if there is no bundle event associated with the call to this method.

object The customized object for the specified bundle.

□ Default implementation of the BundleTrackerCustomizer.removedBundle method.

This method is only called when this BundleTracker has been constructed with a null BundleTrackerCustomizer argument.

This implementation does nothing.

See Also `BundleTrackerCustomizer.removedBundle(Bundle, BundleEvent, Object)`

701.7.3.14 public int size ( )

□ Return the number of bundles being tracked by this BundleTracker.

Returns The number of bundles being tracked.

701.7.4 public interface BundleTrackerCustomizer<T>

<T> The type of the tracked object.

The BundleTrackerCustomizer interface allows a BundleTracker to customize the Bundles that are tracked. A BundleTrackerCustomizer is called when a bundle is being added to a BundleTracker. The BundleTrackerCustomizer can then return an object for the tracked bundle. A BundleTrackerCustomizer is also called when a tracked bundle is modified or has been removed from a BundleTracker.

The methods in this interface may be called as the result of a BundleEvent being received by a BundleTracker. Since BundleEvents are received synchronously by the BundleTracker, it is highly recommended that implementations of these methods do not alter bundle states while being synchronized on any object.

The BundleTracker class is thread-safe. It does not call a BundleTrackerCustomizer while holding any locks. BundleTrackerCustomizer implementations must also be thread-safe.

Since 1.4

Concurrency Thread-safe

701.7.4.1 public T addingBundle (Bundle bundle , BundleEvent event)

bundle The Bundle being added to the BundleTracker.

event The bundle event which caused this customizer method to be called or null if there is no bundle event associated with the call to this method.

□ A bundle is being added to the BundleTracker.

This method is called before a bundle which matched the search parameters of the BundleTracker is added to the BundleTracker. This method should return the object to be tracked for the specified Bundle. The returned object is stored in the BundleTracker and is available from the getObject method.

Returns The object to be tracked for the specified Bundle object or null if the specified Bundle object should not be tracked.

701.7.4.2 public void modifiedBundle (Bundle bundle , BundleEvent event , T object)

bundle The Bundle whose state has been modified.

event The bundle event which caused this customizer method to be called or null if there is no bundle event associated with the call to this method.

object The tracked object for the specified bundle.

□ A bundle tracked by the BundleTracker has been modified.

This method is called when a bundle being tracked by the BundleTracker has had its state modified.

701.7.4.3 public void removedBundle (Bundle bundle , BundleEvent event , T object)

bundle The Bundle that has been removed.

event The bundle event which caused this customizer method to be called or null if there is no bundle event associated with the call to this method.

object The tracked object for the specified bundle.

□ A bundle tracked by the BundleTracker has been removed.

This method is called after a bundle is no longer being tracked by the BundleTracker.

701.7.5 public class ServiceTracker<S,T> implements ServiceTrackerCustomizer<S,T>

<S> The type of the service being tracked.

<T> The type of the tracked object.

The ServiceTracker class simplifies using services from the Framework's service registry.

A ServiceTracker object is constructed with search criteria and a ServiceTrackerCustomizer object. A ServiceTracker can use a ServiceTrackerCustomizer to customize the service objects to be tracked. The ServiceTracker can then be opened to begin tracking all services in the Framework's service registry that match the specified search criteria. The ServiceTracker correctly handles all of the details of listening to ServiceEvents and getting and ungetting services.

The getServiceReferences method can be called to get references to the services being tracked. The getService and getServices methods can be called to get the service objects for the tracked service.

The ServiceTracker class is thread-safe. It does not call a ServiceTrackerCustomizer while holding any locks. ServiceTrackerCustomizer implementations must also be thread-safe.

Concurrency Thread-safe

701.7.5.1 protected final BundleContext context

The Bundle Context used by this ServiceTracker.

701.7.5.2 protected final Filter filter

The Filter used by this ServiceTracker which specifies the search criteria for the services to track.

Since 1.1

701.7.5.3 public ServiceTracker (BundleContext context , ServiceReference<S> reference , ServiceTrackerCustomizer<S,T> customizer)

context The BundleContext against which the tracking is done.

reference The ServiceReference for the service to be tracked.

customizer The customizer object to call when services are added, modified, or removed in this ServiceTracker. If customizer is null, then this ServiceTracker will be used as the ServiceTrackerCustomizer and this ServiceTracker will call the ServiceTrackerCustomizer methods on itself.

□ Create a ServiceTracker on the specified ServiceReference.

The service referenced by the specified ServiceReference will be tracked by this ServiceTracker.

701.7.5.4 public ServiceTracker (BundleContext context , String clazz , ServiceTrackerCustomizer<S,T> customizer)

context The BundleContext against which the tracking is done.

clazz The class name of the services to be tracked.

customizer The customizer object to call when services are added, modified, or removed in this ServiceTracker. If customizer is null, then this ServiceTracker will be used as the ServiceTrackerCustomizer and this ServiceTracker will call the ServiceTrackerCustomizer methods on itself.

□ Create a ServiceTracker on the specified class name.

Services registered under the specified class name will be tracked by this ServiceTracker.

701.7.5.5 public ServiceTracker (BundleContext context , Filter filter , ServiceTrackerCustomizer<S,T> customizer)

context The BundleContext against which the tracking is done.

filter The Filter to select the services to be tracked.

customizer The customizer object to call when services are added, modified, or removed in this ServiceTracker. If customizer is null, then this ServiceTracker will be used as the ServiceTrackerCustomizer and this ServiceTracker will call the ServiceTrackerCustomizer methods on itself.

□ Create a ServiceTracker on the specified Filter object.

Services which match the specified Filter object will be tracked by this ServiceTracker.

Since 1.1

701.7.5.6 public ServiceTracker (BundleContext context , Class<S> clazz , ServiceTrackerCustomizer<S,T> customizer)

context The BundleContext against which the tracking is done.

clazz The class of the services to be tracked.

customizer The customizer object to call when services are added, modified, or removed in this ServiceTracker. If customizer is null, then this ServiceTracker will be used as the ServiceTrackerCustomizer and this ServiceTracker will call the ServiceTrackerCustomizer methods on itself.

□ Create a ServiceTracker on the specified class.

Services registered under the name of the specified class will be tracked by this ServiceTracker.

Since 1.5

701.7.5.7 public T addingService (ServiceReference<S> reference)

reference The reference to the service being added to this ServiceTracker.

□ Default implementation of the ServiceTrackerCustomizer.addingService method.

This method is only called when this ServiceTracker has been constructed with a null ServiceTrackerCustomizer argument.

This implementation returns the result of calling getService on the BundleContext with which this ServiceTracker was created passing the specified ServiceReference.

This method can be overridden in a subclass to customize the service object to be tracked for the service being added. In that case, take care not to rely on the default implementation of removedService to unget the service.

Returns The service object to be tracked for the service added to this ServiceTracker.

See Also ServiceTrackerCustomizer.addingService(ServiceReference)

701.7.5.8 public void close ( )

□ Close this ServiceTracker.

This method should be called when this ServiceTracker should end the tracking of services.

This implementation calls getServiceReferences() to get the list of tracked services to remove.

701.7.5.9 public T getService (ServiceReference<S> reference)

reference The reference to the desired service.

□ Returns the service object for the specified ServiceReference if the specified referenced service is being tracked by this ServiceTracker.

Returns A service object or null if the service referenced by the specified ServiceReference is not being tracked.

701.7.5.10 public T getService ( )

□ Returns a service object for one of the services being tracked by this ServiceTracker.

If any services are being tracked, this implementation returns the result of calling getService(getServiceReference()).

Returns A service object or null if no services are being tracked.

701.7.5.11 public ServiceReference<S> getServiceReference ( )

□ Returns a ServiceReference for one of the services being tracked by this ServiceTracker.

If multiple services are being tracked, the service with the highest ranking (as specified in its service.ranking property) is returned. If there is a tie in ranking, the service with the lowest service ID (as specified in its service.id property); that is, the service that was registered first is returned. This is the same algorithm used by BundleContext.getServiceReference.

This implementation calls getServiceReferences() to get the list of references for the tracked services.

Returns A ServiceReference or null if no services are being tracked.

Since 1.1

701.7.5.12 public ServiceReference<S>[] getServiceReferences ( )

□ Return an array of ServiceReferences for all services being tracked by this ServiceTracker.

Returns Array of ServiceReferences or null if no services are being tracked.

701.7.5.13 public Object[] getServices ( )

□ Return an array of service objects for all services being tracked by this ServiceTracker.

This implementation calls getServiceReferences() to get the list of references for the tracked services and then calls getService(ServiceReference) for each reference to get the tracked service object.

Returns An array of service objects or null if no services are being tracked.

701.7.5.14 public T[] getServices (T[] array)

array An array into which the tracked service objects will be stored, if the array is large enough.

□ Return an array of service objects for all services being tracked by this ServiceTracker. The runtime type of the returned array is that of the specified array.

This implementation calls getServiceReferences() to get the list of references for the tracked services and then calls getService(ServiceReference) for each reference to get the tracked service object.

Returns An array of service objects being tracked. If the specified array is large enough to hold the result, then the specified array is returned. If the specified array is longer then necessary to hold the result, the array element after the last service object is set to null. If the specified array is not large enough to hold the result, a new array is created and returned.

Since 1.5

701.7.5.15 public SortedMap<ServiceReference<S>,T> getTracked ( )

□ Return a SortedMap of the ServiceReferences and service objects for all services being tracked by this ServiceTracker. The map is sorted in reverse natural order of ServiceReference. That is, the first entry is the service with the highest ranking and the lowest service id.

Returns A SortedMap with the ServiceReferences and service objects for all services being tracked by this ServiceTracker. If no services are being tracked, then the returned map is empty.

Since 1.5

701.7.5.16 public int getTrackingCount ( )

□ Returns the tracking count for this ServiceTracker. The tracking count is initialized to 0 when this ServiceTracker is opened. Every time a service is added, modified or removed from this ServiceTracker, the tracking count is incremented.

The tracking count can be used to determine if this ServiceTracker has added, modified or removed a service by comparing a tracking count value previously collected with the current tracking count value. If the value has not changed, then no service has been added, modified or removed from this ServiceTracker since the previous tracking count was collected.

Returns The tracking count for this ServiceTracker or -1 if this ServiceTracker is not open.

Since 1.2

701.7.5.17 public boolean isEmpty ( )

□ Return if this ServiceTracker is empty.

Returns true if this ServiceTracker is not tracking any services.

Since 1.5

701.7.5.18 public void modifiedService (ServiceReference<S> reference , T service)

reference The reference to modified service.

service The service object for the modified service.

□ Default implementation of the ServiceTrackerCustomizer.modifiedService method.

This method is only called when this ServiceTracker has been constructed with a null ServiceTrackerCustomizer argument.

This implementation does nothing.

See Also `ServiceTrackerCustomizer.modifiedService(ServiceReference, Object)`

701.7.5.19 public void open ( )

□ Open this ServiceTracker and begin tracking services.

This implementation calls open(false).

Throws `IllegalStateException` – If the BundleContext with which this ServiceTracker was created is no longer valid.

See Also open(boolean)

701.7.5.20 public void open (boolean trackAllServices)

trackAllServices If true, then this ServiceTracker will track all matching services regardless of class loader accessibility. If false, then this ServiceTracker will only track matching services which are class loader accessible to the bundle whose BundleContext is used by this ServiceTracker.

□ Open this ServiceTracker and begin tracking services.

Services which match the search criteria specified when this ServiceTracker was created are now tracked by this ServiceTracker.

Throws `IllegalStateException` – If the BundleContext with which this ServiceTracker was created is no longer valid.

Since 1.3

701.7.5.21 public void remove (ServiceReference<S> reference)

reference The reference to the service to be removed.

□ Remove a service from this ServiceTracker. The specified service will be removed from this ServiceTracker. If the specified service was being tracked then the ServiceTrackerCustomizer.removedService method will be called for that service.

701.7.5.22 public void removedService (ServiceReference<S> reference , T service)

reference The reference to removed service.

service The service object for the removed service.

□ Default implementation of the ServiceTrackerCustomizer.removedService method.

This method is only called when this ServiceTracker has been constructed with a null ServiceTrackerCustomizer argument.

This implementation calls ungetService, on the BundleContext with which this ServiceTracker was created, passing the specified ServiceReference.

This method can be overridden in a subclass. If the default implementation of addingService method was used, this method must unget the service.

See Also ServiceTrackerCustomizer.removedService(ServiceReference, Object)

701.7.5.23 public int size ( )

□ Return the number of services being tracked by this ServiceTracker.

Returns The number of services being tracked.

701.7.5.24 public T waitForService (long timeout) throws InterruptedException

timeout The time interval in milliseconds to wait. If zero, the method will wait indefinitely.

□ Wait for at least one service to be tracked by this ServiceTracker. This method will also return when this ServiceTracker is closed.

It is strongly recommended that waitForService is not used during the calling of the BundleActivator methods. BundleActivator methods are expected to complete in a short period of time.

This implementation calls getService() to determine if a service is being tracked.

Returns Returns the result of getService().

Throws InterruptedException – If another thread has interrupted the current thread.

IllegalArgumentException – If the value of timeout is negative.

701.7.6 public interface ServiceTrackerCustomizer<S,T>

<S> The type of the service being tracked.

<T> The type of the tracked object.

The ServiceTrackerCustomizer interface allows a ServiceTracker to customize the service objects that are tracked. A ServiceTrackerCustomizer is called when a service is being added to a ServiceTracker. The ServiceTrackerCustomizer can then return an object for the tracked service. A ServiceTrackerCustomizer is also called when a tracked service is modified or has been removed from a ServiceTracker.

The methods in this interface may be called as the result of a ServiceEvent being received by a ServiceTracker. Since ServiceEvents are synchronously delivered by the Framework, it is highly recommended that implementations of these methods do not register (BundleContext.registerService), modify (ServiceRegistration.setProperties) or unregister (ServiceRegistration.unregister) a service while being synchronized on any object.

The ServiceTracker class is thread-safe. It does not call a ServiceTrackerCustomizer while holding any locks. ServiceTrackerCustomizer implementations must also be thread-safe.

Concurrency Thread-safe

701.7.6.1 public T addingService (ServiceReference<S> reference)

reference The reference to the service being added to the ServiceTracker.

□ A service is being added to the ServiceTracker.

This method is called before a service which matched the search parameters of the ServiceTracker is added to the ServiceTracker. This method should return the service object to be tracked for the specified ServiceReference. The returned service object is stored in the ServiceTracker and is available from the getService and getServices methods.

Returns The service object to be tracked for the specified referenced service or null if the specified referenced service should not be tracked.

701.7.6.2 public void modifiedService (ServiceReference<S> reference , T service)

reference The reference to the service that has been modified.

service The service object for the specified referenced service.

□ A service tracked by the ServiceTracker has been modified.

This method is called when a service being tracked by the ServiceTracker has had it properties modified.

701.7.6.3 public void removedService (ServiceReference<S> reference , T service)

reference The reference to the service that has been removed.

service The service object for the specified referenced service.

□ A service tracked by the ServiceTracker has been removed.

This method is called after a service is no longer being tracked by the ServiceTracker.

702 XML Parser Service Specification

Version 1.0

702.1 Introduction

The Extensible Markup Language (XML) has become a popular method of describing data. As more bundles use XML to describe their data, a common XML Parser becomes necessary in an embedded environment in order to reduce the need for space. Not all XML Parsers are equivalent in function, however, and not all bundles have the same requirements on an XML parser.

This problem was addressed in the Java API for XML Processing, see [4] *JAXP* for Java 2 Standard Edition and Enterprise Edition. This specification addresses how the classes defined in JAXP can be used in an OSGi Service Platform. It defines how:

- Implementations of XML parsers can become available to other bundles
- Bundles can find a suitable parser
- A standard parser in a JAR can be transformed to a bundle

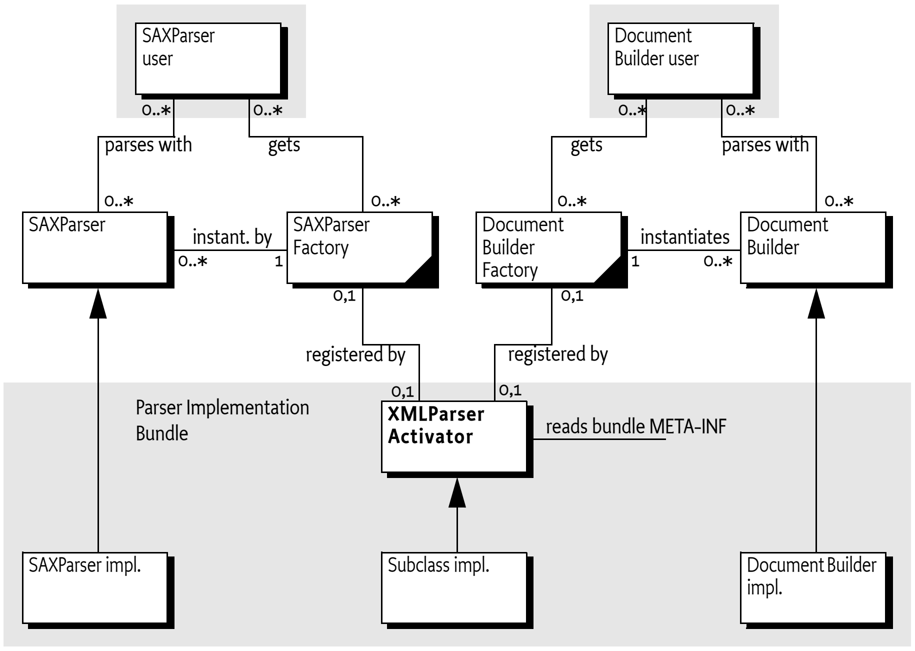
702.1.1 Essentials

- *Standards* – Leverage existing standards in Java based XML parsing: JAXP, SAX and DOM
- *Unmodified JAXP code* – Run unmodified JAXP code
- *Simple* – It should be easy to provide a SAX or DOM parser as well as easy to find a matching parser
- *Multiple* – It should be possible to have multiple implementations of parsers available
- *Extendable* – It is likely that parsers will be extended in the future with more functionality

702.1.2 Entities

- *XMLParserActivator* – A utility class that registers a parser factory from declarative information in the Manifest file.
- *SAXParserFactory* – A class that can create an instance of a `SAXParser` class.
- *DocumentBuilderFactory* – A class that can create an instance of a `DocumentBuilder` class.
- *SAXParser* – A parser, instantiated by a `SaxParserFactory` object, that parses according to the SAX specifications.
- *DocumentBuilder* – A parser, instantiated by a `DocumentBuilderFactory`, that parses according to the DOM specifications.

Figure 702.1 XML Parsing diagram



702.1.3 Operations

A bundle containing a SAX or DOM parser is started. This bundle registers a SAXParserFactory and/or a DocumentBuilderFactory service object with the Framework. Service registration properties describe the features of the parsers to other bundles. A bundle that needs an XML parser will get a SAXParserFactory or DocumentBuilderFactory service object from the Framework service registry. This object is then used to instantiate the requested parsers according to their specifications.

702.2 JAXP

XML has become very popular in the last few years because it allows the interchange of complex information between different parties. Though only a single XML standard exists, there are multiple APIs to XML parsers, primarily of two types:

- The Simple API for XML (SAX1 and SAX2)
- Based on the Document Object Model (DOM 1 and 2)

Both standards, however, define an abstract API that can be implemented by different vendors.

A given XML Parser implementation may support either or both of these parser types by implementing the org.w3c.dom and/or org.xml.sax packages. In addition, parsers have characteristics such as whether they are validating or non-validating parsers and whether or not they are name-space aware.

An application which uses a specific XML Parser must code to that specific parser and become coupled to that specific implementation. If the parser has implemented [4] *JAXP*, however, the application developer can code against SAX or DOM and let the runtime environment decide which parser implementation is used.

JAXP uses the concept of a *factory*. A factory object is an object that abstracts the creation of another object. JAXP defines a DocumentBuilderFactory and a SAXParserFactory class for this purpose.

JAXP is implemented in the javax.xml.parsers package and provides an abstraction layer between an application and a specific XML Parser implementation. Using JAXP, applications can choose to use any JAXP compliant parser without changing any code, simply by changing a System property which specifies the SAX- and DOM factory class names.

In JAXP, the default factory is obtained with a static method in the SAXParserFactory or DocumentBuilderFactory class. This method will inspect the associated System property and create a new instance of that class.

702.3 XML Parser service

The current specification of JAXP has the limitation that only one of each type of parser factories can be registered. This specification specifies how multiple SAXParserFactory objects and DocumentBuilderFactory objects can be made available to bundles simultaneously.

Providers of parsers should register a JAXP factory object with the OSGi service registry under the factory class name. Service properties are used to describe whether the parser:

- Is validating
- Is name-space aware
- Has additional features

With this functionality, bundles can query the OSGi service registry for parsers supporting the specific functionality that they require.

702.4 Properties

Parsers must be registered with a number of properties that qualify the service. In this specification, the following properties are specified:

- PARSER_NAMESPACEAWARE – The registered parser is aware of name-spaces. Name-spaces allow an XML document to consist of independently developed DTDs. In an XML document, they are recognized by the xmlns attribute and names prefixed with an abbreviated name-space identifier, like: <xsl:if ...>. The type is a Boolean object that must be true when the parser supports name-spaces. All other values, or the absence of the property, indicate that the parser does not implement name-spaces.
- PARSER_VALIDATING – The registered parser can read the DTD and can validate the XML accordingly. The type is a Boolean object that must true when the parser is validating. All other values, or the absence of the property, indicate that the parser does not validate.

702.5 Getting a Parser Factory

Getting a parser factory requires a bundle to get the appropriate factory from the service registry. In a simple case in which a non-validating, non-name-space aware parser would suffice, it is best to use getServiceReference(String).

```
DocumentBuilder getParser(BundleContext context)
    throws Exception {
    ServiceReference ref = context.getServiceReference(
        DocumentBuilderFactory.class.getName() );
    if ( ref == null )
        return null;
    DocumentBuilderFactory factory =
        (DocumentBuilderFactory) context.getService(ref);
    return factory.newDocumentBuilder();
}
```

In a more demanding case, the filtered version allows the bundle to select a parser that is validating and name-space aware:

```
SAXParser getParser(BundleContext context)
   throws Exception {
   ServiceReference refs[] = context.getServiceReferences(
      SAXParserFactory.class.getName(),
         "(&(parser.namespaceAware=true)"
      +  "(parser.validating=true))" );
   if ( refs == null )
      return null;
   SAXParserFactory factory =
      (SAXParserFactory) context.getService(refs[0]);
   return factory.newSAXParser();
}
```

702.6 Adapting a JAXP Parser to OSGi

If an XML Parser supports JAXP, then it can be converted to an OSGi aware bundle by adding a BundleActivator class which registers an XML Parser Service. The utility org.osgi.util.xml.XMLParserActivator class provides this function and can be added (copied, not referenced) to any XML Parser bundle, or it can be extended and customized if desired.

702.6.1 JAR Based Services

Its functionality is based on the definition of the [5] *JAR File specification, services directory*. This specification defines a concept for service providers. A JAR file can contain an implementation of an abstractly defined service. The class (or classes) implementing the service are designated from a file in the META-INF/services directory. The name of this file is the same as the abstract service class.

The content of the UTF-8 encoded file is a list of class names separated by new lines. White space is ignored and the number sign ('#' or \u0023) is the comment character.

JAXP uses this service provider mechanism. It is therefore likely that vendors will place these service files in the META-INF/services directory.

702.6.2 XMLParserActivator

To support this mechanism, the XML Parser service provides a utility class that should be normally delivered with the OSGi Service Platform implementation. This class is a Bundle Activator and must start when the bundle is started. This class is copied into the parser bundle, and *not* imported.

The start method of the utility BundleActivator class will look in the META-INF/services service provider directory for the files javax.xml.parsers.SAXParserFactory (SAXFACTORYNAME) or javax.xml.parsers.DocumentBuilderFactory (DOMFACTORYNAME). The full path name is specified in the constants SAXCLASSFILE and DOMCLASSFILE respectively.

If either of these files exist, the utility BundleActivator class will parse the contents according to the specification. A service provider file can contain multiple class names. Each name is read and a new instance is created. The following example shows the possible content of such a file:

```
# ACME example SAXParserFactory file
com.acme.saxparser.SAXParserFast        # Fast
com.acme.saxparser.SAXParserValidating  # Validates
```

Both the javax.xml.parsers.SAXParserFactory and the javax.xml.parsers.DocumentBuilderFactory provide methods that describe the features of the parsers they can create. The XMLParserActivator activator will use these methods to set the values of the properties, as defined in *Properties* on page 461, that describe the instances.

702.6.3 Adapting an Existing JAXP Compatible Parser

To incorporate this bundle activator into a XML Parser Bundle, do the following:

- If SAX parsing is supported, create a /META-INF/services/javax.xml.parsers.SAXParserFactory resource file containing the class names of the SAXParserFactory classes.
- If DOM parsing is supported, create a /META-INF/services/javax.xml.parsers.DocumentBuilderFactory file containing the fully qualified class names of the DocumentBuilderFactory classes.
- Create manifest file which imports the packages org.w3c.dom, org.xml.sax, and javax.xml.parsers.
- Add a Bundle-Activator header to the manifest pointing to the XMLParserActivator, the sub-class that was created, or a fully custom one.
- If the parsers support attributes, properties, or features that should be registered as properties so they can be searched, extend the XMLParserActivator class and override setSAXProperties(javax.xml.parsers.SAXParserFactory,Hashtable) and setDOMProperties(javax.xml.parsers.DocumentBuilderFactory,Hashtable).
- Ensure that custom properties are put into the Hashtable object. JAXP does not provide a way for XMLParserActivator to query the parser to find out what properties were added.
- Bundles that extend the XMLParserActivator class must call the original methods via super to correctly initialize the XML Parser Service properties.
- Compile this class into the bundle.
- Install the new XML Parser Service bundle.
- Ensure that the org.osgi.util.xml.XMLParserActivator class is contained in the bundle.

702.7 Usage of JAXP

A single bundle should export the JAXP, SAX, and DOM APIs. The version of contained packages must be appropriately labeled. JAXP 1.1 or later is required which references SAX 2 and DOM 2. See [4] *JAXP* for the exact version dependencies.

This specification is related to related packages as defined in the JAXP 1.1 document. Table 702.1 contains the expected minimum versions.

Table 702.1 JAXP 1.1 minimum package versions

Package	Minimum Version
javax.xml.parsers	1.1
org.xml.sax	2.0
org.xml.sax.helpers	2.0
org.xsml.sax.ext	1.0
org.w3c.dom	2.0

The Xerces project from the Apache group, [6] *Xerces 2 Java Parser*, contains a number libraries that implement the necessary APIs. These libraries can be wrapped in a bundle to provide the relevant packages.

702.8 Security

A centralized XML parser is likely to see sensitive information from other bundles. Provisioning an XML parser should therefore be limited to trusted bundles. This security can be achieved by providing ServicePermission[javax.xml.parsers.DocumentBuilderFactory | javax.xml.parsers.SAXFactory, REGISTER] to only trusted bundles.

Using an XML parser is a common function, and ServicePermission[javax.xml.parsers.DOMParserFactory | javax.xml.parsers.SAXFactory, GET] should not be restricted.

The XML parser bundle will need FilePermission[<<ALL FILES>>,READ] for parsing of files because it is not known beforehand where those files will be located. This requirement further implies that the XML parser is a system bundle that must be fully trusted.

702.9 org.osgi.util.xml

XML Parser Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest.

Example import for consumers using the API in this package:

Import-Package: org.osgi.util.xml; version="[1.0,2.0)"

702.9.1 public class XMLParserActivator implements BundleActivator , ServiceFactory

A BundleActivator class that allows any JAXP compliant XML Parser to register itself as an OSGi parser service. Multiple JAXP compliant parsers can concurrently register by using this BundleActivator class. Bundles who wish to use an XML parser can then use the framework's service registry to locate available XML Parsers with the desired characteristics such as validating and namespace-aware.

The services that this bundle activator enables a bundle to provide are:

- javax.xml.parsers.SAXParserFactory(SAXFACTORYNAME)
- javax.xml.parsers.DocumentBuilderFactory(DOMFACTORYNAME)

The algorithm to find the implementations of the abstract parsers is derived from the JAR file specifications, specifically the Services API.

An XMLParserActivator assumes that it can find the class file names of the factory classes in the following files:

- /META-INF/services/javax.xml.parsers.SAXParserFactory is a file contained in a jar available to the runtime which contains the implementation class name(s) of the SAXParserFactory.
- /META-INF/services/javax.xml.parsers.DocumentBuilderFactory is a file contained in a jar available to the runtime which contains the implementation class name(s) of the DocumentBuilderFactory

If either of the files does not exist, XMLParserActivator assumes that the parser does not support that parser type.

XMLParserActivator attempts to instantiate both the SAXParserFactory and the DocumentBuilderFactory. It registers each factory with the framework along with service properties:

- PARSER_VALIDATING- indicates if this factory supports validating parsers. It's value is a Boolean.

- PARSER_NAMESPACEAWARE- indicates if this factory supports namespace aware parsers It's value is a Boolean.

Individual parser implementations may have additional features, properties, or attributes which could be used to select a parser with a filter. These can be added by extending this class and overriding the setSAXProperties and setDOMProperties methods.

Concurrency Thread-safe

702.9.1.1 public static final String DOMCLASSFILE = "/META-INF/services/ javax.xml.parsers.DocumentBuilderFactory"

Fully qualified path name of DOM Parser Factory Class Name file

702.9.1.2 public static final String DOMFACTORYNAME = "javax.xml.parsers.DocumentBuilderFactory"

Filename containing the DOM Parser Factory Class name. Also used as the basis for the SERVICE_PID registration property.

702.9.1.3 public static final String PARSER_NAMESPACEAWARE = "parser.namespaceAware"

Service property specifying if factory is configured to support namespace aware parsers. The value is of type Boolean.

702.9.1.4 public static final String PARSER_VALIDATING = "parser.validating"

Service property specifying if factory is configured to support validating parsers. The value is of type Boolean.

702.9.1.5 public static final String SAXCLASSFILE = "/META-INF/services/ javax.xml.parsers.SAXParserFactory"

Fully qualified path name of SAX Parser Factory Class Name file

702.9.1.6 public static final String SAXFACTORYNAME = "javax.xml.parsers.SAXParserFactory"

Filename containing the SAX Parser Factory Class name. Also used as the basis for the SERVICE_PID registration property.

702.9.1.7 public XMLParserActivator ( )

702.9.1.8 public Object getService (Bundle bundle , ServiceRegistration registration)

bundle The bundle using the service.

registration The ServiceRegistration object for the service.

□ Creates a new XML Parser Factory object.

A unique XML Parser Factory object is returned for each call to this method.

The returned XML Parser Factory object will be configured for validating and namespace aware support as specified in the service properties of the specified ServiceRegistration object. This method can be overridden to configure additional features in the returned XML Parser Factory object.

Returns A new, configured XML Parser Factory object or null if a configuration error was encountered

702.9.1.9 public void setDOMProperties (DocumentBuilderFactory factory , Hashtable props)

factory - the DocumentBuilderFactory object

props - Hashtable of service properties.

□ Set the customizable DOM Parser Service Properties.

This method attempts to instantiate a validating parser and a namespace aware parser to determine if the parser can support those features. The appropriate properties are then set in the specified props object.

This method can be overridden to add additional DOM2 features and properties. If you want to be able to filter searches of the OSGi service registry, this method must put a key, value pair into the properties object for each feature or property. For example, properties.put("http://www.acme.com/features/foo", Boolean.TRUE);

702.9.1.10 public void setSAXProperties (SAXParserFactory factory , Hashtable properties)

factory - the SAXParserFactory object

properties - the properties object for the service

☐ Set the customizable SAX Parser Service Properties.

This method attempts to instantiate a validating parser and a namespace aware parser to determine if the parser can support those features. The appropriate properties are then set in the specified properties object.

This method can be overridden to add additional SAX2 features and properties. If you want to be able to filter searches of the OSGi service registry, this method must put a key, value pair into the properties object for each feature or property. For example, properties.put("http://www.acme.com/features/foo", Boolean.TRUE);

702.9.1.11 public void start (BundleContext context) throws Exception

context The execution context of the bundle being started.

☐ Called when this bundle is started so the Framework can perform the bundle-specific activities necessary to start this bundle. This method can be used to register services or to allocate any resources that this bundle needs.

This method must complete and return to its caller in a timely manner.

This method attempts to register a SAX and DOM parser with the Framework's service registry.

Throws `Exception` – If this method throws an exception, this bundle is marked as stopped and the Framework will remove this bundle's listeners, unregister all services registered by this bundle, and release all services used by this bundle.

702.9.1.12 public void stop (BundleContext context) throws Exception

context The execution context of the bundle being stopped.

☐ This method has nothing to do as all active service registrations will automatically get unregistered when the bundle stops.

Throws `Exception` – If this method throws an exception, the bundle is still marked as stopped, and the Framework will remove the bundle's listeners, unregister all services registered by the bundle, and release all services used by the bundle.

702.9.1.13 public void ungetService (Bundle bundle , ServiceRegistration registration , Object service)

bundle The bundle releasing the service.

registration The `ServiceRegistration` object for the service.

service The XML Parser Factory object returned by a previous call to the `getService` method.

☐ Releases a XML Parser Factory object.

702.10 References

[1] *XML*
http://www.w3.org/XML

[2] *SAX*
http://www.saxproject.org/

[3] *DOM Java Language Binding*
http://www.w3.org/TR/REC-DOM-Level-1/java-language-binding.html

[4] *JAXP*
http://jaxp.java.net/

[5] *JAR File specification, services directory*
http://download.oracle.com/javase/1.4.2/docs/guide/jar/jar.html

[6] *Xerces 2 Java Parser*
http://xerces.apache.org/xerces2-j/

End Of Document