OSGi Service Platform, Home Automation Specification Release 4

OSGi Service Platform Home Automation

The OSGi Alliance

Release 4
August 2005

IOS
Press
Ohmsha

Amsterdam • Berlin • Oxford • Tokyo • Washington, DC

Table Of Contents

The Open Services Gateway Initiative
Bishop Ranch 2
2694 Bishop Drive
Suite 275
San Ramon
CA 94583 USA

ISBN 1 58603 311 5 (IOS Press)
ISBN 4-274-90559-4 (Ohmsha)
Library of Congress Control Number: 2003107481

Publisher

IOS Press
Nieuwe Hemweg 6B
1013 BG Amsterdam
The Netherlands
fax: +31 20 620 3419
e-mail: order@iospress.nl

Distributor in the UK and Ireland
IOS Press/Lavis Marketing
73 Lime Walk
Headington
Oxford OX3 7AD
England
fax: +44 1865 75 0079

Distributor in the USA and Canada
IOS Press, Inc.
5795-G Burke Centre Parkway
Burke, VA 22015
USA
fax: +1 703 323 3668
e-mail: iosbooks@iospress.com

Distributor in Germany, Austria and Switzerland
IOS Press/LSL.de
Gerichtsweg 28
D-04103 Leipzig
Germany
fax: +49 341 995 4255

Distributor in Japan

Ohmsha, Ltd.
3-1 Kanda Nishiki-cho
Chiyoda-ku, Tokyo 101-8460
Japan
fax: +81 3 3233 2426

LEGAL NOTICE

The publisher is not responsible for the use which might be made of the following information.

PRINTED IN THE NETHERLANDS

LEGAL TERMS AND CONDITIONS REGARDING SPECIFICATION

Trademarks

Feedback

This specification can be downloaded from the OSGi web site:
http:// www.osgi.org.

Comments about this specification can be mailed to:
speccomments@mail.osgi.org

OSGi Member Companies

4DHomeNet, Inc.

Acunia

Alpine Electronics Europe Gmbh

AMI-C

Atinav Inc.

BellSouth Telecommunications, Inc.

BMW

Bombardier Transportation

Cablevision Systems

Coactive Networks

Connected Systems, Inc.

Deutsche Telekom

Easenergy, Inc.

Echelon Corporation

Electricite de France (EDF)

Elisa Communications Corporation

Ericsson

Espial Group, Inc.

ETRI

France Telecom

Gatespace AB

Hewlett-Packard

IBM Corporation

ITP AS

Jentro AG

KDD R&D Laboratories Inc.

Legend Computer System Ltd.

Lucent Technologies

Metavector Technologies

Mitsubishi Electric Corporation

Motorola, Inc.

NTT

Object XP AG

On Technology UK, Ltd

Oracle Corporation

P&S Datacom Corporation

Panasonic

Patriot Scientific Corp. (PTSC)

Philips

ProSyst Software AG

Robert Bosch Gmbh

Samsung Electronics Co., LTD

Schneider Electric SA

Siemens VDO Automotive

Sharp Corporation

Sonera Corporation

Sprint Communications Company, L.P.

Sony Corporation

Sun Microsystems

TAC AB

Telcordia Technologies

Telefonica I+D

Telia Research

Texas Instruments, Inc.

Toshiba Corporation

Verizon

Whirlpool Corporation

Wind River Systems

OSGi Board and Officers

	Rafiul Ahad	VP of Product Development, Wireless and Voice Division, *Oracle*
VP Americas	Dan Bandera	Program Director & BLM for Client & OEM Technology, *IBM Corporation*
President	John R. Barr, Ph.D.	Director, Standards Realization, Corporate Offices, *Motorola, Inc.*
	Maurizio S. Beltrami	Technology Manager Interconnectivity, *Philips Consumer Electronics*
	Hans-Werner Bitzer M.A.	Head of Section Smart Home Products, *Deutsche Telekom AG*
	Steven Buytaert	Co-Founder and Co-CEO, *ACUNIA*
VP Asia Pacific	R. Lawrence Chan	Vice President Asia Pacific *Echelon Corporation*
CPEG chair	BJ Hargrave	OSGi Fellow and Senior Software Engineer, *IBM Corporation*
Technology Officer	*and editor*	
	Peter Kriens	OSGi Fellow and CEO, *aQute*
Treasurer	Jeff Lund	Vice President, Business Development & Corporate Marketing , *Echelon Corporation*
Executive Director	Dave Marples	Vice President, *Global Inventures, Inc.*
	Hans-Ulrich Michel	Project Manager Information, Communication and Telematics, *BMW*
Secretary	Stan Moyer	Strategic Research Program Manager *Telcordia Technologies, Inc.*
	Behfar Razavi	Sr. Engineering Manager, Java Telematics Technology, *Sun Microsystems, Inc.*
VP Marketing	Susan Schwarze, PhD.	Marketing Director, *ProSyst*
VP Europe, Middle	*East and Africa*	
	Staffan Truvé	Chairman, *Gatespace*

Foreword

John Barr, *President OSGi*

1 Introduction

The Open Services Gateway Initiative (OSGi™) was founded in March 1999. Its mission is to create open specifications for the network delivery of managed services to local networks and devices. The OSGi organization is the leading standard for next-generation Internet services to homes, cars, small offices, and other environments.

The OSGi service platform specification delivers an open, common architecture for service providers, developers, software vendors, gateway operators and equipment vendors to develop, deploy and manage services in a coordinated fashion. It enables an entirely new category of smart devices due to its flexible and managed deployment of services. The primary targets for the OSGi specifications are set top boxes, service gateways, cable modems, consumer electronics, PCs, industrial computers, cars and more. These devices that implement the OSGi specifications will enable service providers like telcos, cable operators, utilities, and others to deliver differentiated and valuable services over their networks.

This is the third release of the OSGi service platform specification developed by representatives from OSGi member companies. The OSGi Service Platform Release 3 mostly extends the existing APIs into new areas. The few modifications to existing APIs are backward compatible so that applications for previous releases should run unmodified on release 3 Frameworks. The built-in version management mechanisms allow bundles written for the new release to adapt to the old Framework implementations, if necessary.

1.1 Sections

1.2 What is New

1.3 Reader Level

This specification is written for the following audiences:

- Application developers
- Framework and system service developers (system developers)
- Architects

This specification assumes that the reader has at least one year of practical experience in writing Java programs. Experience with embedded systems and server environments is a plus. Application developers must be aware that the OSGi environment is significantly more dynamic than traditional desktop or server environments.

System developers require a *very* deep understanding of Java. At least three years of Java coding experience in a system environment is recommended. A Framework implementation will use areas of Java that are not normally encountered in traditional applications. Detailed understanding is required of class loaders, garbage collection, Java 2 security, and Java native library loading.

Architects should focus on the introduction of each subject. This introduction contains a general overview of the subject, the requirements that influenced its design, and a short description of its operation as well as the entities that are used. The introductory sections require knowledge of Java concepts like classes and interfaces, but should not require coding experience.

Most of these specifications are equally applicable to application developers and system developers.

1.4 Conventions and Terms

1.4.1 Typography

A fixed width, non-serif typeface (`sample`) indicates the term is a Java package, class, interface, or member name. Text written in this typeface is always related to coding.

Emphasis (*sample*) is used the first time an important concept is introduced.

When an example contains a line that must be broken over multiple lines, the « character is used. Spaces must be ignored in this case. For example:

```
http://www.acme.com/sp/ «
file?abc=12
```

is equivalent to:

```
http://www.acme.com/sp/file?abc=12
```

In many cases in these specifications, a syntax must be described. This syntax is based on the following symbols:

```
*            Repetition of the previous element zero or
             more times, e.g. ( ’,’ list ) *
+            Repetition one or more times
?            Previous element is optional
( ... )      Grouping
’...’        Literal
|            Or
[...]        Set (one of)
..           list, e.g. 1..5 is the list 1 2 3 4 5
<...>        Externally defined token
digit        ::= [0..9]
alpha        ::= [a..zA..Z]
token        ::= alpha(alpha|digit|’_’|’-’)*
quoted-string::= ’"’ ... ’"’
jar-path     ::= file[’/’file]
file         A valid file name in a zip file which
```

```
                  has no restrictions except that it may not
                  contain a '/'.
```

Spaces are ignored unless specifically noted.

1.4.2 Object Oriented Terminology

Concepts like classes, interfaces, objects, and services are distinct but subtly different. For example, “LogService” could mean an instance of the class LogService, could refer to the class LogService, or could indicate the functionality of the overall Log Service. Experts usually understand the meaning from the context, but this understanding requires mental effort. To highlight these subtle differences, the following conventions are used.

When the class is intended, its name is spelled exactly as in the Java source code and displayed in a fixed width typeface: for example the “HttpService class”, “a method in HttpContext” or “a javax.servlet.Servlet object”. A class name is fully qualified, like javax.servlet.Servlet, when the package is not obvious from the context nor is it in one of the well known java packages like java.lang, java.io, java.util and java.net. Otherwise, the package is omitted like in String.

Exception and permission classes are not followed by the word “object”. Readability is improved when the “object” suffix is avoided. For example, “to throw a SecurityException” and to “to have FilePermission” instead of "to have a FilePermission object".

Permissions can further be qualified with their actions. ServicePermission[GET|REGISTER,com.acme.*] means a ServicePermission with the action GET and REGISTER for all service names starting with com.acme. A ServicePermission[REGISTER, Producer|Consumer] means the GET ServicePermission for the Producer or Consumer class.

When discussing functionality of a class rather than the implementation details, the class name is written as normal text. This convention is often used when discussing services. For example, “the User Admin service”.

Some services have the word “Service” embedded in their class name. In those cases, the word “service” is only used once but is written with an upper case S. For example, “the Log Service performs”.

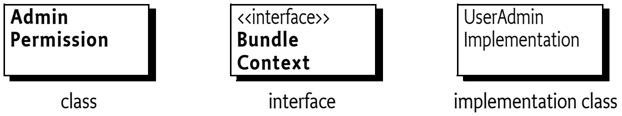
Service objects are registered with the OSGi Framework. Registration consists of the service object, a set of properties, and a list of classes and interfaces implemented by this service object. The classes and interfaces are used for type safety *and* naming. Therefore, it is said that a service object is registered *under* a class/interface. For example, “This service object is registered under PermissionAdmin.”

1.4.3 Diagrams

The diagrams in this document illustrate the specification and are not normative. Their purpose is to provide a high-level overview on a single page. The following paragraphs describe the symbols and conventions used in these diagrams.

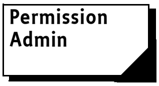
Classes or interfaces are depicted as rectangles, as in Figure 1. Interfaces are indicated with the qualifier <<interface>> as the first line. The name of the class/interface is indicated in bold when it is part of the specification. Implementation classes are sometimes shown to demonstrate a possible implementation. Implementation class names are shown in plain text. In certain cases class names are abbreviated. This is indicated by ending the abbreviation with a period.

Figure 1 *Class and interface symbol*



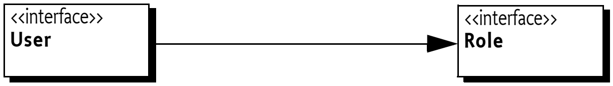
If an interface or class is used as a service object, it will have a black triangle in the bottom right corner.

Figure 2 *Service symbol*



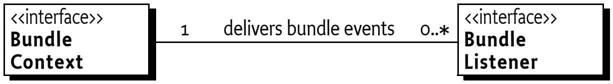
Inheritance (the extends or implements keyword in Java class definitions) is indicated with an arrow. Figure 3 shows that User implements or extends Role.

Figure 3 *Inheritance (implements or extends) symbol*



Relations are depicted with a line. The cardinality of the relation is given explicitly when relevant. Figure 4 shows that each (1) BundleContext object is related to 0 or more BundleListener objects, and that each BundleListener object is related to a single BundleContext object. Relations usually have some description associated with them. This description should be read from left to right and top to bottom, and includes the classes on both sides. For example: "A BundleContext object delivers bundle events to zero or more BundleListener objects."

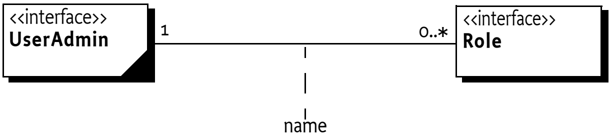
Figure 4 *Relations symbol*



Associations are depicted with a dashed line. Associations are between classes, and an association can be placed on a relation. For example, "every ServiceRegistration object has an associated ServiceReference object." This association does not have to be a hard relationship, but could be derived in some way.

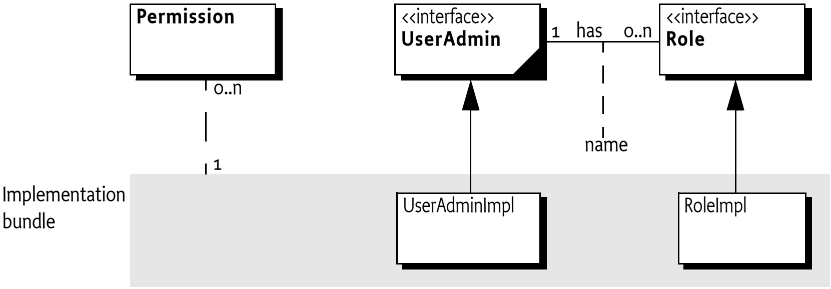
When a relationship is qualified by a name or an object, it is indicated by drawing a dotted line perpendicular to the relation and connecting this line to a class box or a description. Figure 5 shows that the relationship between a UserAdmin class and a Role class is qualified by a name. Such an association is usually implemented with a Dictionary object.

Figure 5 *Associations symbol*



Bundles are entities that are visible in normal application programming. For example, when a bundle is stopped, all its services will be unregistered. Therefore, the classes/interfaces that are grouped in bundles are shown on a grey rectangle.

Figure 6 *Bundles*



1.4.4 Key Words

This specification consistently uses the words *may*, *should*, and *must*. Their meaning is well defined in [1] *Bradner, S., Key words for use in RFCs to Indicate Requirement Levels*. A summary follows.

- *must* – An absolute requirement. Both the Framework implementation and bundles have obligations that are required to be fulfilled to conform to this specification.
- *should* – Recommended. It is strongly recommended to follow the description, but reasons may exist to deviate from this recommendation.
- *may* – Optional. Implementations must still be interoperable when these items are not implemented.

1.5 The Specification Process

Within the OSGi, specifications are developed by Expert Groups (EG). If a member company wants to participate in an EG, it must sign a Statement Of Work (SOW). The purpose of an SOW is to clarify the legal status of the material discussed in the EG. An EG will discuss material which already has

Intellectual Property (IP) rights associated with it, and may also generate new IP rights. The SOW, in conjunction with the member agreement, clearly defines the rights and obligations related to IP rights of the participants and other OSGi members.

To initiate work on a specification, a member company first submits a request for a proposal. This request is reviewed by the Market Requirement Committee which can either submit it to the Technical Steering Committee (TSC) or reject it. The TSC subsequently assigns the request to an EG to be implemented.

The EG will draft a number of proposals that meet the requirements from the request. Proposals usually contain Java code defining the API and semantics of the services under consideration. When the EG is satisfied with a proposal, it votes on it.

To assure that specifications can be implemented, reference implementations are created to implement the proposal. Test suites are also developed, usually by a different member company, to verify that the reference implementation (and future implementations by OSGi member companies) fulfill the requirements of the specifications. Reference implementations and test suites are *only* available to member companies.

Specifications combine a number of proposals to form a single document. The proposals are edited to form a set of consistent specifications, which are voted upon again by the EG. The specification is then submitted to all the member companies for review. During this review period, member companies must disclose any IP claims they have on the specification. After this period, the OSGi board of directors publishes the specification.

This Service Platform Release 3 specification was developed by the Core Platform Expert Group (CPEG), Device Expert Group (DEG), Remote Management Expert Group (RMEG), and Vehicle Expert Group (VEG).

1.6 Version Information

This document specifies OSGi Service Platform Release 3. This specification is backward compatible to releases 1 and 2.

New for this specification are:

- Wire Admin service
- Measurement utility
- Start Level service
- Execution Environments
- URL Stream and Content Handling
- Dynamic Import
- Position utility
- IO service
- XML service
- Jini service
- UPnP service
- OSGi Name-space
- Initial Provisioning service

Components in this specification have their own specification-version, independent of the OSGi Service Platform, Release 3 specification. The following table summarizes the packages and specification-versions for the different subjects.

Item	Package	Version
Framework	org.osgi.framework	1.2
Configuration Admin service	org.osgi.service.cm	1.1
Device Access	org.osgi.service.device	1.1
Http Service	org.osgi.service.http	1.1
IO Connector	org.osgi.service.io	1.0
Jini service	org.osgi.service.jini	1.0
Log Service	org.osgi.service.log	1.2
Metatype	org.osgi.service.metatype	1.0
Package Admin service	org.osgi.service.packageadmin	1.1
Permission Admin service	org.osgi.service.permissionadmin	1.1
Preferences Service	org.osgi.service.prefs	1.0
Initial Provisioning	org.osgi.service.provisioning	1.0
Bundle Start Levels	org.osgi.service.startlevel	1.0
Universal Plug & Play service	org.osgi.service.upnp	1.0
URL Stream and Content	org.osgi.service.url	1.0
User Admin service	org.osgi.service.useradmin	1.0
Wire Admin	org.osgi.service.wireadmin	1.0
Measurement utility	org.osgi.util.measurement	1.0
Position utility	org.osgi.util.position	1.0
Service Tracker	org.osgi.util.tracker	1.2
XML Parsers	org.osgi.util.xml	1.0

Table 1 Packages and versions

When a component is represented in a bundle, a specification-version is needed in the declaration of the Import-Package or Export-Package manifest headers. Package versioning is described in *Sharing Packages* on page 18.

1.7 Compliance Program

The OSGi offers a compliance program for the software product that includes an OSGi Framework and a set of zero or more core bundles collectively referred to as a Service Platform. Any services which exist in the org.osgi name-space and that are offered as part of a Service Platform must pass the conformance test suite in order for the product to be considered for inclusion in the compliance program. A Service Platform may be tested in isolation and is independent of its host Virtual Machine. Certification means that a product has passed the conformance test suite(s) and meets certain criteria necessary for admission to the program, including the requirement for the supplier to warrant and represent that the product conforms to the applicable OSGi specifications, as defined in the compliance requirements.

The compliance program is a voluntary program and participation is the supplier's option. The onus is on the supplier to ensure ongoing compliance with the certification program and any changes which may cause this compliance to be brought into question should result in re-testing and re-submission of the Service Platform. Only members of the OSGi alliance are permitted to submit certification requests.

1.7.1 Compliance Claims.

In addition, any product that contains a certified OSGi Service Platform may be said to contain an *OSGi Compliant Service Platform*. The product itself is not compliant and should not be claimed as such.

More information about the OSGi Compliance program, including the process for inclusion and the list of currently certified products, can be found at http://www.osgi.org/compliance.

1.8 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 Gateway Specification 1.0* http://www.osgi.org/resources/spec_download.asp

[3] *OSGi Service Platform, Release 2, October 2001* http://www.osgi.org/resources/spec_download.asp

2 Log Service Specification

Version 1.2

2.1 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.

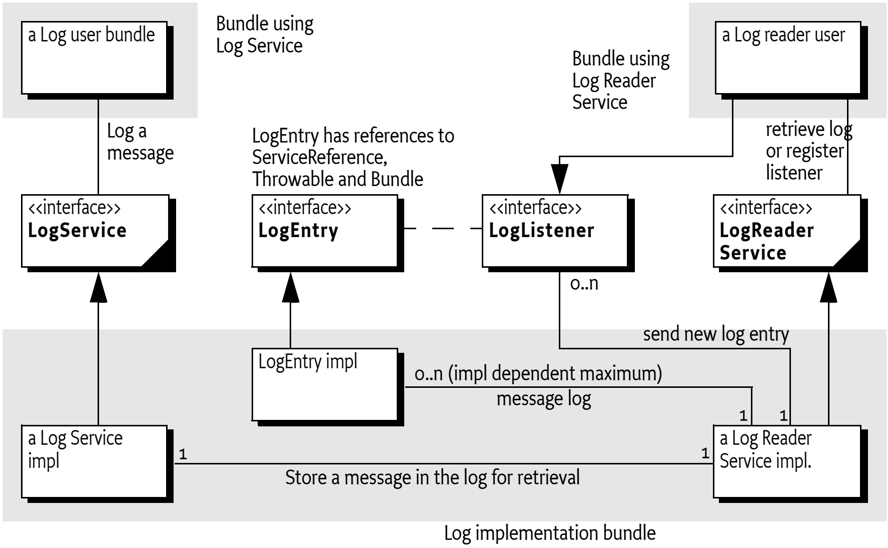
This specification defines the methods and semantics of interfaces which bundle developers can use to log entries and to retrieve log entries.

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.

2.1.1 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 7 Log Service Class Diagram org.osgi.service.log package



2.2 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.

2.3 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 following table lists the 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.

Table 2 *Log Levels*

Level	Descriptions
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.

Table 2 *Log Levels*

2.4 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.

2.5 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.

2.6 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.

2.6.1 Bundle Events Mapping

A Bundle Event is mapped to a LogEntry object according to Table 3, “Mapping of Bundle Events to Log Entries,” on page 15.

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

Table 3 Mapping of Bundle Events to Log Entries

Log Entry method	Information about Bundle Event
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"

Table 3 Mapping of Bundle Events to Log Entries

2.6.2 Service Events Mapping

A Service Event is mapped to a LogEntry object according to Table 4, "Mapping of Service Events to Log Entries," on page 16.

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"

Table 4 Mapping of Service Events to Log Entries

2.6.3 Framework Events Mapping

A Framework Event is mapped to a LogEntry object according to Table 5, "Mapping of Framework Event to Log Entries," on page 17.

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"

Table 5 *Mapping of Framework Event to Log Entries*

2.7 Security

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

2.8 Changes

The following clarifications were made.

- The interpretation of the log level has been clarified to allow arbitrary integers.
- New Framework Event type strings are defined.
- LogEntry.getException is allowed to return a different exception object than the original exception object in order to allow garbage collection of the original object.
- The addLogListener method in the Log Reader Service no longer adds the same listener object twice.
- Delivery of Log Event objects to Log Listener objects must happen asynchronously. This delivery mode was undefined in previous releases.

2.9 org.osgi.service.log

The OSGi Log Service Package. Specification Version 1.2.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.log; specification-version=1.2
```

2.9.1 Summary

- LogEntry - Provides methods to access the information contained in an individual Log Service log entry. [p.11]
- LogListener - Subscribes to LogEntry objects from the LogReaderService. [p.11]
- LogReaderService - Provides methods to retrieve LogEntry objects from the log. [p.11]
- LogService - Provides methods for bundles to write messages to the log. [p.11]

2.9.2 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[p.20], LogListener[p.11]

2.9.2.1 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.

2.9.2.2 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.

2.9.2.3 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[p.21], LogService.LOG_WARNING[p.21], LogService.LOG_INFO[p.21], LogService.LOG_DEBUG[p.20]

2.9.2.4 public String getMessage( )

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

Returns String containing the message associated with this LogEntry object.

2.9.2.5 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.

2.9.2.6 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()

2.9.3 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[p.11], LogEntry[p.11], LogReaderService.addLogListener(LogListener)[p.20], LogReaderService.removeLogListener(LogListener)[p.20]

2.9.3.1 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[p.11]

2.9.4 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[p.11], LogListener[p.11], LogListener.logged(LogEntry)[p.19]

2.9.4.1 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[p.11], LogEntry[p.11], LogListener.logged(LogEntry)[p.19]

2.9.4.2 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.

2.9.4.3 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[p.11]

2.9.5 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[p.21]
2. LOG_WARNING[p.21]
3. LOG_INFO[p.21]
4. LOG_DEBUG[p.20]

2.9.5.1 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.

2.9.5.2 public static final int LOG_ERROR = 1

An error message (Value 1).

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

2.9.5.3 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.

2.9.5.4 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.

2.9.5.5 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`[p.21] , `LOG_WARNING`[p.21] , `LOG_INFO`[p.21] , `LOG_DEBUG`[p.20]

2.9.5.6 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`[p.21] , `LOG_WARNING`[p.21] , `LOG_INFO`[p.21] , `LOG_DEBUG`[p.20]

2.9.5.7 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[p.21] , LOG_WARNING[p.21] , LOG_INFO[p.21] , LOG_DEBUG[p.20]

2.9.5.8 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[p.21] , LOG_WARNING[p.21] , LOG_INFO[p.21] , LOG_DEBUG[p.20]

6 IO Connector Service Specification

Version 1.0

6.1 Introduction

Communication is at the heart of OSGi Service Platform functionality. Therefore, a flexible and extendable communication API is needed: one that can handle all the complications that arise out of the Reference Architecture. These obstacles could include different communication protocols based on different networks, firewalls, intermittent connectivity, and others.

Therefore, this IO Connector Service specification adopts the [12] *Java 2 Micro Edition* (J2ME) javax.microedition.io packages as a basic communications infrastructure. In J2ME, this API is also called the Connector framework. A key aspect of this framework is that the connection is configured by a single string, the URI.

In J2ME, the Connector framework can be extended by the vendor of the Virtual Machine, but cannot be extended at run-time by other code. Therefore, this specification defines a service that adopts the flexible model of the Connector framework, but allows bundles to extend the Connector Services into different communication domains.

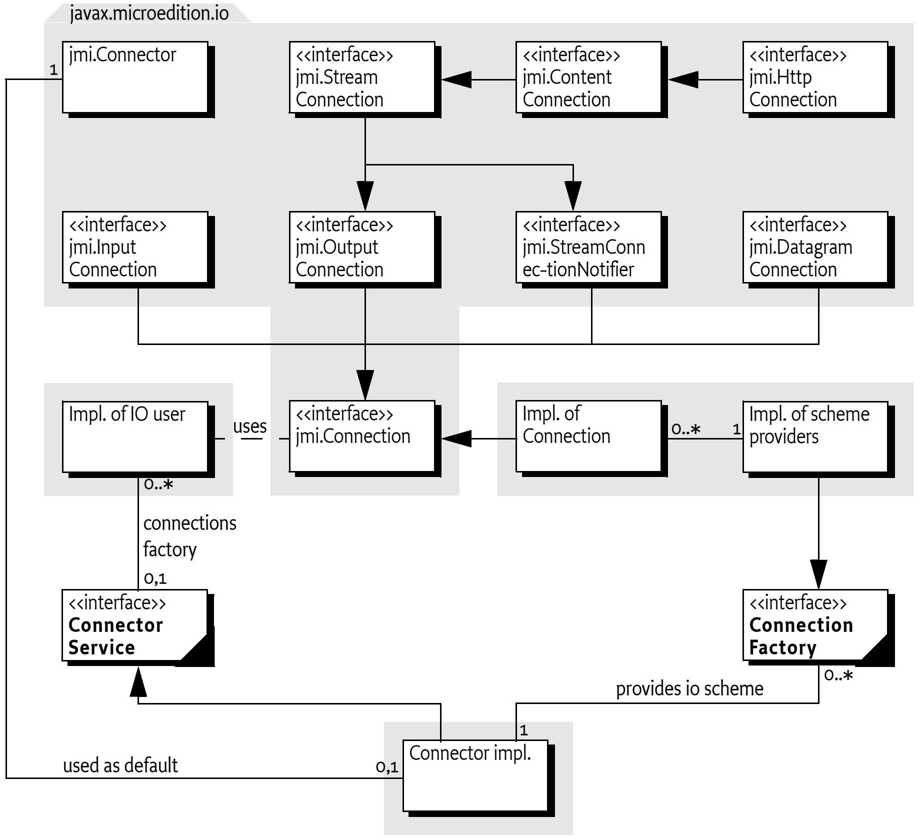
6.1.1 Essentials

- *Abstract* – Provide an intermediate layer that abstracts the actual protocol and devices from the bundle using it.
- *Extendable* – Allow third-party bundles to extend the system with new protocols and devices.
- *Layered* – Allow a protocol to be layered on top of lower layer protocols or devices.
- *Configurable* – Allow the selection of an actual protocol/device by means of configuration data.
- *Compatibility* – Be compatible with existing standards.

6.1.2 Entities

- *ConnectorService* – The service that performs the same function—creating connections from different providers—as the static methods in the Connector framework of javax.microediton.io.
- *ConnectionFactory* – A service that extends the Connector service with more schemes.
- *Scheme* – A protocol or device that is supported in the Connector framework.

Figure 18 Class Diagram, org.osgi.service.io (jmi is javax.microedition.io)


javax.microedition.io
jmi.Connector
1
<<interface>> jmi.Stream Connection
<<interface>> jmi.Content Connection
<<interface>> jmi.Http Connection
<<interface>> jmi.Input Connection
<<interface>> jmi.Output Connection
<<interface>> jmi.StreamConn ec-tionNotifier
<<interface>> jmi.Datagram Connection
Impl. of IO user
uses
<<interface>> jmi.Connection
Impl. of Connection
0..*
1
Impl. of scheme providers
0..*
connections factory
0,1
<<interface>> Connector Service
<<interface>> Connection Factory
0..*
provides io scheme
1
used as default
0,1
Connector impl.

6.2 The Connector Framework

The [12] *Java 2 Micro Edition* specification introduces a package for communicating with back-end systems. The requirements for this package are very similar to the following OSGi requirements:

- Small footprint
- Allows many different implementations simultaneously
- Simple to use
- Simple configuration

The key design goal of the Connector framework is to allow an application to use a communication mechanism/protocol without understanding implementation details.

An application passes a Uniform Resource Identifier (URI) to the `java.microedition.io.Connector` class, and receives an object implementing one or more `Connection` interfaces. The `java.microedition.io.Connector` class uses the scheme in the URI to locate the appropriate Connection Factory service. The remainder of the URI may contain parameters that are used by the Connection Factory service to establish the connection; for example, they may contain the baud rate for a serial connection. Some examples:

- sms://+46705950899;expiry=24h;reply=yes;type=9
- datagram://:53
- socket://www.acme.com:5302
- comm://COM1;baudrate=9600;databits=9
- file:c:/autoexec.bat

The javax.microedition.io API itself does not prescribe any schemes. It is up to the implementor of this package to include a number of extensions that provide the schemes. The javax.microedition.io.Connector class dispatches a request to a class which provides an implementation of a Connection interface. J2ME does not specify how this dispatching takes place, but implementations usually offer a proprietary mechanism to connect user defined classes that can provide new schemes.

The Connector framework defines a taxonomy of communication mechanisms with a number of interfaces. For example, a javax.microedition.io.InputConnection interface indicates that the connection supports the input stream semantics, such as an I/O port. A javax.microedition.io.DatagramConnection interface indicates that communication should take place with messages.

When a javax.microedition.io.Connector.open method is called, it returns a javax.microedition.io.Connection object. The interfaces implemented by this object define the type of the communication session. The following interfaces may be implemented:

- *HttpConnection* – A javax.microedition.io.ContentConnection with specific HTTP support.
- *DatagramConnection* – A connection that can be used to send and receive datagrams.
- *OutputConnection* – A connection that can be used for streaming output.
- *InputConnection* – A connection that can be used for streaming input.
- *StreamConnection* – A connection that is both input and output.
- *StreamConnectionNotifier* – Can be used to wait for incoming stream connection requests.
- *ContentConnection* – A javax.microedition.io.StreamConnection that provides information about the type, encoding, and length of the information.

Bundles using this approach must indicate to the Operator what kind of interfaces they expect to receive. The operator must then configure the bundle with a URI that contains the scheme and appropriate options that match the bundle's expectations. Well-written bundles are flexible enough to communicate with any of the types of javax.microedition.io.Connection interfaces they have specified. For example, a bundle should support javax.microedition.io.StreamConnection as well as javax.microedition.io.DatagramConnection objects in the appropriate direction (input or output).

The following code example shows a bundle that sends an alarm message with the help of the javax.microedition.io.Connector framework:

```
public class Alarm {
  String    uri;
  public Alarm(String uri) { this.uri = uri; }
  private void send(byte[] msg) {
        while ( true ) try {
           Connection  connection = Connector.open( uri );
           DataOutputStream    dout = null;
           if ( connection instanceof OutputConnection ) {
              dout = ((OutputConnection)
                 connection).openDataOutputStream();
             dout.write( msg );
           }
           else if (connection instanceof DatagramConnection) {
             DatagramConnection dgc =
                 (DatagramConnection) connection;
             Datagram datagram = dgc.newDatagram(
                 msg, msg.length );
             dgc.send( datagram );
           } else {
               error( "No configuration for alarm" );
               return;
           }
           connection.close();
         } catch( Exception e ) { ... }
      }
   }
```

6.3 Connector Service

The javax.microedition.io.Connector framework matches the requirements for OSGi applications very well. The actual creation of connections, however, is handled through static methods in the javax.microedition.io.Connector class. This approach does not mesh well with the OSGi service registry and dynamic life-cycle management.

This specification therefore introduces the Connector Service. The methods of the ConnectorService interface have the same signatures as the static methods of the javax.microedition.io.Connector class.

Each javax.microedition.io.Connection object returned by a Connector Service must implement interfaces from the javax.microedition.io package. Implementations must strictly follow the semantics that are associated with these interfaces.

The Connector Service must provide all the schemes provided by the exporter of the javax.microedition.io package. The Connection Factory services must have priority over schemes implemented in the Java run-time environment. For example, if a Connection Factory provides the http scheme and a built-in implementation exists, then the Connector Service must use the Connection Factory service with the http scheme.

Bundles that want to use the Connector Service should first obtain a ConnectorService service object. This object contains open methods that should be called to get a new javax.microedition.io.Connection object.

6.4 Providing New Schemes

The Connector Service must be able to be extended with the Connection Factory service. Bundles that can provide new schemes must register a ConnectionFactory service object.

The Connector Service must listen for registrations of new ConnectionFactory service objects and make the supplied schemes available to bundles that create connections.

Implementing a Connection Factory service requires implementing the following method:

- createConnection(String,int,boolean) – Creates a new connection object from the given URI.

The Connection Factory service must be registered with the IO_SCHEME property to indicate the provided scheme to the Connector Service. The value of this property must be a String[] object.

If multiple Connection Factory services register with the same scheme, the Connector Service should select the Connection Factory service with the highest value for the service.ranking service registration property, or if more than one Connection Factory service has the highest value, the Connection Factory service with the lowest service.id is selected.

The following example shows how a Connection Factory service may be implemented. The example will return a javax.microedition.io.InputConnection object that returns the value of the URI after removing the scheme identifier.

```
public class ConnectionFactoryImpl
   implements BundleActivator, ConnectionFactory {
      public void start( BundleContext context ) {
         Hashtable  properties = new Hashtable();
         properties.put( IO_SCHEME,
            new String[] { "data" } );
         context.registerService(
            ConnectorService.class.getName(),
            this, properties );
      }
      public void stop( BundleContext context ) {}

      public Connection createConnection(
         String uri, int mode, boolean timeouts  ) {
         return new DataConnection(uri);
      }
}

class DataConnection
   implements javax.microedition.io.InputConnection {
   String    uri;
   DataConnection( String uri ) {this.uri = uri;}
   public DataInputStream openDataInputStream()
      throws IOException {
        return new DataInputStream( openInputStream() );
      }

      public InputStream openInputStream() throws IOException {
        byte [] buf = uri.getBytes();
        return new ByteArrayInputStream(buf,5,buf.length-5);
      }
      public void close() {}
    }
```

6.4.1 Orphaned Connection Objects

When a Connection Factory service is unregistered, it must close all Connection objects that are still open. Closing these Connection objects should make these objects unusable, and they should subsequently throw an IOException when used.

Bundles should not unnecessarily hang onto objects they retrieved from services. Implementations of Connection Factory services should program defensively and ensure that resource allocation is minimized when a Connection object is closed.

6.5 Execution Environment

The javax.microedition.io package is available in J2ME configurations/profiles, but is not present in J2SE, J2EE, and the OSGi minimum execution requirements.

Implementations of the Connector Service that are targeted for all environments should carry their own implementation of the javax.microedition.io package and export it.

6.6 Security

The OSGi Connector Service is a key service available in the Service Platform. A malicious bundle which provides this service can spoof any communication. Therefore, it is paramount that the ServicePermission[REGISTER,ConnectorService] is given only to a trusted bundle. ServicePermission[GET,ConnectorService] may be handed to bundles that are allowed to communicate to the external world.

ServicePermission[REGISTER,ConnectionFactory] should also be restricted to trusted bundles because they can implement specific protocols or access devices. ServicePermission[GET,ConnectionFactory] should be limited to trusted bundles that implement the Connector Service.

Implementations of Connection Factory services must perform all I/O operations within a privileged region. For example, an implementation of the sms: scheme must have permission to access the mobile phone, and should not require the bundle that opened the connection to have this permission. Normally, the operations need to be implemented in a doPrivileged method or in a separate thread.

If a specific Connection Factory service needs more detailed permissions than provided by the OSGi or Java 2, it may create a new specific Permission sub-class for its purpose.

6.7 org.osgi.service.io

The OSGi IO Connector Specification Version 1.0.

Bundles wishing to use this package must list the package in the `Import-Package` header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.io; specification-ver-
sion=1.0, javax.microedition.io
```

6.7.1 Summary

- ConnectionFactory - A Connection Factory service is called by the implementation of the Connector Service to create `javax.microedition.io.Connection` objects which implement the scheme named by `IO_SCHEME`. [p.89]
- ConnectorService - The Connector Service should be called to create and open `javax.microedition.io.Connection` objects. [p.91]

6.7.2 public interface ConnectionFactory

A Connection Factory service is called by the implementation of the Connector Service to create `javax.microedition.io.Connection` objects which implement the scheme named by `IO_SCHEME`. When a `ConnectorService.open` method is called, the implementation of the Connector Service will examine the specified name for a scheme. The Connector Service will then look for a Connection Factory service which is registered with the service property `IO_SCHEME` which matches the scheme. The `createConnection`[p.91] method of the selected Connection Factory will then be called to create the actual `Connection` object.

6.7.2.1 public static final String IO_SCHEME = "io.scheme"

Service property containing the scheme(s) for which this Connection Factory can create `Connection` objects. This property is of type `String[]`.

6.7.2.2 public Connection createConnection(String name, int mode, boolean timeouts) throws IOException

name The full URI passed to the `ConnectorService.open` method

mode The mode parameter passed to the `ConnectorService.open` method

timeouts The timeouts parameter passed to the `ConnectorService.open` method

□ Create a new `Connection` object for the specified URI.

Returns A new `javax.microedition.io.Connection` object.

Throws `IOException` – If a `javax.microedition.io.Connection` object can not not be created.

6.7.3 public interface ConnectorService

The Connector Service should be called to create and open `javax.microedition.io.Connection` objects. When an `open*` method is called, the implementation of the Connector Service will examine the specified name for a scheme. The Connector Service will then look for a Connection Factory service which is registered with the service property `IO_SCHEME` which matches the scheme. The `createConnection` method of the selected Connection Factory will then be called to create the actual `Connection` object.

If more than one Connection Factory service is registered for a particular scheme, the service with the highest ranking (as specified in its `service.ranking` property) is called. 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 called. This is the same algorithm used by `BundleContext.getServiceReference`.

6.7.3.1 public static final int READ = 1

Read access mode.

See Also `javax.microedition.io.Connector.READ`

6.7.3.2 public static final int READ_WRITE = 3

Read/Write access mode.

See Also `javax.microedition.io.Connector.READ_WRITE`

6.7.3.3 public static final int WRITE = 2

Write access mode.

See Also `javax.microedition.io.Connector.WRITE`

6.7.3.4 public Connection open(String name) throws IOException

name The URI for the connection.

□ Create and open a `Connection` object for the specified name.

Returns A new `javax.microedition.io.Connection` object.

Throws `IllegalArgumentException` – If a parameter is invalid.

`javax.microedition.io.ConnectionNotFoundException` – If the connection cannot be found.

`IOException` – If some other kind of I/O error occurs.

See Also `javax.microedition.io.Connector.open(String name)`

6.7.3.5 public Connection open(String name, int mode) throws IOException

name The URI for the connection.

mode The access mode.

□ Create and open a `Connection` object for the specified name and access mode.

Returns A new `javax.microedition.io.Connection` object.

Throws `IllegalArgumentException` – If a parameter is invalid.

`javax.microedition.io.ConnectionNotFoundException` – If the connection cannot be found.

`IOException` – If some other kind of I/O error occurs.

See Also `javax.microedition.io.Connector.open(String name, int mode)`

6.7.3.6 public Connection open(String name, int mode, boolean timeouts) throws IOException

name The URI for the connection.

mode The access mode.

timeouts A flag to indicate that the caller wants timeout exceptions.

□ Create and open a `Connection` object for the specified name, access mode and timeouts.

Returns A new `javax.microedition.io.Connection` object.

Throws `IllegalArgumentException` – If a parameter is invalid.

`javax.microedition.io.ConnectionNotFoundException` – If the connection cannot be found.

`IOException` – If some other kind of I/O error occurs.

See Also `javax.microedition.io.Connector.open(String name, int mode, boolean timeouts)`

6.7.3.7 public DataInputStream openDataInputStream(String name) throws IOException

name The URI for the connection.

□ Create and open a `DataInputStream` object for the specified name.

Returns A `DataInputStream` object.

Throws `IllegalArgumentException` – If a parameter is invalid.

`javax.microedition.io.ConnectionNotFoundException` – If the connection cannot be found.

`IOException` – If some other kind of I/O error occurs.

See Also `javax.microedition.io.Connector.openDataInputStream(String name)`

6.7.3.8 public DataOutputStream openDataOutputStream(String name) throws IOException

name The URI for the connection.

□ Create and open a `DataOutputStream` object for the specified name.

Returns A `DataOutputStream` object.

Throws `IllegalArgumentException` – If a parameter is invalid.

`javax.microedition.io.ConnectionNotFoundException` – If the connection cannot be found.

`IOException` – If some other kind of I/O error occurs.

See Also `javax.microedition.io.Connector.openDataOutputStream(String name)`

6.7.3.9 public InputStream openInputStream(String name) throws IOException

name The URI for the connection.

□ Create and open an `InputStream` object for the specified name.

Returns An `InputStream` object.

Throws `IllegalArgumentException` – If a parameter is invalid.

`javax.microedition.io.ConnectionNotFoundException` – If the connection cannot be found.

`IOException` – If some other kind of I/O error occurs.

See Also `javax.microedition.io.Connector.openInputStream(String name)`

6.7.3.10 public OutputStream openOutputStream(String name) throws IOException

name The URI for the connection.

□ Create and open an `OutputStream` object for the specified name.

Returns An `OutputStream` object.

Throws `IllegalArgumentException` – If a parameter is invalid.

`javax.microedition.io.ConnectionNotFoundException` – If the connection cannot be found.

`IOException` – If some other kind of I/O error occurs.

See Also `javax.microedition.io.Connector.openOutputStream(String name)`

6.8 References

[12] *Java 2 Micro Edition*
http://java.sun.com/j2me/

[13] *javax.microedition.io whitepaper*
http://wireless.java.sun.com/midp/chapters/j2mewhite/chap13.pdf

[14] *J2ME Foundation Profile*
http://www.jcp.org/jsr/detail/46.jsp

3 Configuration Admin Service Specification

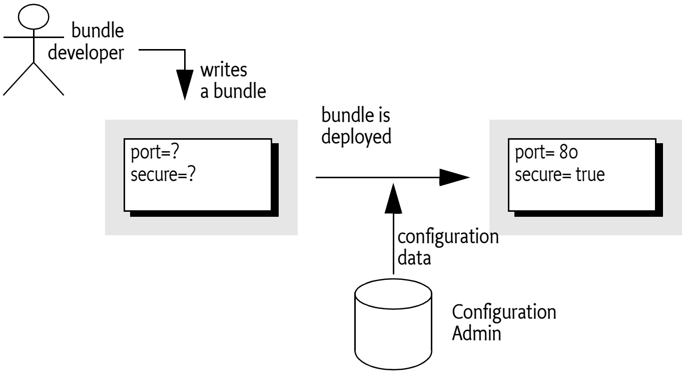
Version 1.1

3.1 Introduction

The Configuration Admin service is an important aspect of the deployment of an OSGi Service Platform. It allows an Operator to set the configuration information of deployed bundles.

Configuration is the process of defining the configuration data of bundles and assuring that those bundles receive that data when they are active in the OSGi Service Platform.

Figure 8 Configuration Admin Service Overview



3.1.1 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.

3.1.2 Operation

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 remote. 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 or when an existing configuration has never been updated.
- *Managed Service Factory* – Services registered with this interface 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.

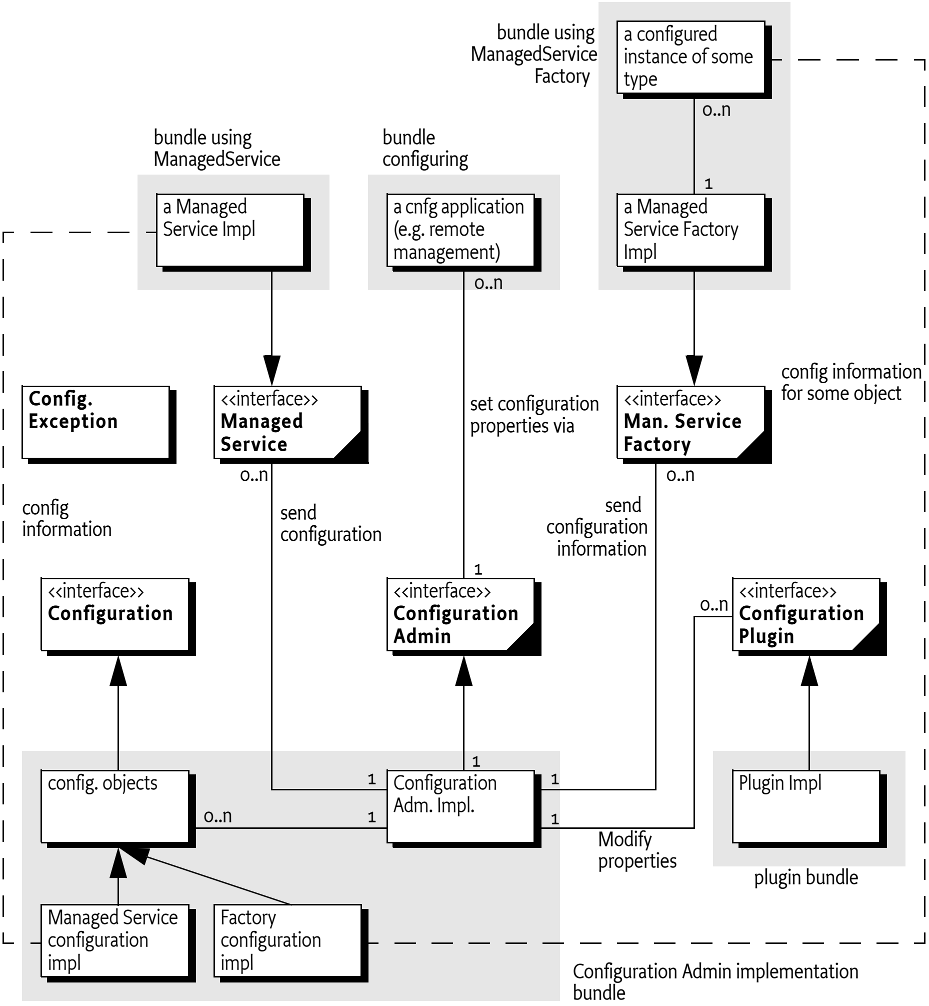
The database can be manipulated either by the Management Agent or bundles that configure themselves.

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.

3.1.3 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 (bundle or 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 or 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 a unique service.pid service registration property 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 a service.pid and receives zero or more configuration dictionaries. Each dictionary has its own 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 9 *Configuration Admin Class Diagram org.osgi.service.cm*


bundle using ManagedService Factory
a configured instance of some type
0..n
1
bundle using ManagedService
bundle configuring
a Managed Service Impl
a cnfg application (e.g. remote management)
0..n
a Managed Service Factory Impl
config information for some object
Config. Exception
<<interface>> Managed Service
set configuration properties via
<<interface>> Man. Service Factory
0..n
0..n
config information
send configuration
send configuration information
1
<<interface>> Configuration
<<interface>> Configuration Admin
0..n
<<interface>> Configuration Plugin
1
config. objects
1
Configuration Adm. Impl.
1
0..n
1
1
Plugin Impl
Modify properties
plugin bundle
Managed Service configuration impl
Factory configuration impl
Configuration Admin implementation bundle

3.2 Configuration Targets

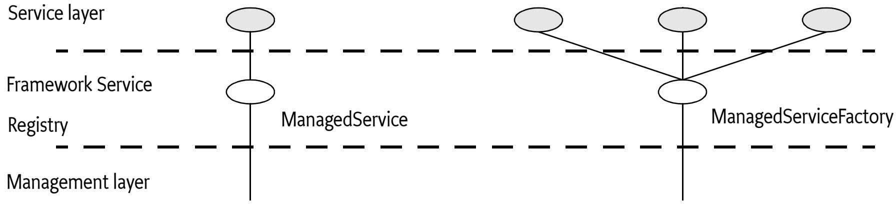
One of the more complicated aspects of this specification is the subtle distinction between the ManagedService and ManagedServiceFactory classes.

Both receive configuration information from the Configuration Admin service and are treated similarly in most respects. Therefore, this specification refers to *configuration targets* 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 entity.

A Managed Service Factory is used when part of the configuration is to define *how many instances are required*. 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 10 Differentiation of ManagedService and ManagedServiceFactory Classes



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.

3.3 The Persistent Identity

A crucial concept in the Configuration Admin service specification is the Persistent IDentity (PID). 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.

A PID is a unique identifier for a service that persists over multiple invocations of the Framework.

When a bundle registers a service with a PID, it should set property service.pid to a unique value. For that service, the same PID should always be used. If the bundle is stopped and later started, the same PID should be used.

PIDs can be useful for all services, but the Configuration Admin service requires their use with Managed Service and Managed Service Factory registrations because it associates its configuration data with PIDs.

PIDs must be unique for each service. A bundle must not register multiple configuration target services with the same PID. If that should occur, the Configuration Admin service must:

- Send the appropriate configuration data to all services registered under that PID from that bundle only.
- Report an error in the log.
- Ignore duplicate PIDs from other bundles and report them to the log.

3.3.1 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.

The schemes for PIDs that are defined in this specification should be followed.

Any globally unique string can be used as a PID. The following sections, however, define schemes for common cases. These schemes are not required, but bundle developers are urged to use them to achieve consistency.

3.3.1.1 Local Bundle PIDs

As a convention, descriptions starting with the bundle identity and a dot (.) are reserved for a bundle. As an example, a PID of "65.536" would belong to the bundle with a bundle identity of 65.

3.3.1.2 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 an example, the PID named com.acme.watchdog would represent a Watchdog service from the ACME company.

3.3.1.3 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..

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

Table 6 Schemes for Device-Oriented PID Names

3.4 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 27 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 again 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.

3.4.1 Location Binding

When a Configuration object is created by either getConfiguration or createFactoryConfiguration, it becomes bound to the location of the calling bundle. This location is obtained with the associated bundle's getLocation 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 SecurityException is thrown if a bundle other than a Management Agent bundle attempts to modify the configuration information of another bundle.

If a Managed Service is registered with a PID that is already bound to another location, the normal callback to ManagedService.updated must not take place.

The two argument versions of getConfiguration and createFactoryConfiguration take a location String as their second argument. These methods require AdminPermission, and they create Configuration objects bound to the specified location, instead of the location of the calling bundle. These methods are intended for management bundles.

The creation of a Configuration object does not in itself initiate a callback to the target.

A null location parameter may be used to create Configuration objects that are not bound. In this case, the objects become bound to a specific location the first time that they are used by a bundle. When this dynamically bound bundle is subsequently uninstalled, the Configuration object's bundle location must be set to null again so it can be bound again later.

A management bundle may create a Configuration object before the associated Managed Service is registered. It may use a null location to avoid any dependency on the actual location of the bundle which registers this service. When the Managed Service is registered later, the Configuration object must be bound to the location of the registering bundle, and its configuration dictionary must then be passed to ManagedService.updated.

3.4.2 Configuration Properties

A configuration dictionary contains a set of properties in a Dictionary object. The value of the property may be of the following types:

```
type        ::=
   String     | Integer    | Long
 | Float      | Double     | Byte
 | Short      | Character  | Boolean
 | vector     | arrays

primitive   ::= long | int | short | char | byte | double
 | float | boolean

arrays      ::= primitive '[]' | type '[]' | null

vector      ::= Vector of type or null
```

This explicitly allows vectors and arrays of mixed types and containing null.

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.

Bundles should not use nested vectors or arrays, nor should they use mixed types. Using mixed types or nesting makes it impossible to use the meta typing specification. See *Metatype Specification* on page 65.

3.4.3 Property Propagation

An implementation of a Managed Service should copy all the properties of the Dictionary object argument in updated(Dictionary), known or unknown, into its service registration properties using ServiceRegistration.setProperties.

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

Bundles that cooperate with the propagation of 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.

3.4.4 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 42, 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 bundle that can use this 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. These system properties are all of type String.

3.4.5 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.

3.5 Managed Service

A Managed Service is used by a bundle that needs one configuration dictionary and is thus associated with one Configuration object in the Configuration Admin service.

A bundle can register any number of ManagedService objects, but each must be identified with its own PID.

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.

3.5.1 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.

3.5.2 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.

3.5.3 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 46.

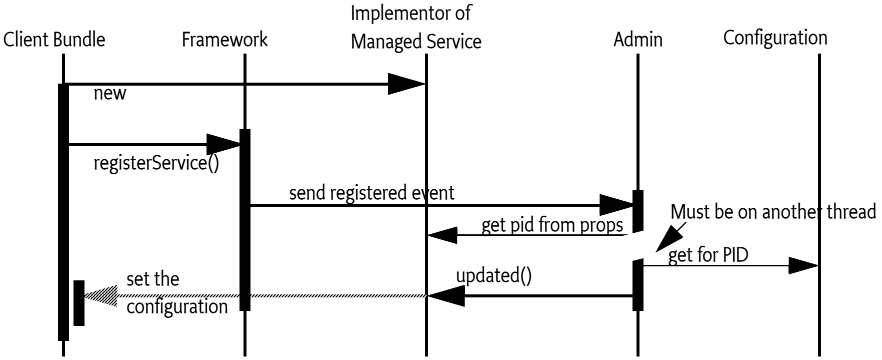
When this registration is detected by the Configuration Admin service, the following steps must occur:

- The configuration stored for the registered PID must be retrieved. If there is a Configuration object for this PID, it is sent to the Managed Service with updated(Dictionary).
- If a Managed Service is registered and no configuration information is available, 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. For this reason, a Managed Service must always get a callback when it registers *and* the Configuration Admin service is started.

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 11 *Managed Service Configuration Action Diagram*



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

3.5.4 Race Conditions

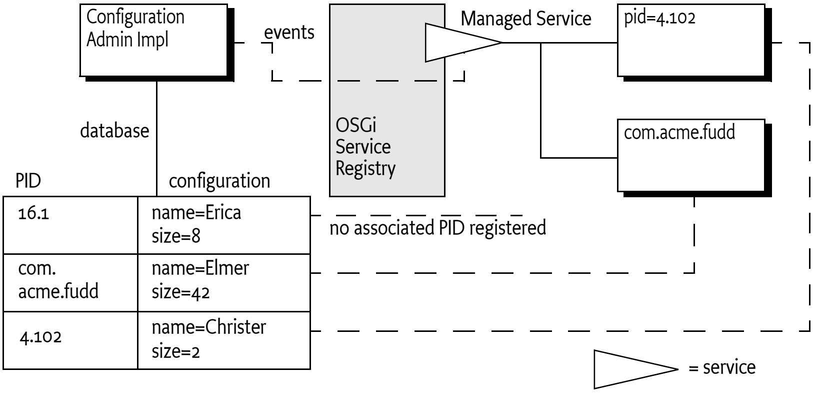
When a Managed Service is registered, the default properties may be visible in the service registry for a short period before they are replaced by the properties of the actual configuration dictionary. Care should be taken that this visibility does not cause race conditions for other bundles.

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.

3.5.5 Examples of Managed Service

Figure 12 shows a Managed Service configuration example. Two services are registered under the ManagedService interface, each with a different PID.

Figure 12 PIDs and External Associations



PID	configuration

16.1	name=Erica
	size=8

com.
acme.fudd
	name=Elmer
	size=42

4.102	name=Christer
	size=2

The Configuration Admin service has a database containing a configuration record for each PID. When the Managed Service with service.pid = com.acme.fudd 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.

3.5.5.1 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 synchronized void start(
    BundleContext context ) throws Exception {
    properties = new Hashtable();
    properties.put( Constants.SERVICE_PID,
      "com.acme.console" );
    properties.put( "port",  new Integer(2011) );

    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
}
```

3.5.6 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.

3.6 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 Configuration object. It passes the identifier of the 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.

3.6.1 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.

3.6.1.1 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).

3.6.1.2 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.

3.6.1.3 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.

3.6.2 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), 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.

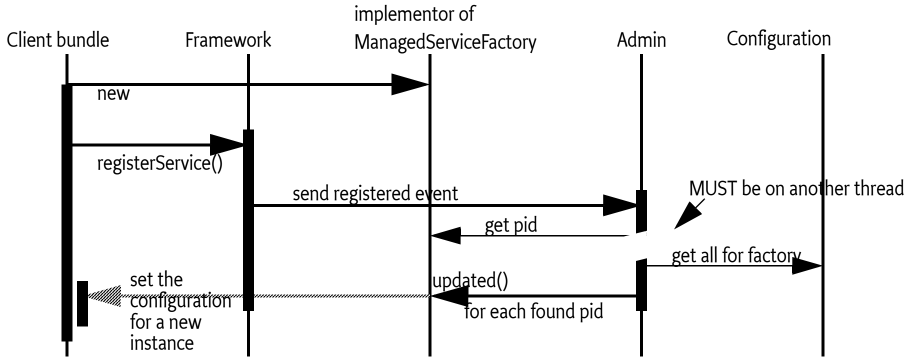
When the Configuration Admin service detects the registration of a Managed Service Factory, it must find all configuration dictionaries for this factory and must then sequentially call ManagedServiceFactory.updated(String,Dictionary) for each configuration dictionary. The first argument is the PID of 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 instances of the associated factory class. 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 dictionary may be used only internally.

The Configuration Admin service must guarantee that the Configuration objects are not deleted before their properties are given to the Managed Service Factory, and must assure that no race conditions exist between initialization and updates.

Figure 13 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.

3.6.3 Deletion

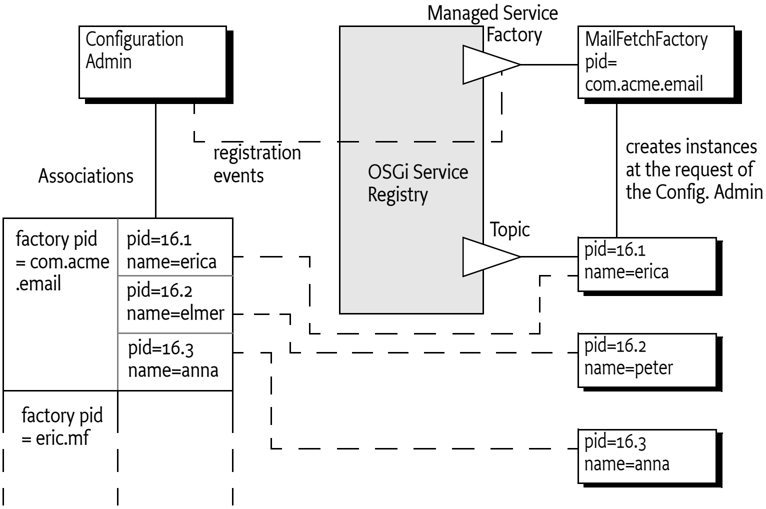
If a configuring bundle deletes an instance of a Managed Service Factory, the deleted(String) method is called. The argument is the PID for this instance. The implementation of the Managed Service Factory must remove all information and stop any behavior associated with that PID. If a service was registered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

3.6.4 Managed Service Factory Example

Figure 14 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 14 *Managed Service Factory Example*



3.6.5 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();
            }
        }
    }
```

3.7 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.

3.7.1 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 creating the same Configuration object. 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 AdminPermission. 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.

3.7.2 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. This management bundle needs AdminPermission. The first argument is the location identifier and the second is the PID 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.

3.7.3 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 filter function must use the properties of the Configuration objects and only return the ones that match the filter expression.

A single Configuration object is identified with a PID and can be obtained with getConfiguration(String).

If the caller has AdminPermission, then all Configuration objects are eligible for search. In other cases, only Configuration objects bound to the calling bundle's location must be returned.

null is returned in both cases when an appropriate Configuration object cannot be found.

3.7.3.1 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 a configuration target's updated method: either the ManagedService.updated or ManagedServiceFactory.updated method. If this target service is not registered, the fresh configuration information must be set when the configuration target service registers.

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

3.7.4 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 updated method is 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. It should then remove the associated *instance*. The ManagedServiceFactory.deleted 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.

3.7.5 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 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.

3.8 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.

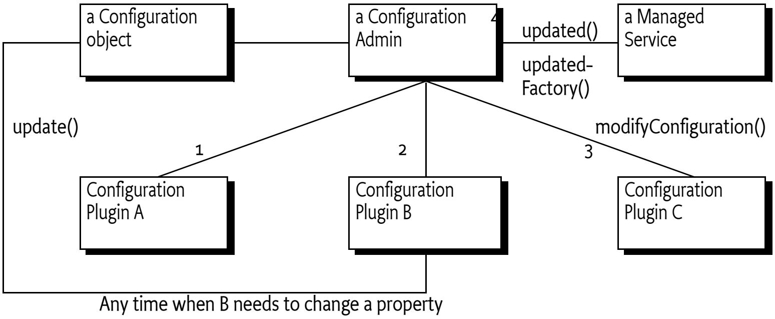
The ConfigurationPlugin interface has only one method: modifyConfiguration(ServiceReference,Dictionary). This method inspects or modifies configuration data.

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 31.

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 properties it receives from the Configuration Admin service to the service registry.

Figure 15 *Order of Configuration Plugin Services*



3.8.1 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. Omitting the cm.target registration property means that it is called for *all* configuration updates.

3.8.2 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 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.

3.8.3 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.

3.8.4 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.

3.8.5 Calling Order

The order in which the ConfigurationPlugin objects are called must depend on the service.cmRanking configuration property of the ConfigurationPlugin object. Table 7 shows the usage of the service.cmRanking property for the order of calling the Configuration Plugin services..

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.

Table 7 service.cmRanking *Usage For Ordering*

3.9 Remote Management

This specification does not attempt to define a remote management interface for the Framework. The purpose of this specification is to define a minimal interface for bundles that is complete enough for testing.

The Configuration Admin service is a primary aspect of remote management, however, and this specification must be compatible with common remote management standards. This section discusses some of the issues of using this specification with [4] *DMTF Common Information Model* (CIM) and [5] *Simple Network Management Protocol* (SNMP), the most likely candidates for remote management today.

These discussions are not complete, comprehensive, or normative. They are intended to point the bundle developer in relevant directions. Further specifications are needed to make a more concrete mapping.

3.9.1 Common Information Model

Common Information Model (CIM) defines the managed objects in [7] *Interface Definition Language* (IDL) language, which was developed for the Common Object Request Broker Architecture (CORBA).

The data types and the data values have a syntax. Additionally, these syntaxes can be mapped to XML. Unfortunately, this XML mapping is very different from the very applicable [6] *XSchema* XML data type definition language. The Framework service registry property types are a proper subset of the CIM data types.

In this specification, a Managed Service Factory maps to a CIM class definition. The primitives create, delete, and set are supported in this specification via the ManagedServiceFactory interface. The possible data types in CIM are richer than those the Framework supports and should thus be limited to cases when CIM classes for bundles are defined.

An important conceptual difference between this specification and CIM is the naming of properties. CIM properties are defined within the scope of a class. In this specification, properties are primarily defined within the scope of the Managed Service Factory, but are then placed in the registry, where they have global scope. This mechanism is similar to [8] *Lightweight Directory Access Protocol*, in which the semantics of the properties are defined globally and a class is a collection of globally defined properties.

This specification does not address the non-Configuration Admin service primitives such as notifications and method calls.

3.9.2 Simple Network Management Protocol

The Simple Network Management Protocol (SNMP) defines the data model in ASN.1. SNMP is a rich data typing language that supports many types that are difficult to map to the data types supported in this specification. A large overlap exists, however, and it should be possible to design a data type that is applicable in this context.

The PID of a Managed Service should map to the SNMP Object IDentifier (OID). Managed Service Factories are mapped to tables in SNMP, although this mapping creates an obvious restriction in data types because tables can only contain scalar values. Therefore, the property values of the Configuration object would have to be limited to scalar values.

Similar scope issues as seen in CIM arise for SNMP because properties have a global scope in the service registry.

SNMP does not support the concept of method calls or function calls. All information is conveyed as the setting of values. The SNMP paradigm maps closely to this specification.

This specification does not address non-Configuration Admin primitives such as traps.

3.10 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 must 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.

3.11 Security

3.11.1 Permissions

Configuration Admin service security is implemented using ServicePermission and AdminPermission. The following table summarizes the permissions needed by the Configuration Admin bundle itself, as well as those needed by the bundles with which it interacts.

Bundle Registering	ServicePermisson Action	AdminPermission
ConfigurationAdmin	REGISTER ConfigurationAdmin	Yes
	GET ManagedService	
	GET ManagedServiceFactory	
	GET ConfigurationPlugin	
ManagedService	REGISTER ManagedService	No
	GET ConfigurationAdmin	
ManagedServiceFactory	REGISTER ManagedServiceFactory	No
	GET ConfigurationAdmin	
ConfigurationPlugin	REGISTER ConfigurationPlugin	No
	GET ConfigurationAdmin	

Table 8 Permission Overview Configuration Admin

The Configuration Admin service must have ServicePermission[REGISTER, ConfigurationAdmin]. It will also be the only bundle that needs the ServicePermission[GET,ManagedService | ManagedServiceFactory |ConfigurationPlugin]. No other bundle should be allowed to have GET permission for these interfaces. The Configuration Admin bundle must also hold AdminPermission.

Bundles that can be configured must have the ServicePermission[REGISTER,ManagedService|ManagedServiceFactory].

Bundles registering ConfigurationPlugin objects must have the ServicePermission[REGISTER, ConfigurationPlugin]. The Configuration Admin service must trust all services registered with the ConfigurationPlugin interface. Only the Configuration Admin service should have ServicePermission[GET, ConfigurationPlugin.

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 AdminPermission (such bundles need AdminPermission to perform this check).

Bundles that want to change their own configuration need ServicePermission[GET, ConfigurationAdmin]. A bundle with AdminPermission is allowed to access and modify any Configuration object.

Pre-configuration of bundles requires AdminPermission because the methods that specify a location require this permission.

3.11.2 Forging PIDs

A risk exists of an unauthorized bundle forging a PID in order to obtain and possibly modify the configuration information of another bundle. To mitigate this risk, Configuration objects are generally *bound* to a specific bundle location, and are not passed to any Managed Service or Managed Service Factory registered by a different bundle.

Bundles with the required AdminPermission can create Configuration objects that are not bound. In other words, they have their location set to null. This can be useful for preconfiguring bundles before they are installed without having to know their actual locations.

In this scenario, the Configuration object must become bound to the first bundle that registers a Managed Service (or Managed Service Factory) with the right PID.

A bundle could still possibly obtain another bundle's configuration by registering a Managed Service with the right PID before the victim bundle does so. This situation can be regarded as a denial-of-service attack, because the victim bundle would never receive its configuration information. Such an attack can be avoided by always binding Configuration objects to the right locations. It can also be detected by the Configuration Admin service when the victim bundle registers the correct PID and two equal PIDs are then registered. This violation of this specification should be logged.

3.11.3 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.

3.12 Configurable Service

Both the Configuration Admin service and the `org.osgi.framework.Configurable` interface address configuration management issues. It is the intention of this specification to replace the Framework interface for configuration management.

The Framework Configurable mechanism works as follows. A registered service object implements the `Configurable` interface to allow a management bundle to configure that service. The `Configurable` interface has only one method: `getConfigurationObject()`. This method returns a Java Bean. Beans can be examined and modified with the `java.reflect` or `java.bean` packages.

This scheme has the following disadvantages:

- *No factory* – Only registered services can be configured, unlike the Managed Service Factory that configures any number of services.
- *Atomicity* – The beans or reflection API can only modify one property at a time and there is no way to tell the bean that no more modifications to the properties will follow. This limitation complicates updates of configurations that have dependencies between properties.
  This specification passes a `Dictionary` object that sets all the configuration properties atomically.
- *Profile* – The Java beans API is linked to many packages that are not likely to be present in OSGi environments. The reflection API may be present but is not simple to use.
  This specification has no required libraries.
- *User Interface support* – UI support in beans is very rudimentary when no AWT is present.
  The associated Metatyping specification does not require any external libraries, and has extensive support for UIs including localization.

3.13 Changes

3.13.1 Clarifications

- It was not clear from the description that a PID received through a Managed Service Factory must not be used to register a Managed Service. This has been highlighted in the appropriate sections.
- It was not clearly specified that a call-back to a target only happens when the data is updated or the target is registered. The creation of a Configuration object does not initiate a call-back. This has been highlighted in the appropriate sections.
- In this release, when a bundle is uninstalled, all Configuration objects that are dynamically bound to that bundle must be unbound again. See *Location Binding* on page 29.
- It was not clearly specified that the data types of a Configuration object allow arrays and vectors that contain elements of mixed types and also null.

3.13.2 Removal of Bundle Location Property

The bundle location property that was required to be set in the Configuration object's properties has been removed because it leaked security sensitive information to all bundles using the Configuration object.

3.13.3 Plug-in Usage

It was not completely clear when a plug-in must be called and how the properties dictionary should behave. This has been clearly specified in *Configuration Plugin* on page 42.

3.13.4 BigInteger/BigDecimal

The classes BigInteger and BigDecimal are not part of the minimal execution requirements and are therefore no longer part of the supported Object types in the Configuration dictionary.

3.13.5 Equals

The behavior of the equals and hashCode methods is now defined. See *Equality* on page 31.

3.13.6 Constant for service.factoryPid

Added a new constant in the ConfigurationAdmin class. See *SERVICE_FACTORYPID* on page 55. This caused this specification to step from version 1.0 to version 1.1.

3.14 org.osgi.service.cm

The OSGi Configuration Admin service Package. Specification Version 1.2

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.cm; specification-version=1.2
```

3.14.1 Summary

- Configuration - The configuration information for a `ManagedService` or `ManagedServiceFactory` object. [p.51]
- ConfigurationAdmin - Service for administering configuration data. [p.54]
- ConfigurationException - An `Exception` class to inform the Configuration Admin service of problems with configuration data. [p.57]
- ConfigurationListener - Listener for Configuration changes. [p.57]
- ConfigurationPlugin - A service interface for processing configuration dictionary before the update. [p.58]
- ManagedService - A service that can receive configuration data from a Configuration Admin service. [p.60]
- ManagedServiceFactory - Manage multiple service instances. [p.62]

3.14.2 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 is preserved from the last set key/value.

A configuration can be *bound* to a bundle location (`Bundle.getLocation()`). The purpose of binding a `Configuration` object to a location is to make it impossible for another bundle to forge a PID that would match this configuration. When a configuration is bound to a specific location, and a bundle with a different location registers a corresponding `ManagedService` object or `ManagedServiceFactory` object, then the configuration is not passed to the updated method of that object.

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.

3.14.2.1 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 intiates a call to any ConfigurationListeners asynchronously.

Throws IOException – If delete fails

IllegalStateException – if this configuration has been deleted

3.14.2.2 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.

3.14.2.3 public String getBundleLocation( )

□ Get the bundle location. Returns the bundle location to which this configuration is bound, or null if it is not yet bound to a bundle location.

This call requires AdminPermission.

Returns location to which this configuration is bound, or null.

Throws SecurityException – if the caller does not have AdminPermission.

IllegalStateException – if this Configuration object has been deleted.

3.14.2.4 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

3.14.2.5 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

3.14.2.6 public Dictionary 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

3.14.2.7 public int hashCode( )

□ Hash code is based on PID. The hashcode for two Configuration objects must be the same when the Configuration PID's are the same.

Returns hash code for this Configuration object

3.14.2.8 public void setBundleLocation(String bundleLocation)

bundleLocation a bundle location or null

□ Bind this Configuration object to the specified bundle location. If the bundleLocation parameter is null then the Configuration object will not be bound to a location. 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 will be set persistently.

This method requires AdminPermission.

Throws SecurityException – if the caller does not have AdminPermission

IllegalStateException – if this configuration has been deleted

3.14.2.9 public void update(Dictionary 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 intiates a call to any ConfigurationListeners asynchronously.

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

3.14.2.10 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. Also intiates a call to any ConfigurationListeners asynchronously.

This is the only way for a bundle that uses a Configuration Plugin service to initate 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[p.58]

3.14.3 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 PID matches the PID registration property (service.pid) of the Managed Service. If found, it calls ManagedService.updated[p.61] method with the new properties. The implementation of a Configuration Admin service must run these call-backs asynchronously to allow proper synchronization.

When the Configuration Admin service detects a Managed Service Factory registration, it checks its storage for configuration objects whose factoryPid matches the PID 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 another bundle requires AdminPermission.

Configuration objects can be *bound* to a specified bundle location. In this case, if a matching Managed Service or Managed Service Factory is registered by a bundle with a different location, then the Configuration Admin service must not do the normal callback, and it should log an error. In the case where a Configuration object is not bound, its location field is null, the Configuration Admin service will bind it to the location of the bundle that registers the first Managed Service or Managed Service Factory that has a corresponding PID property. When a `Configuration` object is bound to a bundle location in this manner, the Confguration Admin service must detect if the bundle corresponding to the location is uninstalled. If this occurs, the `Configuration` object is unbound, that is its location field is set back to `null`.

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.

3.14.3.1 public static final String SERVICE_BUNDLELOCATION = "service.bundleLocation"

Service 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

3.14.3.2 public static final String SERVICE_FACTORYPID = "service.factoryPid"

Service property naming the Factory PID in the configuration dictionary. The property's value is of type `String`.

Since 1.1

3.14.3.3 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)`[p.53] 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.

Returns a new `Configuration` object.

Throws `IOException` – if access to persistent storage fails.

`SecurityException` – if caller does not have `AdminPermission` and `factoryPid` is bound to another bundle.

3.14.3.4 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)`[p.53] 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.

This method requires `AdminPermission`.

Returns a new `Configuration` object.

Throws `IOException` – if access to persistent storage fails.

`SecurityException` – if caller does not have `AdminPermission`.

3.14.3.5 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.

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.

This method requires `AdminPermission`.

Returns an existing or new `Configuration` object.

Throws `IOException` – if access to persistent storage fails.

`SecurityException` – if the caller does not have `AdminPermission`.

3.14.3.6 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.

Else, if the location of the existing `Configuration` object is `null`, set it to the calling bundle's location.

If the location of the `Configuration` object does not match the calling bundle, throw a `SecurityException`.

Returns an existing or new `Configuration` matching the PID.

Throws `IOException` – if access to persistent storage fails.

`SecurityException` – if the `Configuration` object is bound to a location different from that of the calling bundle and it has no `AdminPermission`.

3.14.3.7 public Configuration[] listConfigurations(String filter) throws

IOException, InvalidSyntaxException

filter a `Filter` object, 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.

Normally only `Configuration` objects that are bound to the location of the calling bundle are returned. If the caller has `AdminPermission`, then all matching `Configuration` objects are returned.

The syntax of the filter string is as defined in the `Filter` class. The filter can test any configuration parameters including the following system properties:

- `service.pid`-`String`- the PID under which this is registered
- `service.factoryPid`-`String`- the factory if applicable
- `service.bundleLocation`-`String`- 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

3.14.4 public class ConfigurationException extends Exception

An `Exception` class to inform the Configuration Admin service of problems with configuration data.

3.14.4.1 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.

3.14.4.2 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

3.14.4.3 public String getReason( )

□ Return the reason for this exception.

Returns reason of the failure

3.14.5 public interface ConfigurationListener

Listener for Configuration changes.

`ConfigurationListener` objects are registered with the Framework service registry and are notified when a `Configuration` object is updated or deleted.

`ConfigurationListener` objects are passed the type of configuration change.

One of the change methods will be called with `CM_UPDATED` when `Configuration.update` is called or with `CM_DELETED` when `Configuration.delete` is called. Notification will be asynchronous to the update or delete method call. The design is very lightweight in that is does not pass `Configuration` objects, the listener is merely advised that the configuration information for a given pid has changed. If the listener wants to locate the `Configuration` object for the specified pid, it must use `ConfigurationAdmin`.

Security Considerations. Bundles wishing to monitor `Configuration` changes will require `ServicePermission[ConfigurationListener, REGISTER]` to register a `ConfigurationListener` service. Since `Configuration` objects are not passed to the listener, no sensitive configuration information is available to the listener.

3.14.5.1 public static final int CM_DELETED = 2

Change type that indicates that `Configuration.delete` was called.

3.14.5.2 public static final int CM_UPDATED = 1

Change type that indicates that `Configuration.update` was called.

3.14.5.3 public void configurationChanged(String pid, int type)

pid The pid of the configuration which changed.

type The type of the configuration change.

□ Receives notification a configuration has changed.

This method is only called if the target of the configuration is a ManagedService.

3.14.5.4 public void factoryConfigurationChanged(String factoryPid, String pid, int type)

factoryPid The factory pid for the changed configuration.

pid The pid of the configuration which changed.

type The type of the configuration change.

□ Receives notification a factory configuration has changed.

This method is only called if the target of the configuration is a ManagedServiceFactory.

3.14.6 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 ManagedS ervice or Managed Service Factory.

Configuration Plugin (plugin) services have full read/write access to all configuration information. Therefore, bundles using this facility should be trusted. Access to this facility should be limited with `ServicePermission[REGISTER, ConfigurationPlugin]`. Implementations of a Configuration Plugin service should assure that they only act on appropriate configurations.

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 targetted 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.

3.14.6.1 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

3.14.6.2 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.

3.14.6.3 public void modifyConfiguration(ServiceReference reference, Dictionary 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.

3.14.7 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 identitifier 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,
      null // Properties will come from CM in updated
    );
  }

  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.

3.14.7.1 public void updated(Dictionary 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 which 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.

Throws `ConfigurationException` – when the update fails

3.14.8 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 );
      }
      ...
    }
```

3.14.8.1 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.

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.

3.14.8.2 public String getName( )

□ Return a descriptive name of this factory.

Returns the name for the factory, which might be localized

3.14.8.3 public void updated(String pid, Dictionary 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`[p.57] 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.

Throws `ConfigurationException` – when the configuration properties are invalid.

3.15 References

[4] *DMTF Common Information Model*
http://www.dmtf.org

[5] *Simple Network Management Protocol*
RFCs http://directory.google.com/Top/Computers/Internet/Protocols/SNMP/RFCs

[6] *XSchema*
http://www.w3.org/TR/xmlschema-0/

[7] *Interface Definition Language*
http://www.omg.org

[8] *Lightweight Directory Access Protocol*
http://directory.google.com/Top/Computers/Software/Internet/Servers/Directory/LDAP

[9] *Understanding and Deploying LDAP Directory services*
Timothy Howes et. al. ISBN 1-57870-070-1, MacMillan Technical publishing.

4 Metatype Specification

Version 1.0

4.1 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 23. A Configuration Admin service implementation deploys attributes (key/value pairs) as configuration properties.

During the development of the Configuration Admin service, it became clear that the Framework attribute types needed to be described in a computer readable form. This information (the metadata) could then be used to automatically create user interfaces for management systems or could be translated into management information specifications such as CIM, SNMP, and the like.

4.1.1 Essentials

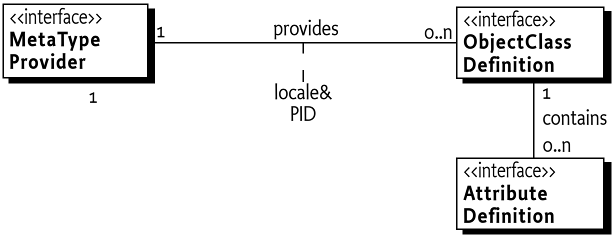
- *Conceptual model* – The specification must have a conceptual model for how classes and attributes are organized.

- *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 attribution 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.

4.1.2 Entities

- *Attribute* – A key/value pair.
- *AttributeDefinition* – Defines a description, name, help text, and type information of an attribute.
- *ObjectClassDefinition* – Defines the type of a datum. It contains a description and name of the type plus a set of AttributeDefinition objects.
- *MetaTypeProvider* – 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.

Figure 16 Class Diagram Meta Typing, org.osgi.service.metatyping



4.1.3 Operation

This specification starts with an object that implements the MetaTypeProvider interface. It is not specified how this object is obtained, and there are several possibilities. Often, however, this object is a service registered with the Framework.

A MetaTypeProvider object provides access to ObjectClassDefinition objects. 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 classes is qualified by a locale and a Persistent IDentity (PID). The locale is a String object that defines for which language the ObjectClassDefinition is intended, allowing for localized user interfaces. The PID is used when a single MetaTypeProvider object can provide ObjectClassDefinition objects for multiple purposes. The context in which the MetaTypeProvider object is used should make this clear.

Attributes have global scope. Two object classes can consist of the same attributes, and attributes with the same name should have the same definition. This global scope is unlike languages like Java that scope instance variables within a class, but it is similar to the Lightweight Directory Access Protocol (LDAP) (SNMP also uses a global attribute name-space).

Attribute Definition objects provide sufficient localized information to generate user interfaces.

4.2 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 [11] *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.

4.3 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.

4.4 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 and 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. The return type of this is a String[ ]. For cardinality = zero, this return type must be an array of one String object. For other cardinalities, the array must not contain more than the absolute value of *cardinality* String objects. In that case, it may contain 0 objects.

4.5 Meta Type Provider

The MetaTypeProvider interface is used to access metatype information. It is used in management systems and run-time management. It supports locales so that the text used in AttributeDefinition and ObjectClassDefinition objects can be adapted to different locales.

The PID is given as an argument with the getObjectClassDefinition method so that a single MetaTypeProvider object can be used for different object classes with their own PIDs.

Locale objects are represented in String objects because not all profiles support Locale. The String holds the standard Locale presentation of:

```
<language> [ "_" <country> [ "_" <variation>]]
```

For example, "en", "nl_BE", "en_CA_posix".

4.6 Metatype Example

AttributeDefinition and ObjectClassDefinition classes are intended to be easy to use for bundles. This example shows a naive implementation for these classes (note that the get methods usages are not shown). Commercial implementations can use XML, Java serialization, or Java Properties for implementations. This example uses plain code to store the definitions.

The example first shows that the ObjectClassDefinition interface is implemented in the OCD class. The name is made very short because the class is used to instantiate the static structures. Normally many of these objects are instantiated very close to each other, and long names would make these lists of instantiations very long.

```
class OCD implements ObjectClassDefinition {
  String                    name;
  String                    id;
  String                    description;
  AttributeDefinition       required[];
  AttributeDefinition       optional[];

  public OCD(
    String name, String id, String description,
    AttributeDefinition required[],
    AttributeDefinition optional[]) {

    this.name = name;
    this.id = id;
    this.description = description;
    this.required = required;
    this.optional = optional;
  }
  .... All the get methods
}
```

The second class is the AD class that implements the AttributeDefinition interface. The name is short for the same reason as in OCD. Note the two different constructors to simplify the common case.

```
class AD implements AttributeDefinition {
  String                    name;
  String                    id;
  String                    description;
  int                       cardinality;
  int                       syntax;
```

```
    String[]                    values;
    String[]                    labels;
    String[]                    deflt;

    public AD( String name, String id, String description,
      int syntax, int cardinality, String values[],
      String labels[], String deflt[]) {
        this.name               = name;
        this.id                 = id;
        this.description        = description;
        this.cardinality        = cardinality;
        this.syntax             = syntax;
        this.values             = values;
        this.labels             = labels;
    }

    public AD( String name, String id, String description,
      int syntax)
    {
      this(name,id,description,syntax,0,null,null, null);
    }
    ... All the get methods and validate method
  }
```

The last part is the example that implements a MetaTypeProvider class. Only one locale is supported, the US locale. The OIDs used in this example are the actual OIDs as defined in X.500.

```
  public class Example implements MetaTypeProvider {
    final static AD cn = new AD(
      "cn",         "2.5.4.3", "Common name", AD.STRING);
    final static AD sn = new AD(
      "sn",         "2.5.4.4", "Sur name", AD.STRING);
    final static AD description = new AD(
      "description", "2.5.4.13","Description", AD.STRING);
    final static AD seeAlso = new AD(
      "seeAlso",    "2.5.4.34", "See Also", AD.STRING);
    final static AD telephoneNumber = new AD(
      "telephoneNumber", "2.5.4.20", "Tel nr", AD.STRING);
    final static AD userPassword = new AD(
      "userPassword", "2.5.4.3", "Password", AD.STRING);

    final static ObjectClassDefinition person = new OCD(
      "person", "2.5.6.6", "Defines a person",
        new AD[] { cn, sn },
        new AD[] { description, seeAlso,
          telephoneNumber, userPassword}
    );

    public ObjectClassDefinition getObjectClassDefinition(
      String pid, String locale) {
      return person;
    }
    public String[] getLocales() {
      return new String[] { "en_US" };
    }
  }
```

This code shows that the attributes are defined in AD objects as final static. The example groups a number of attributes together in an OCD object.

As can be seen from this example, the resource issues for using AttributeDefinition, ObjectClassDefinition and MetaTypeProvider classes are minimized.

4.7 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.

4.8 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.

4.8.1 Beans

The intention of the Beans packages in Java comes very close to the metatype information needed in the OSGi Service Platform. The java.beans.- packages cannot be used, however, for the following reasons:

- Beans packages require a large number of classes that are likely to be optional for an OSGi Service Platform.

- Beans have been closely coupled to the graphic subsystem (AWT) and applets. Neither of these packages is available on an OSGi Service Platform.
- Beans are closely coupled with the type-safe Java classes. The advantage of attributes is that no type-safety is used, allowing two parties to have an independent versioning model (no shared classes).
- Beans packages allow all possible Java objects, not the OSGi subset as required by this specification.
- Beans have no explicit localization.
- Beans have no support for optional attributes.

4.9 Security Considerations

Special security issues are not applicable for this specification.

4.10 Changes

This specification has not been changed since the previous release.

4.11 org.osgi.service.metatype

The OSGi Metatype Package. Specification Version 1.1.

Bundles wishing to use this package must list the package in the `Import-Package` header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.metatype; specification-version=1.1
```

4.11.1 Summary

- AttributeDefinition - An interface to describe an attribute. [p.72]
- MetaTypeInformation - A MetaType Information object is created by the MetaTypeService to return meta type information for a specific bundle. [p.75]
- MetaTypeProvider - Provides access to metatypes. [p.76]
- MetaTypeService - The MetaType Service can be used to obtain meta type information for a bundle. [p.76]
- ObjectClassDefinition - Description for the data type information of an objectclass. [p.67]

4.11.2 public interface AttributeDefinition

An interface to describe an attribute.

An `AttributeDefinition` object defines a description of the data type of a property/attribute.

4.11.2.1 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 Since 1.1

4.11.2.2 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 Since 1.1

4.11.2.3 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().

4.11.2.4 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.

4.11.2.5 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.

4.11.2.6 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.

4.11.2.7 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.

4.11.2.8 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.

4.11.2.9 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.

4.11.2.10 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.

4.11.2.11 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.

4.11.2.12 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
```

4.11.2.13 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.

4.11.2.14 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.

4.11.2.15 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

4.11.2.16 public String getName( )

□ Get the name of the attribute. This name may be localized.

Returns The localized name of the definition.

4.11.2.17 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

4.11.2.18 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

4.11.2.19 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`.

4.11.2.20 public String validate(String value)

value The value before turning it into the basic data type

□ 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
```

Returns `null`, “”, or another string

4.11.3 public interface MetaTypeInformation extends MetaTypeProvider

A MetaType Information object is created by the MetaTypeService to return meta type information for a specific bundle.

4.11.3.1 public Bundle getBundle( )

□ Return the bundle for which this object provides metatype information.

Returns Bundle for which this object provides metatype information.

4.11.3.2 public String[] getFactoryPids( )

□ Return the Factory PIDs (for ManagedServices) for which ObjectClassDefinition information is available.

Returns Array of Factory PIDs.

4.11.3.3 public String[] getPids( )

□ Return the PIDs (for ManagedServices) for which ObjectClassDefinition information is available.

Returns Array of PIDs.

4.11.4 public interface MetaTypeProvider

Provides access to metatypes.

4.11.4.1 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.

4.11.4.2 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

4.11.5 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 and to create the returned MetaTypeInformation object.

4.11.5.1 public MetaTypeInformation getMetaTypeInformation(Bundle bundle)

bundle The bundle for which meta type information is requested.

□ Return the MetaType information for the specified bundle.

Returns MetaTypeInformation object for the specified bundle.

4.11.6 public interface ObjectClassDefinition

Description for the data type information of an objectclass.

4.11.6.1 public static final int ALL = -1

Argument for `getAttributeDefinitions(int)`.

`ALL` indicates that all the definitions are returned. The value is -1.

4.11.6.2 public static final int OPTIONAL = 2

Argument for `getAttributeDefinitions(int)`.

`OPTIONAL` indicates that only the optional definitions are returned. The value is 2.

4.11.6.3 public static final int REQUIRED = 1

Argument for `getAttributeDefinitions(int)`.

`REQUIRED` indicates that only the required definitions are returned. The value is 1.

4.11.6.4 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

4.11.6.5 public String getDescription( )

□ Return a description of this object class. The description may be localized.

Returns The description of this object class.

4.11.6.6 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`

4.11.6.7 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.

4.11.6.8 public String getName( )

□ Return the name of this object class. The name may be localized.

Returns The name of this object class.

4.12 References

[10] *LDAP.*
Available at http://directory.google.com/Top/Computers/Software/Internet/Servers/Directory/LDAP

[11] *Understanding and Deploying LDAP Directory services*
Timothy Howes et. al. ISBN 1-57870-070-1, MacMillan Technical publishing.

5 Service Component Runtime Specification

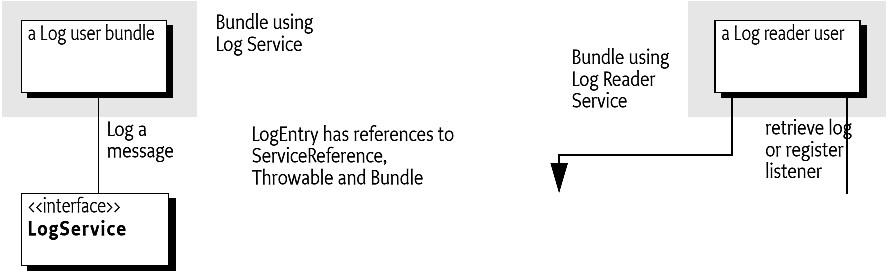
Version 1.0

5.1 Introduction

5.1.1 Entities

- *Application* – ....
- *Application Descrriptor* –

Figure 17 Log Service Class Diagram org.osgi.service.log package



5.2 The Service Component Runtime

5.3 Security

5.4 org.osgi.service.component

The OSGi Service Component Package. Specification Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.component; specification-ver-
sion=1.0
```

5.4.1 Summary

- ComponentConstants - Defines standard names for Service Component constants. [p.80]

- ComponentContext - A ComponentContext interface is used by a Service Component to interact with it execution context including locating services by reference name. [p.80]
- ComponentException - Unchecked exception which may be thrown by the Service Component Runtime. [p.82]
- ComponentFactory - When a component is declared with the `factory` attribute on it's `component` element, the Service Component Runtime will register a ComponentFactory service to allow instances of the component to be created rather than automatically create component instances as necessary. [p.82]
- ComponentInstance - A ComponentInstance encapsulates an instance of a component. [p.83]

5.4.2 public interface ComponentConstants

Defines standard names for Service Component constants.

5.4.2.1 public static final String COMPONENT_FACTORY = "component.factory"

A service registration property for a Service Component Factory. It contains the value of the `factory` attribute. The type of this property must be `String`.

5.4.2.2 public static final String COMPONENT_NAME = "component.name"

A service registration property for a Service Component. It contains the name of the Service Component. The type of this property must be `String`.

5.4.2.3 public static final String REFERENCE_TARGET_SUFFIX = ".target"

A suffix for a service registration property for a reference target. It contains the filter to select the target services for a reference. The type of this property must be `String`.

5.4.2.4 public static final String SERVICE_COMPONENT = "Service-Component"

Manifest header (named "Service-Component") identifying the XML resources within the bundle containing the bundle's Service Component descriptions.

The attribute value may be retrieved from the `Dictionary` object returned by the `Bundle.getHeaders` method.

5.4.3 public interface ComponentContext

A ComponentContext interface is used by a Service Component to interact with it execution context including locating services by reference name. In order to be notified when a component is activated and to obtain a ComponentContext, the component's implementation class must implement a

```
protected void activate(ComponentContext context);
```

method. However, the component is not required to implement this method.

In order to be called when the component is deactivated, a component's implementation class must implement a

```
protected void deactivate(ComponentContext context);
```

method. However, the component is not required to implement this method.

These methods will be called by the Service Component Runtime using reflection and may be private methods to avoid being public methods on the component's provided service object.

5.4.3.1 public void disableComponent(String name)

name of a component.

□ Disables the specified component name. The specified component name must be in the same bundle as this component.

5.4.3.2 public void enableComponent(String name)

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.

5.4.3.3 public BundleContext getBundleContext( )

□ Returns the BundleContext of the bundle which contains this component.

Returns The BundleContext of the bundle containing this component.

5.4.3.4 public ComponentInstance getComponentInstance( )

□ Returns the ComponentInstance object for this component.

Returns The ComponentInstance object for this component.

5.4.3.5 public Dictionary getProperties( )

□ Returns the component properties for this ComponentContext.

Returns properties for this ComponentContext. The properties are read only and cannot be modified.

5.4.3.6 public Bundle getUsingBundle( )

□ If the component is registered as a service using the `servicefactory="true"` attribute, then this method returns the bundle using the service provided by this component.

This method will return `null` if the component is either:

- Not a service, then no bundle can be using it as a service.
- Is a service but did not specify the `servicefactory="true"` attribute, then all bundles will use this component.

Returns The bundle using this component as a service or `null`.

5.4.3.7 public Object locateService(String name)

name The name of a service reference as specified in a `reference` element in this component's description.

□ Returns the service object for the specified service reference name.

Returns A service object for the referenced service or `null` if the reference cardinality is `0..1` or `0..n` and no matching service is available.

Throws `ComponentException` – If the Service Component Runtime catches an exception while activating the target service.

5.4.3.8 public Object[] locateServices(String name)

name The name of a service reference as specified in a `reference` element in this component's description.

□ Returns the service objects for the specified service 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 matching service is available.

Throws `ComponentException` – If the Service Component Runtime catches an exception while activating a target service.

5.4.4 public class ComponentException extends RuntimeException

Unchecked exception which may be thrown by the Service Component Runtime.

5.4.4.1 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.

5.4.4.2 public ComponentException(String message)

message The message for the exception.

□ Construct a new ComponentException with the specified message.

5.4.4.3 public ComponentException(Throwable cause)

cause The cause of the exception. May be `null`.

□ Construct a new ComponentException with the specified cause.

5.4.4.4 public Throwable getCause( )

□ Returns the cause of this exception or `null` if no cause was specified when this exception was created.

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

5.4.4.5 public Throwable initCause(Throwable cause)

□ The cause of this exception can only be set when constructed.

Throws `IllegalStateException` – This method will always throw an `IllegalStateException` since the cause of this exception can only be set when constructed.

5.4.5 public interface ComponentFactory

When a component is declared with the `factory` attribute on it's `component` element, the Service Component Runtime will register a ComponentFactory service to allow instances of the component to be created rather than automatically create component instances as necessary.

5.4.5.1 public ComponentInstance newInstance(Dictionary properties)

properties Additional properties for the component.

□ Create a new instance of the component. Additional properties may be provided for the component instance.

Returns A ComponentInstance object encapsulating the component instance. The returned component instance has been activated.

5.4.6 public interface ComponentInstance

A ComponentInstance encapsulates an instance of a component. ComponentInstances are created whenever an instance of a component is created.

5.4.6.1 public void dispose( )

□ Dispose of this component instance. The instance will be deactivated. If the instance has already been deactivated, this method does nothing.

5.4.6.2 public Object getInstance( )

□ Returns the component instance. The instance has been activated.

Returns The component instance or `null` if the instance has been deactivated.

7 Event Service Specification

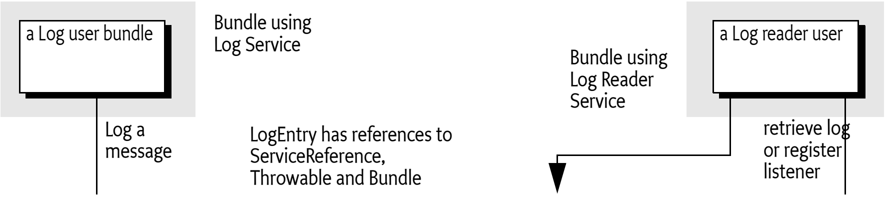
Version 1.0

7.1 Introduction

7.1.1 Entities

- *Application – ....*
- *Application Descrriptor –*

Figure 19 Log Service Class Diagram org.osgi.service.log package


a Log user bundle
Bundle using Log Service
Log a message
LogEntry has references to ServiceReference, Throwable and Bundle
Bundle using Log Reader Service
a Log reader user
retrieve log or register listener

7.2 The Event Service

7.3 Security

7.4 org.osgi.service.event

The OSGi Event Specification Version 1.0.

Bundles wishing to use this package must list the package in the `Import-Package` header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.event; specification-ver-
sion=1.0
```

7.4.1 Summary

- ChannelEvent - [p.95]
- ChannelListener - [p.96]
- EventChannel - [p.96]

7.4.2 public class ChannelEvent

7.4.2.1 public ChannelEvent(String topic, Map properties)

7.4.2.2 public boolean equals(Object obj)

7.4.2.3 public final Object getProperty(String name)

7.4.2.4 public final String[] getPropertyNames( )

7.4.2.5 public final String getTopic( )

7.4.2.6 public int hashCode( )

7.4.2.7 public final boolean matches(Filter filter)

7.4.2.8 public String toString( )

7.4.3 public interface ChannelListener extends EventListener

7.4.3.1 public void channelEvent(ChannelEvent event)

7.4.4 public interface EventChannel

7.4.4.1 public void postEvent(ChannelEvent event)

7.4.4.2 public void sendEvent(ChannelEvent event)

8 Device Access Specification

Version 1.1

8.1 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.

8.1.1 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.

8.1.2 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.

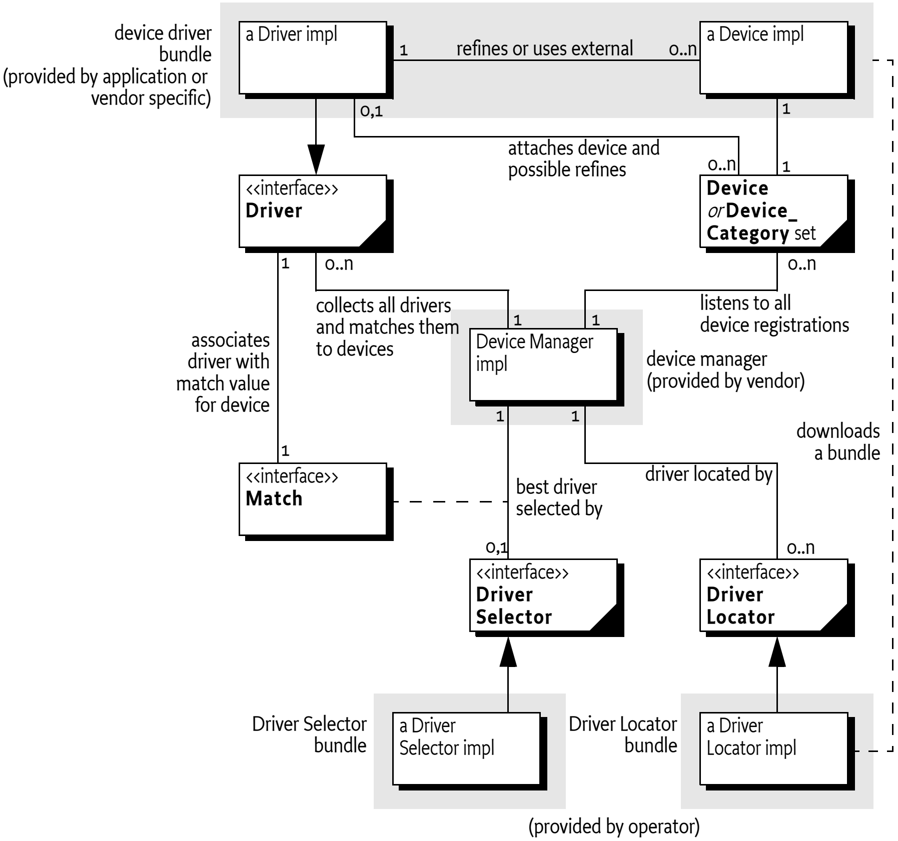
8.1.3 Entities

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 104.
- *Device* – A representation of a physical device or other entity that can be attached by a Driver service. See *Device Services* on page 99.
- *DriverLocator* – Assists in locating bundles that provide a Driver service. See *Driver Locator Service* on page 111.
- *DriverSelector* – Assists in selecting which Driver service is best suited to a Device service. See *The Driver Selector Service* on page 113.

Figure 20 show the classes and their relationships.

Figure 20 *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
listens to all device registrations
1
1
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)

8.2 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 interoperability between bundles that are based on the same underlying technology. Generic Device services will allow interoperability 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.

8.2.1 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.

8.2.2 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.

8.2.2.1 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.

8.2.2.2 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.

8.3 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 interoperability 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.

8.3.1 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 [15] *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 115.

8.3.2 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 9.

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".

Table 9 *Example Device Category Properties, M=Mandatory, O=Optional*

8.3.3 Match Example

Driver services and Device services are connected via a matching process that is explained in *The Device Attachment Algorithm* on page 115. 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.

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.

Table 10 Sample Device Category Match Scale

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.

8.4 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 102.

8.4.1 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.

8.4.2 Driver Taxonomy

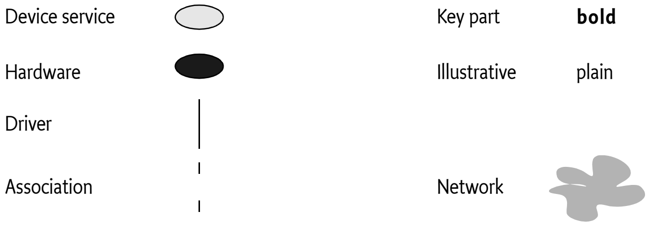
Device Drivers may belong to one of the following categories:

- 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.

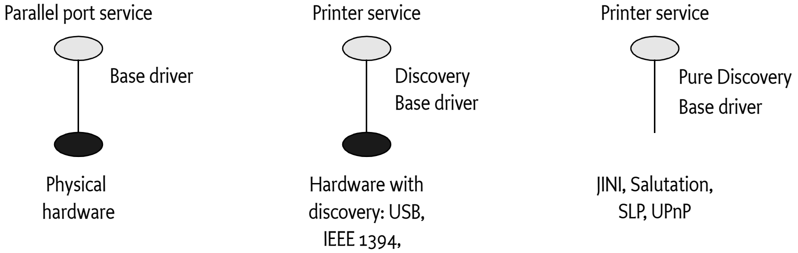
Figure 21 *Legend for Device Driver Services Taxonomy*



8.4.2.1 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 22 *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 [16] *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.

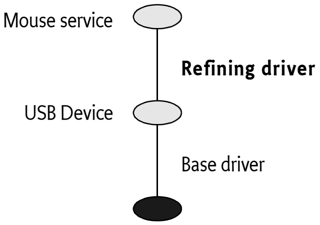
Although each driver bundle should perform discovery on its own, a driver for a non-discoverable serial port requires external help – either through a user interface or by allowing the Configuration Admin service to configure it.

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.

8.4.2.2 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 23 Refining Driver Diagram



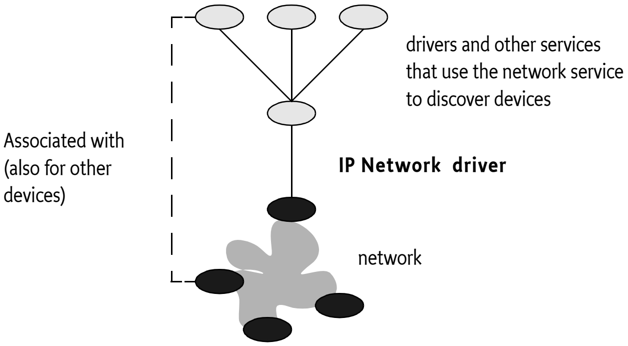
An example of a refining driver is a mouse driver, which is attached to the generic USB Device service representing a physical mouse. It then registers a new Device service which represents it as a Mouse service, defined elsewhere.

The majority of drivers fall into the refining driver type.

8.4.2.3 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.

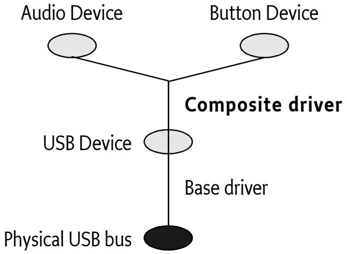
Figure 24 *Network Driver diagram*



8.4.2.4 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 25 *Composite Driver structure*



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

8.4.2.5 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 115.

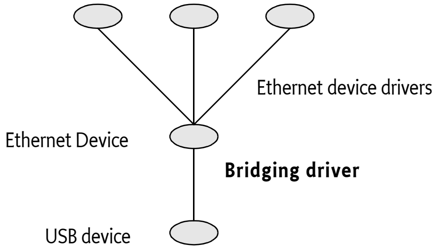
A referring driver implements the call to the `attach` method to inspect the Device service, and decides which Driver bundle would be able to attach to the device. This process can actually involve connecting to the physical device and communicating with it. The `attach` method then returns a `String`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

8.4.2.6 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 26 Bridging Driver Structure



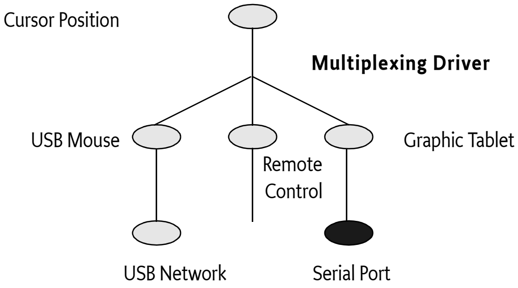
For example, USB to Ethernet bridges exist that allow connection to an Ethernet network through a USB device. In this case, the top level of the USB part of the Device service stack would be an Ethernet Device service. But the same Ethernet Device service can also be the bottom layer of an Ethernet layer 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.

8.4.2.7 Multiplexing Drivers

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

Figure 27 Multiplexing Driver Structure

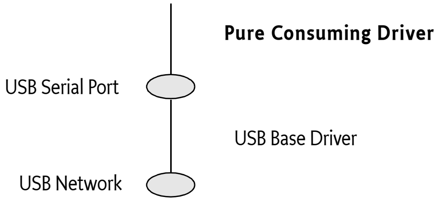


For example, assume that a system has a mouse on USB, a graphic tablet on a serial port, and a remote control facility. Each of these would be registered as a service with the Framework. A multiplexing driver can attach all three, and can merge the different positions in a central Cursor Position service.

8.4.2.8 Pure Consuming Drivers

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

Figure 28 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.

8.4.2.9 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.

8.4.3 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 118 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.

8.4.4 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 118 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.

8.4.5 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 115.
  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 107) 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.

8.4.6 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.

8.5 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 100. 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.

8.5.1 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.

8.5.2 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 102. 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 );
    }
  }
```

8.6 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 115. 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).

8.7 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.

8.7.1 Device Manager Startup

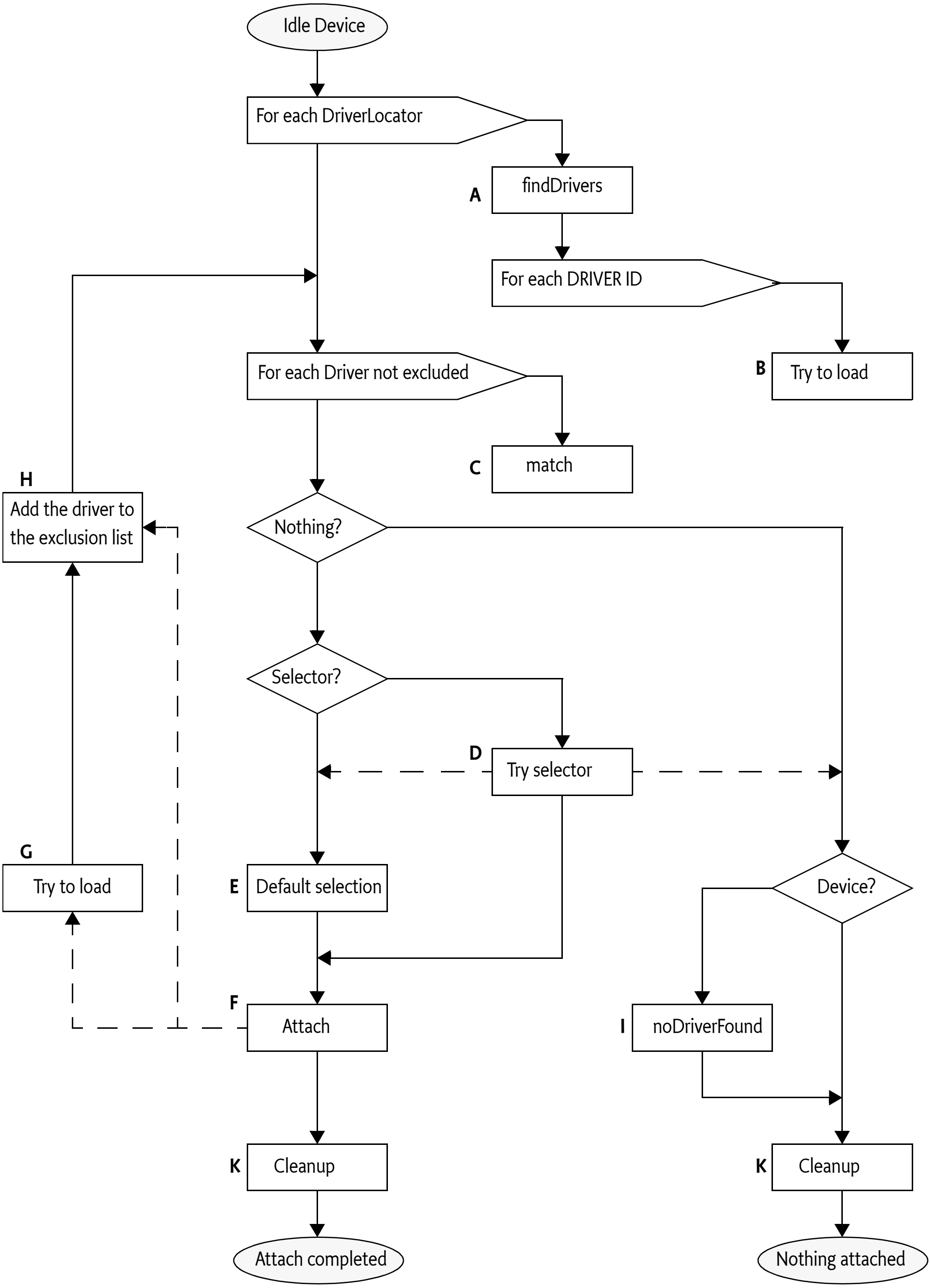
To prevent race conditions during Framework startup, the device manager must monitor the state of Device services and Driver services immediately when it is started. The device manager must not, however, begin attaching Device services until the Framework has been fully started, to prevent superfluous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

8.7.2 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 29 *Device Attachment Algorithm*



8.7.3 Legend

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 118 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 113.
E	The winner is the one with the highest match value. Tie breakers are respectively: • Highest service.ranking property. • Lowest service.id property.

Table 11 *Driver attachment algorithm*

Step	Description
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 107. 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.
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.

Table 11 Driver attachment algorithm

8.7.4 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.

8.7.5 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 preinstalled 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.

8.7.6 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.

8.7.7 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. Howerver, 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.

8.8 Security

The device manager is the only privileged bundle in the Device Access specification and requires the org.osgi.AdminPermission 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[REGISTER, DriverLocator] to register Driver Locator services, and the operator should exercise prudence in assigning this ServicePermission.

Bundles with Driver Selector services only require ServicePermission[REGISTER,DriverSelector] 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.

8.9 Changes

The Device Access specification has not increased its version number because no API change has been necessary. The only change to this specification has been a clarification of the concept of an idle device.

8.10 org.osgi.service.device

The OSGi Device Access Package. Specification Version 1.1.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.device; specification-ver-
sion=1.1
```

8.10.1 Summary

- Constants - This interface defines standard names for property keys associated with Device[p.122] and Driver[p.122] services. [p.121]
- Device - Interface for identifying device services.[p.122]
- Driver - A Driver service object must be registered by each Driver bundle wishing to attach to Device services provided by other drivers. [p.122]
- DriverLocator - A Driver Locator service can find and load device driver bundles given a property set. [p.124]
- DriverSelector - When the device manager detects a new Device service, it calls all registered Driver services to determine if anyone matches the Device service. [p.124]
- Match - Instances of Match are used in the DriverSelector.select[p.124] method to identify Driver services matching a Device service. [p.125]

8.10.2 public interface Constants

This interface defines standard names for property keys associated with Device[p.122] and Driver[p.122] services.

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

See Also Device[p.122], Driver[p.122]

Since 1.1

8.10.2.1 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

8.10.2.2 public static final String DEVICE_DESCRIPTION = "DEVICE_DESCRIPTION"

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

8.10.2.3 public static final String DEVICE_SERIAL = “DEVICE_SERIAL”

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

8.10.2.4 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[p.124] 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.

8.10.3 public interface Device

Interface for identifying device services.

A service must implement this interface or use the Constants.DEVICE_CATEGORY[p.121] 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[p.122] 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[p.122]

8.10.3.1 public static final int MATCH_NONE = 0

Return value from Driver.match[p.123] indicating that the driver cannot refine the device presented to it by the device manager. The value is zero.

8.10.3.2 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.

8.10.4 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[p.122] object, the device manager enters a bidding phase. The Driver object whose match[p.123] 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[p.122], DriverLocator[p.124]

8.10.4.1 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

8.10.4.2 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[p.122] 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

8.10.5 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`[p.122]

8.10.5.1 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

8.10.5.2 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

8.10.6 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

8.10.6.1 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.

8.10.6.2 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`[p.122]) 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

8.10.7 public interface Match

Instances of `Match` are used in the `DriverSelector.select`[p.124] method to identify Driver services matching a Device service.

See Also `DriverSelector`[p.124]

Since 1.1

8.10.7.1 public ServiceReference getDriver( )

□ Return the reference to a Driver service.

Returns `ServiceReference` object to a Driver service.

8.10.7.2 public int getMatchValue( )

□ Return the match value of this object.

Returns the match value returned by this Driver service.

8.11 References

[15] *Java Communications API*
http://java.sun.com/products/javacomm

[16] *USB Specification*
http://www.usb.org/developers/data/usbspec.zip

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

[18] *Jini, Service Discovery and Usage*
http://www.jini.org/resources/

[19] *Salutation, Service Discovery Protocol*
http://www.salutation.org

9 Preferences Service Specification

Version 1.0

9.1 Introduction

Many bundles need to save some data persistently--in other words, the data is required to survive the stopping and restarting of the bundle, Framework and OSGi Service Platform. In some cases, the data is specific to a particular user. For example, imagine a bundle that implements some kind of game. User specific persistent data could include things like the user's preferred difficulty level for playing the game. Some data is not specific to a user, which we call *system* data. An example would be a table of high scores for the game.

Bundles which need to persist data in an OSGi environment can use the file system via org.osgi.framework.BundleContext.getDataFile. A file system, however, can store only bytes and characters, and provides no direct support for named values and different data types.

A popular class used to address this problem for Java applications is the java.util.Properties class. This class allows data to be stored as key/value pairs, called *properties*. For example, a property could have a name com.acme.fudd and a value of elmer. The Properties class has rudimentary support for storage and retrieving with its load and store methods. The Properties class, however, has the following limitations:

- Does not support a naming hierarchy.
- Only supports String property values.
- Does not allow its content to be easily stored in a back-end system.
- Has no user name-space management.

Since the Properties class was introduced in Java 1.0, efforts have been undertaken to replace it with a more sophisticated mechanism. One of these efforts is this Preferences Service specification.

9.1.1 Essentials

The focus of this specification is simplicity, not reliable access to stored data. This specification does *not* define a general database service with transactions and atomicity guarantees. Instead, it is optimized to deliver the stored information when needed, but it will return defaults, instead of throwing an exception, when the back-end store is not available. This approach may reduce the reliability of the data, but it makes the service easier to use, and allows for a variety of compact and efficient implementations.

This API is made easier to use by the fact that many bundles can be written to ignore any problems that the Preferences Service may have in accessing the back-end store, if there is one. These bundles will mostly or exclusively use the methods of the Preferences interface which are not declared to throw a BackingStoreException.

This service only supports the storage of scalar values and byte arrays. It is not intended for storing large data objects like documents or images. No standard limits are placed on the size of data objects which can be stored, but implementations are expected to be optimized for the handling of small objects.

A hierarchical naming model is supported, in contrast to the flat model of the Properties class. A hierarchical model maps naturally to many computing problems. For example, maintaining information about the positions of adjustable seats in a car requires information for each seat. In a hierarchy, this information can be modeled as a node per seat.

A potential benefit of the Preferences Service is that it allows user specific preferences data to be kept in a well defined place, so that a user management system could locate it. This benefit could be useful for such operations as cleaning up files when a user is removed from the system, or to allow a user's preferences to be cloned for a new user.

The Preferences Service does *not* provide a mechanism to allow one bundle to access the preferences data of another. If a bundle wishes to allow another bundle to access its preferences data, it can pass a Preferences or PreferencesService object to that bundle.

The Preferences Service is not intended to provide configuration management functionality. For information regarding Configuration Management, refer to the *Configuration Admin Service Specification* on page 33.

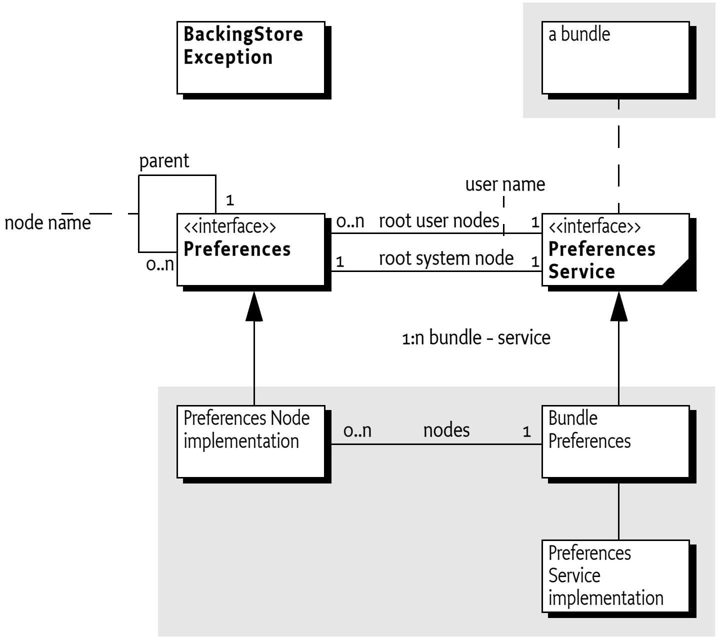
9.1.2 Entities

The PreferencesService is a relatively simple service. It provides access to the different roots of Preferences trees. A single system root node and any number of user root nodes are supported. Each *node* of such a tree is an object that implements the Preferences interface.

This Preferences interface provides methods for traversing the tree, as well as methods for accessing the properties of the node. This interface also contains the methods to flush data into persistent storage, and to synchronize the in-memory data cache with the persistent storage.

All nodes except root nodes have a parent. Nodes can have multiple children.

Figure 30 *Preferences Class Diagram*



9.1.3 Operation

The purpose of the Preferences Service specification is to allow bundles to store and retrieve properties stored in a tree of nodes, where each node implements the Preferences interface. The PreferencesService interface allows a bundle to create or obtain a Preferences tree for system properties, as well as a Preferences tree for each user of the bundle.

This specification allows for implementations where the data is stored locally on the service platform or remotely on a back-end system.

9.2 Preferences Interface

Preferences is an interface that defines the methods to manipulate a node and the tree to which it belongs. A Preferences object contains:

- A set of properties in the form of key/value pairs.
- A parent node.
- A number of child nodes.

9.2.1 Hierarchies

A valid Preferences object always belongs to a *tree*. A tree is identified by its root node. In such a tree, a Preferences object always has a single parent, except for a root node which has a null parent.

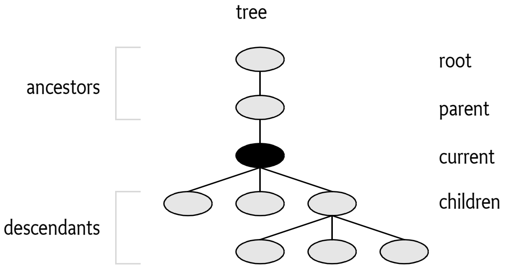
The root node of a tree can be found by recursively calling the parent() method of a node until null is returned. The nodes that are traversed this way are called the *ancestors* of a node.

Each Preferences object has a private name-space for child nodes. Each child node has a name that must be unique among its siblings. Child nodes are created by getting a child node with the node(String) method. The String argument of this call contains a path name. Path names are explained in the next section.

Child nodes can have child nodes recursively. These objects are called the *descendants* of a node.

Descendants are automatically created when they are obtained from a Preferences object, including any intermediate nodes that are necessary for the given path. If this automatic creation is not desired, the nodeExists(String) method can be used to determine if a node already exists.

Figure 31 *Categorization of nodes in a tree*



9.2.2 Naming

Each node has a name relative to its parent. A name may consist of Unicode characters except for the forward slash ("/"). There are no special names, like ".." or ".".

Empty names are reserved for root nodes. Node names that are directly created by a bundle must *always* contain at least one character.

Preferences node names and property keys are *case sensitive*: for example, "org.osgi" and "oRg.oSgI" are two distinct names.

The Preferences Service supports different roots, so there is no absolute root for the Preferences Service. This concept is similar to [21] *Windows Registry* that also supports a number of roots.

A path consists of one or more node names, separated by a slash ("/"). Paths beginning with a "/" are called *absolute paths* while other paths are called *relative paths*. Paths cannot end with a "/" except for the special case of the root node which has absolute path "/".

Path names are always associated with a specific node; this node is called the current node in the following descriptions. Paths identify nodes as follows.

- *Absolute path* – The first "/" is removed from the path, and the remainder of the path is interpreted as a relative path from the tree's root node.
- *Relative path* –
  - If the path is the empty string, it identifies the current node.
  - If the path is a name (does not contain a "/"), then it identifies the child node with that name.

- Otherwise, the first name from the path identifies a child of the current node. The name and slash are then removed from the path, and the remainder of the path is interpreted as a relative path from the child node.

9.2.3 Tree Traversal Methods

A tree can be traversed and modified with the following methods:

- childrenNames()– Returns the names of the child nodes.
- parent() – Returns the parent node.
- removeNode() – Removes this node and all its descendants.
- node(String) – Returns a Preferences object, which is created if it does not already exist. The parameter is an absolute or relative path.
- nodeExists(String) – Returns true if the Preferences object identified by the path parameter exists.

9.2.4 Properties

Each Preferences node has a set of key/value pairs called properties. These properties consist of:

- *Key* – A key is a String object and *case sensitive.*
- The name-space of these keys is separate from that of the child nodes. A Preferences node could have both a child node named fudd and a property named fudd.
- *Value* – A value can always be stored and retrieved as a String object. Therefore, it must be possible to encode/decode all values into/from String objects (though it is not required to store them as such, an implementation is free to store and retrieve the value in any possible way as long as the String semantics are maintained). A number of methods are available to store and retrieve values as primitive types. These methods are provided both for the convenience of the user of the Preferences interface, and to allow an implementation the option of storing the values in a more compact form.

All the keys that are defined in a Preferences object can be obtained with the keys() method. The clear() method can be used to clear all properties from a Preferences object. A single property can be removed with the remove(String) method.

9.2.5 Storing and Retrieving Properties

The Preferences interface has a number of methods for storing and retrieving property values based on their key. All the put* methods take as parameters a key and a value. All the get* methods take as parameters a key and a default value.

- put(String,String), get(String,String)
- putBoolean(String,boolean), getBoolean(String,boolean)
- putInt(String,int), getInt(String,int)
- putLong(String,long), getLong(String,long)
- putFloat(String,float), getFloat(String,float)
- putDouble(String,double), getDouble(String,double)
- putByteArray(String,byte[]), getByteArray(String,byte[])

The methods act as if all the values are stored as String objects, even though implementations may use different representations for the different types. For example, a property can be written as a String object and read back as a float, providing that the string can be parsed as a valid Java float object. In the event of a parsing error, the get* methods do not raise exceptions, but instead return their default parameters.

9.2.6 Defaults

All get* methods take a default value as a parameter. The reasons for having such a default are:

- When a property for a Preferences object has not been set, the default is returned instead. In most cases, the bundle developer does not have to distinguish whether or not a property exists.
- A *best effort* strategy has been a specific design choice for this specification. The bundle developer should not have to react when the back-end store is not available. In those cases, the default value is returned without further notice.
  Bundle developers who want to assure that the back-end store is available should call the flush or sync method. Either of these methods will throw a BackingStoreException if the back-end store is not available.

9.3 Concurrency

This specification specifically allows an implementation to modify Preferences objects in a back-end store. If the back-end store is shared by multiple processes, concurrent updates may cause differences between the back-end store and the in-memory Preferences objects.

Bundle developers can partly control this concurrency with the flush() and sync() method. Both methods operate on a Preferences object.

The flush method performs the following actions:

- Stores (makes persistent) any ancestors (including the current node) that do not exist in the persistent store.
- Stores any properties which have been modified in this node since the last time it was flushed.
- Removes from the persistent store any child nodes that were removed from this object since the last time it was flushed.
- Flushes all existing child nodes.

The sync method will first flush, and then ensure that any changes that have been made to the current node and its descendents in the back-end store (by some other process) take effect. For example, it could fetch all the descendants into a local cache, or it could clear all the descendants from the cache so that they will be read from the back-end store as required.

If either method fails, a BackingStoreException is thrown.

The flush or sync methods provide no atomicity guarantee. When updates to the same back-end store are done concurrently by two different processes, the result may be that changes made by different processes are intermingled. To avoid this problem, implementations may simply provide a dedicated section (or name-space) in the back-end store for each OSGi environment, so that clashes do not arise, in which case there is no reason for bundle programmers to ever call sync.

In cases where sync is used, the bundle programmer needs to take into account that changes from different processes may become intermingled, and the level of granularity that can be assumed is the individual property level. Hence, for example, if two properties need to be kept in lockstep, so that one should not be changed without a corresponding change to the other, consider combining them into a single property, which would then need to be parsed into its two constituent parts.

9.4 PreferencesService Interface

The PreferencesService is obtained from the Framework's service registry in the normal way. Its purpose is to provide access to Preferences root nodes.

A Preferences Service maintains a system root and a number of user roots. User roots are automatically created, if necessary, when they are requested. Roots are maintained on a per bundle basis. For example, a user root called elmer in one bundle is distinct from a user root with the same name in another bundle. Also, each bundle has its own system root. Implementations should use a ServiceFactory service object to create a separate PreferencesService object for each bundle.

The precise description of *user* and *system* will vary from one bundle to another. The Preference Service only provides a mechanism, the bundle may use this mechanism in any desired way.

The PreferencesService interface has the following methods to access the system root and user roots:

- getSystemPreferences() – Return a Preferences object that is the root of the system preferences tree.
- getUserPreferences(String) – Return a Preferences object associated with the user name that is given as argument. If the user does not exist, a new root is created atomically.
- getUsers() – Return an array of the names of all the users for whom a Preferences tree exists.

9.5 Cleanup

The Preferences Service must listen for bundle uninstall events, and remove all the preferences data for the bundle that is being uninstalled.

It also must handle the possibility of a bundle getting uninstalled while the Preferences Service is stopped. Therefore, it must check on startup whether preferences data exists for any bundle which is not currently installed. If it does, that data must be removed.

9.6 Changes

- Added several exception clauses to the methods.
- Removed the description of JSR 10 from this specification.

9.7 org.osgi.service.prefs

The OSGi Preferences Service Package. Specification Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.prefs; specification-version=1.0
```

9.7.1 Summary

- BackingStoreException - Thrown to indicate that a preferences operation could not complete because of a failure in the backing store, or a failure to contact the backing store. [p.134]
- Preferences - A node in a hierarchical collection of preference data. [p.134]
- PreferencesService - The Preferences Service. [p.144]

9.7.2 public class BackingStoreException extends Exception

Thrown to indicate that a preferences operation could not complete because of a failure in the backing store, or a failure to contact the backing store.

9.7.2.1 public BackingStoreException(String s)

s the detail message.

□ Constructs a `BackingStoreException` with the specified detail message.

9.7.3 public interface Preferences

A node in a hierarchical collection of preference data.

This interface allows applications to store and retrieve user and system preference data. This data is stored persistently in an implementation-dependent backing store. Typical implementations include flat files, OS-specific registries, directory servers and SQL databases.

For each bundle, there is a separate tree of nodes for each user, and one for system preferences. The precise description of "user" and "system" will vary from one bundle to another. Typical information stored in the user preference tree might include font choice, and color choice for a bundle which interacts with the user via a servlet. Typical information stored in the system preference tree might include installation data, or things like high score information for a game program.

Nodes in a preference tree are named in a similar fashion to directories in a hierarchical file system. Every node in a preference tree has a *node name* (which is not necessarily unique), a unique *absolute path name* , and a path name *relative* to each ancestor including itself.

The root node has a node name of the empty `String` object (""). Every other node has an arbitrary node name, specified at the time it is created. The only restrictions on this name are that it cannot be the empty string, and it cannot contain the slash character ('/').

The root node has an absolute path name of "`/`". Children of the root node have absolute path names of "`/`" + *<node name>* . All other nodes have absolute path names of *<parent's absolute path name>* + "`/`" + *<node name>* . Note that all absolute path names begin with the slash character.

A node *n* 's path name relative to its ancestor *a* is simply the string that must be appended to *a* 's absolute path name in order to form *n* 's absolute path name, with the initial slash character (if present) removed. Note that:

- No relative path names begin with the slash character.
- Every node's path name relative to itself is the empty string.
- Every node's path name relative to its parent is its node name (except for the root node, which does not have a parent).
- Every node's path name relative to the root is its absolute path name with the initial slash character removed.

Note finally that:

- No path name contains multiple consecutive slash characters.
- No path name with the exception of the root's absolute path name end in the slash character.
- Any string that conforms to these two rules is a valid path name.

Each `Preference` node has zero or more properties associated with it, where a property consists of a name and a value. The bundle writer is free to choose any appropriate names for properties. Their values can be of type `String`, `long`,`int`,`boolean`, `byte[]`,`float`, or `double` but they can always be accessed as if they were `String` objects.

All node name and property name comparisons are case-sensitive.

All of the methods that modify preference data are permitted to operate asynchronously; they may return immediately, and changes will eventually propagate to the persistent backing store, with an implementation-dependent delay. The `flush` method may be used to synchronously force updates to the backing store.

Implementations must automatically attempt to flush to the backing store any pending updates for a bundle's preferences when the bundle is stopped or otherwise ungets the Preferences Service.

The methods in this class may be invoked concurrently by multiple threads in a single Java Virtual Machine (JVM) without the need for external synchronization, and the results will be equivalent to some serial execution. If this class is used concurrently *by multiple JVMs* that store their preference data in the same backing store, the data store will not be corrupted, but no other guarantees are made concerning the consistency of the preference data.

9.7.3.1 public String absolutePath( )

□ Returns this node's absolute path name. Note that:

- Root node - The path name of the root node is "`/`".
- Slash at end - Path names other than that of the root node may not end in slash ('`/`').
- Unusual names -"`.`" and "`..`" have *no* special significance in path names.
- Illegal names - The only illegal path names are those that contain multiple consecutive slashes, or that end in slash and are not the root.

Returns this node's absolute path name.

9.7.3.2 public String[] childrenNames( ) throws BackingStoreException

□ Returns the names of the children of this node. (The returned array will be of size zero if this node has no children and not `null`!)

Returns the names of the children of this node.

Throws `BackingStoreException` – if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

`IllegalStateException` – if this node (or an ancestor) has been removed with the `removeNode()`[p.144] method.

9.7.3.3 public void clear( ) throws BackingStoreException

□ Removes all of the properties (key-value associations) in this node. This call has no effect on any descendants of this node.

Throws `BackingStoreException` – if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

`IllegalStateException` – if this node (or an ancestor) has been removed with the `removeNode()`[p.144] method.

See Also `remove(String)`[p.144]

9.7.3.4 public void flush( ) throws BackingStoreException

□ Forces any changes in the contents of this node and its descendants to the persistent store.

Once this method returns successfully, it is safe to assume that all changes made in the subtree rooted at this node prior to the method invocation have become permanent.

Implementations are free to flush changes into the persistent store at any time. They do not need to wait for this method to be called.

When a flush occurs on a newly created node, it is made persistent, as are any ancestors (and descendants) that have yet to be made persistent. Note however that any properties value changes in ancestors are *not* guaranteed to be made persistent.

Throws `BackingStoreException` – if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

`IllegalStateException` – if this node (or an ancestor) has been removed with the `removeNode()`[p.144] method.

See Also sync() [p.144]

9.7.3.5 public String get(String key, String def)

key key whose associated value is to be returned.

def the value to be returned in the event that this node has no value associated with key or the backing store is inaccessible.

□ Returns the value associated with the specified key in this node. Returns the specified default if there is no value associated with the key, or the backing store is inaccessible.

Returns the value associated with key, or def if no value is associated with key.

Throws IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

NullPointerException – if key is null. (A null default *is* permitted.)

9.7.3.6 public boolean getBoolean(String key, boolean def)

key key whose associated value is to be returned as a boolean.

def the value to be returned in the event that this node has no value associated with key or the associated value cannot be interpreted as a boolean or the backing store is inaccessible.

□ Returns the boolean value represented by the String object associated with the specified key in this node. Valid strings are “true”, which represents true, and “false”, which represents false. Case is ignored, so, for example, “TRUE” and “False” are also valid. This method is intended for use in conjunction with the putBoolean[p.141] method.

Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if the associated value is something other than “true” or “false”, ignoring case.

Returns the boolean value represented by the String object associated with key in this node, or null if the associated value does not exist or cannot be interpreted as a boolean.

Throws NullPointerException – if key is null.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also get(String, String) [p.137] , putBoolean(String, boolean) [p.141]

9.7.3.7 public byte[] getByteArray(String key, byte[] def)

key key whose associated value is to be returned as a byte[] object.

def the value to be returned in the event that this node has no value associated with key or the associated value cannot be interpreted as a byte[] type, or the backing store is inaccessible.

□ Returns the byte[] value represented by the String object associated with the specified key in this node. Valid String objects are *Base64* encoded binary data, as defined in RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt), Section 6.8, with one minor change: the string must consist solely of characters from the *Base64 Alphabet*; no newline characters or extraneous characters are permitted. This method is intended for use in conjunction with the putByteArray[p.142] method.

Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if the associated value is not a valid Base64 encoded byte array (as defined above).

Returns the byte[] value represented by the String object associated with key in this node, or def if the associated value does not exist or cannot be interpreted as a byte[].

Throws NullPointerException – if key is null. (A null value for def*is* permitted.)

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also get(String, String)[p.137], putByteArray(String, byte[])[p.142]

9.7.3.8 public double getDouble(String key, double def)

key key whose associated value is to be returned as a double value.

def the value to be returned in the event that this node has no value associated with key or the associated value cannot be interpreted as a double type or the backing store is inaccessible.

□ Returns the double value represented by the String object associated with the specified key in this node. The String object is converted to a double value as by Double.parseDouble(String). Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if Double.parseDouble(String) would throw a NumberFormatException if the associated value were passed. This method is intended for use in conjunction with the putDouble[p.142] method.

Returns the double value represented by the String object associated with key in this node, or def if the associated value does not exist or cannot be interpreted as a double type.

Throws IllegalStateException – if this node (or an ancestor) has been removed with the the removeNode()[p.144] method.

NullPointerException – if key is null.

See Also putDouble(String, double)[p.142], get(String, String)[p.137]

9.7.3.9 public float getFloat(String key, float def)

key key whose associated value is to be returned as a float value.

def the value to be returned in the event that this node has no value associated with key or the associated value cannot be interpreted as a float type or the backing store is inaccessible.

□ Returns the float value represented by the String object associated with the specified key in this node. The String object is converted to a float value as by Float.parseFloat(String). Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if Float.parseFloat(String) would throw a NumberFormatException if the associated value were passed. This method is intended for use in conjunction with the putFloat[p.142] method.

Returns the float value represented by the string associated with key in this node, or def if the associated value does not exist or cannot be interpreted as a float type.

Throws IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

NullPointerException – if key is null.

See Also putFloat(String, float) [p.142] , get(String, String) [p.137]

9.7.3.10 public int getInt(String key, int def)

key key whose associated value is to be returned as an int.

def the value to be returned in the event that this node has no value associated with key or the associated value cannot be interpreted as an int or the backing store is inaccessible.

□ Returns the int value represented by the String object associated with the specified key in this node. The String object is converted to an int as by Integer.parseInt(String). Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if Integer.parseInt(String) would throw a NumberFormatException if the associated value were passed. This method is intended for use in conjunction with the putInt[p.143] method.

Returns the int value represented by the String object associated with key in this node, or def if the associated value does not exist or cannot be interpreted as an int type.

Throws NullPointerException – if key is null.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also putInt(String, int) [p.143] , get(String, String) [p.137]

9.7.3.11 public long getLong(String key, long def)

key key whose associated value is to be returned as a long value.

def the value to be returned in the event that this node has no value associated with key or the associated value cannot be interpreted as a long type or the backing store is inaccessible.

□ Returns the long value represented by the String object associated with the specified key in this node. The String object is converted to a long as by Long.parseLong(String). Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if Long.parseLong(String) would throw a NumberFormatException if the associated value were passed. This method is intended for use in conjunction with the putLong[p.143] method.

Returns the long value represented by the String object associated with key in this node, or def if the associated value does not exist or cannot be interpreted as a long type.

Throws NullPointerException – if key is null.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also putLong(String, long)[p.143], get(String, String)[p.137]

9.7.3.12 public String[] keys( ) throws BackingStoreException

□ Returns all of the keys that have an associated value in this node. (The returned array will be of size zero if this node has no preferences and not null!)

Returns an array of the keys that have an associated value in this node.

Throws BackingStoreException – if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

9.7.3.13 public String name( )

□ Returns this node's name, relative to its parent.

Returns this node's name, relative to its parent.

9.7.3.14 public Preferences node(String pathName)

pathName the path name of the Preferences object to return.

□ Returns a named Preferences object (node), creating it and any of its ancestors if they do not already exist. Accepts a relative or absolute pathname. Absolute pathnames (which begin with '/') are interpreted relative to the root of this node. Relative pathnames (which begin with any character other than '/') are interpreted relative to this node itself. The empty string ("") is a valid relative pathname, referring to this node itself.

If the returned node did not exist prior to this call, this node and any ancestors that were created by this call are not guaranteed to become persistent until the flush method is called on the returned node (or one of its descendants).

Returns the specified Preferences object.

Throws IllegalArgumentException – if the path name is invalid.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

NullPointerException – if path name is null.

See Also flush()[p.136]

9.7.3.15 public boolean nodeExists(String pathName) throws BackingStoreException

pathName the path name of the node whose existence is to be checked.

□ Returns true if the named node exists. Accepts a relative or absolute pathname. Absolute pathnames (which begin with '/') are interpreted relative to the root of this node. Relative pathnames (which begin with any character other than '/') are interpreted relative to this node itself. The pathname "" is valid, and refers to this node itself.

If this node (or an ancestor) has already been removed with the `removeNode()`[p.144] method, it *is* legal to invoke this method, but only with the pathname ""; the invocation will return `false`. Thus, the idiom `p.nodeExists("")` may be used to test whether `p` has been removed.

Returns true if the specified node exists.

Throws `BackingStoreException` – if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

`IllegalStateException` – if this node (or an ancestor) has been removed with the `removeNode()`[p.144] method and `pathname` is not the empty string ("").

`IllegalArgumentException` – if the path name is invalid (i.e., it contains multiple consecutive slash characters, or ends with a slash character and is more than one character long).

9.7.3.16 public Preferences parent( )

□ Returns the parent of this node, or `null` if this is the root.

Returns the parent of this node.

Throws `IllegalStateException` – if this node (or an ancestor) has been removed with the `removeNode()`[p.144] method.

9.7.3.17 public void put(String key, String value)

key key with which the specified value is to be associated.

value value to be associated with the specified key.

□ Associates the specified value with the specified key in this node.

Throws `NullPointerException` – if `key` or `value` is `null`.

`IllegalStateException` – if this node (or an ancestor) has been removed with the `removeNode()`[p.144] method.

9.7.3.18 public void putBoolean(String key, boolean value)

key `key` with which the string form of value is to be associated.

value value whose string form is to be associated with `key`.

□ Associates a `String` object representing the specified `boolean` value with the specified key in this node. The associated string is "true" if the value is `true`, and "false" if it is `false`. This method is intended for use in conjunction with the `getBoolean`[p.137] method.

Implementor's note: it is *not* necessary that the value be represented by a string in the backing store. If the backing store supports `boolean` values, it is not unreasonable to use them. This implementation detail is not visible through the `Preferences` API, which allows the value to be read as a `boolean` (with `getBoolean`) or a `String` (with `get`) type.

Throws `NullPointerException` – if `key` is `null`.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also getBoolean(String, boolean)[p.137], get(String, String)[p.137]

9.7.3.19 public void putByteArray(String key, byte[] value)

key key with which the string form of value is to be associated.

value value whose string form is to be associated with key.

□ Associates a String object representing the specified byte[] with the specified key in this node. The associated String object the *Base64* encoding of the byte[], as defined in RFC 2045 (http://www.ietf.org/rfc/rfc2045.txt), Section 6.8, with one minor change: the string will consist solely of characters from the *Base64 Alphabet*; it will not contain any newline characters. This method is intended for use in conjunction with the getByteArray[p.137] method.

Implementor's note: it is *not* necessary that the value be represented by a String type in the backing store. If the backing store supports byte[] values, it is not unreasonable to use them. This implementation detail is not visible through the Preferences API, which allows the value to be read as an a byte[] object (with getByteArray) or a String object (with get).

Throws NullPointerException – if key or value is null.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also getByteArray(String, byte[])[p.137], get(String, String)[p.137]

9.7.3.20 public void putDouble(String key, double value)

key key with which the string form of value is to be associated.

value value whose string form is to be associated with key.

□ Associates a String object representing the specified double value with the specified key in this node. The associated String object is the one that would be returned if the double value were passed to Double.toString(double). This method is intended for use in conjunction with the getDouble[p.138] method

Implementor's note: it is *not* necessary that the value be represented by a string in the backing store. If the backing store supports double values, it is not unreasonable to use them. This implementation detail is not visible through the Preferences API, which allows the value to be read as a double (with getDouble) or a String (with get) type.

Throws NullPointerException – if key is null.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also getDouble(String, double)[p.138]

9.7.3.21 public void putFloat(String key, float value)

key key with which the string form of value is to be associated.

value value whose string form is to be associated with key.

□ Associates a String object representing the specified float value with the specified key in this node. The associated String object is the one that would be returned if the float value were passed to Float.toString(float). This method is intended for use in conjunction with the getFloat[p.138] method.

Implementor's note: it is *not* necessary that the value be represented by a string in the backing store. If the backing store supports float values, it is not unreasonable to use them. This implementation detail is not visible through the Preferences API, which allows the value to be read as a float (with getFloat) or a String (with get) type.

Throws NullPointerException – if key is null.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also getFloat(String, float)[p.138]

9.7.3.22 public void putInt(String key, int value)

key key with which the string form of value is to be associated.

value value whose string form is to be associated with key.

□ Associates a String object representing the specified int value with the specified key in this node. The associated string is the one that would be returned if the int value were passed to Integer.toString(int). This method is intended for use in conjunction with getInt[p.139] method.

Implementor's note: it is *not* necessary that the property value be represented by a String object in the backing store. If the backing store supports integer values, it is not unreasonable to use them. This implementation detail is not visible through the Preferences API, which allows the value to be read as an int (with getInt or a String (with get) type.

Throws NullPointerException – if key is null.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also getInt(String, int)[p.139]

9.7.3.23 public void putLong(String key, long value)

key key with which the string form of value is to be associated.

value value whose string form is to be associated with key.

□ Associates a String object representing the specified long value with the specified key in this node. The associated String object is the one that would be returned if the long value were passed to Long.toString(long). This method is intended for use in conjunction with the getLong[p.139] method.

Implementor's note: it is *not* necessary that the value be represented by a String type in the backing store. If the backing store supports long values, it is not unreasonable to use them. This implementation detail is not visible through the Preferences API, which allows the value to be read as a long (with getLong or a String (with get) type.

Throws NullPointerException – if key is null.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also getLong(String, long)[p.139]

9.7.3.24 public void remove(String key)

key key whose mapping is to be removed from this node.

□ Removes the value associated with the specified key in this node, if any.

Throws IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also get(String, String)[p.137]

9.7.3.25 public void removeNode( ) throws BackingStoreException

□ Removes this node and all of its descendants, invalidating any properties contained in the removed nodes. Once a node has been removed, attempting any method other than name(),absolutePath() or nodeExists("") on the corresponding Preferences instance will fail with an IllegalStateException. (The methods defined on Object can still be invoked on a node after it has been removed; they will not throw IllegalStateException.)

The removal is not guaranteed to be persistent until the flush method is called on the parent of this node. (It is illegal to remove the root node.)

Throws IllegalStateException – if this node (or an ancestor) has already been removed with the removeNode()[p.144] method.

RuntimeException – if this is a root node.

BackingStoreException – if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

See Also flush()[p.136]

9.7.3.26 public void sync( ) throws BackingStoreException

□ Ensures that future reads from this node and its descendants reflect any changes that were committed to the persistent store (from any VM) prior to the sync invocation. As a side-effect, forces any changes in the contents of this node and its descendants to the persistent store, as if the flush method had been invoked on this node.

Throws BackingStoreException – if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

IllegalStateException – if this node (or an ancestor) has been removed with the removeNode()[p.144] method.

See Also flush()[p.136]

9.7.4 public interface PreferencesService

The Preferences Service.

Each bundle using this service has its own set of preference trees: one for system preferences, and one for each user.

A `PreferencesService` object is specific to the bundle which obtained it from the service registry. If a bundle wishes to allow another bundle to access its preferences, it should pass its `PreferencesService` object to that bundle.

9.7.4.1 public Preferences getSystemPreferences( )

□ Returns the root system node for the calling bundle.

9.7.4.2 public Preferences getUserPreferences(String name)

□ Returns the root node for the specified user and the calling bundle.

9.7.4.3 public String[] getUsers( )

□ Returns the names of users for which node trees exist.

9.8 References

[20] *JSR 10 Preferences API*
http://www.jcp.org/jsr/detail/10.jsp

[21] *Windows Registry*
http://www.microsoft.com/technet/win98/reg.asp

[22] *RFC 2045 Base 64 encoding*
http://www.ietf.org/rfc/rfc2045.txt

10 User Admin Service Specification

Version 1.0

10.1 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 [23] *The Java Security Architecture for JDK 1.2.* It performs a role similar to [24] *Java Authentication and Authorization Service.*

10.1.1 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.

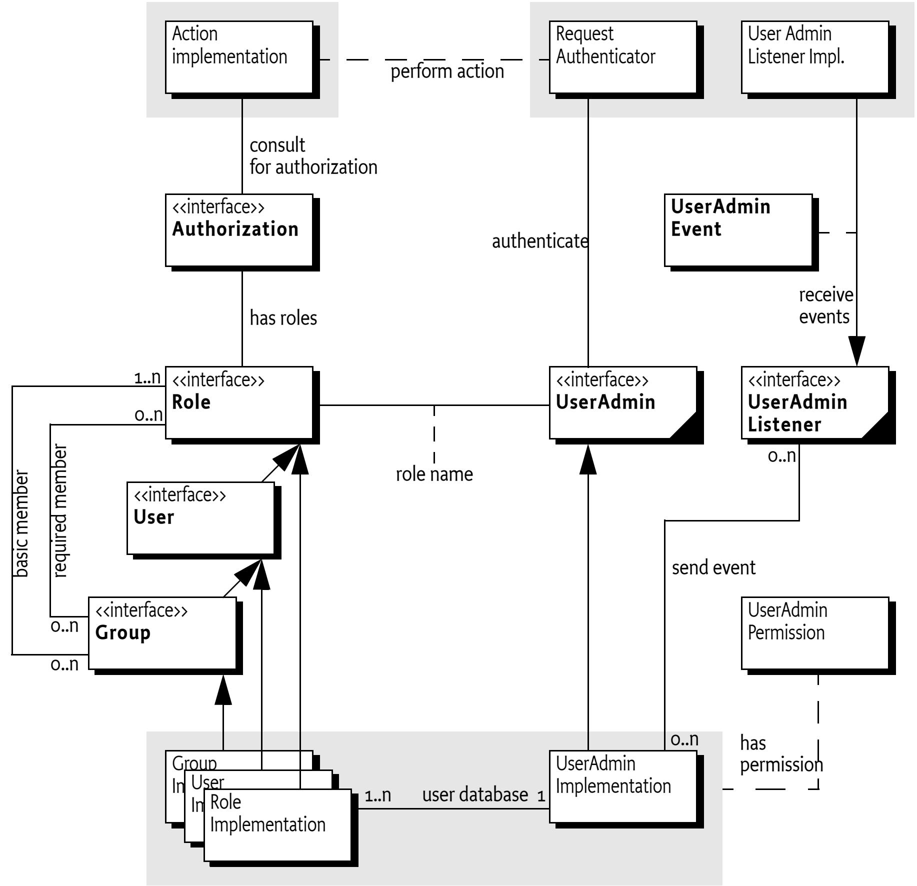
- *Administration* – Administering authorizations for each possible action and initiator is time-consuming and error-prone. It is therefore necessary to have mechanisms to group end users and make it simple to assign authorizations to all members of a group at one time.

10.1.2 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 32 *User Admin Service*, org.osgi.service.useradmin



10.1.3 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 179.

10.2 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.

10.2.1 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.

10.2.2 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;
}
```

10.2.3 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.

10.3 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.

10.3.1 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" );
}
```

10.3.2 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.

Groups	Elmer	Fudd	Marvin	Pepe	Daffy	Foghorn
Residents	Basic	Basic	Basic	Basic	-	-
Buddies	-	-	-	-	Basic	Basic
Children	-	-	Basic	Basic	-	-
Adults	Basic	Basic	-	-	-	-
Administrators	Basic	-	-	-	-	-

Table 12 Example Groups with 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

Table 13 Example Action Groups with their Basic and Required Members

10.4 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.

10.5 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.

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.

10.6 Security

The User Admin service is related to the security model of the OSGi Service Platform, but is complementary to the [23] *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.

10.6.1 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.

10.7 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 J2SE version 1.3 ("JDK 1.3") and existing mechanisms in the previous OSGi Service Gateway 1.0 specification.

10.7.1 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.

10.7.2 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.

10.7.3 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.

10.8 Changes

The description of the Http Service authentication has been removed because it duplicated the description in the Http Service Specification.

10.9 org.osgi.service.useradmin

The OSGi User Admin service Package. Specification Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.useradmin; specification-version=1.0
```

10.9.1 Summary

- Authorization - The Authorization interface encapsulates an authorization context on which bundles can base authorization decisions, where appropriate. [p.158]
- Group - A named grouping of roles (Role objects). [p.159]

- Role - The base interface for Role objects managed by the User Admin service. [p.161]
- User - A User role managed by a User Admin service. [p.162]
- UserAdmin - This interface is used to manage a database of named Role objects, which can be used for authentication and authorization purposes. [p.163]
- UserAdminEvent - Role change event. [p.165]
- UserAdminListener - Listener for UserAdminEvents. [p.166]
- UserAdminPermission - Permission to configure and access the Role[p.161] objects managed by a User Admin service. [p.166]

10.9.2 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[p.164] 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.

10.9.2.1 public String getName( )

□ Gets the name of the User[p.162] that this Authorization context was created for.

Returns The name of the User[p.162] object that this Authorization context was created for, or null if no user was specified when this Authorization context was created.

10.9.2.2 public String[] getRoles( )

□ Gets the names of all roles encapsulated by this Authorization context.

Returns The names of all roles encapsulated 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.

10.9.2.3 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[p.159] object for each role and adding User[p.162] objects to it.

Returns true if this Authorization context implies the specified role, otherwise false.

10.9.3 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
```

10.9.3.1 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.

10.9.3.2 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.

10.9.3.3 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.

10.9.3.4 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.

10.9.3.5 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.

10.9.4 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[p.166] 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.

10.9.4.1 public static final int GROUP = 2

The type of a Group[p.159] role.

The value of GROUP is 2.

10.9.4.2 public static final int ROLE = 0

The type of a predefined role.

The value of ROLE is 0.

10.9.4.3 public static final int USER = 1

The type of a User[p.162] role.

The value of USER is 1.

10.9.4.4 public static final String USER_ANYONE = "user.anyone"

The name of the predefined role, user.anyone, that all users and groups belong to.

10.9.4.5 public String getName( )

□ Returns the name of this role.

Returns The role's name.

10.9.4.6 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[p.165] 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[p.166] named after the property name (or a prefix of it) with action changeProperty is required.

Returns Dictionary containing the properties of this Role object.

10.9.4.7 public int getType( )

□ Returns the type of this role.

Returns The role's type.

10.9.5 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[p.161] class) associated with it. Specific UserAdminPermission[p.166] 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.

10.9.5.1 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[p.165] 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[p.166] 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[p.166] 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.

10.9.5.2 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.

10.9.6 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[p.162] object can be configured with credentials (e.g., a password) and properties (e.g., a street address, phone number, etc.).

A Group[p.159] object represents an aggregation of User[p.162] and Group[p.159] 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.

10.9.6.1 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[p.161] type or Role.GROUP[p.161] 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[p.165] 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.

10.9.6.2 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.

10.9.6.3 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.

10.9.6.4 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.

10.9.6.5 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.

10.9.6.6 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.

If the Role object was removed, a UserAdminEvent object of type UserAdminEvent.ROLE_REMOVED[p.165] 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.

10.9.7 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[p.165] type, ROLE_CHANGED[p.165] type, and ROLE_REMOVED[p.165] type. Additional event types may be defined in the future.

See Also UserAdmin[p.163] , UserAdminListener[p.166]

10.9.7.1 public static final int ROLE_CHANGED = 2

A Role object has been modified.

The value of ROLE_CHANGED is 0x00000002.

10.9.7.2 public static final int ROLE_CREATED = 1

A Role object has been created.

The value of ROLE_CREATED is 0x00000001.

10.9.7.3 public static final int ROLE_REMOVED = 4

A Role object has been removed.

The value of ROLE_REMOVED is 0x00000004.

10.9.7.4 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.

10.9.7.5 public Role getRole( )

□ Gets the Role object this event was generated for.

Returns The Role object this event was generated for.

10.9.7.6 public ServiceReference getServiceReference( )

□ Gets the ServiceReference object of the User Admin service that generated this event.

Returns The User Admin service's ServiceReference object.

10.9.7.7 public int getType( )

□ Returns the type of this event.

The type values are ROLE_CREATED[p.165] type, ROLE_CHANGED[p.165] type, and ROLE_REMOVED[p.165] type.

Returns The event type.

10.9.8 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[p.163] , UserAdminEvent[p.165]

10.9.8.1 public void roleChanged(UserAdminEvent event)

event The UserAdminEvent object.

□ Receives notification that a Role object has been created, removed, or modified.

10.9.9 public final class UserAdminPermission extends BasicPermission

Permission to configure and access the Role[p.161] 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[p.162] 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 re-
move)
                    Role object properties whose names start
with
                    the name argument specified in the con-
structor.
  changeCredential  Permission to change (i.e., add and re-
move)
                    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,getCredential,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.

10.9.9.1 public static final String ADMIN = “admin”

The permission name “admin”.

10.9.9.2 public static final String CHANGE_CREDENTIAL = “changeCredential”

The action string “changeCredential”.

10.9.9.3 public static final String CHANGE_PROPERTY = “changeProperty”

The action string “changeProperty”.

10.9.9.4 public static final String GET_CREDENTIAL = “getCredential”

The action string “getCredential”.

10.9.9.5 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.

10.9.9.6 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.

10.9.9.7 public String getActions( )

□ Returns the canonical string representation of the actions, separated by comma.

Returns the canonical string representation of the actions.

10.9.9.8 public int hashCode( )

□ Returns the hash code of this `UserAdminPermission` object.

10.9.9.9 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.

10.9.9.10 public PermissionCollection newPermissionCollection( )

□ Returns a new `PermissionCollection` object for storing `UserAdminPermission` objects.

Returns a new `PermissionCollection` object suitable for storing `UserAdminPermission` objects.

10.9.9.11 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`

10.10 References

[23] *The Java Security Architecture for JDK 1.2*
Version 1.0, Sun Microsystems, October 1998
http://java.sun.com/products/jdk/1.4/docs/guide/security/spec/security-spec.doc.html

[24] *Java Authentication and Authorization Service*
http://java.sun.com/products/jaas

11 Http Service Specification

Version 1.1

11.1 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:

- [25] *HTTP 1.0 Specification RFC-1945*
- [26] *HTTP 1.1 Specification RFC-2616*

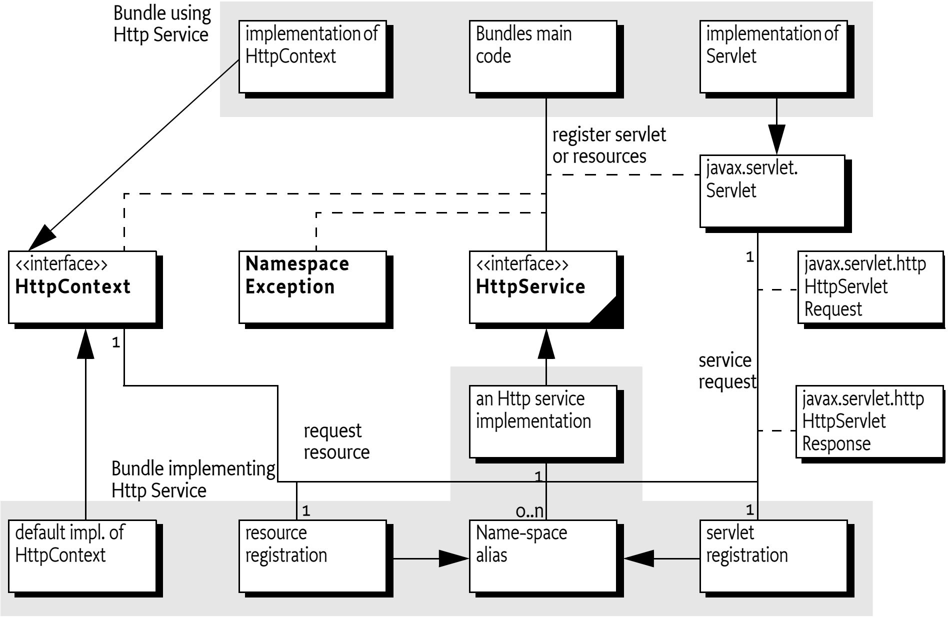
Alternatively, implementations of this service can support other protocols if these protocols can conform to the semantics of the javax.servlet API. This additional support is necessary because the Http Service is closely related to [27] *Java Servlet Technology*. Http Service implementations must support at least version 2.1 of the Java Servlet API.

11.1.1 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 33 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
1
javax.servlet.http HttpServlet Request
service request
1
an Http service implementation
javax.servlet.http HttpServlet Response
request resource
Bundle implementing Http Service
1
1
0..n
1
default impl. of HttpContext
resource registration
Name-space alias
servlet registration

11.2 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/servlet?name=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 176 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 179.

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.)

11.3 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 172. 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 getMime(String) implementation of the default HttpContext object should return a reasonable mapping. Its handleSecurity(HttpServlet Request,HttpServletResponse) may implement an authentication mechanism that is implementation-dependent.

11.4 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 179. 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 179. 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.

Alias	Internal Name	URI	getResource Parameter
/	(empty string)	/fudd/bugs	/fudd/bugs
/	/	/fudd/bugs	/fudd/bugs
/	/tmp	/fudd/bugs	/tmp/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

Table 14 Examples of Name-space Mapping

11.5 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
```

11.6 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 [28] *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/plan ; charset=iso-8859-1
```

The Internet Assigned Number Authority (IANA) maintains a set of defined MIME media types. This list can be found at [29] *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 [30] *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.

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
.htm .html	text/html	Hyper Text Markup Language
.wbmp	image/vnd.wap.wbmp	Bitmaps for WAP

Table 15 *Sample Extension to MIME Media Mapping*

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.

11.7 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 [31] *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 [31] *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.

11.8 Security

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

11.8.1 Accessing Resources in Bundles

The Http Service must be granted AdminPermission 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.

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.

11.8.2 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. Servlet and HttpContext objects must use a doPrivileged construct in their implementations when performing privileged operations.

11.9 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.

11.10 Changes

The API of the HTTP service has not been changed and the version is therefore also not changed.

11.10.1 Example

The example in *Mapping HTTP Requests to Servlet and Resource Registrations* on page 176 contained two errors in calling non-existing methods. These were corrected.

11.10.2 Use of single /

Ambiguities in the use of a single '/' were corrected and an example was added to Table 14, “Examples of Name-space Mapping,” on page 177.

11.10.3 MIME Type Table

Table 15, “Sample Extension to MIME Media Mapping,” on page 179 contained the .html extension twice. The first occurrence was replaced with .htm.

11.11 org.osgi.service.http

The OSGi Http Service Package. Specification Version 1.1.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.http; specification-version=1.1
```

11.11.1 Summary

- HttpContext - This interface defines methods that the Http Service may call to get information about a registration. [p.183]
- HttpService - The Http Service allows other bundles in the OSGi environment to dynamically register resources and servlets into the URI namespace of Http Service. [p.185]
- 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. [p.187]

11.11.2 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.

11.11.2.1 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

11.11.2.2 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

11.11.2.3 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

11.11.2.4 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.

11.11.2.5 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.

11.11.2.6 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[p.183] request attribute to the type of authentication used, and the REMOTE_USER[p.183] 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[p.183] 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.

11.11.3 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[p.183]

11.11.3.1 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 the org.osgi.framework.AdminPermission.

Returns a default HttpContext object.

Since 1.1

11.11.3.2 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 ('/'). 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`[p.185]).

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

`IllegalArgumentException` – if any of the parameters are invalid

11.11.3.3 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, initpar-
ams, 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[p.185]).

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

11.11.3.4 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[p.187] 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[p.187] must be explicitly called to cause the destroy method of the servlet to be called. This can be done in the org.osgi.framework.BundleActivator.stopmethod 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.

11.11.4 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.

11.11.4.1 public NamespaceException(String message)

message the detail message

□ Construct a NamespaceException object with a detail message.

11.11.4.2 public NamespaceException(String message, Throwable exception)

message the detail message

exception the nested exception

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

11.11.4.3 public Throwable getException( )

□ Returns the nested exception.

Returns the nested exception or null if there is no nested exception.

11.12 References

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

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

[27] *Java Servlet Technology*
http://java.sun.com/products/servlet/index.html

[28] *MIME Multipurpose Internet Mail Extension*
http://www.nacs.uci.edu/indiv/ehood/MIME/MIME.html

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

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

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

12 UPnP™ Device Service Specification

Version 1.0

12.1 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 UPnP specifications leverage Internet protocols, including IP, TCP, UDP, HTTP, and XML. 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 succesful 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 [32] *UPnP Device Architecture* and does not further explain the UPnP specifications. The UPnP specifications are maintained by [33] *UPnP Forum*.

UPnP is a trademark of the UPnP Implementers Corporation.

12.1.1 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 publishedto 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.

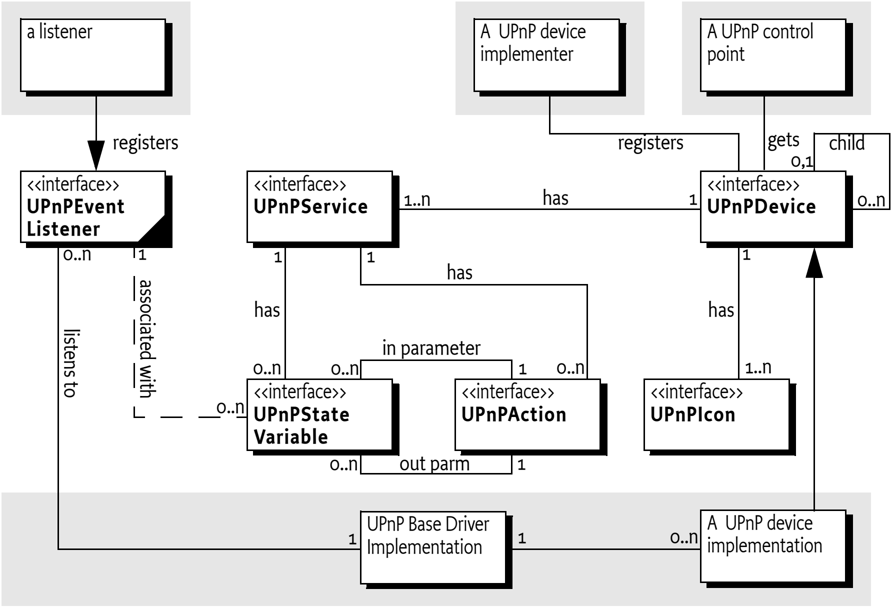
12.1.2 Entities

- *UPnP Base Driver* – The bundle that implements the bridge between OSGi and UPnP networks. This entity is not represented as a service.
- *UPnP RootDevice* –A physical device can contain one or more root devices. Root devices contain one ore more devices. A root device is mod-

elled 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.
- *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 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.
- *UDN* – Unique Device Name, a name that uniquely identifies the a specific device.

Figure 34 UPnP Service Specification class Diagram org.osgi.service.upnp package



12.1.3 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 detcted 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.

12.2 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 (refridgerators, coffeemakers) 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.

12.2.1 UPnP Base Driver

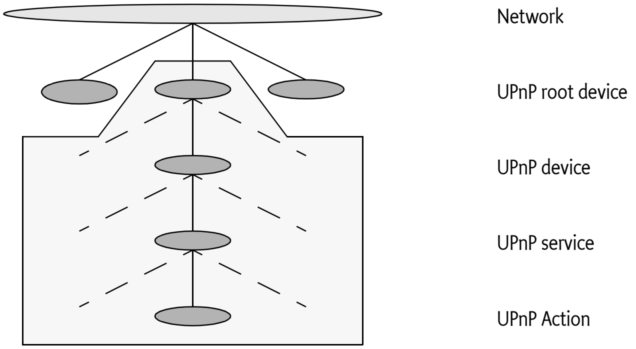
The functionality of the UPnP service is implemented in a UPnP *base driver*. This is a bundle that implements the UPnP protocols and handles the interaction with bundles that use the UPnP devices. A UPnP base driver bundle must provide the following functions:

- 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.

12.3 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 35 on page 193 illustrates this hierarchy.

Figure 35 *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.

12.3.1 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 – 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 – An array of UDNs of this device's children. Type is a String[] object.

12.3.2 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 97 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.

12.3.3 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.

12.4 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.

12.5 UPnPService

A UPnP Device contains a number of UPnPService objects. UPnPService objects combine actions and state variables.

12.5.1 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.

12.6 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 );
   }
   ...
}
```

12.7 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.

12.8 Event API

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.

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) 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) 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) Service identity. Only events generated by services matching the given service ID are delivered.
- UPnPService.TYPE – (UPnP.service.type) Only events generated by services of of the given type are delivered.

If an event is generated, 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.

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.

12.8.1 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 status of each state variable. This behavior simplifies the synchronization of a device and an event-driven client.

The UPnP Driver must mimic this event distribution for all UPnP Event Listener services when they are registered. The driver must guarantee the same behavior for all registrations by keeping an internal history of the events.

The call to the listener's notification method must be done asynchronously.

12.9 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.

12.10 Dates and Times

The UPnP specification uses different types for date and time concepts. An overview of these types is given in Table 16 on page 197.

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)

Table 16 Mapping UPnP Date/Time types to Java

The UPnP specification points to [37] *XML Schema*. In this standard, [38] *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.

12.11 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.

12.12 Networking considerations

12.12.1 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.

12.13 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[REGISTER|GET,UPnPDevice|UPnPEventListener] be used sparingly and only for bundles that are trusted.

12.14 org.osgi.service.upnp

The OSGi UPnP API Package. Specification Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.upnp; specification-ver-
sion=1.0
```

12.14.1 Summary

- UPnPAction - A UPnP action. [p.199]
- UPnPDevice - Represents a UPnP device. [p.200]
- UPnPEventListener - UPnP Events are mapped and delivered to applications according to the OSGi whiteboard model. [p.204]
- UPnPException - [p.205]
- UPnPIcon - A UPnP icon representation. [p.205]
- UPnPLocalStateVariable - [p.206]
- UPnPService - A representation of a UPnP Service. [p.206]
- UPnPStateVariable - The meta-information of a UPnP state variable as declared in the device's service state table (SST). [p.208]

12.14.2 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.

12.14.2.1 public String[] getInputArgumentNames( )

□ Lists all input arguments for this action.

Each action may have zero or more input arguments.

Returns Array of input argument names or `null` if no input arguments.

See Also `UPnPStateVariable`[p.208]

12.14.2.2 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_`.

Returns Name of action, must not contain a hyphen character or a hash character

12.14.2.3 public String[] getOutputArgumentNames( )

□ List all output arguments for this action.

Returns Array of output argument names or `null` if there are no output arguments.

See Also `UPnPStateVariable`[p.208]

12.14.2.4 public String getReturnArgumentName( )

□ Returns the name of the designated return argument.

One of the output arguments can be flagged as a designated return argument.

Returns The name of the designated return argument or `null` if none is marked.

12.14.2.5 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.

See Also `UPnPStateVariable`[p.208]

12.14.2.6 public Dictionary invoke(Dictionary args) throws Exception, UPnPException

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 `Exception` – The execution fails for some reason.

See Also `UPnPStateVariable`[p.208]

12.14.3 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)`.

12.14.3.1 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"

12.14.3.2 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`

12.14.3.3 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".

12.14.3.4 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".

12.14.3.5 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".

12.14.3.6 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".

12.14.3.7 public static final int MATCH_GENERIC = 1

Constant for the UPnP device match scale, indicating a generic match for the device. Value is 1.

12.14.3.8 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.

12.14.3.9 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.

12.14.3.10 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.

12.14.3.11 public static final int MATCH_TYPE = 3

Constant for the UPnP device match scale, indicating a match with the device type. Value is 3.

12.14.3.12 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".

12.14.3.13 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".

12.14.3.14 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".

12.14.3.15 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".

12.14.3.16 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".

12.14.3.17 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".

12.14.3.18 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".

12.14.3.19 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".

12.14.3.20 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.

12.14.3.21 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".

12.14.3.22 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".

12.14.3.23 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.

Returns Dictionary mapping property name Strings to property value Strings

12.14.3.24 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.

12.14.3.25 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.

12.14.3.26 public UPnPService[] getServices( )

□ Lists all services provided by this device.

Returns Array of services or null if no services are available.

12.14.4 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.

12.14.4.1 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.

12.14.4.2 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

12.14.5 public class UPnPException extends Exception

12.14.5.1 public UPnPException(int errorCode, String errordesc)

errorCode errorCode which defined UPnP Device Architecture V1.0.

errordesc errorDescription which explain the type of propblem.

□ This constructor creates a UPnPException on the specified error code and error description.

12.14.5.2 public int getUPnPError_Code( )

□ Returns the UPnPError Code occured by UPnPDevices during invocation.

Returns The UPnPErrorCode defined by a UPnP Forum working committee or specified by a UPnP vendor.

12.14.6 public interface UPnPIcon

A UPnP icon representation. Each UPnP device can contain zero or more icons.

12.14.6.1 public int getDepth( )

□ Returns the color depth of the icon in bits.

Returns The color depth in bits. If the actual color depth of the icon is unknown, -1 is returned.

12.14.6.2 public int getHeight( )

□ Returns the height of the icon in pixels. If the actual height of the icon is unknown, -1 is returned.

Returns The height in pixels, or -1 if unknown.

12.14.6.3 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 availble via the `getMimeType()` method.

Returns An InputStream to read the icon graphics data from.

See Also `UPnPIcon.getMimeType()`[p.206]

12.14.6.4 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 at ftp://ftp.isi.edu/in-notes/iana/assignments/media-types/media-types (ftp://ftp.isi.edu/in-notes/iana/assignments/media-types/media-types) .

Typical values returned include: "image/jpeg" or "image/gif"

Returns The MIME type of the encoded icon.

12.14.6.5 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.

12.14.6.6 public int getWidth( )

□ Returns the width of the icon in pixels. If the actual width of the icon is unknown, -1 is returned.

Returns The width in pixels, or -1 if unknown.

12.14.7 public interface UPnPLocalStateVariable extends UPnPStateVariable

12.14.7.1 public Object getCurrentValue( )

□ This method will keep the current values of UPnPStateVariables of UPnPDevice whenever UPnPStateVariable'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.

12.14.8 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.

12.14.8.1 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".

12.14.8.2 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()` [p.208]

12.14.8.3 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.

12.14.8.4 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.

12.14.8.5 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`.

Returns The service ID suffix defined by a UPnP Forum working committee or specified by a UPnP vendor. Must be <= 64 characters. Single URI.

12.14.8.6 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.

12.14.8.7 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.

12.14.8.8 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`.

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.

12.14.8.9 public String getVersion( )

□ Returns the version suffix encoded in the `serviceType` field in the UPnP service description.

Returns The integer service version defined by a UPnP Forum working committee or specified by a UPnP vendor.

12.14.9 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.

12.14.9.1 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.

12.14.9.2 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.

12.14.9.3 public static final String TYPE_BOOLEAN = "boolean"

True or false.

Mapped to `Boolean` object.

12.14.9.4 public static final String TYPE_CHAR = "char"

Unicode string.

One character long.

Mapped to `Character` object.

12.14.9.5 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.

12.14.9.6 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.

12.14.9.7 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.

12.14.9.8 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.

12.14.9.9 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.

12.14.9.10 public static final String TYPE_I1 = "i1"

1 Byte int.

Mapped to `Integer` object.

12.14.9.11 public static final String TYPE_I2 = "i2"

2 Byte int.

Mapped to `Integer` object.

12.14.9.12 public static final String TYPE_I4 = "i4"

4 Byte int.

Must be between -2147483648 and 2147483647

Mapped to `Integer` object.

12.14.9.13 public static final String TYPE_INT = "int"

Integer number.

Mapped to `Integer` object.

12.14.9.14 public static final String TYPE_NUMBER = "number"

Same as r8.

Mapped to `Double` object.

12.14.9.15 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.

12.14.9.16 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.

12.14.9.17 public static final String TYPE_STRING = "string"

Unicode string.

No limit on length.

Mapped to `String` object.

12.14.9.18 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.

12.14.9.19 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.

12.14.9.20 public static final String TYPE_UI1 = "ui1"

Unsigned 1 `Byte` int.

Mapped to an `Integer` object.

12.14.9.21 public static final String TYPE_UI2 = "ui2"

Unsigned 2 Byte int.

Mapped to `Integer` object.

12.14.9.22 public static final String TYPE_UI4 = “ui4”

Unsigned 4 Byte int.

Mapped to Long object.

12.14.9.23 public static final String TYPE_URI = “uri”

Universal Resource Identifier.

Mapped to `String` object.

12.14.9.24 public static final String TYPE_UUID = “uuid”

Universally Unique ID.

Hexadecimal digits representing octets. Optional embedded hyphens are ignored.

Mapped to `String` object.

12.14.9.25 public String[] getAllowedValues( )

□ Returns the allowed values, if defined. Allowed values can be defined only for String types.

Returns The allowed values or `null` if not defined. Should be less than 32 characters.

12.14.9.26 public Object getDefaultValue( )

□ Returns the default value, if defined.

Returns The default value or `null` if not defined. The type of the returned object can be determined by `getJavaDataType`.

12.14.9.27 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`

Returns A class object corresponding to the Java type of this argument.

12.14.9.28 public Number getMaximum( )

□ Returns the maximum value, if defined. Maximum values can only be defined for numeric types.

Returns The maximum value or `null` if not defined.

12.14.9.29 public Number getMinimum( )

□ Returns the minimum value, if defined. Minimum values can only be defined for numeric types.

Returns The minimum value or `null` if not defined.

12.14.9.30 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_`.

Returns Name of state variable. Must not contain a hyphen character nor a hash character. Should be <32 characters.

12.14.9.31 public Number getStep( )

□ Returns the size of an increment operation, if defined. Step sizes can be defined only for numeric types.

Returns The increment size or null if not defined.

12.14.9.32 public String getUPnPDataType( )

□ Returns the UPnP type of this state variable. Valid types are defined as constants.

Returns The UPnP data type of this state variable, as defined in above constants.

12.14.9.33 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.

Returns `true` if the `StateVariable` generates events, `false` otherwise.

12.15 References

[32] *UPnP Device Architecture*
http://www.upnp.org/download/UPnPDA10_20000613.htm

[33] *UPnP Forum*
http://www.upnp.org

[34] *Simple Object Access Protocol, SOAP*
http://www.w3.org/TR/SOAP

[35] *General Event Notification Architecture, GENA*
http://www.upnp.org/download/draft-cohen-gena-client-01.txt

[36] *Simple Service Discovery Protocol, SSDP*
http://www.upnp.org/download/draft_cai_ssdp_v1_03.txt

[37] *XML Schema*
http://www.w3.org/TR/xmlschema-2

[38] *ISo 8601 Date And Time formats*
www.iso.ch

14 XML Parser Service Specification

Version 1.0

14.1 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 [18] *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

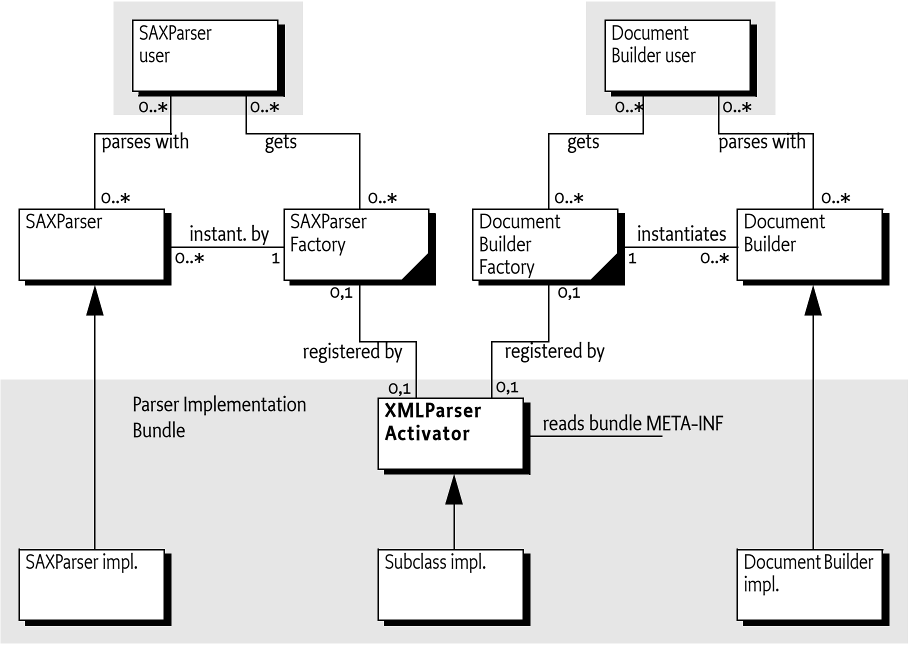
14.1.1 Essentials

- *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

14.1.2 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 26 XML Parsing diagram



14.1.3 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.

14.2 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 [18] *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.

14.3 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.

14.4 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.

14.5 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();
}
```

14.6 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.

14.6.1 JAR Based Services

Its functionality is based on the definition of the [19] *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.

14.6.2 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 utlity 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 153, that describe the instances.

14.6.3 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 is contained in the bundle.

14.7 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 [18] *JAXP* for the exact version dependencies.

This specification is related to related packages as defined in the JAXP 1.1 document. Table 9 contains the expected minimum 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

Table 9 JAXP 1.1 minimum package versions

The Xerces project from the Apache group, [20] *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.

14.8 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[REGISTER,javax.xml.parsers.DocumentBuilderFactory | javax.xml.parsers.SAXFactory] to only trusted bundles.

Using an XML parser is a common function, and ServicePermission[GET, javax.xml.parsers.DOMParserFactory | javax.xml.parsers.SAXFactory] 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.

14.9 org.osgi.util.xml

The OSGi XML Parser service Package. Specification Version 1.0.

14.9.1 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[p.158])
- javax.xml.parsers.DocumentBuilderFactory(DOMFACTORYNAME[p.158])

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[p.158] - indicates if this factory supports validating parsers. It's value is a Boolean.
- PARSER_NAMESPACEAWARE[p.158] - 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.

14.9.1.1 public static final String DOMCLASSFILE = "/META-INF/services/javax.xml.parsers.DocumentBuilderFactory"

Fully qualified path name of DOM Parser Factory Class Name file

14.9.1.2 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.

14.9.1.3 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.

14.9.1.4 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.

14.9.1.5 public static final String SAXCLASSFILE = "/META-INF/services/javax.xml.parsers.SAXParserFactory"

Fully qualified path name of SAX Parser Factory Class Name file

14.9.1.6 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.

14.9.1.7 public XMLParserActivator( )

14.9.1.8 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

14.9.1.9 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 namespaceaware 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);

14.9.1.10 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 namespaceaware 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);

14.9.1.11 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.

See Also `Bundle.start`

14.9.1.12 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.

See Also `Bundle.stop`

14.9.1.13 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.

14.10 References

[15] *XML*
http://www.w3.org/XML

[16] *SAX*
http://www.saxproject.org/

[17] *DOM Java Language Binding*
http://www.w3.org/TR/REC-DOM-Level-1/java-language-binding.html

[18] *JAXP*
http://java.sun.com/xml/jaxp

[19] *JAR File specification, services directory*
http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html

[20] *Xerces 2 Java Parser*
http://xml.apache.org/xerces2-j

14 Initial Provisioning

Version 1.0

14.1 Introduction

To allow freedom regarding the choice of management protocol, the OSGi *Remote Management Reference Architecture* on page 29, specifies 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.

14.1.1 Essentials

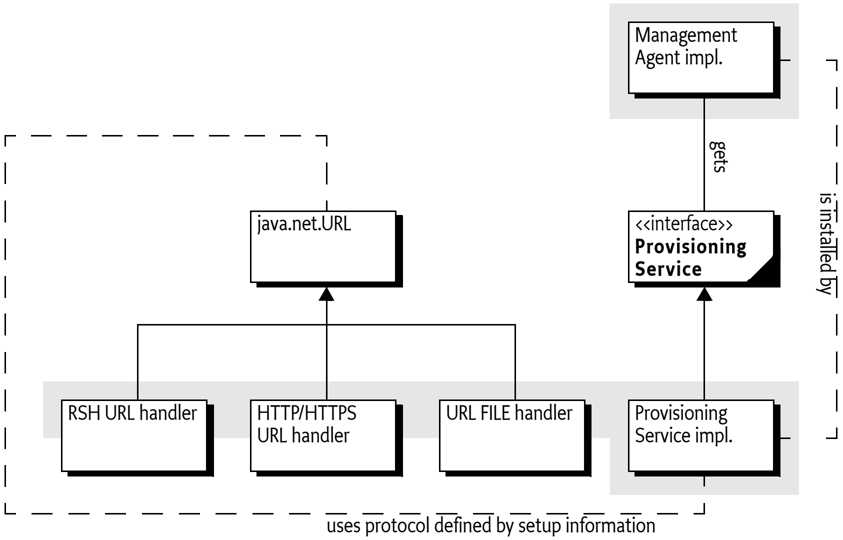
- *Policy Free* – The proposed solution must be business model agnostic; none of the affected parties (Operators, SPS Manufacturers, etc.) should be forced into any particular business model.
- *Interoperability* – The Initial Provisioning must permit arbitrary interoperability between management systems 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.

14.1.2 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 37 *Initial Provisioning*



14.2 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 19) 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 types are specified in the ZIP entry's extra field, and must be a MIME type as defined in [51] *MIME Types*. The text and bundle-url entries are translated into a String object. All other entries must be stored as a byte[].

Type	MIME Type	Description
text	MIME_STRING text/plain;charset=utf-8	Must be represented as a String object
binary	MIME_BYTE_ARRAY application/octet-stream	Must be represented as a byte array (byte[]).

Table 18 Content types of provisioning ZIP file

Type	MIME Type	Description
bundle	MIME_BUNDLE application/x-osgi-bundle	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.
bundle-url	MIME_BUNDLE_URL text/x-osgi-bundle-url; charset=utf-8	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.

Table 18 Content types of provisioning ZIP file

The Provisioning Service must install (but not start) all entries in the ZIP file that are typed in the extra field 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 18. 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.

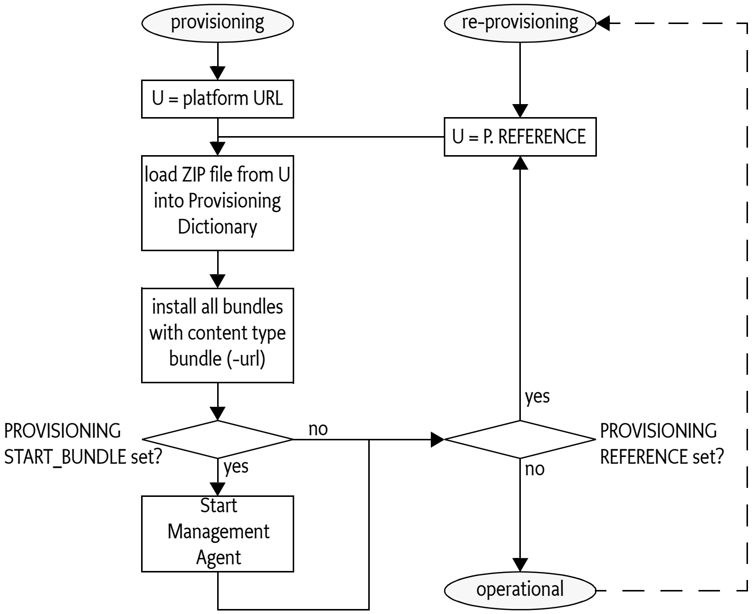
Information retrieved while loading subsequent PROVISIONING_REFERENCE URLs may replace previous key/values in the Provisioning Dictionary, but must not erase unrecognized key/values. For example, if an assignment has assigned the key proprietary-x, with a value '3', then later assignments must not override 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 18) 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 38 as a flow chart.

Figure 38 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
Start Management Agent
yes
no
PROVISIONING REFERENCE set?
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.

14.3 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.

14.3.1 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.

14.3.2 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.

14.4 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 226.
- setInformation(Dictionary) – Set a new Provisioning Dictionary. This will remove all existing entries.

Each of these method will increment the PROVISIONING_UPDATE_COUNT entry.

14.5 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.

14.6 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.

14.6.1 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 application/x-osgi-bundle or application/x-osgi-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
```

14.7 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.

14.7.1 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 233.

14.7.2 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);
}
```

14.7.3 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 [50] *RFC 2396 - Uniform Resource Identifier (URI)*.

14.8 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 $\mathrm{K_e}$ calculation.

- *m* – The calculated message authentication code.
- *3DES* – Triple DES, encryption function, see [52] *3DES*. The bytes of the key must be set to odd parity. CBC mode must be used where the padding method is defined in [53] *RFC 1423 Part III: Algorithms, Modes, and Identifiers*. In [55] *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 Autentication Code, see [56] *SHA-1*. The function takes a single parameter, the block to be worked upon.
- *HMAC* – The fuction that calculates a message authentication code, which must HMAC-SHA1. HMAC-SHA1 is defined in [45] *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.

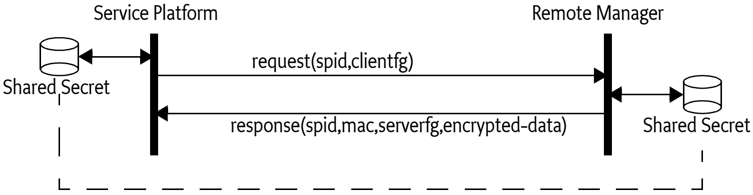
The encryption/decryption key used is specified by the following:
$e \leftarrow 3DES(K_e, IV, r)$

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 39 *Action Diagram for RSH*



14.8.1 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 [46] *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.

14.8.2 Request Coding

RSH is mapped to HTTP or HTTPS. Thus, the request parameters are URL encoded as discussed in 14.7.3 *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 238.

14.8.3 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 19.

Bytes	Description	Value hex
4	Number of bytes in header	2E
1	Major version number	01
1	Minor version number	00
16	`serverfg`	...
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	...

Table 19 RSH Header description

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.

14.8.4 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
```

14.8.5 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.

14.8.6 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
```

14.9 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.

14.9.1 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.

14.9.2 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.

14.9.3 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[GET, ProvisioningService]`.

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.

14.10 org.osgi.service.provisioning

The OSGi Provisioning Service Package. Specification Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. For example:

```
Import-Package: org.osgi.service.provisioning; specification-
version=1.0
```

14.10.1 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.

14.10.1.1 public static final String MIME_BUNDLE = "application/x-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.

14.10.1.2 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.

14.10.1.3 public static final String MIME_BYTE_ARRAY = "application/octet-stream"

MIME type to be stored in the extra field of a `ZipEntry` object for `byte[]` data.

14.10.1.4 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.

14.10.1.5 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[]`.

14.10.1.6 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`.

14.10.1.7 public static final String PROVISIONING_ROOTX509 = "provisioning.rootx509"

The key to the provisioning information that contains the root X509 certificate used to esatblish trust with operator when using HTTPS.

14.10.1.8 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.

14.10.1.9 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`.

14.10.1.10 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.

14.10.1.11 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.

14.10.1.12 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.

14.10.1.13 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.

14.10.1.14 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.

14.10.1.15 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.

14.11 References

[45] *HMAC:* Keyed-Hashing for Message Authentication
http://www.ietf.org/rfc/rfc2104.txt Krawczyk ,et. al. 1997.

[46] *NIST, FIPS PUB 180-1: Secure Hash Standard, April 1995.*

[47] *Hypertext Transfer Protocol - HTTP/1.1*
http://www.ietf.org/rfc/rfc2616.txt *Fielding, R., et. al.*

[48] *Rescorla, E., HTTP over TLS, IETF RFC 2818, May 2000*
http://www.ietf.org/rfc/rfc2818.txt.

[49] *ZIP Archive format*
ftp://ftp.uu.net/pub/archiving/zip/doc/appnote-970311-iz.zip

[50] *RFC 2396 - Uniform Resource Identifier (URI)*
http://www.ietf.org/rfc/rfc2396.txt

[51] *MIME Types*
http://www.ietf.org/rfc/rfc2046.txt and http://www.iana.org/assignments/media-types

[52] *3DES*
W/ Tuchman, "Hellman Presents No Shortcut Solution to DES," IEEE Spectrum, v. 16, n. 7 July 1979, pp40-41.

[53] *RFC 1423 Part III: Algorithms, Modes, and Identifiers*
http://www.ietf.org/rfc/rfc1423.txt

[54] *PKCS 5*
ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2

[55] *Java Cryptography API (part of Java 1.4)*
http://java.sun.com/products/jce/index-14.html

[56] *SHA-1*
U.S. Government, Proposed Federal Information Processing Standard for Secure Hash Standard, January 1992

[57] *Transport Layer Security*
http://www.ietf.org/rfc/rfc2246.txt, January 1999, The TLS Protocol Version 1.0, T. Dierks & C. Allen.

End Of Document