The OSGi Alliance
OSGi Enterprise

Release 6
July 2015

Table of Contents

105 Metatype Service Specification 105

107 User Admin Service Specification 137

121 Blueprint Container Specification   275

122 Remote Service Admin Service Specification   375

123 JTA Transaction Services Specification   423

124 JMX™ Management Model Specification 433

125 JDBC™ Service Specification 489

126 JNDI Services Specification 497

127 JPA Service Specification 517

134   Subsystem Service Specification   613

135   Common Namespaces Specification   681

702   XML Parser Service Specification   795

705   Promises Specification   805

1   Introduction

The OSGi Enterprise Expert Group (EEG) is chartered to define the technical requirements and specifications to tailor and extend the OSGi framework to address information technology software infrastructure use cases found in enterprise scenarios.

The EEG technical areas of concern include:

- Scaling, including multi-container and multi-process environments
- Distributed and/or federated service model for:
  - Multiple OSGi frameworks
  - External, heterogeneous systems
- Requirements for extensions to the OSGi publish/find/bind service model
- Enterprise-class life cycle and configuration management
- Integration of established Java EE technology into OSGi

This specification is based on *OSGi Core Release 6*. The specification combines previously published, as well as new, OSGi services that address the common use cases of enterprise application and application server developers. It serves as a first reference point for the suggested audience when considering the use of OSGi in their environment to fulfill their own needs or to better serve the needs of their customers. This collection of services is taken from the complete set of available specifications and narrowed down to what can be relevant to the enterprise domain.

The services of the Enterprise Specification have been designed to integrate with OSGi and cooperate with each other. None of the listed service specifications is mandatory; all service specifications are optional. However, services provided must follow their specification completely.

It is not suggested, or expected, that an enterprise solution will incorporate support for all listed specifications, instead a customized subset to satisfy the requirements at hand is recommended. A solution can further include other core and compendium services that are not listed as part of the Enterprise Specification. The selection of appropriate services should be driven by requirements and use cases.

The Enterprise Specification includes the recommended specifications for a number of areas. Together they address use-cases found in the enterprise context and provide a powerful set of tools to build enterprise OSGi deployments.

These Enterprise Specification areas are described in the following sections.

1.1   Overview of Services

1.1.1   Component Models

While the OSGi framework API is relatively simple to use, it is still considered infrastructure that can bleed into the application code of a bundle. This Enterprise Specification therefore provides multiple, interoperable, Dependency Injection based component models. These DI models ensure decoupling of the application code from the OSGi APIs; they provide an OSGi bundle programming model with minimal implementation dependencies and virtually no accidental complexity in the Java code. Both models provide support for handling the life cycle of services, albeit in different ways. These component models are the Declarative Services Specification and the Blueprint Container Specification:

- *Declarative Services Specification* - The Declarative Services specification provides dependency injection for services. It handles the service life cycle dynamics by notifying the component or managing the component's life cycle. See chapter *Declarative Services Specification* on page 175.
- *Blueprint Container Specification* - The Blueprint Container is derived from the Spring Dynamic Module project. It provides a general DI framework; services are supported by proxying them and damping their life cycle. See chapter *Blueprint Container Specification* on page 275.

1.1.2 Distributed Services

The OSGi framework provides a local service registry for bundles to communicate through service objects, where a service is an object that one bundle registers and another bundle looks up. The Enterprise Specification enhances this model by defining endpoints that represent services hosted in a remote systems. It allows for seamless access to remote services within the OSGi framework without changing the service layer. The remote system may or may not be based on OSGi.

The Enterprise Specification includes the specifications of:

- *Remote Services* - The Remote Services specification defines a number of service properties that participating bundles can use to convey information to a distribution provider. The distribution provider creates endpoints that are accessible to remote clients or registers proxies that access services hosted external to the OSGi framework. See chapter *Remote Services* on page 21.
- *Remote Service Admin Specification* - The Remote Services Admin Service Specification defines an API for the distribution provider and discovery of services in a network. A management agent can use this API to provide an actual distribution policy. This management agent can export and import services as well as discovering services in the network. See *Remote Service Admin Service Specification* on page 375.

1.1.3 Web Applications and HTTP Servlets

Current Enterprise Java architectures almost always require support for web technologies in the form of Java Servlets or Web Applications. The Enterprise Specification includes three complementary service specifications in support of web technologies.

- *Web Application Specification* - The Web Application specification provides support for web applications written to the Servlet 2.5 specification as well as the JSP 2.1 specification. This specification details how web applications packaged as a WAR or as bundles (WABs) can be installed into an OSGi framework, as well as how this application can use OSGi services. See *Web Applications Specification* on page 535.
- *Http Service Specification* - 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 and registering resources. See *Http Service Specification* on page 45.
- *Http Whiteboard Specification* - The [1] *Whiteboard Pattern* pattern has shown to be a powerful and flexible mechanism for registering customized functionality with a container. The Http Whiteboard Specification allows the registration of Servlets, Servlet Filters, Resources and Servlet-related listeners via the Whiteboard pattern, providing a convenient approach to working with servlets to the Http Service Specification. See *Http Whiteboard Specification* on page 755.

1.1.4 Asynchronous Processing and Event models

A number of specifications focus specifically on asynchronous programming and executing of components, as well as the sending and receiving of events.

- *Event Admin Service Specification* - The Event Admin service provides an inter-bundle communication mechanism. It is based on a event publish and subscribe model, popular in many message based systems. See *Event Admin Service Specification* on page 253.

- *Asynchronous Services* - Asynchronous processing can be the key to scalability for large enterprise applications, especially under heavy load. OSGi Services have traditionally followed the Java interface-based design which by default provides synchronous semantics. The Asynchronous Services specification adds an asynchronous programming model to new and existing OSGi services. See *Asynchronous Service Specification* on page 741.
- *Promises* - Many JavaScript applications use Promises-based APIs to facilitate asynchronous processing of a workflow in which executions are time consuming or subject to waiting for I/O operations. The OSGi Promises specification defines a Promises API for use in OSGi applications. The Promises API is used by other specifications, such as the Asynchronous Services specification, but can also be used independently.

1.1.5 Management and Configuration services

Support for managing the servers and their applications is essential to all enterprise systems. The Enterprise Specification includes several services addressing the need to manage the framework from the outside as well as configuring individual bundles and applications from within the OSGi framework.

- *JMX™ Management Model Specification* - The Java Management Extensions (JMX) is the standard API specification for providing a management interface to Java SE and Java EE applications. The JMX Management Model specification provides an MBean interface adaptation of the existing OSGi framework artifacts; these can then be used to expose an OSGi Framework manipulation API over JMX. See *JMX™ Management Model Specification* on page 433.
- *REST Management Service Specification* - REST is a powerful paradigm for accessing resources over a network and is widely used in Enterprise and Cloud settings as a protocol of choice, especially since it generally avoids problems with internet firewalls, from which other protocols may suffer. The REST Management Service Specification provides an API to manage and control an OSGi framework using REST operations. See *REST Management Service Specification* on page 709.
- *User Admin Service Specification* - The User Admin Service Specification provides authorization for OSGi framework actions based on authenticated users, instead of using the Java code-based permission model. See *User Admin Service Specification* on page 137.
- *Initial Provisioning Specification* - This specification defines how the Management Agent can make its way into the OSGi framework, and gives a structured view of the problems and their corresponding resolution methods. The purpose of this specification is enable the management of a OSGi framework by an operator, and (optionally) to hand over the management of the OSGi framework later to another operator. See *Initial Provisioning Specification* on page 157.
- *Configuration Admin Service Specification* - The Configuration Admin service allows an operator to set the configuration information of bundles. See *Configuration Admin Service Specification* on page 61.
- *Metatype Service Specification* - The Metatype specification defines interfaces that allow bundle developers to describe attribute types in a computer readable form using metadata. It is mostly used in conjunction with the Configuration Admin Service. See *Metatype Service Specification* on page 105.

1.1.6 Naming and Directory services

Naming and directory services are well established and useful tools in enterprise applications. The Enterprise Specification includes the:

- *JNDI Services Specification* - The Java Naming and Directory Interface (JNDI) is a registry technology in Java applications, both in the Java SE and Java EE space. JNDI provides a vendor-neutral set of APIs that allow clients to interact with a naming service. See *JNDI Services Specification* on page 497.

1.1.7   Database Access

There are multiple approaches available to model and persist data in databases. The Enterprise Specification includes support for the common technologies:

- *JDBC™ Service Specification* - provides an API for applications to interact with relational database systems from different vendors. See *JDBC™ Service Specification* on page 489.
- *JPA Service Specification* - The Java Persistence API (JPA) is a specification that sets a standard for persistence in enterprise and non-enterprise JRE™-based environments. The JPA Service Specification defines how bundles may access and use JPA persistence units in applications, as well as how a JPA implementation can become available and be invoked within an OSGi framework. See *JPA Service Specification* on page 517.

1.1.8   Transaction Support

The support for transactions in Java is well defined outside of the OSGi specification. The Enterprise Specification includes the:

- *JTA Transaction Services Specification* - This specification provides the User Transaction, Transaction Manager, and Synchronization Registry services, which are based on their counterparts in the Java EE™ JTA Specifications. These services can be used to demarcate transaction boundaries, enlists durable and volatile resources, and provides transactional aware code to influence the outcome of a transaction and synchronize with the ending of a transaction. See *JTA Transaction Services Specification* on page 423.

1.1.9   Miscellaneous Supporting Services

Services providing solutions to common infrastructure requirements include:

- *Log Service Specification* - Provides a general purpose message logger for the OSGi framework. See *Log Service Specification* on page 33.
- *XML Parser Service Specification* - Addresses how the classes defined in JAXP can be used in an OSGi framework. See *XML Parser Service Specification* on page 795.
- *Service Loader Mediator Specification* - Addresses common problems of bundles that rely on the JRE provided Service Loader API to load custom Service Provider Implementations. This specification describes how to use the service registry for lookup of Service Providers as well as a solution for existing code to continue functioning using Service Loader API in a OSGi environment. See *Service Loader Mediator Specification* on page 599.
- *Coordinator Specification* - The Coordinator service provides a mechanism for multiple parties to collaborate on a common task without a priori knowledge of who will collaborate in that task. The service provides a rendezvous for an initiator to create a Coordination where collaborators can decide to participate. When the Coordination is ended, all participants are informed. See *Coordinator Service Specification* on page 549.

1.2   Application and Provisioning Support

The support for applications in OSGi environments has long been missing. The term 'application' may mean different things to different people, therefore, rather than defining what an application is, the OSGi specification provides a set of enabling services and specifications to aid in the definition, composition, deployment, and governance of a group of bundles and resources in an OSGi environment. The specifications are essential building blocks and provide from low level resolution to higher level composition abstractions that a management agent can use to build the necessary tools for managing OSGi applications.

- *Resolver Hook Service Specification* - An enabling technology defined by *OSGi Core Release 6* for OSGi frameworks to support isolation of bundles.
- *Resolver Specification* - A service that can be the base for provisioning, deployment, build, and diagnostic tooling. Based on the generic Requirement/Capability model from *OSGi Core Release 6* a management agent can use the Resolver service to compute the set of necessary resources needed to satisfy the given set of requirements. These requirements could represent the dependencies of a core set of bundles that need to be deployed. While the framework already hosts a Resolver for Bundle wiring, this specification makes a generic Resolver available as a service to resolve dependencies on resources beyond Bundle wiring. See *Resolver Service Specification* on page 691.
- *Repository Specification* - The Repository specification provides a standard API to access (possibly remote) repositories. Resources can be obtained from the repository by specifying declarative requirements, which might include for example 'provide all resources that export a given package' or 'provide the bundle with the given symbolic name and version', but can also be used with any other generic capabilities. While the Repository API can be used on its own, in conjunction with the Resolver Specification it provides the capability to manage retrieval of external resources during the resolution process. Typically one or multiple repositories provide the metadata for the Resolver service to draw the resolution from. A management agent can then use the repositories to apply the result of the resolution. See *Repository Service Specification* on page 577.
- *Subsystems Specification* - The core framework defines the life cycle of bundles and their relationships, but it is missing the ability to define a common life cycle and scoping rules for a set of bundles that are conceptually tied together. The Subsystems Specification provides the ability not only to group multiple bundles into a single manageable entity, but furthermore include arbitrary resources in this grouping. This allows for complete isolation as well as various sharing models of code, services, and resources through a management agent. See *Subsystem Service Specification* on page 613.

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 enterprise 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 Version Information

This document is the Enterprise Specification for the OSGi Enterprise Release 6.

1.4.1 OSGi Core Release 6

This specification is based on the OSGi Core Release 6. This specification can be downloaded from:

`http://www.osgi.org/Specifications/HomePage`

1.4.2 Component Versions

Components in this specification have their own specification version, independent of this speci-fication. The following table summarizes the packages and specification versions for the different subjects.

Table 1.1   Packages and versions

Item	Package(s)	Version
100 *Remote Services*	–	Version 1.0
101 *Log Service Specification*	org.osgi.service.log	Version 1.3
102 *Http Service Specification*	org.osgi.service.http	Version 1.2
104 *Configuration Admin Service Specification*	org.osgi.service.cm	Version 1.5
105 *Metatype Service Specification*	org.osgi.service.metatype	Version 1.3
	org.osgi.service.metatype.annotations	
107 *User Admin Service Specification*	org.osgi.service.useradmin	Version 1.1
110 *Initial Provisioning Specification*	org.osgi.service.provisioning	Version 1.2
112 *Declarative Services Specification*	org.osgi.service.component	Version 1.3
	org.osgi.service.component.annotations	
	org.osgi.service.component.runtime	
	org.osgi.service.component.runtime.dto	
113 *Event Admin Service Specification*	org.osgi.service.event	Version 1.3
121 *Blueprint Container Specification*	org.osgi.service.blueprint.container	Version 1.0
	org.osgi.service.blueprint.reflect	
122 *Remote Service Admin Service Specification*	org.osgi.service.remoteserviceadmin	Version 1.1
123 *JTA Transaction Services Specification*	–	Version 1.0
124 *JMX™ Management Model Specification*	org.osgi.jmx[1]	Version 1.1
125 *JDBC™ Service Specification*	org.osgi.service.jdbc	Version 1.0
126 *JNDI Services Specification*	org.osgi.service.jndi	Version 1.0
127 *JPA Service Specification*	org.osgi.service.jpa	Version 1.0
128 *Web Applications Specification*	–	Version 1.0
130 *Coordinator Service Specification*	org.osgi.service.coordinator	Version 1.0
132 *Repository Service Specification*	org.osgi.service.repository	Version 1.1
133 *Service Loader Mediator Specification*	org.osgi.service.serviceloader	Version 1.0
134 *Subsystem Service Specification*	org.osgi.service.subsystem	Version 1.1

Item	Package(s)	Version
135 *Common Namespaces Specification*	org.osgi.namespace.contract	Version 1.1
	org.osgi.namespace.extender	
	org.osgi.namespace.implementation	
	org.osgi.namespace.service	
136 *Resolver Service Specification*	org.osgi.service.resolver	Version 1.0
137 *REST Management Service Specification*	org.osgi.service.rest	Version 1.0
	org.osgi.service.rest.client	
138 *Asynchronous Service Specification*	org.osgi.service.async	Version 1.0
	org.osgi.service.async.delegate	
140 *Http Whiteboard Specification*	org.osgi.service.http.whiteboard	Version 1.0
	org.osgi.service.http.context	
	org.osgi.service.http.runtime	
	org.osgi.service.http.runtime.dto	
702 *XML Parser Service Specification*	org.osgi.util.xml	Version 1.0
705 *Promises Specification*	org.osgi.util.promise	Version 1.0
	org.osgi.util.function	

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

1.4.3   Note

1. The org.osgi.jmx sub-packages are individually versioned to be aligned with the service they manage.

1.5   References

[1]   *Whiteboard Pattern*
http://www.osgi.org/wiki/uploads/Links/whiteboard.pdf

1.6   Changes

- Updated *Metatype Service Specification* on page 105.
- Updated *Declarative Services Specification* on page 175.
- Updated *Remote Service Admin Service Specification* on page 375.
- Updated *Repository Service Specification* on page 577.
- Updated *Subsystem Service Specification* on page 613.
- Updated *Common Namespaces Specification* on page 681.
- Added *REST Management Service Specification* on page 709.
- Added *Asynchronous Service Specification* on page 741.
- Added *Http Whiteboard Specification* on page 755.
- Added *Promises Specification* on page 805.

100   Remote Services
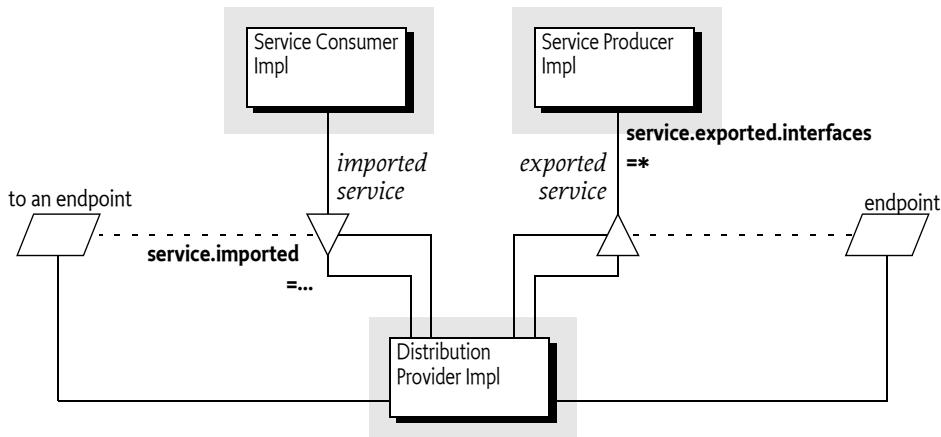
Version 1.0

The OSGi framework provides a *local* service registry for bundles to communicate through service objects, where a service is an object that one bundle registers and another bundle gets. A *distribution provider* can use this loose coupling between bundles to *export* a registered service by creating an *endpoint*. Vice versa, the distribution provider can create a *proxy* that accesses an endpoint and then registers this proxy as an *imported* service. A Framework can contain multiple distribution providers simultaneously, each independently importing and exporting services.

An endpoint is a communications access mechanisms to a service in another framework, a (web) service, another process, or a queue or topic destination, etc., requiring some protocol for communications. The constellation of the mapping between services and endpoints as well as their communication characteristics is called the *topology*. A common case for distribution providers is to be present on multiple frameworks importing and exporting services; effectively distributing the service registry.

The local architecture for remote services is depicted in Figure 100.1 on page 21.

Figure 100.1   *Architecture*

Local services imply in-VM call semantics. Many of these semantics cannot be supported over a communications connection, or require special configuration of the communications connection. It is therefore necessary to define a mechanism for bundles to convey their assumptions and requirements to the distribution provider. This chapter defines a number of service properties that a distribution provider can use to establish a topology while adhering to the given constraints.

100.1   The Fallacies

General abstractions for distributed systems have been tried before and often failed. Well known are the fallacies described in [1] *The Fallacies of Distributed Computing Explained*:

• The network is reliable

- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- There is one administrator
- Transport cost is zero
- The network is homogeneous

Most fallacies represent non-functional trade-offs that should be considered by administrators, their decisions can then be reflected in the topology. For example, in certain cases limited bandwidth is acceptable and the latency in a datacenter is near zero. However, the reliability fallacy is the hardest because it intrudes into the application code. If a communication channel is lost, the application code needs to take specific actions to recover from this failure.

This reliability aspect is also addressed with OSGi services because services are dynamic. Failures in the communications layer can be mapped to the unregistration of the imported service. OSGi bundles are already well aware of these dynamics, and a number of programming models have been developed to minimize the complexity of writing these dynamic applications.

100.2 Remote Service Properties

This section introduces a number of properties that participating bundles can use to convey information to the distribution provider according to this Remote Service specification.

The following table lists the properties that must be listed by a distribution provider.

Table 100.1   *Remote Service Properties registered by the Distribution Provider*

Service Property Name	Type	Description
remote.configs.supported	String+	Registered by the distribution provider on one of its services to indicate the supported configuration types. See *Configuration Types* on page 27 and *Dependencies* on page 30.
remote.intents.supported	String+	Registered by the distribution provider on one of its services to indicate the vocabulary of implemented intents. See *Dependencies* on page 30.
service.imported	*	Must be set by a distribution provider to any value when it registers the endpoint proxy as an imported service. A bundle can use this property to filter out imported services.
service.imported.configs	String+	The configuration information used to import this service, as described in service.exported.configs. Any associated properties for this configuration types must be properly mapped to the importing system. For example, a URL in these properties must point to a valid resource when used in the importing framework. If multiple configuration types are listed in this property, then they must be synonyms for exactly the same remote endpoint that is used to export this service.

Service Property Name	Type	Description
service.intents	String+	A distribution provider must use this property to convey the combined intents of: • The exporting service, and • The intents that the exporting distribution provider adds. • The intents that the importing distribution provider adds.

The properties for bundles providing services to be exported or require services to be imported are listed alphabetically in the following table. The scenarios that these properties are used in are discussed in later sections.
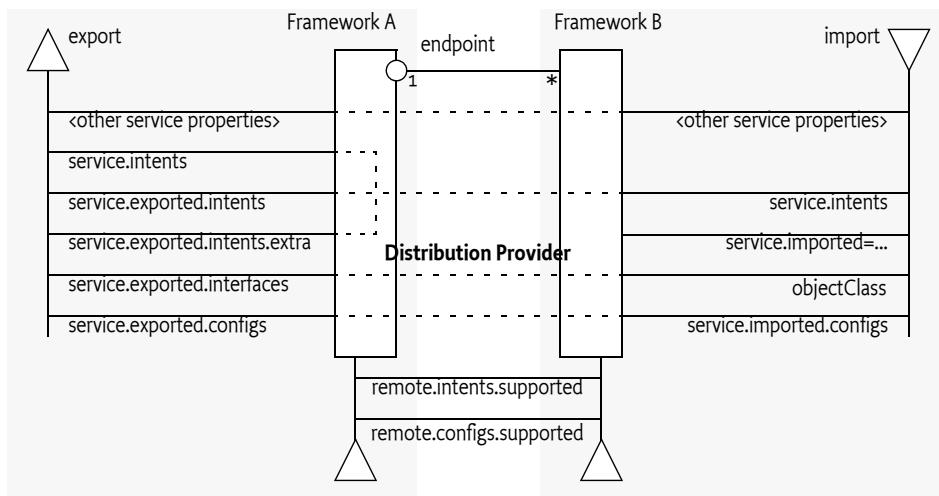
Table 100.2   *Remote Service Properties registered by Exporting bundles*

Service Property Name	Type	Description
service.exported.configs	String+	A list of configuration types that should be used to export the service. Each configuration type represents the configuration parameters for one or more Endpoints. A distribution provider should create endpoints for each configuration type that it supports. See *Configuration Types* on page 27 for more details. If this property is not set or empty a distribution provider is free to choose a default configuration type for the service.
service.exported.intents	String+	A list of *intents* that the distribution provider must implement to distribute the service. Intents listed in this property are reserved for intents that are critical for the code to function correctly, for example, ordering of messages. These intents should not be configurable. For more information about intents, see *Intents* on page 26. This property is optional.
service.exported.intents.extra	String+	This property is merged with the service.exported.intents property before the distribution provider interprets the listed intents; it has therefore the same semantics but the property should be configurable so the administrator can choose the intents based on the topology. Bundles should therefore make this property configurable, for example through the Configuration Admin service. See *Intents* on page 26. This property is optional. If absent or empty no specific intents are required.
service.exported.interfaces	String+	Setting this property marks this service for export. It defines the interfaces under which this service can be exported. This list must be a subset of the types listed in the objectClass service property. The single value of an asterisk ('*' \u002A) indicates all interfaces in the registration's objectClass property and ignore the classes. It is strongly recommended to only export interfaces and not concrete classes due to the complexity of creating proxies for some type of concrete classes. See *Registering a Service for Export* on page 24.

Service Property Name	Type	Description
service.intents	String+	A list of intents that this service implements. A distribution provider must use this property to convey the combined intents of:
		• The exporting service, and
		• The intents that the exporting distribution provider adds.
		• The intents that the importing distribution provider adds.
		To export a service, a distribution provider must expand any qualified intents to include those supported by the endpoint. This can be a subset of all known qualified intents. See *Intents* on page 26. This property is optional for registering bundles.
service.pid	String+	Services that are exported should have a service.pid property. The service.pid (PID) is a unique persistent identity for the service, the PID is defined in *Persistent Identifier (PID)* of *OSGi Core Release 6*. This property enables a distribution provider to associate persistent proprietary data with a service registration.

The properties and their treatment by the distribution provider is depicted in Figure 100.2.

Figure 100.2   Distribution Service Properties

100.2.1   Registering a Service for Export

A distribution provider should create one or more endpoints for an exported service when the following conditions are met:

- The service has the service property service.exported.interfaces set.
- All intents listed in service.exported.intents, service.exported.intents.extra and service.intents are part of the distributed provider's vocabulary
- None of the intents are mutually exclusive.

- The distribution provider can use the configuration types in `service.exported.configs` to create one or more endpoints.

The endpoint must at least implement all the intents that are listed in the `service.exported.intents` and `service.exported.intents.extra` properties.

The configuration types listed in the `service.exported.configs` can contain *alternatives* and/or *synonyms*. Alternatives describe different endpoints for the same service while a synonym describes a different configuration type for the same endpoint.

A distribution provider should create endpoints for each of the configuration types it supports; these configuration types should be alternatives. Synonyms are allowed.

If no configuration types are recognized, the distribution provider should create an endpoint with a default configuration type except when one of the listed configuration types is `<<nodefault>>`.

For more information about the configuration types, see further *Configuration Types* on page 27.

100.2.2 Getting an Imported Service

An imported service must be a normal service, there are therefore no special rules for getting it. An imported service has a number of additional properties that must be set by the distribution provider.

If the endpoint for an exported service is imported as an OSGi service in another framework, then the following properties must be treated as special.

- `service.imported` - Must be set to some value.
- `service.intents` - This must be the combination of the following:
  - The `service.intents` property on the exported service
  - The `service.exported.intents` and `service.exported.intents.extra` properties on the exported service
  - Any additional intents implemented by the distribution providers on both sides.
- `service.imported.configs` - Contains the configuration types that can be used to import this service. The types listed in this property must be *synonymous*, that is, they must refer to exactly the same endpoint that is exporting the service. See *Configuration Types* on page 27.
- `service.exported.*` - Properties starting with `service.exported.` must not be set on the imported service.
- `service.exported.interfaces` - This property must not be set, its content is reflected in the `objectClass` property.

All other *public* service properties (not starting with a full stop ('.' \u002E)) must be listed on the imported service if they use the basic service property types. If the service property cannot be communicated because, for example, it uses a type that can not be marshaled by the distribution provider then the distribution provider must ignore this property.

The `service.imported` property indicates that a service is an imported service. If this service property is set to any value, then the imported service is a proxy for an endpoint. If a bundle wants to filter out imported services, then it can add the following filter:

```
(&(!(service.imported=*)) <previousfilter>)
```

Distribution providers can also use the *Service Hook Service Specification* of *OSGi Core Release 6* to hide services from specific bundles.

100.2.3 On Demand Import

The Service Hooks Service Specification of *OSGi Core Release 6*, allows a distribution provider to detect when a bundle is listening for specific services. Bundles can request imported services with spe-

cific intents by building an appropriate filter. The distribution provider can use this information to import a service on demand.

The following example creates a Service Tracker that is interested in an imported service.

```
Filter f = context.createFilter(
        "(&(objectClasss=com.acme.Foo)"
    +   "(service.intents=confidentiality))"
);
ServiceTracker tracker =
    new ServiceTracker(context, f, null );
tracker.open();
```

Such a Service Tracker will inform the Listener Hook and will give it the filter expression. If the distribution provider has registered such a hook, it will be informed about the need for an imported com.acme.Foo service that has a confidentiality intent. It can then use some proprietary means to find a service to import that matches the given object class and intent.

How the distribution provider finds an appropriate endpoint is out of scope for this specification.

100.3 Intents

An intent is a name for an abstract distribution capability. An intent can be *implemented* by a service; this can then be reflected in the service.intents property. An intent can also *constrain* the possible communication mechanisms that a distribution provider can choose to distribute a service. This is reflected in the service.export.intents and service.exported.intents.extra properties.

The purpose of the intents is to have a *vocabulary* that is shared between distribution aware bundles and the distribution provider. This vocabulary allows the bundles to express constraints on the export of their services as well as providing information on what intents are implemented by a service.

Intents have the following syntax

```
intent  ::= token ( '.' token )?
```

Qualified intents use a full stop ('.' \u002E) to separate the intent from the qualifier. A qualifier provides additional details, however, it implies its prefix. For example:

```
confidentiality.message
```

This example, can be *expanded* into confidentiality and confidentiality.message. Qualified intents can be used to provide additional details how an intent is achieved. However, a Distribution Provider must expand any qualified intents to include those supported by the endpoint. This can be a subset of all known qualified intents.

The concept of intents is derived from the [3] *SCA Policy Framework specification*. When designing a vocabulary for a distribution provider it is recommended to closely follow the vocabulary of intents defined in the SCA Policy Framework.

100.4 General Usage

100.4.1 Call by Value

Normal service semantics are call-by-reference. An object passed as an argument in a service call is a direct reference to that object. Any changes to this object will be shared on both sides of the service registry.

Distributed services are different. Arguments are normally passed by value, which means that a copy is sent to the remote system, changes to this value are not reflected in the originating framework. When using distributed services, call-by-value should always be assumed by all participants in the distribution chain.

100.4.2 Data Fencing

Services are syntactically defined by their Java interfaces. When exposing a service over a remote protocol, typically such an interface is mapped to a protocol-specific interface definition. For example, in CORBA the Java interfaces would be converted to a corresponding IDL definition. This mapping does not always result in a complete solution.

Therefore, for many practical distributed applications it will be necessary to constrain the possible usage of data types in service interfaces. A distribution provider must at least support interfaces (not classes) that only use the basic types as defined for the service properties. These are the primitive types and their wrappers as well as arrays and collections. See *Filter Syntax* of *OSGi Core Release 6* for a list of service property types.

Distribution providers will in general provide a richer set of types that can be distributed.

100.4.3 Remote Services Life Cycle

A distributed service must closely track any modifications on the corresponding service registration. If service properties are modified, these modifications should be propagated to the distributed service and associated service proxies. If the exported service is unregistered, the endpoint must be withdrawn as soon as possible and any imported service proxies unregistered.

100.4.4 Runtime

An imported service is just like any other service and can be used as such. However, certain non-functional characteristics of this service can differ significantly from what is normal for an in-VM object call. Many of these characteristics can be mapped to the normal service operations. That is, if the connection fails in any way, the service can be unregistered. According to the standard OSGi contract, this means that the users of that service must perform the appropriate cleanup to prevent stale references.

100.4.5 Exceptions

It is impossible to guarantee that a service is not used when it is no longer valid. Even with the synchronous callbacks from the Service Listeners, there is always a finite window where a service can be used while the underlying implementation has failed. In a distributed environment, this window can actually be quite large for an imported service.

Such failure situations must be exposed to the application code that uses a failing imported service. In these occasions, the distribution provider must notify the application by throwing a Service Exception, or subclass thereof, with the reason REMOTE. The Service Exception is a Runtime Exception, it can be handled higher up in the call chain. The cause of this Service Exception must be the Exception that caused the problem.

A distribution provider should log any problems with the communications layer to the Log Service, if available.

100.5 Configuration Types

An exported service can have a `service.exported.configs` service property. This property lists configuration types for endpoints that are provided for this service. Each type provides a specification that defines how the configuration data for one or more endpoints is provided. For example, a hypothetical configuration type could use a service property to hold a URL for the RMI naming registry.

Configuration types that are not defined by the OSGi Alliance should use a name that follows the reverse capabilities domain name scheme defined in [4] *Java Language Specification* for Java packages. For example, com.acme.wsdl would be the proprietary way for the ACME company to specify a WSDL configuration type.

100.5.1 Configuration Type Properties

The service.exported.configs and service.imported.configs use the configuration types in very different ways. That is, the service.imported.configs property is not a copy of the service.exported.configs as the name might seem to imply.

An exporting service can list its desired configuration types in the service.exported.configs property. This property is potentially seen and interpreted by multiple distribution providers. Each of these providers can independently create endpoints from the configuration types. In principle, the service.exported.configs lists *alternatives* for a single distribution provider and can list *synonyms* to support alternative distribution providers. If only one of the synonyms is useful, there is an implicit assumption that when the service is exported, only one of the synonyms should be supported by the installed distribution providers. If it is detected that this assumption is violated, then an error should be logged and the conflicting configuration is further ignored.

The interplay of synonyms and alternatives is depicted in Table 100.3. In this table, the first columns on the left list different combinations of the configuration types in the service.exported.configs property. The next two columns list two distribution providers that each support an overlapping set of configuration types. The x's in this table indicate if a configuration type or distribution provider is active in a line. The description then outlines the issues, if any. It is assumed in this table that hypothetical configuration types net.rmi and com.rmix map to an identical endpoint, just like net.soap and net.soapx.

Table 100.3   Synonyms and Alternatives in Exported Configurations

service.exported.configs					Distribution Provider A	Distribution Provider B	Description
net.rmi	com.rmix	net.soap	com.soapx	<<no default>>	Supports: net.rmi com.rmix com.soapx	Supports: net.rmi net.soap	
x		x			x		*Ok*, A will create an endpoint for the RMI and SOAP alternatives.
x					x	x	*Configuration error*. There is a clash for net.rmi because A and B can both create an endpoint for the same configuration. It is likely that one will fail.
		x	x		x		*Ok*, exported on com.soapx by A, the net.soap is ignored.
		x	x		x	x	*Synonym error* because A and B export to same SOAP endpoint, it is likely that one will fail.
	x	x			x	x	*Ok*, two alternative endpoints over RMI (by A) and SOAP (by B) are created. This is a typical use case.
		x	x		x		*Ok*. Synonyms are used to allow frameworks that have either A or B installed. In this case A exports over SOAP.
		x	x			x	*Ok*. Synonyms are used to allow frameworks that have either A or B installed. In this case B exports.
					x		*Ok*. A creates an endpoint with default configuration type.

service.exported. configs		Distribution Provider A	Distribution Provider B	Description
		x	x	*Ok*. Both A and B each create an endpoint with their default configuration type.
	x	x		*Ok*. No endpoint is created.
x	x		x	Provider B does not recognize the configuration types it should therefore use a default configuration type.

To summarize, the following rules apply for a single distribution provider:

- Only configuration types that are supported by this distribution provider must be used. All other configuration types must be ignored.
- All of the supported configuration types must be *alternatives*, that is, they must map to different endpoints. Synonyms for the same distribution provider should be logged as errors.
- If a configuration type results in an endpoint that is already in use, then an error should be logged. It is likely then that another distribution provider already had created that endpoint.

An export of a service can therefore result in multiple endpoints being created. For example, a service can be exported over RMI as well as SOAP. Creating an endpoint can fail, in that case the distribution provider must log this information in the Log Service, if available, and not export the service to that endpoint. Such a failure can, for example, occur when two configuration types are synonym and multiple distribution providers are installed that supporting this type.

On the importing side, the `service.imported.configs` property lists configuration types that must refer to the same endpoint. That is, it can list alternative configuration types for this endpoint but all configuration types must result in the same endpoint.
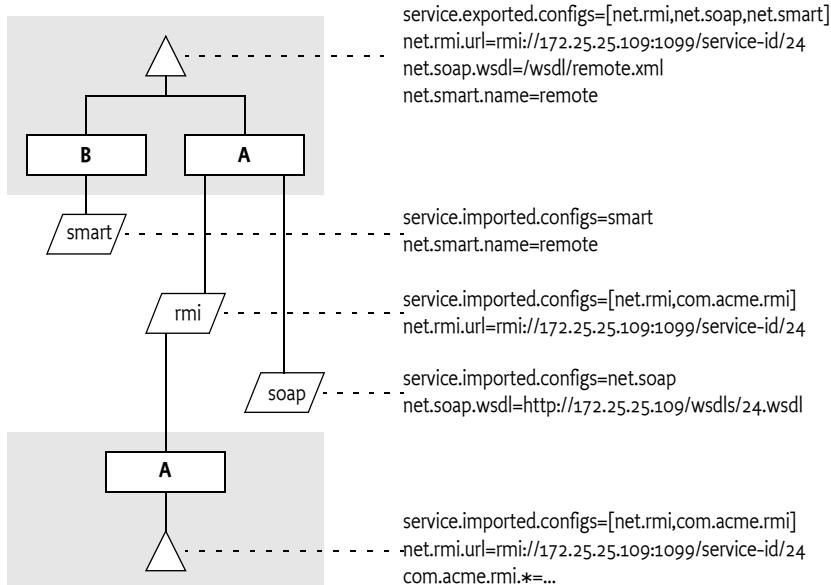
For example, there are two distribution providers installed at the exporting and importing frameworks. Distribution provider A supports the hypothetical configuration type `net.rmi` and `net.soap`. Distribution provider B supports the hypothetical configuration type `net.smart`. A service is registered that list all three of those configuration types.

Distribution provider A will create two endpoints, one for RMI and one for SOAP. Distribution provider B will create one endpoint for the smart protocol. The distribution provider A knows how to create the configuration data for the `com.acme.rmi` configuration type as well and can therefore create a synonymous description of the endpoint in that configuration type. It will therefore set the imported configuration type for the RMI endpoint to:

```
service.imported.configs = net.rmi, com.acme.rmi
net.rmi.url = rmi://172.25.25.109:1099/service-id/24
com.acme.rmi.address = 172.25.25.109
com.acme.rmi.port = 1099
com.acme.rmi.path = service-id/24
```

service.exported.configs=[net.rmi,net.soap,net.smart]
net.rmi.url=rmi://172.25.25.109:1099/service-id/24
net.soap.wsdl=/wsdl/remote.xml
net.smart.name=remote

service.imported.configs=smart
net.smart.name=remote

service.imported.configs=[net.rmi,com.acme.rmi]
net.rmi.url=rmi://172.25.25.109:1099/service-id/24

service.imported.configs=net.soap
net.soap.wsdl=http://172.25.25.109/wsdls/24.wsdl

service.imported.configs=[net.rmi,com.acme.rmi]
net.rmi.url=rmi://172.25.25.109:1099/service-id/24
com.acme.rmi.*=...

100.5.2   Dependencies

A bundle that uses a configuration type has an implicit dependency on the distribution provider. To make this dependency explicit, the distribution provider must register a service with the following properties:

- `remote.intents.supported` - (String+) The vocabulary of the given distribution provider.
- `remote.configs.supported` - (String+) The configuration types that are implemented by the distribution provider.

A bundle that depends on the availability of specific intents or configuration types can create a service dependency on an anonymous service with the given properties. The following filter is an example of depending on a hypothetical `net.rmi` configuration type:

`(remote.configs.supported=net.rmi)`

100.6   Security

The distribution provider will be required to invoke methods on any exported service. This implies that it must have the combined set of permissions of all methods it can call. It also implies that the distribution provider is responsible for ensuring that a bundle that calls an imported service is not granted additional permissions through the fact that the distribution provider will call the exported service, not the original invoker.

The actual mechanism to ensure that bundles can get additional permissions through the distribution is out of scope for this specification. However, distribution providers should provide mechanisms to limit the set of available permissions for a remote invocation, preferably on a small granularity basis.

One possible means is to use the `getAccessControlContext` method on the Conditional Permission Admin service to get an Access Control Context that is used in a `doPrivileged` block where the invocation takes place. The `getAccessControlContext` method takes a list of signers which could repre-

sent the remote bundles that cause an invocation. How these are authenticated is up to the distribution provider.

A distribution provider is a potential attack point for intruders. Great care should be taken to properly setup the permissions or topology in an environment that requires security.

100.6.1 Limiting Exports and Imports

Service registration and getting services is controlled through the ServicePermission class. This permission supports a filter based constructor that can assert service properties. This facility can be used to limit bundles from being able to register exported services or get imported services if they are combined with Conditional Permission Admin's ALLOW facility. The following example shows how all bundles except from www.acme.com are denied the registration and getting of distributed services.

```
DENY {
   [...BundleLocationCondition("http://www.acme.com/*" "!")]
   (...ServicePermission "(service.imported=*)" "GET" )
   (...ServicePermission "(service.exported.interfaces=*)"
                              "REGISTER" )
}
```

100.7 References

[1] *The Fallacies of Distributed Computing Explained*
     http://www.rgoarchitects.com/Files/fallacies.pdf

[2] *Service Component Architecture (SCA)*
     http://www.oasis-opencsa.org/

[3] *SCA Policy Framework specification*
     http://www.oasis-open.org/committees/sca-policy/

[4] *Java Language Specification*
     http://docs.oracle.com/javase/specs/

101   Log Service Specification

Version 1.3

101.1   Introduction

The Log Service provides a general purpose message logger for the OSGi framework. It consists of two services, one for logging information and another for retrieving current or previously recorded log information.
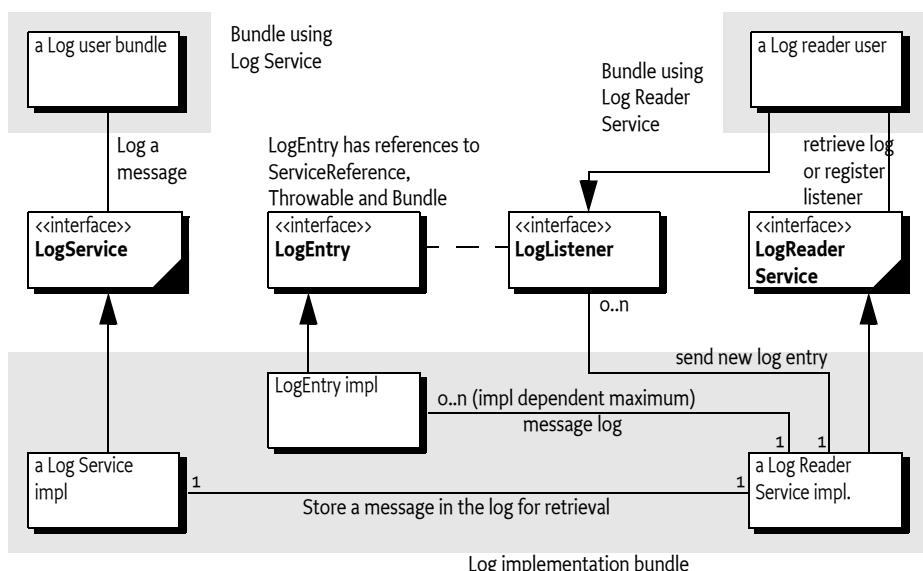
This specification defines the methods and semantics of interfaces which bundle developers can use to log entries and to retrieve log entries.

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

101.1.1   Entities

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

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

101.2   The Log Service Interface

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

The `LogService` interface allows the bundle developer to:

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

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

The `LogService` interface defines these methods for logging messages:

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

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

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

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

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

The following example code records error conditions as log messages.

```
try {
    FileInputStream fis = new FileInputStream("myFile");
    int b;
    while ( (b = fis.read()) != -1 ) {
      ...
    }
    fis.close();
}
catch ( IOException exception ) {
    logService.log(
        myServiceReference,
```

```
            LogService.LOG_ERROR,
            "Cannot access file",
            exception );
}
```

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

101.3   Log Level and Error Severity

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

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

The following table lists the log levels.

Table 101.1   *Log Levels*

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

101.4   Log Reader Service

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

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

The LogReaderService interface defines these methods for retrieving log entries.

- getLog() - This method retrieves past log entries as an enumeration with the most recent entry first.
- addLogListener(LogListener) - This method is used to subscribe to the Log Reader Service in order to receive log messages as they occur. Unlike the previously recorded log entries, all log messages must be sent to subscribers of the Log Reader Service as they are recorded.

  A subscriber to the Log Reader Service must implement the LogListener interface.

  After a subscription to the Log Reader Service has been started, the subscriber's LogListener.logged method must be called with a Log-Entry 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 `Log-Entry` object created. A Log Reader Service implementation must not filter entries to the `LogListener` interface as it is allowed to do for its log. A `LogListener` object should see all `LogEntry` objects that are created.

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

101.5 Log Entry Interface

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

- `getBundle()` - This method returns the Bundle object related to a `Log-Entry` object.
- `getException()` - This method returns the exception related to a `Log-Entry` 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 `Log-Entry` object.
- `getMessage()` - This method returns the message related to a `Log-Entry` object.
- `getServiceReference()` - This method returns the `ServiceReference` object of the service related to a `Log-Entry` object.
- `getTime()` - This method returns the time that the log entry was created.

101.6 Mapping of Events

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

101.6.1 Bundle Events Mapping

A Bundle Event is mapped to a `LogEntry` object according to the following table.

Table 101.2   *Mapping of Bundle Events to Log Entries*

Log Entry method	Information about Bundle Event
`getLevel()`	LOG_INFO
`getBundle()`	Identifies the bundle to which the event happened. In other words, it identifies the bundle that was installed, started, stopped, updated, or uninstalled. This identification is obtained by calling `getBundle()` on the `BundleEvent` object.
`getException()`	null
`getServiceReference()`	null

Log Entry method	Information about Bundle Event
getMessage()	The message depends on the event type:

- INSTALLED - "BundleEvent INSTALLED"
- STARTED - "BundleEvent STARTED"
- STOPPED - "BundleEvent STOPPED"
- UPDATED - "BundleEvent UPDATED"
- UNINSTALLED - "BundleEvent UNINSTALLED"
- RESOLVED - "BundleEvent RESOLVED"
- UNRESOLVED - "BundleEvent UNRESOLVED"

101.6.2   Service Events Mapping

A Service Event is mapped to a LogEntry object according to the following table.

Table 101.3   *Mapping of Service Events to Log Entries*

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

- REGISTERED - "ServiceEvent REGISTERED"
- MODIFIED - "ServiceEvent MODIFIED"
- UNREGISTERING - "ServiceEvent UNREGISTERING"

101.6.3   Framework Events Mapping

A Framework Event is mapped to a LogEntry object according to the following table.

Table 101.4   *Mapping of Framework Event to Log Entries*

Log Entry method	Information about Framework Event
getLevel()	LOG_INFO, except for the FrameworkEvent.ERROR event. This event represents an error and is logged with a level of LOG_ERROR.
getBundle()	Identifies the bundle associated with the event. This may be the system bundle. It is obtained by calling getBundle() on the FrameworkEvent object.
getException()	Identifies the exception associated with the error. This will be null for event types other than ERROR. It is obtained by calling getThrowable() on the FrameworkEvent object.
getServiceReference()	null

Log Entry method	Information about Framework Event
getMessage()	This message depends on the actual event type. The messages are mapped as follows:

- STARTED - "FrameworkEvent STARTED"
- ERROR - "FrameworkEvent ERROR"
- PACKAGES_REFRESHED - "FrameworkEvent PACKAGES REFRESHED"
- STARTLEVEL_CHANGED - "FrameworkEvent STARTLEVEL CHANGED"
- WARNING - "FrameworkEvent WARNING"
- INFO - "FrameworkEvent INFO"

101.6.4 Log Events

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

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

The logging level is used as event type:

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

The properties of a log event are:

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

If the log entry has an associated Exception:

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

If the getServiceReference method returns a non- null value:

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

101.7   Security

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

101.8   org.osgi.service.log

Log Service Package Version 1.3.

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

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

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

101.8.1   Summary

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

101.8.2   public interface LogEntry

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

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

See Also   LogReaderService.getLog, LogListener

Concurrency   Thread-safe

No Implement   Consumers of this API must not implement this interface

101.8.2.1   **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 ob-
ject.

101.8.2.2   **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 ref-
erences to a bundle defined exception class, thus preventing an uninstalled bundle from being
garbage collected, the Log Service may return an exception object of an implementation defined
Throwable subclass. The returned object will attempt to provide as much information as possible
from the original exception object such as the message and stack trace.

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

101.8.2.3 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, LogService.LOG_WARNING, LogService.LOG_INFO, LogService.LOG_DEBUG

101.8.2.4 public String getMessage()

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

Returns String containing the message associated with this `LogEntry` object.

101.8.2.5 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.

101.8.2.6 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()

101.8.3 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, LogEntry, LogReaderService.addLogListener(LogListener), LogReaderService.removeLogListener(LogListener)

Concurrency Thread-safe

101.8.3.1 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

101.8.4 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, LogListener, LogListener.logged(LogEntry)

Concurrency Thread-safe

101.8.4.1 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, LogEntry, LogListener.logged(LogEntry)

101.8.4.2 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.

Returns An Enumeration of all LogEntry objects in the log.

101.8.4.3 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

101.8.5 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
2. LOG_WARNING
3. LOG_INFO
4. LOG_DEBUG

Concurrency Thread-safe

No Implement Consumers of this API must not implement this interface

101.8.5.1   **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.

101.8.5.2   **public static final int LOG_ERROR = 1**

An error message (Value 1).

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

101.8.5.3   **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.

101.8.5.4   **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.

101.8.5.5   **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, LOG_WARNING, LOG_INFO, LOG_DEBUG

101.8.5.6   **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, LOG_WARNING, LOG_INFO, LOG_DEBUG

101.8.5.7   **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, LOG_WARNING, LOG_INFO, LOG_DEBUG

101.8.5.8 **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, LOG_WARNING, LOG_INFO, LOG_DEBUG

102   Http Service Specification

Version 1.2

102.1   Introduction

An OSGi framework 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 framework using a standard web browser.

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

The Http Service supports two standard techniques for this purpose:

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

Implementations of the Http Service can be based on:

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

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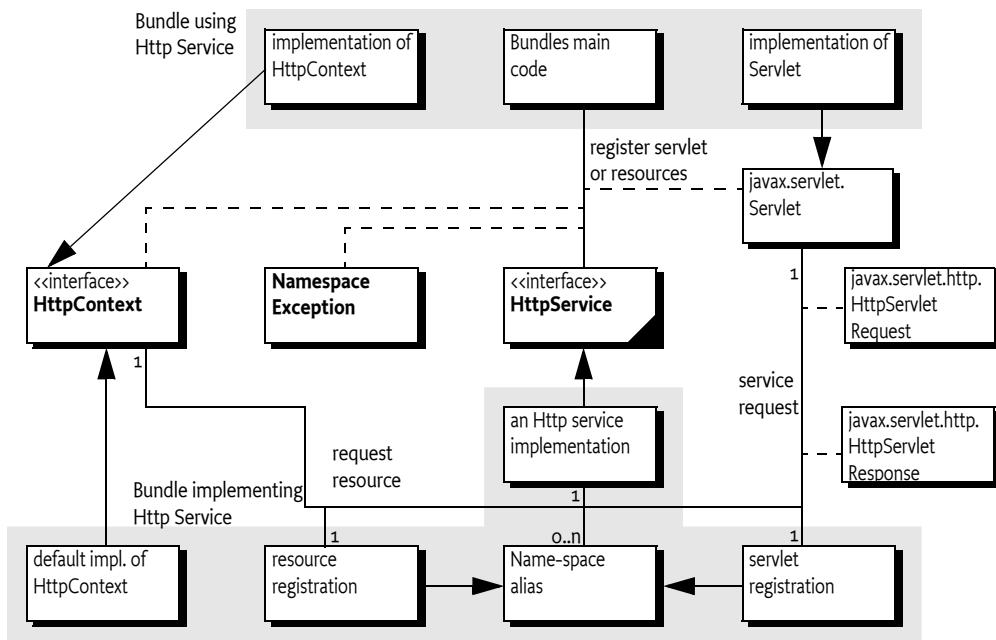
102.1.1   Entities

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

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

Figure 102.1   *Http Service Overview Diagram*

102.2   Registering Servlets

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

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

http: //www. acme. com/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 49 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 52.

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

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

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

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

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

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

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

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

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

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

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

102.3   Registering Resources

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

Resources could be handled by Servlet objects as explained in *Registering Servlets* on page 46. Transferring a resource over HTTP, however, would require very similar Servlet objects for each bundle. To prevent this redundancy, resources can be registered directly with the Http Service via the HttpService interface. This HttpService interface defines the registerResources(String,String,HttpContext) method for registering a resource into the Http Service URI name-space.

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

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

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

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

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

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

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

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

```
getHttpService().unregister("/files");
```

This example registers the alias /files on the Http Service. Requests for resources below this name-space are transferred to the HttpContext object with an internal name of www/<name>. This example uses the Class.get Resource(String) method. Because the internal name does not start with a "/", it must map to a resource in the "com/acme/www" directory of the JAR file. If the internal name did start with a "/", the package name would not have to be prefixed and the JAR file would be searched from the root. Consult the java.lang.Class.getResource(String) method for more information.

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

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

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

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

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

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

102.4 Mapping HTTP Requests to Servlet and Resource Registrations

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

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

1. If the registration corresponds to a servlet, the authorization is verified by calling the handleSe-curity method of the associated HttpContext object. See *Authentication* on page 52. 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 han-dleSecurity method of the associated HttpContext object. See *Authentication* on page 52. If the request is authorized, a target resource name is constructed from the requested URI by sub-stituting 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 request-

ed URI with the internal name. An internal name of "/" is considered to have the value of the empty string ("") during this process.

3. The target resource name must be passed to the getResource method of the associated HttpContext object.

4. If the returned URL object is not null, the Http Service must return the contents of the URL to the client completing the HTTP request. The translated target name, as opposed to the original requested URI, must also be used as the argument to HttpContext.getMimeType.

5. If the returned URL object is null, the Http Service continues as if there was no match.

6. If there is no match, the Http Service must attempt to match sub-strings of the requested URI to registered aliases. The sub-strings of the requested URI are selected by removing the last "/" and everything to the right of the last "/".

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

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

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

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

Table 102.1   *Examples of Name-space Mapping*

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

102.5 The Default Http Context Object

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

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

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

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

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

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

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

102.6 Multipurpose Internet Mail Extension (MIME) Types

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

An important aspect of this extension is the type (file format) mechanism of the binary messages. The type is defined by a string containing a general category (text, application, image, audio and video, multipart, and message) followed by a "/" and a specific media type, as in the example, "`text/html`" for HTML formatted text files. A MIME type string can be followed by additional specifiers by separating `key=value` pairs with a semicolon ('`;`' \u003B). These specifiers can be used, for example, to define character sets as follows:

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

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

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

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

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

Table 102.2   *Sample Extension to MIME Media Mapping*

Extension	MIME media type	Description
.jpg .jpeg	image/jpeg	JPEG Files
.gif	image/gif	GIF Files
.css	text/css	Cascading Style Sheet Files
.txt	text/plain	Text Files

Extension	MIME media type	Description
.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

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

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

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

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

102.7 Authentication

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

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

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

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

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

Using basic authentication requires the following steps:

1. If no Authorization header is set in the request, the method should set the WWW-Authenticate header in the response. This header indicates the desired authentication mechanism and the realm. For example, WWW-Authenticate: Basic realm="ACME".

   The header should be set with the response object that is given as a parameter to the handleSecurity method. The handleSecurity method should set the status to HttpServletResponse.SC_UNAUTHORIZED (401) and return false.
2. Secure connections can be verified with the ServletRequest.getScheme() method. This method returns, for example, "https" for an SSL connection; the handleSecurity method can use this and other information to decide if the connection's security level is acceptable. If not, the handleSecurity method should set the status to HttpServletResponse.SC_FORBIDDEN (403) and return false.

3. Next, the request must be authenticated. When basic authentication is used, the Authorization header is available in the request and should be parsed to find the user and password. See [7] *RFC 2617: HTTP Authentication: Basic and Digest Access Authentication* for more information.

   If the user cannot be authenticated, the status of the response object should be set to HttpServletResponse.SC_UNAUTHORIZED (401) and return false.

4. The authentication mechanism that is actually used and the identity of the authenticated user can be of interest to the Servlet object. Therefore, the implementation of the handleSecurity method should set this information in the request object using the ServletRequest.setAttribute method. This specification has defined a number of OSGi-specific attribute names for this purpose:

   - AUTHENTICATION_TYPE - Specifies the scheme used in authentication. A Servlet may retrieve the value of this attribute by calling the HttpServletRequest.getAuthType method. This attribute name is org.osgi.service.http.authentication.type.

   - REMOTE_USER - Specifies the name of the authenticated user. A Servlet may retrieve the value of this attribute by calling the HttpServletRequest.getRemoteUser method. This attribute name is org.osgi.service.http.authentication.remote.user.

   - AUTHORIZATION - If a User Admin service is available in the environment, then the handleSecurity method should set this attribute with the Authorization object obtained from the User Admin service. Such an object encapsulates the authentication of its remote user. A Servlet may retrieve the value of this attribute by calling ServletRequest.getAttribute(HttpContext.AUTHORIZATION). This header name is org.osgi.service.useradmin.authorization.

5. Once the request is authenticated and any attributes are set, the handleSecurity method should return true. This return indicates to the Http Service that the request is authorized and processing may continue. If the request is for a Servlet, the Http Service must then call the service method on the Servlet object.

102.8 Security

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

102.8.1 Accessing Resources with the Default Http Context

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

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

102.8.2 Accessing Other Types of Resources

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

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

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

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

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

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

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

102.8.3   Servlet and HttpContext objects

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

102.9   Configuration Properties

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

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

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

102.10   org.osgi.service.http

Http Service Package Version 1.2.

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

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

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

102.10.1   Summary

- HttpContext - Context for HTTP Requests.
- HttpService - The Http Service allows other bundles in the OSGi environment to dynamically register resources and servlets into the URI namespace of Http Service.
- NamespaceException - A NamespaceException is thrown to indicate an error with the caller's request to register a servlet or resources into the URI namespace of the Http Service.

102.10.2   public interface HttpContext

Context for HTTP Requests.

This service defines methods that the Http Service may call to get information for a request.

Servlets may be associated with an HttpContext service. Servlets that are associated using the same HttpContext object will share the same ServletContext object.

If no HttpContext service is associated, a default HttpContext is used. 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 bundle of the servlet service. This method calls the servlet bundle's Bundle.getResource method, and returns the appropriate URL to access the resource. On a Java runtime environment that supports permissions, the Http Service needs to be granted org.osgi.framework.AdminPermission[*,RESOURCE].

102.10.2.1   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

102.10.2.2   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

102.10.2.3   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

102.10.2.4   public String getMimeType(String name)

name The name for which to determine the MIME type.

       □ Maps a name to a MIME type.

Called by the Http Service to determine the MIME type for the specified name. For servlets, the Http Service will call this method to support the ServletContext method getMimeType. For resources, the Http Service will call this method to determine the MIME type for the Content-Type header in the response.

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

102.10.2.5 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.

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

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

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

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

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

102.10.3   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

No Implement   Consumers of this API must not implement this interface

102.10.3.1   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 org.osgi.framework.AdminPermission[*,RESOURCE].

Returns   a default HttpContext object.

Since   1.1

102.10.3.2   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 ('/') with the exception that a name of the form "/" is used to denote the root of the bundle. See the specification text for details on how HTTP requests are mapped to servlet and resource registrations.

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

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

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

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

IllegalArgumentException– if any of the parameters are invalid

102.10.3.3   **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, initparams, context);
```

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

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

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

IllegalArgumentException– if any of the arguments are invalid

102.10.3.4   **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(String) then Http Service must automatically unregister the registration. However, if the registration was for a servlet, the destroy method of the servlet will not be called in this case since the bundle may be stopped. unregister(String) must be explicitly called to cause the destroy method of the servlet to be called. This can be done in the BundleActivator.stop method of the bundle registering the servlet.

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

102.10.4   **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.

102.10.4.1   **public NamespaceException(String message)**

message the detail message

☐ Construct a NamespaceException object with a detail message.

102.10.4.2   **public NamespaceException(String message,Throwable cause)**

message The detail message.

cause The nested exception.

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

102.10.4.3   **public Throwable getCause()**

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

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

Since 1.2

102.10.4.4   **public Throwable getException()**

☐ Returns the nested exception.

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

Returns The result of calling getCause().

102.10.4.5   **public Throwable initCause(Throwable cause)**

cause The cause of this exception.

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

Returns This exception.

Throws IllegalArgumentException– If the specified cause is this exception.

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

Since 1.2

102.11   References

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

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

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

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

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

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

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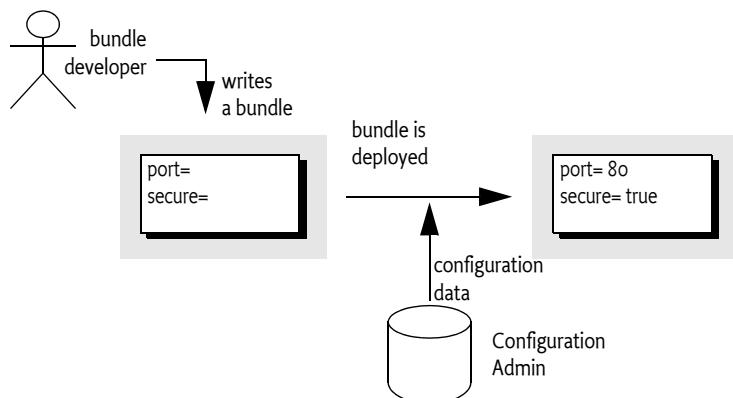
104 Configuration Admin Service Specification

Version 1.5

104.1 Introduction

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

Figure 104.1   *Configuration Admin Service Overview*

104.1.1 Essentials

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

- *Local Configuration* - The Configuration Admin service must support bundles that have their own user interface to change their configurations.
- *Reflection* - The Configuration Admin service must be able to deduce the names and types of the needed configuration data.
- *Legacy* - The Configuration Admin service must support configuration data of existing entities (such as devices).
- *Object Oriented* - The Configuration Admin service must support the creation and deletion of instances of configuration information so that a bundle can create the appropriate number of services under the control of the Configuration Admin service.
- *Embedded Devices* - The Configuration Admin service must be deployable on a wide range of platforms. This requirement means that the interface should not assume file storage on the platform. The choice to use file storage should be left to the implementation of the Configuration Admin service.

- *Remote versus Local Management* - The Configuration Admin service must allow for a remotely managed OSGi framework, and must not assume that con-figuration information is stored local-ly. Nor should it assume that the Configuration Admin service is always done remotely. Both implementation approaches should be viable.
- *Availability* - The OSGi environment is a dynamic environment that must run continuously (24/7/365). Configuration updates must happen dynamically and should not require restarting of the system or bundles.
- *Immediate Response* - Changes in configuration should be reflected immediately.
- *Execution Environment* - The Configuration Admin service will not require more than an environment that fulfills the minimal execution requirements.
- *Communications* - The Configuration Admin service should not assume "always-on" connectivity, so the API is also applicable for mobile applications in cars, phones, or boats.
- *Extendability* - The Configuration Admin service should expose the process of configuration to other bundles. This exposure should at a minimum encompass initiating an update, removing certain configuration properties, adding properties, and modifying the value of properties potentially based on existing property or service values.
- *Complexity Trade-offs* - Bundles in need of configuration data should have a simple way of obtaining it. Most bundles have this need and the code to accept this data. Additionally, updates should be simple from the perspective of the receiver.

  Trade-offs in simplicity should be made at the expense of the bundle implementing the Configuration Admin service and in favor of bundles that need configuration information. The reason for this choice is that normal bundles will outnumber Configuration Admin bundles.
- *Regions* - It should be possible to create groups of bundles and a manager in a single system that share configuration data that is not accessible outside the region.
- *Shared Information* - It should be possible to share configuration data between bundles.
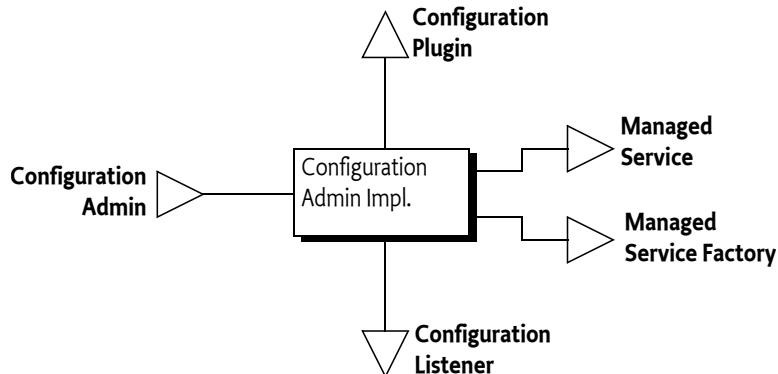
104.1.2   Entities

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

- *Configuration Object* - Implements the `Configuration` interface and contains the configuration dictionary for a Managed Service or one of the configuration dictionaries for a Managed Service Factory. These objects are manipulated by configuring bundles.
- *Configuration Plugin Services* - Configuration Plugin services are called before the configuration dictionary is given to the configuration targets. The plug-in can modify the configuration dictionary, which is passed to the Configuration Target.

Figure 104.2   *Overall Service Diagram*

104.1.3   Synopsis

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

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

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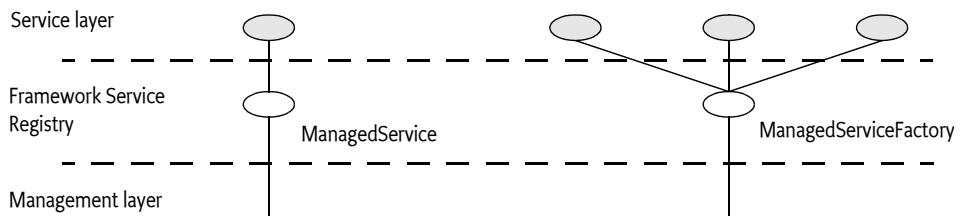
104.2   Configuration Targets

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

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

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

Figure 104.3   *Differentiation of ManagedService and ManagedServiceFactory Classes*

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

To summarize:

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

104.3   The Persistent Identity

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

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

A service can register with multiple PIDs and PIDs can be shared between multiple targets (both Managed Service and Managed Service Factory targets) to receive the same information. If PIDs are to be shared between Bundles then the location of the Configuration must be a multi-location, see *Location Binding* on page 67.

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

104.3.1   PID Syntax

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

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

104.3.1.1 Local Bundle PIDs

As a convention, descriptions starting with the bundle identity and a full stop ('.' \u002E) are reserved for a bundle. As an example, a PID of "65.536" would belong to the bundle with a bundle identity of 65.

104.3.1.2 Software PIDs

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

104.3.1.3 Devices

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

Table 104.1   *Schemes for Device-Oriented PID Names*

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

104.3.2 Targeted PIDs

PIDs are defined as primary keys for the configuration object; any target that uses the PID in its service registration (and has the proper permissions if security is on) will receive the configuration associated with it, regardless of the bundle that registered the target service. Though in general the PID is designed to ignore the bundle, there are a number of cases where the bundle becomes relevant. The most typical case is where a bundle is available in different versions. Each version will request the same PID and will get therefore configured identically.

Targeted PIDs are specially formatted PIDs that are interpreted by the Configuration Admin service. Targeted PIDs work both as a normal Managed Service PID and as a Managed Service Factory PID. In the case of factories, the targeted PID is the Factory PID since the other PID is chosen by CM for each instance.

The target PID scopes the applicability of the PID to a limited set of target bundles. The syntax of a target pid is:

```
target-pid  ::=  PID
    ( '|' symbolic-name ( '|' version ( '|' location )? )? )?
```

Targets never register with a target PID, target PIDs should only be used when creating, getting, or deleting a Configuration through the Configuration Admin service. The target PID is still the primary key of the Configuration and is thus in itself a PID. The distinction is only made when the Configuration Admin must update a target service. Instead of using the non-target PID as the primary key it must first search if there exists a target PID in the Configuration store that matches the requested target PID.

When a target registers and needs to be updated the Configuration Admin must first find the Configuration with the *best matching* PID. It must logically take the requested PID, append it with the bundle symbolic name, the bundle version, and the bundle location. The version must be formatted canonically, that is, according to the toString() method of the Version class. The rules for best matching are then as follows:

Look for a Configuration, in the given order, with a key of:

```
<pid>|<bsn>|<version>|<location>
<pid>|<bsn>|<version>
<pid>|<bsn>
<pid>
```

For example:

```
com.example.web.WebConf|com.acme.example|3.2.0|http://www.xyz.com/acme.jar
com.example.web.WebConf|com.acme.example|3.2.0
com.example.web.WebConf|com.acme.example
com.example.web.WebConf
```

If a registered target service has a PID that contains a vertical line ('|' \u007c)|then the value must be taken as is and must not be interpreted as a targeted PID.

The service.pid configuration property for a targeted PID configuration must always be set to the targeted PID. That is, if the PID is com.example.web.WebConf and the targeted PID com.example.web.WebConf|com.acme.example|3.2.0 then the property in the Configuration dictionary must be the targeted PID.

If a Configuration with a targeted PID is deleted or a Configuration with a new targeted PID is added then all targets that would be stale must be reevaluated against the new situation and updated accordingly if they are no longer bound against the best matching target PID.

104.3.3   Extenders and Targeted PIDs

Extenders like Declarative Services use Configurations but bypass the general Managed Service or Managed Service Factory method. It is the responsibility of these extenders to access the Configurations using the targeted PIDs.

Since getting a Configuration tends to create that Configuration it is necessary for these extenders to use the listConfigurations(String) method to find out if a more targeted Configuration exists. There are many ways the extender can find the most targeted PID. For example, the following code gets the most targeted PID for a given bundle.

```
String mostTargeted(String key, String pid, Bundle bundle) throws Exception {
    String bsn = bundle.getSymbolicName();
    Version version = bundle.getVersion();
    String location = bundle.getLocation();
    String f = String.format("(|(%1$s=%2$s)(%1$s=%2$s|%3$s)" +
        "(%1$s=%2$s|%3$s|%4$s)(%1$s=%2$s|%3$s|%4$s|%5$s))",
        key, pid, bsn, version, location );

    Configuration[] configurations = cm.listConfigurations(f);
    if (configurations == null)
        return null;

    String largest = null;
    for (Configuration c : configurations) {
        String s = (String) c.getProperties().get(key);
        if ((largest == null) || (largest.length() < s.length()))
```

```
                largest = s;
        }
        return largest;
}
```

104.4   The Configuration Object

A Configuration object contains the configuration dictionary, which is a set of properties that configure an aspect of a bundle. A bundle can receive Configuration objects by registering a configuration target service with a PID service property. See *The Persistent Identity* on page 64 for more information about PIDs.

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

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

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

104.4.1   Location Binding

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

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

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

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

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

For example

```
?com.acme
```

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

- *Single-Location* - If c.location is not a multi-location then a Bundle only has visibility if the Bundle's location exactly matches c.location. In this case there is never a security check.
- *Multi-Location* - If c.location is a multi-location (that is, starts with a question mark):

- *Security Off* - The Bundle always has visibility
- *Security On* - The target's Bundle must have ConfigurationPermission[c.location, TARGET] as defined by the Bundle's hasPermission method. The resource name of the permission must include the question mark.

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

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

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

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

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

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

104.4.2   Dynamic Binding

Dynamic binding is available for backward compatibility with earlier versions. It is recommended that management agents explicitly set the location to a ? (a multi-location) to allow multiple bundles to share PIDs and not use the dynamic binding facility. If a management agent uses ?, it must at least have ConfigurationPermission[?, CONFIGURE] when security is on, it is also possible to use ConfigurationPermission[?*, CONFIGURE] to not limit the management agent. See *Regions* on page 79 for some examples of using the locations in isolation scenarios.

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

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

104.4.3   Configuration Properties

A configuration dictionary contains a set of properties in a Dictionary object. The value of the property must be the same type as the set of Primary Property Types specified in *OSGi Core Release 6* Filter Syntax.

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

```
property-name ::= public | private
public       ::= symbolic-name // See General Syntax in Core Framework
private      ::= '.' symbolic-name
```

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

Bundles must not use nested lists or arrays, nor must they use mixed types. Using mixed types or nesting makes it impossible to use the meta typing specification. See *Metatype Service Specification* on page 105.

Property values that are collections may have an ordering that must be preserved when persisting the configuration so that later access to the property value will see the preserved ordering of the collection.

104.4.4 Property Propagation

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

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

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

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

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

104.4.5 Automatic Properties

The Configuration Admin service must automatically add a number of properties to the configuration dictionary. If these properties are also set by a configuring bundle or a plug-in, they must always be overridden before they are given to the target service, see *Configuration Plugin* on page 82. 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. This is the full the targeted PID if a targeted PID is used, see *Targeted PIDs* on page 65.
- service.factoryPid - Only set for a Managed Service Factory. It is then set to the PID of the associated Managed Service Factory. This is the full the targeted PID if a targeted PID is used.
- service.bundleLocation - Set to the location of the Configuration object. This property can only be used for searching, it may not appear in the configuration dictionary returned from the getProperties method due to security reasons, nor may it be used when the target is updated.

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

104.4.6 Equality

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

104.5   Managed Service

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

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

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

A Managed Service may be registered with more than one PID and therefore be associated with multiple Configuration objects, one for each PID. Using multiple PIDs for a Managed Service is not recommended. For example, when a configuration is deleted for a Managed Service there is no way to identify which PID is associated with the deleted configuration.

104.5.1   Singletons

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

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

104.5.2   Networks

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

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

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

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

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

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

104.5.3   Configuring Managed Services

A bundle that needs configuration information should register one or more `ManagedService` objects with a PID service property. If it has a default set of properties for its configuration, it may include them as service properties of the Managed Service. These properties may be used as a configuration template when a `Configuration` object is created for the first time. A Managed Service optionally implements the `MetaTypeProvider` interface to provide information about the property types. See *Meta Typing* on page 84.

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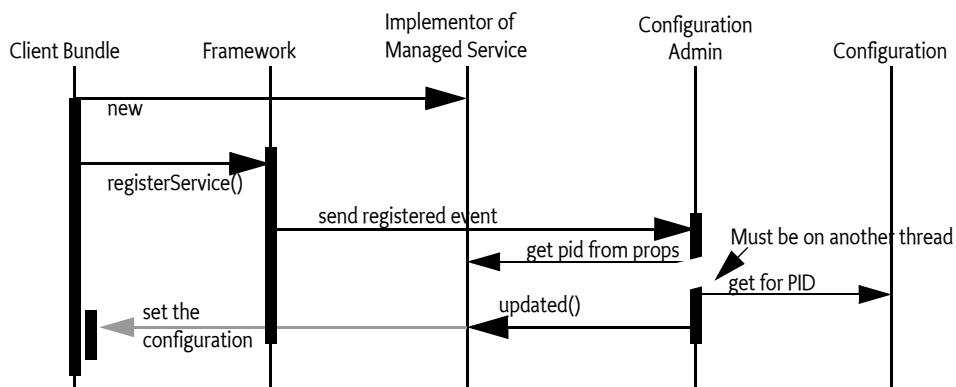
- The configuration stored for the registered PID must be retrieved. If there is a `Configuration` object for this PID and the configuration is visible for the associated bundle then it is sent to the Managed Service with `updated(Dictionary)`.
- If a Managed Service is registered and no configuration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

Multiple Managed Services can register with the same PID, they are all updated as long as they have visibility to the configuration as defined by the location, see *Location Binding* on page 67.

If the Managed Service is registered with more than one PID and more than one PID has no configuration information available, then `updated(Dictionary)` will be called multiple times with a `null` parameter.

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

Figure 104.4   Managed Service Configuration Action Diagram

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

104.5.4 Race Conditions

When a Managed Service is registered, the default properties may be visible in the service registry for a short period before they are replaced by the properties of the actual configuration dictionary. Care should be taken that this visibility does not cause race conditions for other bundles.
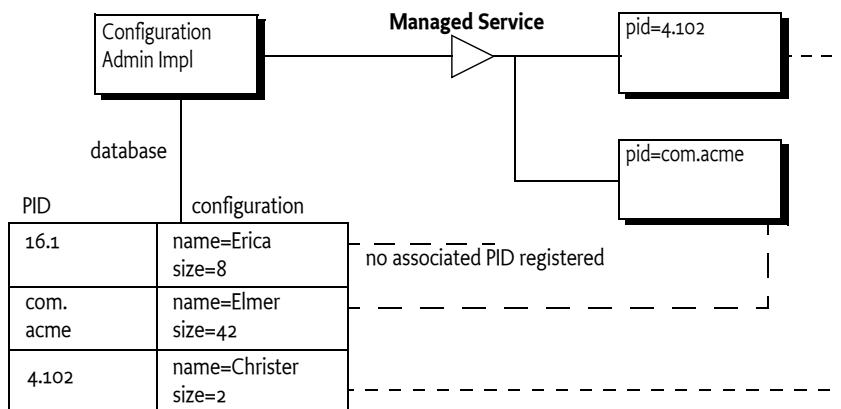
In cases where race conditions could be harmful, the Managed Service must be split into two pieces: an object performing the actual service and a Managed Service. First, the Managed Service is registered, the configuration is received, and the actual service object is registered. In such cases, the use of a Managed Service Factory that performs this function should be considered.

104.5.5 Examples of Managed Service

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

Figure 104.5   *PIDs and External Associations*

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

104.5.5.1 Configuring A Console Bundle

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

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

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

        registration = context.registerService(
```

```
                ManagedService.class.getName(),
                this,
                properties
        );
    }

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

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

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

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

104.5.6   Deletion

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

104.6   Managed Service Factory

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

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

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

104.6.1   When to Use a Managed Service Factory

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

104.6.1.1   Example Email Fetcher

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

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

104.6.1.2   Example Temperature Conversion Service

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

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

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

104.6.1.3   Serial Ports

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

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

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

104.6.2   Registration

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

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

When the Configuration Admin service detects the registration of a Managed Service Factory, it must find all visible configuration dictionaries for this factory and must then sequentially call ManagedServiceFactory.updated(String,Dictionary) for each configuration dictionary. The first ar-

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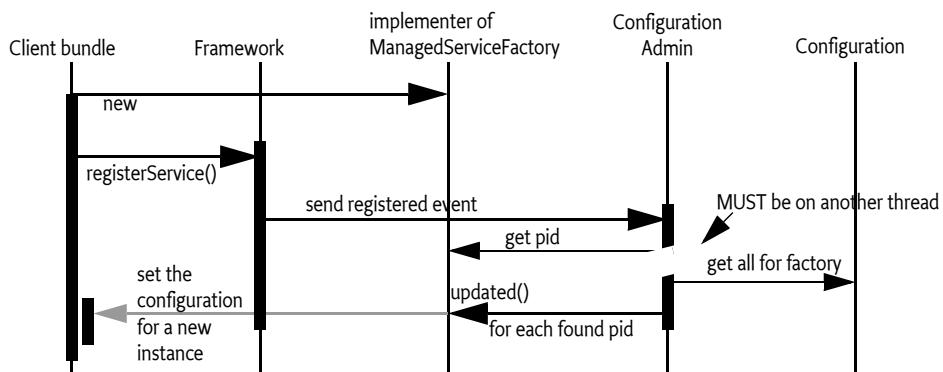
The Managed Service Factory should then create any artifacts associated with that factory. Using the PID given in the Configuration object, the bundle may register new services (other than a Managed Service) with the Framework, but this is not required. This may be necessary when the PID is useful in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

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

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

Figure 104.6   *Managed Service Factory Action Diagram*

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

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

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

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

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

104.6.3 Deletion

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

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

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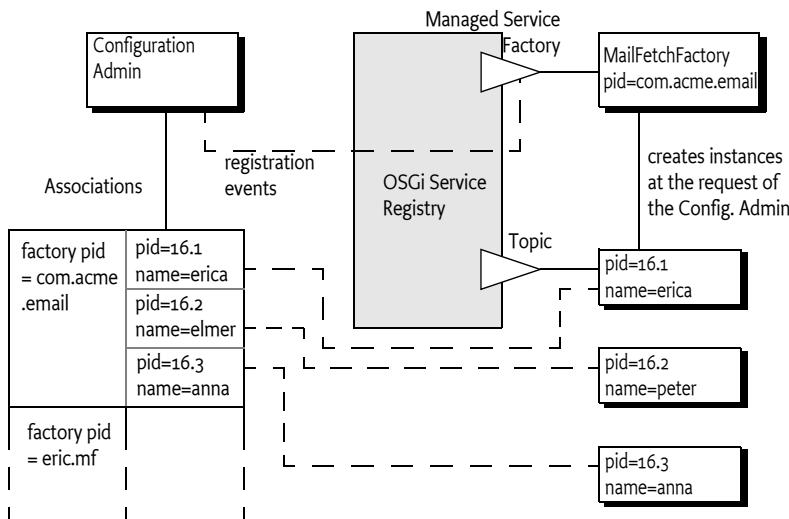
Figure 104.7 highlights the differences between a Managed Service and a Managed Service Factory. It shows how a Managed Service Factory implementation receives configuration information that was created before it was registered.

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

If the `EMailFetcher` object is registered, it may safely use the PID of the `Configuration` object because the Configuration Admin service must guarantee its suitability for this purpose.

Figure 104.7   *Managed Service Factory Example*

104.6.5   Multiple Consoles Example

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

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

```
                    this,
                    local );
        }

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

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

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

104.7 Configuration Admin Service

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

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

104.7.1 Creating a Managed Service Configuration Object

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

Two variants of this method are:

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

is the PID and the second argument is the location identifier of the targeted ManagedService object.

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

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

104.7.2   Creating a Managed Service Factory Configuration Object

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

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

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

104.7.3   Accessing Existing Configurations

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

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

The Configuration Admin service must only return Configurations that are visible to the calling bundle, see *Location Binding* on page 67.

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

The PIDs that are filtered on can be targeted PIDs, see *Targeted PIDs* on page 65.

104.7.4   Updating a Configuration

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

New properties can be set by calling Configuration.update. The Configuration Admin service must first store the configuration information and then call all configuration targets that have visibility with the updated method: either the ManagedService.updated(Dictionary) or ManagedServiceFactory.updated(String,Dictionary) method. If a target service is not registered, the fresh configuration information must be given to the target when the configuration target service registers and it has visibility. Each update of the Configuration properties must update a counter in the Configuration object after the data has been persisted but before the target(s) have been updated and any events are sent out. This counter is available from the getChangeCount() method.

The update method calls in Configuration objects are not executed synchronously with the related target services updated method. The updated method must be called asynchronously. The Configu-

ration Admin service, however, must have updated the persistent storage before the update method returns.

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

104.7.5 Using Multi-Locations

Sharing configuration between different bundles can be done using multi-locations, see *Location Binding* on page 67. A multi-location for a Configuration enables this Configuration to be delivered to any bundle that has visibility to that configuration. It is also possible that Bundles are interested in multiple PIDs for one target service, for this reason they can register multiple PIDs for one service.

For example, a number of bundles require access to the URL of a remote host, associated with the PID com.acme.host. A manager, aware that this PID is used by different bundles, would need to specify a location for the Configuration that allows delivery to any bundle. A multi-location, any location starting with a question mark achieves this. The part after the question mark has only use if the system runs with security, it allows the implementation of regions, see *Regions* on page 79. In this example a single question mark is used because any Bundle can receive this Configuration. The manager's code could look like:

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

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

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

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

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

104.7.6 Regions

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

Regions are groups of bundles that share location information among each other but are not willing to share this information with others. Using the multi-locations, see *Location Binding* on page 67, and security it is possible to limit access to a Configuration by using a location name. A Bundle can only receive a Configuration when it has ConfigurationPermission [location name, TARGET]. It is therefore possible to create region by choosing a region name for the location. A management agent

then requires ConfigurationPermission [?region-name, CONFIGURE] and a Bundle in the region requires ConfigurationPermission [?region-name, TARGET].

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

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

ConfigurationPermission [?com.acme.∗, CONFIGURE]

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

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

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

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

Another, similar, example with two regions:

- system
- application

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

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

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

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

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

104.7.7   Deletion

A Configuration object that is no longer needed can be deleted with Configuration.delete, which removes the Configuration object from the database. The database must be updated before the target service's updated or deleted method is called. Only services that have received the configuration dictionary before must be called.

If the target service is a Managed Service Factory, the factory is informed of the deleted Configuration object by a call to ManagedServiceFactory.deleted(String) method. It should then remove the associated *instance*. The ManagedServiceFactory.deleted(String) call must be done asynchronously with respect to Configuration.delete().

When a Configuration object of a Managed Service is deleted, ManagedService.updated is called with null for the properties argument. This method may be used for clean-up, to revert to default values, or to unregister a service. This method is called asynchronously from the delete method.

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

104.7.8 Updating a Bundle's Own Configuration

The Configuration Admin service specification does not distinguish between updates via a Management Agent and a bundle updating its own configuration information (as defined by its location). Even if a bundle updates its own configuration information, the Configuration Admin service must callback the associated target service's updated method.

As a rule, to update its own configuration, a bundle's user interface should *only* update the configuration information and never its internal structures directly. This rule has the advantage that the events, from the bundle implementation's perspective, appear similar for internal updates, remote management updates, and initialization.

104.8 Configuration Events

Configuration Admin can update interested parties of changes in its repository. The model is based on the white board pattern where Configuration Listener services are registered with the service registry.

There are two types of Configuration Listener services:

- ConfigurationListener - The default Configuration Listener receives events asynchronously from the method that initiated the event and on another thread.
- SynchronousConfigurationListener - A Synchronous Configuration Listener is guaranteed to be called on the same thread as the method call that initiated the event.

The Configuration Listener service will receive ConfigurationEvent objects if important changes take place. The Configuration Admin service must call the configurationEvent(ConfigurationEvent) method with such an event. Configuration Events must be delivered in order for each listener as they are generated. The way events must be delivered is the same as described in *Delivering Events* of *OSGi Core Release 6*.

The ConfigurationEvent object carries a factory PID (getFactoryPid()) and a PID (getPid()). If the factory PID is null, the event is related to a Managed Service Configuration object, else the event is related to a Managed Service Factory Configuration object.

The ConfigurationEvent object can deliver the following events from the getType() method:

- CM_DELETED - The Configuration object is deleted.
- CM_UPDATED - The Configuration object is updated.
- CM_LOCATION_CHANGED - The location of the Configuration object changed.

The Configuration Event also carries the ServiceReference object of the Configuration Admin service that generated the event.

104.8.1 Event Admin Service and Configuration Change Events

Configuration events must be delivered asynchronously by the Configuration Admin implementation, if present. The topic of a configuration event must be:

```
org/osgi/service/cm/ConfigurationEvent/<eventtype>
```

The ‹event type› can be any of the following:

CM_DELETED
CM_UPDATED
CM_LOCATION_CHANGED

The properties of a configuration event are:

- cm.factoryPid - (String) The factory PID of the associated Configuration object, if the target is a Managed Service Factory. Otherwise not set.
- cm.pid - (String) The PID of the associated Configuration object.
- service - (ServiceReference) The Service Reference of the Configuration Admin service.
- service.id - (Long) The Configuration Admin service's ID.
- service.objectClass - (String[]) The Configuration Admin service's object class (which must include org.osgi.service.cm.ConfigurationAdmin)
- service.pid - (String) The Configuration Admin service's persistent identity, if set.

104.9   Configuration Plugin

The Configuration Admin service allows third-party applications to participate in the configuration process. Bundles that register a service object under a ConfigurationPlugin interface can process the configuration dictionary just before it reaches the configuration target service.

Plug-ins allow sufficiently privileged bundles to intercept configuration dictionaries just *before* they must be passed to the intended Managed Service or Managed Service Factory but *after* the properties are stored. The changes the plug-in makes are dynamic and must not be stored. The plug-in must only be called when an update takes place while it is registered and there is a valid dictionary. The plugin is not called when a configuration is deleted.

The ConfigurationPlugin interface has only one method: modifyConfiguration(ServiceReference,Dictionary). This method inspects or modifies configuration data.
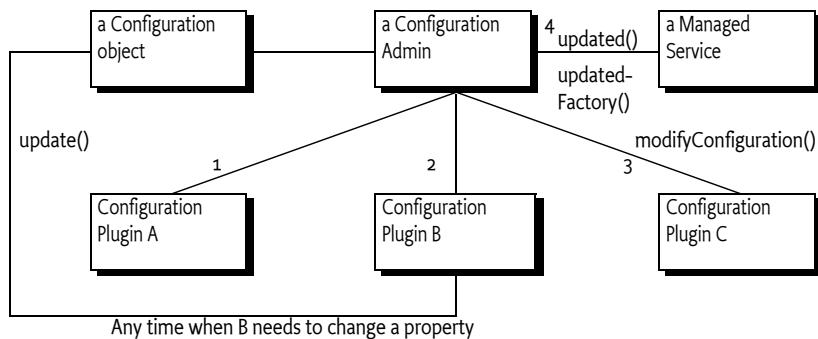
All plug-ins in the service registry must be traversed and called before the properties are passed to the configuration target service. Each Configuration Plugin object gets a chance to inspect the existing data, look at the target object, which can be a ManagedService object or a ManagedServiceFactory object, and modify the properties of the configuration dictionary. The changes made by a plugin 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 69.

For example, a Configuration Plugin service may add a physical location property to a service. This property can be leveraged by applications that want to know where a service is physically located. This scenario could be carried out without any further support of the service itself, except for the general requirement that the service should propagate the public properties it receives from the Configuration Admin service to the service registry.

Figure 104.8   *Order of Configuration Plugin Services*

104.9.1 Limiting The Targets

A ConfigurationPlugin object may optionally specify a cm.target registration property. This value is the PID of the configuration target whose configuration updates the ConfigurationPlugin object wants to intercept.

The ConfigurationPlugin object must then only be called with updates for the configuration target service with the specified PID. For a factory target service, the factory PID is used and the plugin will see all instances of the factory. Omitting the cm.target registration property means that it is called for *all* configuration updates.

104.9.2 Example of Property Expansion

Consider a Managed Service that has a configuration property service.to with the value (objectclass=com.acme.Alarm). When the Configuration Admin service sets this property on the target service, a ConfigurationPlugin object may replace the (objectclass=com.acme.Alarm) filter with an array of existing alarm systems' PIDs as follows:

ID "service.to=[32434,232,12421,1212]"

A new Alarm Service with service.pid=343 is registered, requiring that the list of the target service be updated. The bundle which registered the Configuration Plugin service, therefore, wants to set the service.to registration property on the target service. It does *not* do this by calling ManagedService.updated directly for several reasons:

- In a securely configured system, it should not have the permission to make this call or even obtain the target service.
- It could get into race conditions with the Configuration Admin service if it had the permissions in the previous bullet. Both services would compete for access simultaneously.

Instead, it must get the Configuration object from the Configuration Admin service and call the update method on it.

The Configuration Admin service must schedule a new update cycle on another thread, and sometime in the future must call ConfigurationPlugin.modifyProperties. The ConfigurationPlugin object could then set the service.to property to [32434,232,12421,1212, 343]. After that, the Configuration Admin service must call updated on the target service with the new service.to list.

104.9.3 Configuration Data Modifications

Modifications to the configuration dictionary are still under the control of the Configuration Admin service, which must determine whether to accept the changes, hide critical variables, or deny the changes for other reasons.

The ConfigurationPlugin interface must also allow plugins to detect configuration updates to the service via the callback. This ability allows them to synchronize the configuration updates with transient information.

104.9.4 Forcing a Callback

If a bundle needs to force a Configuration Plugin service to be called again, it must fetch the appropriate Configuration object from the Configuration Admin service and call the update() method (the no parameter version) on this object. This call forces an update with the current configuration dictionary so that all applicable plug-ins get called again.

104.9.5 Calling Order

The order in which the ConfigurationPlugin objects are called must depend on the service.cmRanking configuration property of the ConfigurationPlugin object. Table 104.2 shows the usage of the service.cmRanking property for the order of calling the Configuration Plugin services.

Table 104.2   *service.cmRanking Usage For Ordering*

service.cmRanking value	Description
< 0	The Configuration Plugin service should not modify properties and must be called before any modifications are made.
>= 0 && <= 1000	The Configuration Plugin service modifies the configuration data. The calling order should be based on the value of the service.cmRanking property.
> 1000	The Configuration Plugin service should not modify data and is called after all modifications are made.

104.10 Meta Typing

This section discusses how the Metatype specification is used in the context of a Configuration Admin service.

When a Managed Service or Managed Service Factory is registered, the service object may also implement the MetaTypeProvider interface.

If the Managed Service or Managed Service Factory object implements the MetaTypeProvider interface, a management bundle may assume that the associated ObjectClassDefinition object can be used to configure the service.

The ObjectClassDefinition and AttributeDefinition objects contain sufficient information to automatically build simple user interfaces. They can also be used to augment dedicated interfaces with accurate validations.

When the Metatype specification is used, care should be taken to match the capabilities of the metatype package to the capabilities of the Configuration Admin service specification. Specifically:

• The metatype specification cannot describe nested arrays and lists or arrays/lists of mixed type.

This specification does not address how the metatype is made available to a management system due to the many open issues regarding remote management.

104.11 Security

104.11.1 Configuration Permission

Every bundle has the implicit right to receive and configure configurations with a location that exactly matches the Bundle's location or that is null. For all other situations the Configuration Admin must verify that the configuring and to be updated bundles have a Configuration Permission that matches the Configuration's location.

The resource name of this permission maps to the location of the Configuration, the location can control the visibility of a Configuration for a bundle. The resource name is compared with the actual configuration location using the OSGi Filter sub-string matching. The question mark for multi-locations is part of the given resource name. The Configure Permission has the following actions:

- CONFIGURE - Can manage matching configurations
- TARGET - Can be updated with a matching configuration

To be able to set the location to null requires a ConfigurationPermission[*, CONFIGURE].

It is possible to deny bundles the use of multi-locations by using Conditional Permission Admin's deny model.

104.11.2 Permissions Summary

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

Configuration Admin:

```
ServicePermission[ ..ConfigurationAdmin, REGISTER]
ServicePermission[ ..ManagedService, GET ]
ServicePermission[ ..ManagedServiceFactory, GET ]
ServicePermission[ ..ConfigurationPlugin, GET ]
ConfigurationPermission[ *, CONFIGURE ]
AdminPermission[ *, METADATA ]
```

Managed Service:

```
ServicePermission[ ..ConfigurationAdmin, GET]
ServicePermission[ ..ManagedService, REGISTER ]
ConfigurationPermission[ ... , TARGET ]
```

Managed Service Factory:

```
ServicePermission[ ..ConfigurationAdmin, GET]
ServicePermission[ ..ManagedServiceFactory, REGISTER ]
ConfigurationPermission[ ... , TARGET ]
```

Configuration Plugin:

```
ServicePermission[ ..ConfigurationPlugin,REGISTER ]
```

Configuration Listener:

```
ServicePermission[ ..ConfigurationListener,REGISTER ]
```

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

have GET permission for these interfaces. The Configuration Admin bundle must also hold ConfigurationPermission[*,CONFIGURE].

Bundles that can be configured must have the ServicePermission[ManagedService | ManagedServiceFactory, REGISTER]. Bundles registering ConfigurationPlugin objects must have ServicePermission[ConfigurationPlugin, REGISTER]. The Configuration Admin service must trust all services registered with the ConfigurationPlugin interface. Only the Configuration Admin service should have ServicePermission[ConfigurationPlugin, GET].

If a Managed Service or Managed Service Factory is implemented by an object that is also registered under another interface, it is possible, although inappropriate, for a bundle other than the Configuration Admin service implementation to call the updated method. Security-aware bundles can avoid this problem by having their updated methods check that the caller has ConfigurationPermission[*,CONFIGURE].

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

Pre-configuration of bundles requires ConfigurationPermission[location,CONFIGURE] (location can use the sub-string matching rules of the Filter) because the methods that specify a location require this permission.

104.11.3   Configuration and Permission Administration

Configuration information has a direct influence on the permissions needed by a bundle. For example, when the Configuration Admin Bundle orders a bundle to use port 2011 for a console, that bundle also needs permission for listening to incoming connections on that port.

Both a simple and a complex solution exist for this situation.

The simple solution for this situation provides the bundle with a set of permissions that do not define specific values but allow a range of values. For example, a bundle could listen to ports above 1024 freely. All these ports could then be used for configuration.

The other solution is more complicated. In an environment where there is very strong security, the bundle would only be allowed access to a specific port. This situation requires an atomic update of both the configuration data and the permissions. If this update was not atomic, a potential security hole would exist during the period of time that the set of permissions did not match the configuration.

The following scenario can be used to update a configuration and the security permissions:

1. Stop the bundle.
2. Update the appropriate Configuration object via the Configuration Admin service.
3. Update the permissions in the Framework.
4. Start the bundle.

This scenario would achieve atomicity from the point of view of the bundle.

104.12   org.osgi.service.cm

Configuration Admin Package Version 1.5.

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

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.cm; version="[1.5,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.cm; version="[1.5,1.6)"

104.12.1 Summary

- Configuration - The configuration information for a ManagedService or ManagedServiceFactory object.
- ConfigurationAdmin - Service for administering configuration data.
- ConfigurationEvent - A Configuration Event.
- ConfigurationException - An Exception class to inform the Configuration Admin service of problems with configuration data.
- ConfigurationListener - Listener for Configuration Events.
- ConfigurationPermission - Indicates a bundle's authority to configure bundles or be updated by Configuration Admin.
- ConfigurationPlugin - A service interface for processing configuration dictionary before the update.
- ManagedService - A service that can receive configuration data from a Configuration Admin service.
- ManagedServiceFactory - Manage multiple service instances.
- SynchronousConfigurationListener - Synchronous Listener for Configuration Events.

104.12.2 Permissions

104.12.2.1 Configuration

- setBundleLocation(String)
  - ConfigurationPermission[this.location,CONFIGURE] - if this.location is not null
  - ConfigurationPermission[location,CONFIGURE] - if location is not null
  - ConfigurationPermission["∗",CONFIGURE] - if this.location is null or if location is null
- getBundleLocation()
  - ConfigurationPermission[this.location,CONFIGURE] - if this.location is not null
  - ConfigurationPermission["∗",CONFIGURE] - if this.location is null

104.12.2.2 ConfigurationAdmin

- createFactoryConfiguration(String,String)
  - ConfigurationPermission[location,CONFIGURE] - if location is not null
  - ConfigurationPermission["∗",CONFIGURE] - if location is null
- getConfiguration(String,String)
  - ConfigurationPermission[∗,CONFIGURE] - if location is null or if the returned configuration c already exists and c.location is null
  - ConfigurationPermission[location,CONFIGURE] - if location is not null
  - ConfigurationPermission[c.location,CONFIGURE] - if the returned configuration c already exists and c.location is not null
- getConfiguration(String)
  - ConfigurationPermission[c.location,CONFIGURE] - If the configuration c already exists and c.location is not null
- listConfigurations(String)
  - ConfigurationPermission[c.location,CONFIGURE] - Only configurations c are returned for which the caller has this permission

104.12.2.3   **ManagedService**

- updated(Dictionary)
  - ConfigurationPermission[c.location,TARGET] - Required by the bundle that registered this service

104.12.2.4   **ManagedServiceFactory**

- updated(String,Dictionary)
  - ConfigurationPermission[c.location,TARGET] - Required by the bundle that registered this service

104.12.3   public interface Configuration

The configuration information for a ManagedService or ManagedServiceFactory object. The Configuration Admin service uses this interface to represent the configuration information for a ManagedService or for a service instance of a ManagedServiceFactory.

A Configuration object contains a configuration dictionary and allows the properties to be updated via this object. Bundles wishing to receive configuration dictionaries do not need to use this class - they register a ManagedService or ManagedServiceFactory. Only administrative bundles, and bundles wishing to update their own configurations need to use this class.

The properties handled in this configuration have case insensitive String objects as keys. However, case must be preserved from the last set key/value.

A configuration can be *bound* to a specific bundle or to a region of bundles using the *location*. In its simplest form the location is the location of the target bundle that registered a Managed Service or a Managed Service Factory. However, if the location starts with ? then the location indicates multiple delivery. In such a case the configuration must be delivered to all targets. If security is on, the Configuration Permission can be used to restrict the targets that receive updates. The Configuration Admin must only update a target when the configuration location matches the location of the target's bundle or the target bundle has a Configuration Permission with the action ConfigurationPermission.TARGET and a name that matches the configuration location. The name in the permission may contain wildcards ('*') to match the location using the same substring matching rules as Filter. Bundles can always create, manipulate, and be updated from configurations that have a location that matches their bundle location.

If a configuration's location is null, it is not yet bound to a location. It will become bound to the location of the first bundle that registers a ManagedService or ManagedServiceFactory object with the corresponding PID.

The same Configuration object is used for configuring both a Managed Service Factory and a Managed Service. When it is important to differentiate between these two the term "factory configuration" is used.

Concurrency   Thread-safe

Provider Type   Consumers of this API must not implement this type

104.12.3.1   **public void delete() throws IOException**

☐   Delete this Configuration object. Removes this configuration object from the persistent store. Notify asynchronously the corresponding Managed Service or Managed Service Factory. A ManagedService object is notified by a call to its updated method with a null properties argument. A ManagedServiceFactory object is notified by a call to its deleted method.

Also notifies all Configuration Listeners with a ConfigurationEvent.CM_DELETED event.

Throws   IOException– If delete fails.

IllegalStateException– If this configuration has been deleted.

104.12.3.2   **public boolean equals(Object other)**

other   Configuration object to compare against

☐   Equality is defined to have equal PIDs Two Configuration objects are equal when their PIDs are equal.

Returns   true if equal, false if not a Configuration object or one with a different PID.

104.12.3.3   **public String getBundleLocation()**

☐   Get the bundle location. Returns the bundle location or region to which this configuration is bound, or null if it is not yet bound to a bundle location or region. If the location starts with ? then the configuration is delivered to all targets and not restricted to a single bundle.

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

Throws   IllegalStateException– If this configuration has been deleted.

         SecurityException– when the required permissions are not available

Security   ConfigurationPermission[this.location,CONFIGURE]] – if this.location is not null

         ConfigurationPermission["*",CONFIGURE]] – if this.location is null

104.12.3.4   **public long getChangeCount()**

☐   Get the change count. Each Configuration must maintain a change counter that is incremented with a positive value every time the configuration is updated and its properties are stored. The counter must be incremented before the targets are updated and events are sent out.

Returns   A monotonically increasing value reflecting changes in this Configuration.

Throws   IllegalStateException– If this configuration has been deleted.

Since   1.5

104.12.3.5   **public String getFactoryPid()**

☐   For a factory configuration return the PID of the corresponding Managed Service Factory, else return null.

Returns   factory PID or null

Throws   IllegalStateException– If this configuration has been deleted.

104.12.3.6   **public String getPid()**

☐   Get the PID for this Configuration object.

Returns   the PID for this Configuration object.

Throws   IllegalStateException– if this configuration has been deleted

104.12.3.7   **public Dictionary‹String,Object› getProperties()**

☐   Return the properties of this Configuration object. The Dictionary object returned is a private copy for the caller and may be changed without influencing the stored configuration. The keys in the returned dictionary are case insensitive and are always of type String.

If called just after the configuration is created and before update has been called, this method returns null.

Returns   A private copy of the properties for the caller or null. These properties must not contain the "service.bundleLocation" property. The value of this property may be obtained from the getBundleLocation() method.

Throws   IllegalStateException– If this configuration has been deleted.

104.12.3.8   **public int hashCode()**

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

Returns   hash code for this Configuration object

104.12.3.9   **public void setBundleLocation(String location)**

location   a location, region, or null

□ Bind this Configuration object to the specified location. If the location parameter is null then the Configuration object will not be bound to a location/region. It will be set to the bundle's location before the first time a Managed Service/Managed Service Factory receives this Configuration object via the updated method and before any plugins are called. The bundle location or region will be set persistently.

If the location starts with ? then all targets registered with the given PID must be updated.

If the location is changed then existing targets must be informed. If they can no longer see this configuration, the configuration must be deleted or updated with null. If this configuration becomes visible then they must be updated with this configuration.

Also notifies all Configuration Listeners with a ConfigurationEvent.CM_LOCATION_CHANGED event.

Throws   IllegalStateException– If this configuration has been deleted.

SecurityException– when the required permissions are not available

SecurityException– when the required permissions are not available

Security   ConfigurationPermission[this.location,CONFIGURE]] − if this.location is not null

ConfigurationPermission[location,CONFIGURE]] − if location is not null

ConfigurationPermission["*",CONFIGURE]] − if this.location is null or if location is null

104.12.3.10   **public void update(Dictionary<String,?> properties) throws IOException**

properties   the new set of properties for this configuration

□ Update the properties of this Configuration object. Stores the properties in persistent storage after adding or overwriting the following properties:

- "service.pid" : is set to be the PID of this configuration.
- "service.factoryPid" : if this is a factory configuration it is set to the factory PID else it is not set.

These system properties are all of type String.

If the corresponding Managed Service/Managed Service Factory is registered, its updated method must be called asynchronously. Else, this callback is delayed until aforementioned registration occurs.

Also notifies all Configuration Listeners with a ConfigurationEvent.CM_UPDATED event.

Throws   IOException– if update cannot be made persistent

IllegalArgumentException– if the Dictionary object contains invalid configuration types or contains case variants of the same key name.

IllegalStateException– If this configuration has been deleted.

104.12.3.11   **public void update() throws IOException**

□ Update the Configuration object with the current properties. Initiate the updated callback to the Managed Service or Managed Service Factory with the current properties asynchronously.

This is the only way for a bundle that uses a Configuration Plugin service to initiate a callback. For example, when that bundle detects a change that requires an update of the Managed Service or Managed Service Factory via its ConfigurationPlugin object.

Throws IOException– if update cannot access the properties in persistent storage

IllegalStateException– If this configuration has been deleted.

See Also ConfigurationPlugin

104.12.4　public interface ConfigurationAdmin

Service for administering configuration data.

The main purpose of this interface is to store bundle configuration data persistently. This information is represented in Configuration objects. The actual configuration data is a Dictionary of properties inside a Configuration object.

There are two principally different ways to manage configurations. First there is the concept of a Managed Service, where configuration data is uniquely associated with an object registered with the service registry.

Next, there is the concept of a factory where the Configuration Admin service will maintain 0 or more Configuration objects for a Managed Service Factory that is registered with the Framework.

The first concept is intended for configuration data about "things/services" whose existence is defined externally, e.g. a specific printer. Factories are intended for "things/services" that can be created any number of times, e.g. a configuration for a DHCP server for different networks.

Bundles that require configuration should register a Managed Service or a Managed Service Factory in the service registry. A registration property named service.pid (persistent identifier or PID) must be used to identify this Managed Service or Managed Service Factory to the Configuration Admin service.

When the ConfigurationAdmin detects the registration of a Managed Service, it checks its persistent storage for a configuration object whose service.pid property matches the PID service property (service.pid) of the Managed Service. If found, it calls ManagedService.updated(Dictionary) method with the new properties. The implementation of a Configuration Admin service must run these callbacks asynchronously to allow proper synchronization.

When the Configuration Admin service detects a Managed Service Factory registration, it checks its storage for configuration objects whose service.factoryPid property matches the PID service property of the Managed Service Factory. For each such Configuration objects, it calls the ManagedServiceFactory.updated method asynchronously with the new properties. The calls to the updated method of a ManagedServiceFactory must be executed sequentially and not overlap in time.

In general, bundles having permission to use the Configuration Admin service can only access and modify their own configuration information. Accessing or modifying the configuration of other bundles requires ConfigurationPermission[location,CONFIGURE], where location is the configuration location.

Configuration objects can be *bound* to a specified bundle location or to a region (configuration location starts with ?). If a location is not set, it will be learned the first time a target is registered. If the location is learned this way, the Configuration Admin service must detect if the bundle corresponding to the location is uninstalled. If this occurs, the Configuration object must be unbound, that is its location field is set back to null.

If target's bundle location matches the configuration location it is always updated.

If the configuration location starts with ?, that is, the location is a region, then the configuration must be delivered to all targets registered with the given PID. If security is on, the target bundle must have Configuration Permission[location,TARGET], where location matches given the configu-

ration location with wildcards as in the Filter substring match. The security must be verified using the org.osgi.framework.Bundle.hasPermission(Object) method on the target bundle.

If a target cannot be updated because the location does not match or it has no permission and security is active then the Configuration Admin service must not do the normal callback.

The method descriptions of this class refer to a concept of "the calling bundle". This is a loose way of referring to the bundle which obtained the Configuration Admin service from the service registry. Implementations of ConfigurationAdmin must use a org.osgi.framework.ServiceFactory to support this concept.

Concurrency   Thread-safe

Provider Type   Consumers of this API must not implement this type

104.12.4.1   **public static final String SERVICE_BUNDLELOCATION = "service.bundleLocation"**

Configuration property naming the location of the bundle that is associated with a Configuration object. This property can be searched for but must not appear in the configuration dictionary for security reason. The property's value is of type String.

Since   1.1

104.12.4.2   **public static final String SERVICE_FACTORYPID = "service.factoryPid"**

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

Since   1.1

104.12.4.3   **public Configuration createFactoryConfiguration(String factoryPid) throws IOException**

factoryPid   PID of factory (not null).

☐   Create a new factory Configuration object with a new PID. The properties of the new Configuration object are null until the first time that its Configuration.update(Dictionary) method is called.

It is not required that the factoryPid maps to a registered Managed Service Factory.

The Configuration object is bound to the location of the calling bundle. It is possible that the same factoryPid has associated configurations that are bound to different bundles. Bundles should only see the factory configurations that they are bound to or have the proper permission.

Returns   A new Configuration object.

Throws   IOException– if access to persistent storage fails.

104.12.4.4   **public Configuration createFactoryConfiguration(String factoryPid,String location) throws IOException**

factoryPid   PID of factory (not null).

location   A bundle location string, or null.

☐   Create a new factory Configuration object with a new PID. The properties of the new Configuration object are null until the first time that its Configuration.update(Dictionary) method is called.

It is not required that the factoryPid maps to a registered Managed Service Factory.

The Configuration is bound to the location specified. If this location is null it will be bound to the location of the first bundle that registers a Managed Service Factory with a corresponding PID. It is possible that the same factoryPid has associated configurations that are bound to different bundles. Bundles should only see the factory configurations that they are bound to or have the proper permission.

If the location starts with ? then the configuration must be delivered to all targets with the corresponding PID.

Returns a new Configuration object.

Throws IOException– if access to persistent storage fails.

SecurityException– when the require permissions are not available

Security ConfigurationPermission[location,CONFIGURE]] – if location is not null

ConfigurationPermission["*",CONFIGURE]] – if location is null

104.12.4.5 **public Configuration getConfiguration(String pid,String location) throws IOException**

pid Persistent identifier.

location The bundle location string, or null.

☐ Get an existing Configuration object from the persistent store, or create a new Configuration object.

If a Configuration with this PID already exists in Configuration Admin service return it. The location parameter is ignored in this case though it is still used for a security check.

Else, return a new Configuration object. This new object is bound to the location and the properties are set to null. If the location parameter is null, it will be set when a Managed Service with the corresponding PID is registered for the first time. If the location starts with ? then the configuration is bound to all targets that are registered with the corresponding PID.

Returns An existing or new Configuration object.

Throws IOException– if access to persistent storage fails.

SecurityException– when the require permissions are not available

Security ConfigurationPermission[*,CONFIGURE]] – if location is null or if the returned configuration c already exists and c.location is null

ConfigurationPermission[location,CONFIGURE]] – if location is not null

ConfigurationPermission[c.location,CONFIGURE]] – if the returned configuration c already exists and c.location is not null

104.12.4.6 **public Configuration getConfiguration(String pid) throws IOException**

pid persistent identifier.

☐ Get an existing or new Configuration object from the persistent store. If the Configuration object for this PID does not exist, create a new Configuration object for that PID, where properties are null. Bind its location to the calling bundle's location.

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

Returns an existing or new Configuration matching the PID.

Throws IOException– if access to persistent storage fails.

SecurityException– when the required permission is not available

Security ConfigurationPermission[c.location,CONFIGURE]] – If the configuration c already exists and c.location is not null

104.12.4.7 **public Configuration[] listConfigurations(String filter) throws IOException, InvalidSyntaxException**

filter A filter string, or null to retrieve all Configuration objects.

☐ List the current Configuration objects which match the filter.

Only Configuration objects with non- null properties are considered current. That is, Configuration.getProperties() is guaranteed not to return null for each of the returned Configuration objects.

When there is no security on then all configurations can be returned. If security is on, the caller must have ConfigurationPermission[location,CONFIGURE].

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

- service.pid - the persistent identity
- service.factoryPid - the factory PID, if applicable
- service.bundleLocation - the bundle location

The filter can also be null, meaning that all Configuration objects should be returned.

Returns All matching Configuration objects, or null if there aren't any.

Throws IOException – if access to persistent storage fails

InvalidSyntaxException – if the filter string is invalid

Security ConfigurationPermission[c.location,CONFIGURE]] – Only configurations c are returned for which the caller has this permission

104.12.5   public class ConfigurationEvent

A Configuration Event.

ConfigurationEvent objects are delivered to all registered ConfigurationListener service objects. ConfigurationEvents must be delivered in chronological order with respect to each listener.

A type code is used to identify the type of event. The following event types are defined:

- CM_UPDATED
- CM_DELETED
- CM_LOCATION_CHANGED

Additional event types may be defined in the future.

Security Considerations. ConfigurationEvent objects do not provide Configuration objects, so no sensitive configuration information is available from the event. If the listener wants to locate the Configuration object for the specified pid, it must use ConfigurationAdmin.

See Also ConfigurationListener

Since 1.2

Concurrency Immutable

104.12.5.1   public static final int CM_DELETED = 2

A Configuration has been deleted.

This ConfigurationEvent type that indicates that a Configuration object has been deleted. An event is fired when a call to Configuration.delete() successfully deletes a configuration.

104.12.5.2   public static final int CM_LOCATION_CHANGED = 3

The location of a Configuration has been changed.

This ConfigurationEvent type that indicates that the location of a Configuration object has been changed. An event is fired when a call to Configuration.setBundleLocation(String) successfully changes the location.

Since 1.4

104.12.5.3   public static final int CM_UPDATED = 1

A Configuration has been updated.

This ConfigurationEvent type that indicates that a Configuration object has been updated with new properties. An event is fired when a call to Configuration.update(Dictionary) successfully changes a configuration.

104.12.5.4   **public ConfigurationEvent(ServiceReference‹ConfigurationAdmin› reference,int type,String factoryPid,String pid)**

reference   The ServiceReference object of the Configuration Admin service that created this event.

type   The event type. See getType().

factoryPid   The factory pid of the associated configuration if the target of the configuration is a ManagedServiceFactory. Otherwise null if the target of the configuration is a ManagedService.

pid   The pid of the associated configuration.

☐   Constructs a ConfigurationEvent object from the given ServiceReference object, event type, and pids.

104.12.5.5   **public String getFactoryPid()**

☐   Returns the factory pid of the associated configuration.

Returns   Returns the factory pid of the associated configuration if the target of the configuration is a ManagedServiceFactory. Otherwise null if the target of the configuration is a ManagedService.

104.12.5.6   **public String getPid()**

☐   Returns the pid of the associated configuration.

Returns   Returns the pid of the associated configuration.

104.12.5.7   **public ServiceReference‹ConfigurationAdmin› getReference()**

☐   Return the ServiceReference object of the Configuration Admin service that created this event.

Returns   The ServiceReference object for the Configuration Admin service that created this event.

104.12.5.8   **public int getType()**

☐   Return the type of this event.

The type values are:

- CM_UPDATED
- CM_DELETED
- CM_LOCATION_CHANGED

Returns   The type of this event.

104.12.6   **public class ConfigurationException**
extends Exception

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

104.12.6.1   **public ConfigurationException(String property,String reason)**

property   name of the property that caused the problem, null if no specific property was the cause

reason   reason for failure

☐   Create a ConfigurationException object.

104.12.6.2   **public ConfigurationException(String property,String reason,Throwable cause)**

property   name of the property that caused the problem, null if no specific property was the cause

reason   reason for failure

cause   The cause of this exception.

□ Create a ConfigurationException object.

Since   1.2

104.12.6.3   **public Throwable getCause()**

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

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

Since   1.2

104.12.6.4   **public String getProperty()**

□ Return the property name that caused the failure or null.

Returns   name of property or null if no specific property caused the problem

104.12.6.5   **public String getReason()**

□ Return the reason for this exception.

Returns   reason of the failure

104.12.6.6   **public Throwable initCause(Throwable cause)**

cause   The cause of this exception.

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

Returns   This exception.

Throws   IllegalArgumentException– If the specified cause is this exception.

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

Since   1.2

104.12.7   public interface ConfigurationListener

Listener for Configuration Events. When a ConfigurationEvent is fired, it is asynchronously delivered to all ConfigurationListeners.

ConfigurationListener objects are registered with the Framework service registry and are notified with a ConfigurationEvent object when an event is fired.

ConfigurationListener objects can inspect the received ConfigurationEvent object to determine its type, the pid of the Configuration object with which it is associated, and the Configuration Admin service that fired the event.

Security Considerations. Bundles wishing to monitor configuration events will require ServicePermission[ConfigurationListener,REGISTER] to register a ConfigurationListener service.

Since   1.2

Concurrency   Thread-safe

104.12.7.1   **public void configurationEvent(ConfigurationEvent event)**

event   The ConfigurationEvent.

□ Receives notification of a Configuration that has changed.

104.12.8 public final class ConfigurationPermission extends BasicPermission

Indicates a bundle's authority to configure bundles or be updated by Configuration Admin.

Since 1.2

Concurrency Thread-safe

104.12.8.1 public static final String CONFIGURE = "configure"

Provides permission to create new configurations for other bundles as well as manipulate them. The action string "configure".

104.12.8.2 public static final String TARGET = "target"

The permission to be updated, that is, act as a Managed Service or Managed Service Factory. The action string "target".

Since 1.4

104.12.8.3 public ConfigurationPermission(String name,String actions)

name Name of the permission. Wildcards ('*') are allowed in the name. During implies(Permission), the name is matched to the requested permission using the substring matching rules used by Filters.

actions Comma separated list of CONFIGURE, TARGET (case insensitive).

☐ Create a new ConfigurationPermission.

104.12.8.4 public boolean equals(Object obj)

obj The object being compared for equality with this object.

☐ Determines the equality of two ConfigurationPermission objects.

Two ConfigurationPermission objects are equal.

Returns true if obj is equivalent to this ConfigurationPermission; false otherwise.

104.12.8.5 public String getActions()

☐ Returns the canonical string representation of the ConfigurationPermission actions.

Always returns present ConfigurationPermission actions in the following order: "configure", "target"

Returns Canonical string representation of the ConfigurationPermission actions.

104.12.8.6 public int hashCode()

☐ Returns the hash code value for this object.

Returns Hash code value for this object.

104.12.8.7 public boolean implies(Permission p)

p The target permission to check.

☐ Determines if a ConfigurationPermission object "implies" the specified permission.

Returns true if the specified permission is implied by this object; false otherwise.

104.12.8.8 public PermissionCollection newPermissionCollection()

☐ Returns a new PermissionCollection object suitable for storing ConfigurationPermissions.

Returns A new `PermissionCollection` object.

104.12.9   public interface ConfigurationPlugin

A service interface for processing configuration dictionary before the update.

A bundle registers a `ConfigurationPlugin` object in order to process configuration updates before they reach the Managed Service or Managed Service Factory. The Configuration Admin service will detect registrations of Configuration Plugin services and must call these services every time before it calls the `ManagedService` or `ManagedServiceFactory` updated method. The Configuration Plugin service thus has the opportunity to view and modify the properties before they are passed to the Managed Service or Managed Service Factory.

Configuration Plugin (plugin) services have full read/write access to all configuration information that passes through them.

The `Integer` service.cmRanking registration property may be specified. Not specifying this registration property, or setting it to something other than an `Integer`, is the same as setting it to the `Integer` zero. The `service.cmRanking` property determines the order in which plugins are invoked. Lower ranked plugins are called before higher ranked ones. In the event of more than one plugin having the same value of `service.cmRanking`, then the Configuration Admin service arbitrarily chooses the order in which they are called.

By convention, plugins with `service.cmRanking < 0` or `service.cmRanking > 1000` should not make modifications to the properties.

The Configuration Admin service has the right to hide properties from plugins, or to ignore some or all the changes that they make. This might be done for security reasons. Any such behavior is entirely implementation defined.

A plugin may optionally specify a `cm.target` registration property whose value is the PID of the Managed Service or Managed Service Factory whose configuration updates the plugin is intended to intercept. The plugin will then only be called with configuration updates that are targeted at the Managed Service or Managed Service Factory with the specified PID. Omitting the `cm.target` registration property means that the plugin is called for all configuration updates.

Concurrency Thread-safe

104.12.9.1   public static final String CM_RANKING = "service.cmRanking"

A service property to specify the order in which plugins are invoked. This property contains an `Integer` ranking of the plugin. Not specifying this registration property, or setting it to something other than an `Integer`, is the same as setting it to the `Integer` zero. This property determines the order in which plugins are invoked. Lower ranked plugins are called before higher ranked ones.

Since 1.2

104.12.9.2   public static final String CM_TARGET = "cm.target"

A service property to limit the Managed Service or Managed Service Factory configuration dictionaries a Configuration Plugin service receives. This property contains a `String[]` of PIDs. A Configuration Admin service must call a Configuration Plugin service only when this property is not set, or the target service's PID is listed in this property.

104.12.9.3   public void modifyConfiguration(ServiceReference<?> reference,Dictionary<String,Object> properties)

reference reference to the Managed Service or Managed Service Factory

properties The configuration properties. This argument must not contain the "service.bundleLocation" property. The value of this property may be obtained from the `Configuration.getBundleLocation` method.

☐ View and possibly modify the a set of configuration properties before they are sent to the Managed Service or the Managed Service Factory. The Configuration Plugin services are called in increasing

order of their service.cmRanking property. If this property is undefined or is a non‑Integer type, 0 is used.

This method should not modify the properties unless the service.cmRanking of this plugin is in the range 0 <= service.cmRanking <= 1000.

If this method throws any Exception, the Configuration Admin service must catch it and should log it.

A Configuration Plugin will only be called for properties from configurations that have a location for which the Configuration Plugin has permission when security is active. When security is not active, no filtering is done.

104.12.10   public interface ManagedService

A service that can receive configuration data from a Configuration Admin service.

A Managed Service is a service that needs configuration data. Such an object should be registered with the Framework registry with the service.pid property set to some unique identifier called a PID.

If the Configuration Admin service has a Configuration object corresponding to this PID, it will call-back the updated() method of the ManagedService object, passing the properties of that Configuration object.

If it has no such Configuration object, then it calls back with a null properties argument. Registering a Managed Service will always result in a callback to the updated() method provided the Configuration Admin service is, or becomes active. This callback must always be done asynchronously.

Else, every time that either of the updated() methods is called on that Configuration object, the ManagedService.updated() method with the new properties is called. If the delete() method is called on that Configuration object, ManagedService.updated() is called with a null for the properties parameter. All these callbacks must be done asynchronously.

The following example shows the code of a serial port that will create a port depending on configuration information.

```
class SerialPort implements ManagedService {

  ServiceRegistration registration;
  Hashtable configuration;
  CommPortIdentifier id;

  synchronized void open(CommPortIdentifier id,
  BundleContext context) {
    this.id = id;
    registration = context.registerService(
      ManagedService.class.getName(),
      this,
      getDefaults()
    );
  }

  Hashtable getDefaults() {
    Hashtable defaults = new Hashtable();
    defaults.put( "port", id.getName() );
    defaults.put( "product", "unknown" );
    defaults.put( "baud", "9600" );
    defaults.put( Constants.SERVICE_PID,
```

```
          "com.acme.serialport." + id.getName() );
        return defaults;
      }

      public synchronized void updated(
        Dictionary configuration  ) {
        if ( configuration == null )
          registration.setProperties( getDefaults() );
        else {
          setSpeed( configuration.get("baud") );
          registration.setProperties( configuration );
        }
      }
      ...
    }
```

As a convention, it is recommended that when a Managed Service is updated, it should copy all the properties it does not recognize into the service registration properties. This will allow the Configuration Admin service to set properties on services which can then be used by other applications.

Normally, a single Managed Service for a given PID is given the configuration dictionary, this is the configuration that is bound to the location of the registering bundle. However, when security is on, a Managed Service can have Configuration Permission to also be updated for other locations.

Concurrency Thread-safe

104.12.10.1   **public void updated(Dictionary<String,?> properties) throws ConfigurationException**

properties A copy of the Configuration properties, or null . This argument must not contain the "service.bundleLocation" property. The value of this property may be obtained from the Configuration.getBundleLocation method.

☐ Update the configuration for a Managed Service.

When the implementation of updated(Dictionary) detects any kind of error in the configuration properties, it should create a new ConfigurationException which describes the problem. This can allow a management system to provide useful information to a human administrator.

If this method throws any other Exception, the Configuration Admin service must catch it and should log it.

The Configuration Admin service must call this method asynchronously with the method that initiated the callback. This implies that implementors of Managed Service can be assured that the callback will not take place during registration when they execute the registration in a synchronized method.

If the location allows multiple managed services to be called back for a single configuration then the callbacks must occur in service ranking order. Changes in the location must be reflected by deleting the configuration if the configuration is no longer visible and updating when it becomes visible.

If no configuration exists for the corresponding PID, or the bundle has no access to the configuration, then the bundle must be called back with a null to signal that CM is active but there is no data.

Throws ConfigurationException– when the update fails

Security ConfigurationPermission[c.location,TARGET]] – Required by the bundle that registered this service

104.12.11   **public interface ManagedServiceFactory**

Manage multiple service instances. Bundles registering this interface are giving the Configuration Admin service the ability to create and configure a number of instances of a service that the imple-

menting 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 );
  }
  ...
}
```

Concurrency Thread-safe

104.12.11.1 **public void deleted(String pid)**

pid the PID of the service to be removed

☐ Remove a factory instance. Remove the factory instance associated with the PID. If the instance was registered with the service registry, it should be unregistered. The Configuration Admin must call deleted for each instance it received in updated(String, Dictionary).

If this method throws any Exception, the Configuration Admin service must catch it and should log it.

The Configuration Admin service must call this method asynchronously.

104.12.11.2 **public String getName()**

☐ Return a descriptive name of this factory.

Returns the name for the factory, which might be localized

104.12.11.3 **public void updated(String pid,Dictionary<String,?> properties) throws ConfigurationException**

pid The PID for this configuration.

properties A copy of the configuration properties. This argument must not contain the service.bundleLocation" property. The value of this property may be obtained from the Configuration.getBundleLocation method.

☐ Create a new instance, or update the configuration of an existing instance. If the PID of the Configuration object is new for the Managed Service Factory, then create a new factory instance, using the configuration properties provided. Else, update the service instance with the provided properties.

If the factory instance is registered with the Framework, then the configuration properties should be copied to its registry properties. This is not mandatory and security sensitive properties should obviously not be copied.

If this method throws any Exception, the Configuration Admin service must catch it and should log it.

When the implementation of updated detects any kind of error in the configuration properties, it should create a new ConfigurationException which describes the problem.

The Configuration Admin service must call this method asynchronously. This implies that implementors of the ManagedServiceFactory class can be assured that the callback will not take place during registration when they execute the registration in a synchronized method.

If the security allows multiple managed service factories to be called back for a single configuration then the callbacks must occur in service ranking order.

It is valid to create multiple factory instances that are bound to different locations. Managed Service Factory services must only be updated with configurations that are bound to their location or that start with the ? prefix and for which they have permission. Changes in the location must be reflected by deleting the corresponding configuration if the configuration is no longer visible or updating when it becomes visible.

Throws ConfigurationException– when the configuration properties are invalid.

Security ConfigurationPermission[c.location,TARGET]] – Required by the bundle that registered this service

104.12.12 **public interface SynchronousConfigurationListener**
extends ConfigurationListener

Synchronous Listener for Configuration Events. When a ConfigurationEvent is fired, it is synchronously delivered to all SynchronousConfigurationListeners.

SynchronousConfigurationListener objects are registered with the Framework service registry and are synchronously notified with a ConfigurationEvent object when an event is fired.

SynchronousConfigurationListener objects can inspect the received ConfigurationEvent object to determine its type, the PID of the Configuration object with which it is associated, and the Configuration Admin service that fired the event.

Security Considerations. Bundles wishing to synchronously monitor configuration events will require ServicePermission[SynchronousConfigurationListener,REGISTER] to register a Synchronous-ConfigurationListener service.

Since 1.5

Concurrency Thread-safe

104.13   Changes

- Clarified that collection property values may have an ordering that must be preserved.
- Changed vector references to list.
- Clarified that a if Managed Service is registered with more than one PID and more than one PID has no configuration information available, then updated(Dictionary) can be called multiple time with a null parameter.

105   Metatype Service Specification

Version 1.3

105.1   Introduction

The Metatype specification defines interfaces that allow bundle developers to describe attribute types in a computer readable form using so-called *metadata*.

The purpose of this specification is to allow services to specify the type information of data that they can use as arguments. The data is based on *attributes*, which are key/value pairs like properties.

A designer in a type-safe language like Java is often confronted with the choice of using the language constructs to exchange data or using a technique based on attributes/properties that are based on key/value pairs. Attributes provide an escape from the rigid type-safety requirements of modern programming languages.

Type-safety works very well for software development environments in which multiple programmers work together on large applications or systems, but often lacks the flexibility needed to receive structured data from the outside world.

The attribute paradigm has several characteristics that make this approach suitable when data needs to be communicated between different entities which "speak" different languages. Attributes are uncomplicated, resilient to change, and allow the receiver to dynamically adapt to different types of data.

As an example, the OSGi framework 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 61. A Configuration Admin service implementation deploys attributes (key/value pairs) as configuration properties.

The Meta Type Service provides a unified access point to the Meta Type information that is associated with bundles. This Meta Type information can be defined by an XML resource in a bundle (OSGI-INF/metatype directories must be scanned for any XML resources), it can come from the Meta Type Provider service, or it can be obtained from Managed Service or Managed Service Factory services.

105.1.1   Essentials
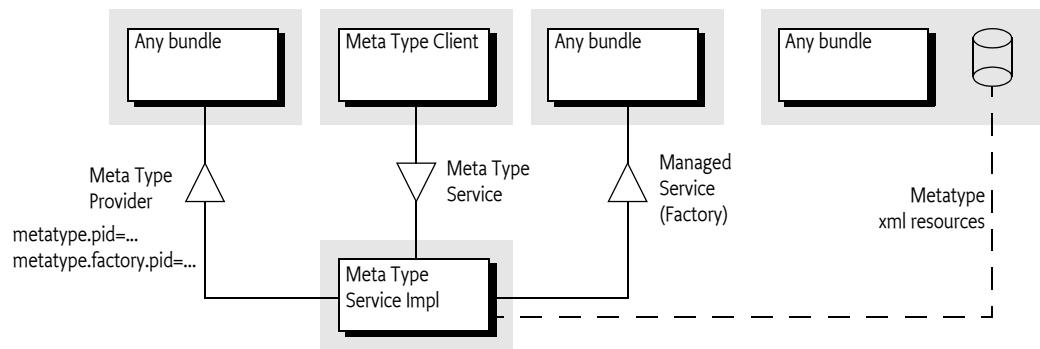
- *Conceptual model* - The specification must have a conceptual model for how classes and attributes are organized.
- *Standards* - The specification should be aligned with appropriate standards, and explained in situations where the specification is not aligned with, or cannot be mapped to, standards.
- *Remote Management* - Remote management should be taken into account.
- *Size* - Minimal overhead in size for a bundle using this specification is required.
- *Localization* - It must be possible to use this specification with different languages at the same time. This ability allows servlets to serve information in the language selected in the browser.
- *Type information* - The definition of an attribute should contain the name (if it is required), the cardinality, a label, a description, labels for enumerated values, and the Java class that should be used for the values.
- *Validation* - It should be possible to validate the values of the attributes.

105.1.2   Entities

- *Meta Type Service* - A service that provides a unified access point for meta type information.
- *Attribute* - A key/value pair.
- *PID* - A unique persistent ID, defined in configuration management.
- *Attribute Definition* - Defines a description, name, help text, and type information of an attribute.
- *Object Class Definition* - Defines the type of a datum. It contains a description and name of the type plus a set of AttributeDefinition objects.
- *Meta Type Provider* - Provides access to the object classes that are available for this object. Access uses the PID and a locale to find the best ObjectClassDefinition object.
- *Meta Type Information* - Provides meta type information for a bundle.

Figure 105.1   Class Diagram Meta Type Service, org.osgi.service.metatype

105.1.3   Operation

The Meta Type service defines a rich dynamic typing system for properties. The purpose of the type system is to allow reasonable User Interfaces to be constructed dynamically.

The type information is normally carried by the bundles themselves. Either by implementing the MetaTypeProvider interface on the Managed Service or Managed Service Factory, by carrying one or more XML resources that define a number of Meta Types in the OSGI-INF/metatype directories, or registering a Meta Type Provider as a service. Additionally, a Meta Type service could have other sources that are not defined in this specification.

The Meta Type Service provides unified access to Meta Types that are carried by the resident bundles. The Meta Type Service collects this information from the bundles and provides uniform access to it. A client can requests the Meta Type Information associated with a particular bundle. The MetaTypeInformation object provides a list of ObjectClassDefinition objects for a bundle. These objects define all the information for a specific *object class*. An object class is a some descriptive information and a set of named attributes (which are key/value pairs).

Access to Object Class Definitions is qualified by a locale and a Persistent IDentity (PID). This specification does not specify what the PID means. One application is OSGi Configuration Management where a PID is used by the Managed Service and Managed Service Factory services. In general, a PID should be regarded as the name of a variable where an Object Class Definition defines its type.

105.2   Attributes Model

The Framework uses the LDAP filter syntax for searching the Framework registry. The usage of the attributes in this specification and the Framework specification closely resemble the LDAP attribute

model. Therefore, the names used in this specification have been aligned with LDAP. Consequently, the interfaces which are defined by this Specification are:

- AttributeDefinition
- ObjectClassDefinition
- MetaTypeProvider

These names correspond to the LDAP attribute model. For further information on ASN.1-defined attributes and X.500 object classes and attributes, see [2] *Understanding and Deploying LDAP Directory services*.

The LDAP attribute model assumes a global name-space for attributes, and object classes consist of a number of attributes. So, if an object class inherits the same attribute from different parents, only one copy of the attribute must become part of the object class definition. This name-space implies that a given attribute, for example cn, should *always* be the common name and the type must always be a String. An attribute cn cannot be an Integer in another object class definition. In this respect, the OSGi approach towards attribute definitions is comparable with the LDAP attribute model.

105.3   Object Class Definition

The ObjectClassDefinition interface is used to group the attributes which are defined in AttributeDefinition objects.

An ObjectClassDefinition object contains the information about the overall set of attributes and has the following elements:

- A name which can be returned in different locales.
- A global name-space in the registry, which is the same condition as LDAP/X.500 object classes. In these standards the OSI Object Identifier (OID) is used to uniquely identify object classes. If such an OID exists, (which can be requested at several standard organizations, and many companies already have a node in the tree) it can be returned here. Otherwise, a unique id should be returned. This id can be a Java class name (reverse domain name) or can be generated with a GUID algorithm. All LDAP-defined object classes already have an associated OID. It is strongly advised to define the object classes from existing LDAP schemes which provide many preexisting OIDs. Many such schemes exist ranging from postal addresses to DHCP parameters.
- A human-readable description of the class.
- A list of attribute definitions which can be filtered as required, or optional. Note that in X.500 the mandatory or required status of an attribute is part of the object class definition and not of the attribute definition.
- An icon, in different sizes.

105.4   Attribute Definition

The AttributeDefinition interface provides the means to describe the data type of attributes.

The AttributeDefinition interface defines the following elements:

- Defined names (final ints) for the data types as restricted in the Framework for the attributes, called the syntax in OSI terms, which can be obtained with the getType() method.
- AttributeDefinition objects should use an ID that is similar to the OID as described in the ID field for ObjectClassDefinition.
- A localized name intended to be used in user interfaces.

- A localized description that defines the semantics of the attribute and possible constraints, which should be usable for tooltips.
- An indication if this attribute should be stored as a unique value, a List, or an array of values, as well as the maximum cardinality of the type.
- The data type, as limited by the Framework service registry attribute types.
- A validation function to verify if a possible value is correct.
- A list of values and a list of localized labels. Intended for popup menus in GUIs, allowing the user to choose from a set.
- A default value (String[]). The return depends on the following cases:
  - *not specified* - Return null if this attribute is not specified.
  - *cardinality = 0* - Return an array with one element.
  - *otherwise* - Return an array with less or equal than the absolute value of cardinality, possibly empty if the value is an empty string.
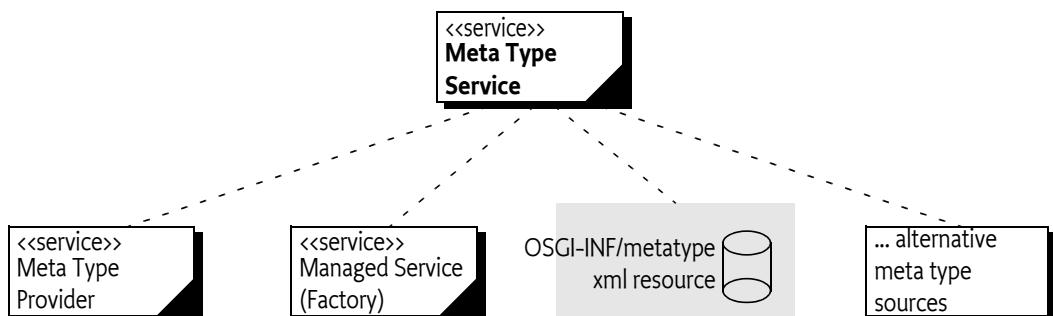
105.5   Meta Type Service

The Meta Type Service provides unified access to Meta Type information that is associated with a Bundle. It can get this information through the following means:

- *Meta Type Resource* - A bundle can provide one or more XML resources that are contained in its JAR file. These resources contain an XML definition of meta types as well as to what PIDs these Meta Types apply. These XML resources must reside in the OSGI-INF/metatype directories of the bundle (including any fragments).
- *Managed Service [Factory] objects* - As defined in the configuration management specification, ManagedService and ManagedServiceFactory service objects can optionally implement the MetaTypeProvider interface. The Meta Type Service will only search for ManagedService and ManagedServiceFactory service objects that implement MetaTypeProvider if no meta type resources are found in the bundle.
- *Meta Type Provider service* - Bundles can register Meta Type Provider services to dynamically provide meta types for PIDs and factory PIDs.

Figure 105.2   *Sources for Meta Types*

This model is depicted in Figure 105.2.

The Meta Type Service can therefore be used to retrieve meta type information for bundles which contain Meta Type resources or which provide MetaTypeProvider objects and/or services. If multiple sources define the same Object Class Definition, the Meta Type service must select which source to use. Meta Type Provider services must take precedence over Managed Service [Factory] objects implementing MetaTypeProvider or Meta Type Resources.

The MetaTypeService interface has a single method:

- getMetaTypeInformation(Bundle) - Given a bundle, it must return the Meta Type Information for that bundle, even if there is no meta type information available at the moment of the call.

The returned MetaTypeInformation object maintains a map of PID to ObjectClassDefinition objects. The map is keyed by locale and PID. The list of maintained PIDs is available from the MetaTypeInformation object with the following methods:

- getPids() - PIDs for which Meta Types are available.
- getFactoryPids() - PIDs associated with Managed Service Factory services.

These methods and their interaction with the Meta Type resource are described in *Designate Element* on page 114.

The MetaTypeInformation interface extends the MetaTypeProvider interface. The MetaType-Provider interface is used to access meta type information. It supports locale dependent information so that the text used in AttributeDefinition and ObjectClassDefinition objects can be adapted to different locales.

Which locales are supported by the MetaTypeProvider object are defined by the implementer or the meta type resources. The list of available locales can be obtained from the MetaTypeProvider object.

The MetaTypeProvider interface provides the following methods:

- getObjectClassDefinition(String,String) - Get access to an ObjectClassDefinition object for the given PID. The second parameter defines the locale.
- getLocales() - List the locales that are available.

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

```
locale = language ( '_' country ( '_' variation))
language ::= < defined by ISO 3166 >
country  ::= < defined by ISO 639 >
```

For example, en, nl_BE, en_CA_posix are valid locales. The use of null for locale indicates that java.util.Locale.getDefault() must be used.

The Meta Type Service implementation class is the main class. It registers the org.osgi.service.metatype.MetaTypeService service and has a method to get a MetaTypeInforma-tion object for a bundle.

Following is some sample code demonstrating how to print out all the Object Class Definitions and Attribute Definitions contained in a bundle:

```
void printMetaTypes( MetaTypeService mts,Bundle b ) {
    MetaTypeInformation mti =
        mts.getMetaTypeInformation(b);
    String [] pids = mti.getPids();
    String [] locales = mti.getLocales();

    for ( int locale = 0; locale<locales.length; locale++) {
        System.out.println("Locale " + locales[locale] );
        for (int i=0; i< pids.length; i++) {
            ObjectClassDefinition ocd =
                mti.getObjectClassDefinition(pids[i], null);
            AttributeDefinition[] ads =
                ocd.getAttributeDefinitions(
                    ObjectClassDefinition.ALL);
            for (int j=0; j< ads.length; j++) {
```

```
                          System.out.println("OCD="+ocd.getName()
                              + "AD="+ads[j].getName());
                      }
                  }
              }
          }
```

105.6   Meta Type Provider Service

A Meta Type Provider service allows third party contributions to the internal Object Class Definition repository. A Meta Type Provider can contribute multiple PIDs, both factory and singleton PIDs. A Meta Type Provider service must register with both or one of the following service properties:

- METATYPE_PID - (String+) Provides a list of PIDs that this Meta Type Provider can provide Object Class Definitions for. The listed PIDs are intended to be used as normal singleton PIDs used by Managed Services.
- METATYPE_FACTORY_PID - (String+) Provides a list of factory PIDs that this Meta Type Provider can provide Object Class Definitions for. The listed PIDs are intended to be used as factory PIDs used by Managed Service Factories.

The Object Class Definitions must originate from the bundle that registered the Meta Type Provider service. Third party extenders should therefore use the bundle of their extendee. A Meta Type Service must report these Object Class Definitions on the Meta Type Information of the registering bundle, merged with any other information from that bundle.

The Meta Type Service must track these Meta Type Provider services and make their Meta Types available as if they were provided on the Managed Service (Factory) services. The Meta Types must become unavailable when the Meta Type Provider service is unregistered.

105.7   Using the Meta Type Resources

A bundle that wants to provide meta type resources must place these resources in the OSGI-INF/metatype directory. The name of the resource must be a valid bundle entry path. All resources in that directory must be meta type documents. Fragments can contain additional meta type resources in the same directory and they must be taken into account when the meta type resources are searched. A meta type resource must be encoded in UTF-8.

The MetaType Service must support localization of the

- name
- icon
- description
- label attributes

The localization mechanism must be identical using the same mechanism as described in the Core module layer, see *Localization*, using the same property resource. However, it is possible to override the property resource in the meta type definition resources with the localization attribute of the MetaData element.

The Meta Type Service must examine the bundle and its fragments to locate all localization resources for the localization base name. From that list, the Meta Type Service derives the list of locales which are available for the meta type information. This list can then be returned by MetaTypeInformation.getLocales method. This list can change at any time because the bundle could be refreshed. Clients should be prepared that this list changes after they received it.

105.7.1   XML Schema of a Meta Type Resource

This section describes the schema of the meta type resource. This schema is not intended to be used during runtime for validating meta type resources. The schema is intended to be used by tools and external management systems.

The XML namespace for meta type documents must be:

```
http://www.osgi.org/xmlns/metatype/v1.3.0
```
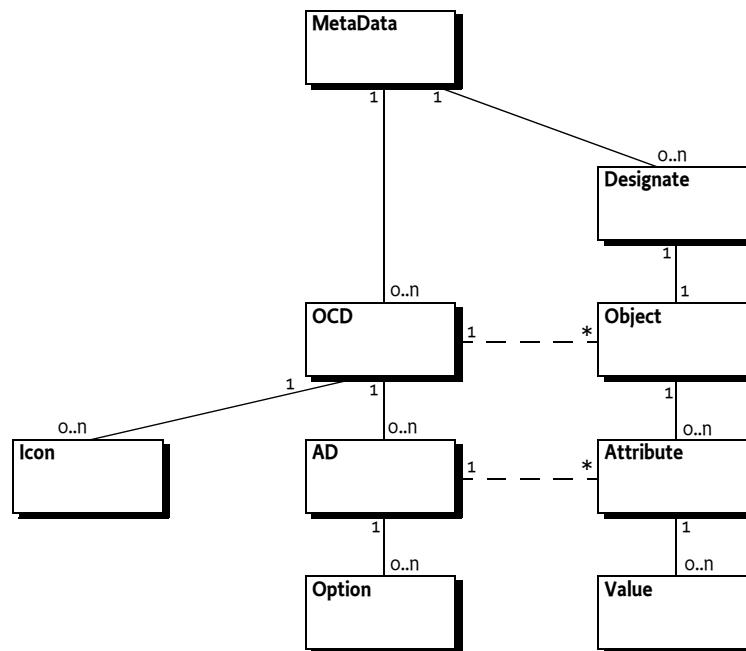
The namespace abbreviation should be `metatype`. That is, the following header should be:

```
<metatype:MetaData
    xmlns:metatype="http://www.osgi.org/xmlns/metatype/v1.3.0">
```

The file can be found in the osgi.jar file that can be downloaded from the www.osgi.org web site.

Figure 105.3   *XML Schema Instance Structure (Type name = Element name)*

The element structure of the XML file is:

```
MetaData    ::= OCD* Designate*

OCD         ::= AD*  Icon*
AD          ::= Option*

Designate   ::= Object
Object      ::= Attribute*

Attribute   ::= Value*
```

The different elements are described in Table 105.1.

Table 105.1   *XML Schema for Meta Type resources*

Attribute	Deflt	Type	Method	Description
MetaData				Top Element
localization		string		Points to the Properties file that can localize this XML. See *Localization* in *OSGi Core Release 6.*
OCD				Object Class Definition
name	<>	string	getName()	A human readable name that can be localized.
description			getDescription()	A human readable description of the Object Class Definition that can be localized.
id	<>		getID()	A unique id, cannot be localized.
Designate				An association between one PID and an Object Class Definition. This element *designates* a PID to be of a certain *type*.
pid	<>	string		The PID that is associated with an OCD. This can be a reference to a factory or singleton configuration object. The PID can be a Targeted PID, if factoryPid is not set or empty. Either pid or factoryPid must be specified. See *Designate Element* on page 114.
factoryPid		string		If the factoryPid attribute is set, this Designate element defines a factory configuration for the given factory. If it is not set or empty, it designates a singleton configuration. The PID can be a Targeted PID. Either pid or factoryPid must be specified. See *Designate Element* on page 114.
bundle		string		The value is used to set the location of any configuration created using this Meta Type resource. This may contain a bundle location or a multi-location. In a Meta Type resource, using the wildcard value ('*' \u002A) indicates the bundle location of the bundle containing the resource must be used as the location. See *Location Binding* on page 67
				This is an optional attribute but can be mandatory in certain usage schemes, for example the Autoconf Resource Processor.
optional	false	boolean		If true, then this Designate element is optional, errors during processing must be ignored.
merge	false	boolean		If the PID refers to an existing configuration, then merge the properties with the existing properties if this attribute is true. Otherwise, replace the properties.

Attribute	Deflt	Type	Method	Description
AD				Attribute Definition
name		string	getName()	A localizable name for the Attribute Definition. description
description		string	getDescription()	A localizable description for the Attribute Definition.
id			getID()	The unique ID of the Attribute Definition.
type		string	getType()	The type of an attribute is an enumeration of the different scalar types. The string is mapped to one of the constants on the AttributeDefinition interface. Valid values, which are defined in the Scalar type, are: String ↔ STRING Long ↔ LONG Double ↔ DOUBLE Float ↔ FLOAT Integer ↔ INTEGER Byte ↔ BYTE Char ↔ CHARACTER Boolean ↔ BOOLEAN Short ↔ SHORT Password ↔ PASSWORD
cardinality	0		getCardinality()	The number of elements an instance can take. Positive numbers describe an array ([]) and negative numbers describe a List object.
min		string	validate(String)	A validation value. This value is not directly available from the AttributeDefinition interface. However, the validate(String) method must verify this. The semantics of this field depend on the type of this Attribute Definition.
max		string	validate(String)	A validation value. Similar to the min field. When min or max are numbers, attribute values with a numeric data type are valid if min <= value <= max. Attribute values with a string (or equivalent) data type are valid if min <= value.length() <= max.

Attribute	Deflt	Type	Method	Description
default		string	getDefaultValue()	The default value. A default is an array of String objects. The XML attribute must contain a comma delimited list. The default value is trimmed and escaped in the same way as described in the validate(String) method. The empty string is significant and must be seen as an empty List or array if specified as the default for an attribute with a cardinality that is not equal to zero. Default values must be valid or otherwise ignored.
required	true	boolean		Required attribute. The required attribute indicates whether or not the attribute key must appear within the configuration dictionary to be valid.
Option				One option label/value for the options in an AD. Options are exclusive. The validate(String) method must verify that an attribute value matches one of the option values when present.
label	<>	string	getOptionLabels()	The label
value	<>	string	getOptionValues()	The value
Icon				An icon definition.
resource	<>	string	getIcon(int)	The resource is a URL. The base URL is assumed to be the root of the bundle containing the XML file. That is, this URL can reference another resource in the bundle using a relative URL.
size	<>	string	getIcon(int)	The number of pixels of the icon, maps to the size parameter of the getIcon(int) method.
Object				A definition of an instance.
ocdref	<>	string		A reference to the id attribute of an OCD element. That is, this attribute defines the OCD type of this object.
Attribute				A value for an attribute of an object.
adref	<>	string		A reference to the id of the AD in the OCD as referenced by the parent Object.
content		string		The content of the attributes. If this is an array, the content must be separated by commas (',' \u002C). Commas must be escaped as described at the default attribute of the AD element.
Value				Holds a single value. This element can be repeated multiple times under an Attribute

105.7.2 Designate Element

For the MetaType Service, the Designate definition is used to declare the available PIDs and factory PIDs; the Attribute elements are never used by the MetaType service.

The getPids() method returns an array of PIDs that were specified in the pid attribute of the Object elements. The getFactoryPids() method returns an array of the factoryPid attributes. For factories, the related pid attribute is ignored because all instances of a factory must share the same meta type.

The following example shows a metatype reference to a singleton configuration and a factory configuration.

```
<Designate pid="com.acme.designate.1">
    <Object ocdref="com.acme.designate"/>
</Designate>
<Designate factoryPid="com.acme.designate.factory"
    bundle="*">
    <Object ocdref="com.acme.designate"/>
</Designate>
```

Other schemes can embed the Object element in the Designate element to define actual instances for the Configuration Admin service. In that case the pid attribute must be used together with the factoryPid attribute. However, in that case an aliasing model is required because the Configuration Admin service does not allow the creator to choose the Configuration object's PID.

105.7.3 Example Metadata File

This example defines a meta type file for a Person record, based on ISO attribute types. The ids that are used are derived from ISO attributes.

```
<?xml version="1.0" encoding="UTF-8"?>
<MetaData
    xmlns="http://www.osgi.org/xmlns/metatype/v1.3.0"
    localization="person">
  <OCD name="%person" id="2.5.6.6"
       description="%person record">
    <AD name="%sex" id="2.5.4.12" type="Integer">
      <Option label="%male" value="1"/>
      <Option label="%female" value="0"/>
    </AD>
    <AD name="%sn" id="2.5.4.4" type="String"/>
    <AD name="%cn" id="2.5.4.3" type="String"/>
    <AD name="%seeAlso" id="2.5.4.34" type="String"
        cardinality="8"
        default="http://www.google.com,http://www.yahoo.com"/>
    <AD name="%telNumber" id="2.5.4.20" type="String"/>
  </OCD>

  <Designate pid="com.acme.addressbook">
    <Object ocdref="2.5.6.6"/>
  </Designate>
</MetaData>
```

Translations for this file, as indicated by the localization attribute must be stored in the root directory (e.g. person_du_NL.properties). The default localization base name for the properties is OSGI-INF/l10n/bundle, but can be overridden by the manifest Bundle-Localization header and the localization attribute of the Meta Data element. The property files have the base name of person. The Dutch, French and English translations could look like:

person_du_NL.properties:

```
person=Persoon
person\ record=Persoons beschrijving
```

```
cn=Naam
sn=Voornaam
seeAlso=Zie ook
telNumber=Tel. Nummer
sex=Geslacht
male=Mannelijk
female=Vrouwelijk

person_fr.properties:

person=Personne
person\ record=Description de la personne
cn=Nom
sn=Surnom
seeAlso=Reference
telNumber=Tel.
sex=Sexe
male=Homme
female=Femme

person_en_US.properties:

person=Person
person\ record=Person Record
cn=Name
sn=Sur Name
seeAlso=See Also
telNumber=Tel.
sex=Sex
male=Male
female=Female
```

105.7.4   Object Element

The OCD element can be used to describe the possible contents of a Dictionary object. In this case, the attribute name is the key. The Object element can be used to assign a value to a Dictionary object.

For example:

```
<Designate pid="com.acme.b">
  <Object ocdref="b">
    <Attribute adref="foo" content="Zaphod Beeblebrox"/>
    <Attribute adref="bar">
      <Value>1</Value>
      <Value>2</Value>
      <Value>3</Value>
      <Value>4</Value>
      <Value>5</Value>
    </Attribute>
  </Object>
</Designate>
```

105.8   Meta Type Resource XML Schema

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:metatype="http://www.osgi.org/xmlns/metatype/v1.3.0"
```

```
    targetNamespace="http://www.osgi.org/xmlns/metatype/v1.3.0"
    version="1.3.0">

    <element name="MetaData" type="metatype:Tmetadata" />

    <complexType name="Tmetadata">
        <choice minOccurs="0" maxOccurs="unbounded">
            <element name="OCD" type="metatype:Tocd" />
            <element name="Designate" type="metatype:Tdesignate" />
            <!-- It is non-deterministic, per W3C XML Schema 1.0: http://www.w3.org/TR/xmlschema-1/#cos-nonambig
                 to use namespace="##any" below. -->
            <any namespace="##other" processContents="lax" />
        </choice>
        <attribute name="localization" type="string" use="optional" />
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Tocd">
        <choice minOccurs="0" maxOccurs="unbounded">
            <element name="AD" type="metatype:Tad" />
            <element name="Icon" type="metatype:Ticon" />
            <!-- It is non-deterministic, per W3C XML Schema 1.0: http://www.w3.org/TR/xmlschema-1/#cos-nonambig
                 to use namespace="##any" below. -->
            <any namespace="##other" processContents="lax" />
        </choice>
        <attribute name="name" type="string" use="required" />
        <attribute name="description" type="string" use="optional" />
        <attribute name="id" type="string" use="required" />
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Tad">
        <choice minOccurs="0" maxOccurs="unbounded">
            <element name="Option" type="metatype:Toption" />
            <!-- It is non-deterministic, per W3C XML Schema 1.0: http://www.w3.org/TR/xmlschema-1/#cos-nonambig
                 to use namespace="##any" below. -->
            <any namespace="##other" processContents="lax"  />
        </choice>
        <attribute name="name" type="string" use="optional" />
        <attribute name="description" type="string" use="optional" />
        <attribute name="id" type="string" use="required" />
        <attribute name="type" type="metatype:Tscalar" use="required" />
        <attribute name="cardinality" type="int" use="optional"
            default="0" />
        <attribute name="min" type="string" use="optional" />
        <attribute name="max" type="string" use="optional" />
        <attribute name="default" type="string" use="optional" />
        <attribute name="required" type="boolean" use="optional"
            default="true" />
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Tobject">
        <choice minOccurs="0" maxOccurs="unbounded">
            <element name="Attribute" type="metatype:Tattribute" />
            <!-- It is non-deterministic, per W3C XML Schema 1.0: http://www.w3.org/TR/xmlschema-1/#cos-nonambig
                 to use namespace="##any" below. -->
            <any namespace="##other" processContents="lax" />
        </choice>
        <attribute name="ocdref" type="string" use="required" />
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Tattribute">
        <choice minOccurs="0" maxOccurs="unbounded">
            <element name="Value" type="string" />
            <!-- It is non-deterministic, per W3C XML Schema 1.0: http://www.w3.org/TR/xmlschema-1/#cos-nonambig
                 to use namespace="##any" below. -->
            <any namespace="##other" processContents="lax" />
        </choice>
        <attribute name="adref" type="string" use="required" />
        <attribute name="content" type="string" use="optional" />
        <anyAttribute processContents="lax" />
    </complexType>
```

```
    <complexType name="Tdesignate">
        <sequence>
            <element name="Object" type="metatype:Tobject" minOccurs="1"
                maxOccurs="1" />
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="pid" type="string" use="optional" />
        <attribute name="factoryPid" type="string" use="optional" />
        <attribute name="bundle" type="string" use="optional" />
        <attribute name="optional" type="boolean" default="false"
            use="optional" />
        <attribute name="merge" type="boolean" default="false"
            use="optional" />
        <anyAttribute processContents="lax" />
    </complexType>

    <simpleType name="Tscalar">
        <restriction base="string">
            <enumeration value="String" />
            <enumeration value="Long" />
            <enumeration value="Double" />
            <enumeration value="Float" />
            <enumeration value="Integer" />
            <enumeration value="Byte" />
            <enumeration value="Character" />
            <enumeration value="Boolean" />
            <enumeration value="Short" />
            <enumeration value="Password" />
        </restriction>
    </simpleType>

    <complexType name="Toption">
        <sequence>
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="label" type="string" use="required" />
        <attribute name="value" type="string" use="required" />
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Ticon">
        <sequence>
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="resource" type="string" use="required" />
        <attribute name="size" type="positiveInteger" use="required" />
        <anyAttribute processContents="lax" />
    </complexType>

    <attribute name="must-understand" type="boolean">
        <annotation>
            <documentation xml:lang="en">
                This attribute should be used by extensions to documents
                to require that the document consumer understand the
                extension.
            </documentation>
        </annotation>
    </attribute>
</schema>
```

105.9 Meta Type Annotations

A developer can use Meta Type Annotations on a Component Property Type, see *Component Property Types* on page 216, or an interface to define an Object Class Definition in a type safe manner. The Meta Type Annotations are CLASS retention annotations intended to be used during build time to generate Meta Type Resources from the Java class files providing a convenient way to create the Meta Type Resource XML documents.

Tools processing these annotations must always generate valid Meta Type Resource XML documents. If the Meta Type Annotations are used in a way that is not supported or in error, then the tool must report the error to enable the developer to take corrective action.

105.9.1 ObjectClassDefinition Annotation

The ObjectClassDefinition annotation can be applied to a Component Property Type or an interface. From that type, tooling can generate an OCD element. When applied to an interface, all the methods inherited from supertypes are include as Attribute Definitions. The tool processing the annotations must be able to examine all the types in the hierarchy of the annotated type to generate the Meta Type Resource. It is an error if the tool cannot examine a type in the hierarchy.

It is an error to apply the ObjectClassDefinition annotation to concrete and abstract class types. It is also an error to apply it to an interface if any of the methods of the interface take arguments.

The ObjectClassDefinition annotation can be applied without defining any element values as default values for the ObjectClassDefinition annotation elements can be generated from the annotated type. For example:

```
@ObjectClassDefinition
@interface Config {
  boolean enabled();
  String[] names();
  String topic();
}
```

In the following larger example, the ObjectClassDefinition annotation defines the description and name of the OCD which are to be localized using the specified resource as well as an icon resource. Also, AttributeDefinition annotations are applied to the methods to supply some non-default values for the generated AD elements.

```
@ObjectClassDefinition(localization = "OSGI-INF/l10n/member",
    description = "%member.description",
    name = "%member.name"
    icon = @Icon(resource = "icon/member-32.png", size = 32))
@interface Member {
  @AttributeDefinition(type = AttributeType.PASSWORD,
    description = "%member.password.description",
    name = "%member.password.name")
  public String _password();

  @AttributeDefinition(options = {
    @Option(label = "%strategic", value = "strategic"),
    @Option(label = "%principal", value = "principal"),
    @Option(label = "%contributing", value = "contributing")
    },
    defaultValue = "contributing",
    description = "%member.membertype.description",
```

```
            name = "%member.membertype.name")
    public String type();
}
```

105.9.2 AttributeDefinition Annotation

The AttributeDefinition annotation is an optional annotation which can applied to the methods in a type annotated by ObjectClassDefinition. Each method of the type annotated by ObjectClassDefinition is mapped to an AD child element of the OCD element in the generated Meta Type Resource XML document. The AttributeDefinition annotation only needs to be applied to a method if values other than the defaults are desired.

The id of the Attribute Definition is generated from the method name as follows:

- A single dollar sign ('$' \u0024) is removed unless it is followed by another dollar sign in which case the two consecutive dollar signs ("$$") are converted to a single dollar sign.
- A single low line ('_' \u005F) is converted into a full stop ('.' \u002E) unless is it followed by another low line in which case the two consecutive low lines ("__") are converted to a single low line.
- All other characters are unchanged.

The generated id becomes the value of the id attribute of the AD element in the generated Meta Type Resource XML document.

105.9.3 Designate Annotation

The Designate annotation can be applied to a Declarative Services component class to make the connection between the pid of the component and an Object Class Definition. This annotation must be used on a type that is also annotated with the Declarative Services Component annotation. The component must only have a single PID which is used for the generated Designate element.

In the following example, the Designate annotation is applied to a Declarative Services component and references the Object Class Definition type.

```
@ObjectClassDefinition(id="my.config.ocd")
@interface Config {
  boolean enabled() default true;
  String[] names() default {"a", "b"};
  String topic() default "default/topic";
}

@Component(configurationPid="my.component.pid")
@Designate(ocd = Config.class)
public class MyComponent {
    static final String DEFAULT_TOPIC_PREFIX = "topic.prefix";
    protected void activate(Config configuration) {
        String t = configuration.topic();
    }
}
```

Tools processing these annotations will generate a Designate element in the generated Meta Type Resource XML document using the PID of the component and the id of the Object Class Definition. For example:

```
<Designate pid="my.component.pid">
    <Object ocdref="my.config.ocd"/>
</Designate>
```

105.10　Limitations

The OSGi MetaType specification is intended to be used for simple applications. It does not, there-fore, support recursive data types, mixed types in arrays/lists, or nested arrays/lists.

105.11　Related Standards

One of the primary goals of this specification is to make metatype information available at run-time with minimal overhead. Many related standards are applicable to metatypes; except for Java beans, however, all other metatype standards are based on document formats (e.g. XML). In the OSGi framework, 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 devel-oped 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 infor-mation as it is *implementing* this information.

In some traditional cases, a management system uses the metatype information to control objects in an OSGi framework. 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 framework. This ability is usually not a problem, because the metatype languages used by current management systems are very powerful.

105.12　Capabilities

Implementations of the Metatype Service specification must provide the following capabilities.

- A capability in the osgi.implementation namespace declaring a specification implementation with the name METATYPE_CAPABILITY_NAME. This capability must also declare a uses constraint for the org.osgi.service.metatype package. For example:

```
Provide-Capability: osgi.implementation;
    osgi.implementation="osgi.metatype";
    version:Version="1.3";
    uses:="org.osgi.service.metatype"
```

This capability must follow the rules defined for the *osgi.implementation Namespace* on page 685.

- A capability in the osgi.extender namespace declaring an extender with the name METATYPE_CAPABILITY_NAME. This capability must also declare a uses constraint for the org.osgi.service.metatype package. For example:

```
Provide-Capability: osgi.extender;
    osgi.extender="osgi.metatype";
    version:Version="1.3";
    uses:="org.osgi.service.metatype"
```

This capability must follow the rules defined for the *osgi.extender Namespace* on page 681.

- A capability in the osgi.service namespace representing the MetaTypeService service. This capability must also declare a uses constraint for the org.osgi.service.metatype package. For example:

```
Provide-Capability: osgi.service;
    objectClass:List<String>="org.osgi.service.metatype.MetaTypeService";
    uses:="org.osgi.service.metatype"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 685.

105.13 Security Considerations

Special security issues are not applicable for this specification.

105.14 org.osgi.service.metatype

Metatype Package Version 1.3.

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

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

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

105.14.1 Summary

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

105.14.2 public interface AttributeDefinition

An interface to describe an attribute.

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

Concurrency Thread-safe

105.14.2.1 public static final int BIGDECIMAL = 10

The BIGDECIMAL type. Attributes of this type should be stored as BigDecimal, List<BigDecimal> or BigDecimal[] objects depending on getCardinality().

Deprecated As of 1.1.

105.14.2.2   **public static final int BIGINTEGER = 9**

The BIGINTEGER type. Attributes of this type should be stored as BigInteger, List<BigInteger> or BigInteger[] objects, depending on the getCardinality() value.

Deprecated As of 1.1.

105.14.2.3   **public static final int BOOLEAN = 11**

The BOOLEAN type. Attributes of this type should be stored as Boolean, List<Boolean> or boolean[] objects depending on getCardinality().

105.14.2.4   **public static final int BYTE = 6**

The BYTE type. Attributes of this type should be stored as Byte, List<Byte> or byte[] objects, depending on the getCardinality() value.

105.14.2.5   **public static final int CHARACTER = 5**

The CHARACTER type. Attributes of this type should be stored as Character, List<Character> or char[] objects, depending on the getCardinality() value.

105.14.2.6   **public static final int DOUBLE = 7**

The DOUBLE type. Attributes of this type should be stored as Double, List<Double> or double[] objects, depending on the getCardinality() value.

105.14.2.7   **public static final int FLOAT = 8**

The FLOAT type. Attributes of this type should be stored as Float, List<Float> or float[] objects, depending on the getCardinality() value.

105.14.2.8   **public static final int INTEGER = 3**

The INTEGER type. Attributes of this type should be stored as Integer, List<Integer> or int[] objects, depending on the getCardinality() value.

105.14.2.9   **public static final int LONG = 2**

The LONG type. Attributes of this type should be stored as Long, List<Long> or long[] objects, depending on the getCardinality() value.

105.14.2.10   **public static final int PASSWORD = 12**

The PASSWORD type. Attributes of this type must be stored as String, List<String> or String[] objects depending on getCardinality(). A PASSWORD must be treated as a string but the type can be used to disguise the information when displayed to a user to prevent others from seeing it.

Since 1.2

105.14.2.11   **public static final int SHORT = 4**

The SHORT type. Attributes of this type should be stored as Short, List<Short> or short[] objects, depending on the getCardinality() value.

105.14.2.12   **public static final int STRING = 1**

The STRING type.

Attributes of this type should be stored as String, List<String> or String[] objects, depending on the getCardinality() value.

105.14.2.13   **public int getCardinality()**

☐ Return the cardinality of this attribute. The OSGi environment handles multi valued attributes in arrays ([]) or in List objects. The return value is defined as follows:

```
x = Integer.MIN_VALUE    no limit, but use List
x < 0                    -x = max occurrences, store in List
x > 0                     x = max occurrences, store in array []
x = Integer.MAX_VALUE    no limit, but use array []
x = 0                     1 occurrence required
```

Returns   The cardinality of this attribute.

105.14.2.14   **public String[] getDefaultValue()**

☐   Return a default for this attribute. The object must be of the appropriate type as defined by the cardinality and getType(). The return type is a list of String objects that can be converted to the appropriate type. The cardinality of the return array must follow the absolute cardinality of this type. For example, 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 lists of 0 elements.

Returns   Return a default value or null if no default exists.

105.14.2.15   **public String getDescription()**

☐   Return a description of this attribute. The description may be localized and must describe the semantics of this type and any constraints.

Returns   The localized description of the definition.

105.14.2.16   **public String getID()**

☐   Unique identity for this attribute. Attributes share a global namespace in the registry. For example, 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 organizations and many companies already have a node in the tree) it can be returned here. Otherwise, a unique id should be returned which can be a Java class name (reverse domain name) or generated with a GUID algorithm. Note that all LDAP defined attributes already have an OID. It is strongly advised to define the attributes from existing LDAP schemes which will give the OID. Many such schemes exist ranging from postal addresses to DHCP parameters.

Returns   The id or oid

105.14.2.17   **public String getName()**

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

Returns   The localized name of the definition.

105.14.2.18   **public String[] getOptionLabels()**

☐   Return a list of labels of option values.

The purpose of this method is to allow menus with localized labels. It is associated with getOption-Values. 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. That is, for each index i in getOptionLabels, i in getOptionValues() should be the associated value.

For example, if an attribute can have the value male, female, unknown, this list can return (for dutch) new String[] { "Man", "Vrouw", "Onbekend" }.

Returns   A list values

105.14.2.19 **public String[] getOptionValues()**

□ Return a list of option values that this attribute can take.

If the function returns null, there are no option values available.

Each value must be acceptable to validate() (return "") and must be a String object that can be converted to the data type defined by getType() for this attribute.

This list must be in the same sequence as getOptionLabels(). That is, for each index i in getOption-Values, i in getOptionLabels() should be the label.

For example, if an attribute can have the value male, female, unknown, this list can return new String[] { "male", "female", "unknown" }.

Returns A list values

105.14.2.20 **public int getType()**

□ Return the type for this attribute.

Defined in the following constants which map to the appropriate Java type.
STRING,LONG,INTEGER, SHORT, CHARACTER, BYTE,DOUBLE,FLOAT, BOOLEAN, PASSWORD.

Returns The type for this attribute.

105.14.2.21 **public String validate(String value)**

value The value before turning it into the basic data type. If the cardinality indicates a multi-valued attribute then the given string must be escaped.

□ Validate an attribute in String form. An attribute might be further constrained in value. This method will attempt to validate the attribute according to these constraints. It can return three different values:

```
null          No validation present
""            No problems detected
"..."         A localized description of why the value is wrong
```

If the cardinality of this attribute is multi-valued then this string must be interpreted as a comma delimited string. The complete value must be trimmed from white space as well as spaces around commas. Commas (',' \u002C) and spaces (' ' \u0020) and backslashes ('\' \u005C) can be escaped with another backslash. Escaped spaces must not be trimmed. For example:

```
value="  a\,b,b\,c,\ c\\,d   " => [ "a,b", "b,c", " c\", "d" ]
```

Returns null, "", or another string

105.14.3 **public interface MetaTypeInformation
extends MetaTypeProvider**

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

Since 1.1

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

105.14.3.1 **public Bundle getBundle()**

□ Return the bundle for which this object provides meta type information.

Returns Bundle for which this object provides meta type information.

105.14.3.2   **public String[] getFactoryPids()**

☐ Return the Factory PIDs (for ManagedServiceFactories) for which ObjectClassDefinition informa-
tion is available.

Returns Array of Factory PIDs.

105.14.3.3   **public String[] getPids()**

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

Returns Array of PIDs.

105.14.4   public interface MetaTypeProvider

Provides access to metatypes. This interface can be implemented on a Managed Service or Managed
Service Factory as well as registered as a service. When registered as a service, it must be registered
with a METATYPE_FACTORY_PID or METATYPE_PID service property (or both). Any PID men-
tioned in either of these factories must be a valid argument to the getObjectClassDefinition(String,
String) method.

Concurrency Thread-safe

105.14.4.1   **public static final String METATYPE_FACTORY_PID = "metatype.factory.pid"**

Service property to signal that this service has ObjectClassDefinition objects for the given factory
PIDs. The type of this service property is String+.

Since 1.2

105.14.4.2   **public static final String METATYPE_PID = "metatype.pid"**

Service property to signal that this service has ObjectClassDefinition objects for the given PIDs. The
type of this service property is String+.

Since 1.2

105.14.4.3   **public String[] getLocales()**

☐ Return a list of available locales. The results must be names that consists of language [_ country [_
variation]] as is customary in the Locale class.

Returns An array of locale strings or null if there is no locale specific localization can be found.

105.14.4.4   **public ObjectClassDefinition getObjectClassDefinition(String id,String locale)**

id The ID of the requested object class. This can be a pid or factory pid returned by getPids or getFacto-
ryPids.

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 cus-
tomary in the Locale class. This Locale class is not used because certain profiles do not contain it.

Returns A ObjectClassDefinition object.

Throws IllegalArgumentException– If the id or locale arguments are not valid

105.14.5   public interface MetaTypeService

The MetaType Service can be used to obtain meta type information for a bundle. The MetaType Ser-
vice will examine the specified bundle for meta type documents to create the returned MetaTypeIn-
formation object.

If the specified bundle does not contain any meta type documents, then a `MetaTypeInformation` object will be returned that wrappers any `ManagedService` or `ManagedServiceFactory` services registered by the specified bundle that implement `MetaTypeProvider`. Thus the MetaType Service can be used to retrieve meta type information for bundles which contain a meta type documents or which provide their own `MetaTypeProvider` objects.

Since 1.1

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

105.14.5.1 **public static final String METATYPE_CAPABILITY_NAME = "osgi.metatype"**

Capability name for meta type document processors.

Used in `Provide-Capability` and `Require-Capability` manifest headers with the `osgi.extender` namespace. For example:

```
Require-Capability: osgi.extender;
 filter:="(&(osgi.extender=osgi.metatype)(version>=1.3)(!(version>=2.0)))"
```

Since 1.3

105.14.5.2 **public static final String METATYPE_DOCUMENTS_LOCATION = "OSGI-INF/metatype"**

Location of meta type documents. The MetaType Service will process each entry in the meta type documents directory.

105.14.5.3 **public MetaTypeInformation getMetaTypeInformation(Bundle bundle)**

bundle The bundle for which meta type information is requested.

☐ Return the MetaType information for the specified bundle.

Returns A MetaTypeInformation object for the specified bundle.

105.14.6 public interface ObjectClassDefinition

Description for the data type information of an objectclass.

Concurrency Thread-safe

105.14.6.1 **public static final int ALL = –1**

Argument for `getAttributeDefinitions(int)`.

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

105.14.6.2 **public static final int OPTIONAL = 2**

Argument for `getAttributeDefinitions(int)`.

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

105.14.6.3 **public static final int REQUIRED = 1**

Argument for `getAttributeDefinitions(int)`.

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

105.14.6.4 **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

105.14.6.5   **public String getDescription()**

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

Returns The description of this object class.

105.14.6.6   **public InputStream getIcon(int size) throws IOException**

size Requested size of an icon. For example, a 16x16 pixel icon has a size of 16

☐ Return an InputStream object that can be used to create an icon from.

Indicate the size and return an InputStream object containing an icon. The returned icon maybe larger or smaller than the indicated size.

The icon may depend on the localization.

Returns An InputStream representing an icon or null

Throws IOException– If the InputStream cannot be returned.

105.14.6.7   **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 organizations and many companies already have a node in the tree) it can be returned here. Otherwise, a unique id should be returned which can be a java class name (reverse domain name) or generated with a GUID algorithm. Note that all LDAP defined object classes already have an OID associated. It is strongly advised to define the object classes from existing LDAP schemes which will give the OID for free. Many such schemes exist ranging from postal addresses to DHCP parameters.

Returns The id of this object class.

105.14.6.8   **public String getName()**

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

Returns The name of this object class.

105.15   org.osgi.service.metatype.annotations

Metatype Annotations Package Version 1.3.

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

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

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

105.15.1   Summary

- AttributeDefinition - AttributeDefinition information for the annotated method.
- AttributeType - Attribute types for the AttributeDefinition annotation.

- Designate - Generate a Designate element in the Meta Type Resource for an ObjectClassDefinition using the annotated Declarative Services component.
- Icon - Icon information for an ObjectClassDefinition.
- ObjectClassDefinition - Generate a Meta Type Resource using the annotated type.
- Option - Option information for an AttributeDefinition.

105.15.2 @AttributeDefinition

AttributeDefinition information for the annotated method.

Each method of a type annotated by ObjectClassDefinition has an implied AttributeDefinition annotation. This annotation is only used to specify non-default AttributeDefinition information.

The id of this AttributeDefinition is generated from the name of the annotated method. The annotated method name is processed from left to right changing each character as follows:

- A dollar sign ('$' \u0024) is removed unless it is followed by another dollar sign in which case the two consecutive dollar signs ('$$') are changed to a single dollar sign.
- A low line ('_' \u005F) is changed to a full stop ('.' \u002E) unless is it followed by another low line in which case the two consecutive low lines ('__') are changed to a single low line.
- All other characters are unchanged.

This id is the value of the id attribute of the generate AD element and is used as the name of the corresponding configuration property.

This annotation is not processed at runtime. It must be processed by tools and used to contribute to a Meta Type Resource document for the bundle.

See Also   The AD element of a Meta Type Resource.

Retention   CLASS

Target   METHOD

105.15.2.1 String name default ""

☐ The human readable name of this AttributeDefinition.

If not specified, the name of this AttributeDefinition is derived from the name of the annotated method. For example, low line ('_' \u005F) and dollar sign ('$' \u0024) are replaced with space (' ' \u0020) and space is inserted between camel case words.

If the name begins with the percent sign ('%' \u0025), the name can be localized.

See Also   The name attribute of the AD element of a Meta Type Resource.

105.15.2.2 String description default ""

☐ The human readable description of this AttributeDefinition.

If not specified, the description of this AttributeDefinition is the empty string.

If the description begins with the percent sign ('%' \u0025), the description can be localized.

See Also   The description attribute of the AD element of a Meta Type Resource.

105.15.2.3 AttributeType type default STRING

☐ The type of this AttributeDefinition.

This must be one of the defined attributes types.

If not specified, the type is derived from the return type of the annotated method. Return types of Class and Enum are mapped to STRING. If the return type is List, Set, Collection, Iterable or some type which can be determined at annotation processing time to

1. be a subtype of Collection and
2. have a public no argument constructor,

then the type is derived from the generic type. For example, a return type of List<String> will be mapped to STRING. A return type of a single dimensional array is supported and the type is the component type of the array. Multi dimensional arrays are not supported. Annotation return types are not supported. Any unrecognized type is mapped to STRING. A tool processing the annotation should declare an error for unsupported return types.

See Also   The type attribute of the AD element of a Meta Type Resource.

105.15.2.4   **int cardinality default 0**

□ The cardinality of this AttributeDefinition.

If not specified, the cardinality is derived from the return type of the annotated method. For an array return type, the cardinality is a large positive value. If the return type is List, Set, Collection, Iterable or some type which can be determined at annotation processing time to

1. be a subtype of Collection and
2. have a public no argument constructor,

the cardinality is a large negative value. Otherwise, the cardinality is 0.

See Also   The cardinality attribute of the AD element of a Meta Type Resource.

105.15.2.5   **String min default ""**

□ The minimum value for this AttributeDefinition.

If not specified, there is no minimum value.

See Also   The min attribute of the AD element of a Meta Type Resource.

105.15.2.6   **String max default ""**

□ The maximum value for this AttributeDefinition.

If not specified, there is no maximum value.

See Also   The max attribute of the AD element of a Meta Type Resource.

105.15.2.7   **String[] defaultValue default {}**

□ The default value for this AttributeDefinition.

The specified values are concatenated into a comma delimited list to become the value of the default attribute of the generated AD element.

If not specified and the annotated method is an annotation element that has a default value, then the value of this element is the default value of the annotated element. Otherwise, there is no default value.

See Also   The default attribute of the AD element of a Meta Type Resource.

105.15.2.8   **boolean required default true**

□ The required value for this AttributeDefinition.

If not specified, the value is true.

See Also   The required attribute of the AD element of a Meta Type Resource.

105.15.2.9   **Option[] options default {}**

□ The option information for this AttributeDefinition.

For each specified Option, an Option element is generated for this AttributeDefinition.

If not specified, the option information is derived from the return type of the annotated method. If the return type is an enum, a single dimensional array of an enum, or a List, Set, Collection, Iterable or some type which can be determined at annotation processing time to

1. be a subtype of Collection and
2. have a public no argument constructor,

with a generic type of an enum, then the value of this element has an Option for each value of the enum. The label and value of each Option are set to the name of the corresponding enum value. Otherwise, no Option elements will be generated.

See Also   The Option element of a Meta Type Resource.

105.15.3   enum AttributeType

Attribute types for the AttributeDefinition annotation.

See Also   AttributeDefinition.type()

105.15.3.1   STRING

The String type.

Attributes of this type should be stored as String, List<String> or String[] objects, depending on the cardinality value.

105.15.3.2   LONG

The Long type.

Attributes of this type should be stored as Long, List<Long> or long[] objects, depending on the AttributeDefinition#cardinality() cardinality value.

105.15.3.3   INTEGER

The Integer type.

Attributes of this type should be stored as Integer, List<Integer> or int[] objects, depending on the AttributeDefinition#cardinality() cardinality value.

105.15.3.4   SHORT

The Short type.

Attributes of this type should be stored as Short, List<Short> or short[] objects, depending on the AttributeDefinition#cardinality() cardinality value.

105.15.3.5   CHARACTER

The Character type.

Attributes of this type should be stored as Character, List<Character> or char[] objects, depending on the AttributeDefinition#cardinality() cardinality value.

105.15.3.6   BYTE

The Byte type.

Attributes of this type should be stored as Byte, List<Byte> or byte[] objects, depending on the AttributeDefinition#cardinality() cardinality value.

105.15.3.7   DOUBLE

The Double type.

Attributes of this type should be stored as Double, List<Double> or double[] objects, depending on the AttributeDefinition#cardinality() cardinality value.

105.15.3.8   FLOAT

The Float type.

Attributes of this type should be stored as Float, List<Float> or float[] objects, depending on the AttributeDefinition#cardinality() cardinality value.

105.15.3.9   BOOLEAN

The Boolean type.

Attributes of this type should be stored as Boolean, List<Boolean> or boolean[] objects depending on AttributeDefinition#cardinality() cardinality.

105.15.3.10   PASSWORD

The Password type.

Attributes of this type must be stored as String, List<String> or String[] objects depending on cardinality.

A Password must be treated as a String but the type can be used to disguise the information when displayed to a user to prevent it from being seen.

105.15.4   @Designate

Generate a Designate element in the Meta Type Resource for an ObjectClassDefinition using the annotated Declarative Services component.

This annotation must be used on a type that is also annotated with the Declarative Services Component annotation. The component must only have a single PID which is used for the generated Designate element.

This annotation is not processed at runtime. It must be processed by tools and used to contribute to a Meta Type Resource document for the bundle.

See Also The Designate element of a Meta Type Resource.

Retention CLASS

Target TYPE

105.15.4.1   Class<?> ocd

□ The type of the ObjectClassDefinition for this Designate.

The specified type must be annotated with ObjectClassDefinition.

See Also The ocdref attribute of the Designate element of a Meta Type Resource.

105.15.4.2   boolean factory default false

□ Specifies whether this Designate is for a factory PID.

If false, then the PID value from the annotated component will be used in the pid attribute of the generated Designate element. If true, then the PID value from the annotated component will be used in the factoryPid attribute of the generated Designate element.

See Also The pid and factoryPid attributes of the Designate element of a Meta Type Resource.

105.15.5   @Icon

Icon information for an ObjectClassDefinition.

See Also   ObjectClassDefinition.icon()

Retention   CLASS

Target

105.15.5.1   String resource

□   The resource name for this Icon.

The resource is a URL. The resource URL can be relative to the root of the bundle containing the Meta Type Resource.

If the resource begins with the percent sign ('%' \u0025), the resource can be localized.

See Also   The resource attribute of the Icon element of a Meta Type Resource.

105.15.5.2   int size

□   The pixel size of this Icon.

For example, 32 represents a 32x32 icon.

See Also   The size attribute of the Icon element of a Meta Type Resource.

105.15.6   @ObjectClassDefinition

Generate a Meta Type Resource using the annotated type.

This annotation can be used without defining any element values since defaults can be generated from the annotated type. Each method of the annotated type has an implied AttributeDefinition annotation if not explicitly annotated.

This annotation may only be used on annotation types and interface types. Use on concrete or abstract class types is unsupported. If applied to an interface then all methods inherited from super types are included as attributes.

This annotation is not processed at runtime. It must be processed by tools and used to generate a Meta Type Resource document for the bundle.

See Also   The OCD element of a Meta Type Resource.

Retention   CLASS

Target   TYPE

105.15.6.1   String id default ""

□   The id of this ObjectClassDefinition.

If not specified, the id of this ObjectClassDefinition is the fully qualified name of the annotated type using the dollar sign ('$' \u0024) to separate nested class names from the name of their enclosing class. The id is not to be confused with a PID which can be specified by the pid() or factoryPid() element.

See Also   The id attribute of the OCD element of a Meta Type Resource.

105.15.6.2   String name default ""

□   The human readable name of this ObjectClassDefinition.

If not specified, the name of this ObjectClassDefinition is derived from the id(). For example, low line ('_' \u005F) and dollar sign ('$' \u0024) are replaced with space (' ' \u0020) and space is inserted between camel case words.

If the name begins with the percent sign ('%' \u0025), the name can be localized.

See Also   The name attribute of the OCD element of a Meta Type Resource.

105.15.6.3   **String description default ""**

   ☐ The human readable description of this ObjectClassDefinition.

If not specified, the description of this ObjectClassDefinition is the empty string.

If the description begins with the percent sign ('%' \u0025), the description can be localized.

See Also   The description attribute of the OCD element of a Meta Type Resource.

105.15.6.4   **String localization default ""**

   ☐ The localization resource of this ObjectClassDefinition.

This refers to a resource property entry in the bundle that can be augmented with locale information. If not specified, the localization resource for this ObjectClassDefinition is the string "OSGI-INF/l10n/" followed by the id().

See Also   The localization attribute of the MetaData element of a Meta Type Resource.

105.15.6.5   **String[] pid default {}**

   ☐ The PIDs associated with this ObjectClassDefinition.

For each specified PID, a Designate element with a pid attribute is generated that references this ObjectClassDefinition.

The Designate annotation can also be used to associate a Declarative Services component with an ObjectClassDefinition and generate a Designate element.

See Also   The pid attribute of the Designate element of a Meta Type Resource., Designate

105.15.6.6   **String[] factoryPid default {}**

   ☐ The factory PIDs associated with this ObjectClassDefinition.

For each specified factory PID, a Designate element with a factoryPid attribute is generated that references this ObjectClassDefinition.

The Designate annotation can also be used to associate a Declarative Services component with an ObjectClassDefinition and generate a Designate element.

See Also   The factoryPid attribute of the Designate element of a Meta Type Resource., Designate

105.15.6.7   **Icon[] icon default {}**

   ☐ The icon resources associated with this ObjectClassDefinition.

For each specified Icon, an Icon element is generated for this ObjectClassDefinition. If not specified, no Icon elements will be generated.

See Also   The Icon element of a Meta Type Resource.

105.15.7   **@Option**

Option information for an AttributeDefinition.

See Also   AttributeDefinition.options()

Retention   CLASS

Target

105.15.7.1   **String label default ""**

   ☐ The human readable label of this Option.

If not specified, the label of this Option is the empty string.

If the label begins with the percent sign ('%' \u0025), the label can be localized.

    See Also The label attribute of the Option element of a Meta Type Resource.

105.15.7.2   **String value**

    ▫ The value of this Option.

    See Also The value attribute of the Option element of a Meta Type Resource.

105.16 References

[1]   *LDAP.*
http://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol

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

105.17 Changes

- Changed vector references to list.
- Updates schema to 1.3.0. This includes changing `Char` to `Character` to conform with other OSGi schema and allowing some more flexibility in ordering of elements.
- Clarified that Meta Type Service will only search for ManagedService and ManagedServiceFactory service objects that implement MetaTypeProvider if no meta type resources are found in the bundle.
- Clarify that if multiple sources define the same Object Class Definition, that Meta Type Provider services must take precedence over Managed Service [Factory] objects implementing `MetaType-Provider` or Meta Type Resources.
- The Meta Type Resource schema is updated to allow an OCD element to contain zero AD elements.
- New Meta Type Annotations. See *Meta Type Annotations* on page 119.
- Added capabilities that Metatype Service specification implementations must provide. See *Capabilities* on page 121.

107   User Admin Service Specification

Version 1.1

107.1   Introduction

OSGi frameworks 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, biometrics, and certificates.

Once the initiator is authenticated, it is necessary to verify that this principal is authorized to perform the requested action. This authorization can only be decided by the operator of the OSGi environment, and thus requires administration.

The User Admin service provides this type of functionality. Bundles can use the User Admin service to authenticate an initiator and represent this authentication as an Authorization object. Bundles that execute actions on behalf of this user can use the Authorization object to verify if that user is authorized.

The User Admin service provides authorization based on who runs the code, instead of using the Java code-based permission model. See [1] *The Java Security Architecture for JDK 1.2.* It performs a role similar to [2] *Java Authentication and Authorization Service.*

107.1.1   Essentials

- *Authentication* - A large number of authentication schemes already exist, and more will be developed. The User Admin service must be flexible enough to adapt to the many different authentication schemes that can be run on a computer system.
- *Authorization* - All bundles should use the User Admin service to authenticate users and to find out if those users are authorized. It is therefore paramount that a bundle can find out authorization information with little effort.
- *Security* - Detailed security, based on the Framework security model, is needed to provide safe access to the User Admin service. It should allow limited access to the credentials and other properties.
- *Extensibility* - Other bundles should be able to build on the User Admin service. It should be possible to examine the information from this service and get real-time notifications of changes.
- *Properties* - The User Admin service must maintain a persistent database of users. It must be possible to use this database to hold more information about this user.
- *Administration* - Administering authorizations for each possible action and initiator is time-consuming and error-prone. It is therefore necessary to have mechanisms to group end users and make it simple to assign authorizations to all members of a group at one time.
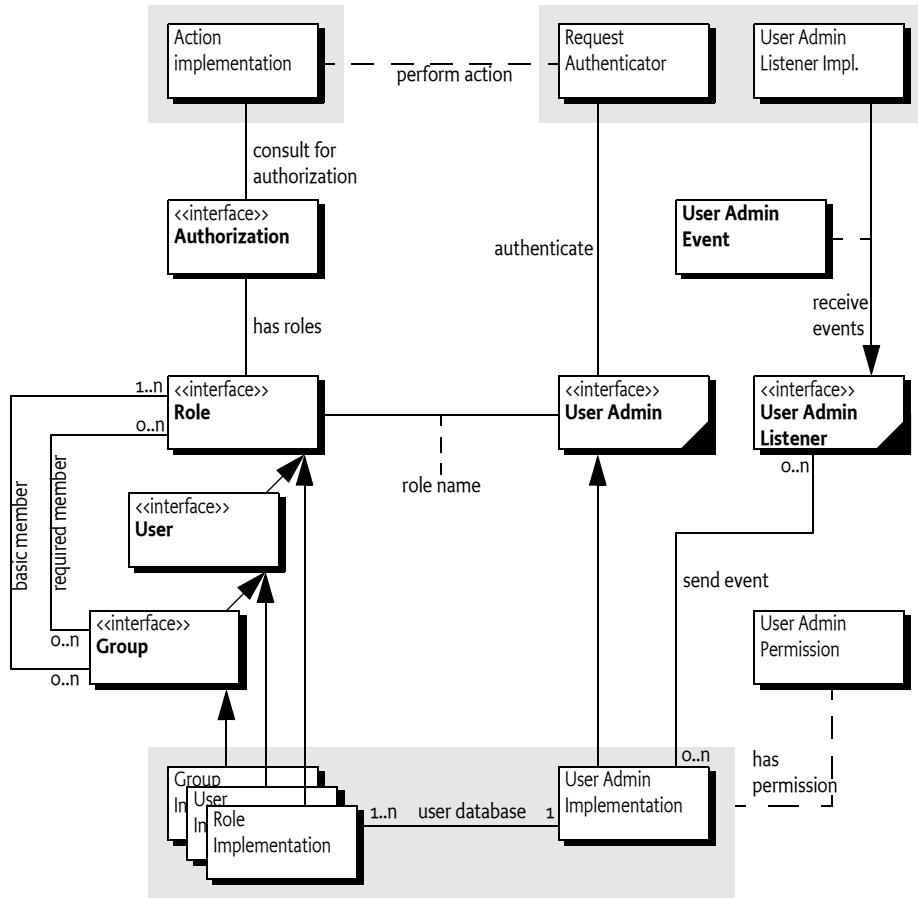
107.1.2   Entities

This Specification defines the following User Admin service entities:

- *User Admin* - 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.
- *User Admin Event* - This class is used to represent a role change event.
- *User Admin Listener* - This interface provides a listener for events of type UserAdminEvent that can be registered as a service.
- *User Admin Permission* - This permission is needed to configure and access the roles managed by a User Admin service.
- *Role.USER_ANYONE* - This is a special User object that represents *any* user, it implies all other User objects. It is also used when a Group is used with only basic members. The Role.USER_ANYONE is then the only required member.

Figure 107.1   *User Admin Service, org.osgi.service.useradmin*

107.1.3   Operation

An Operator uses the User Admin service to define OSGi framework 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 framework 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 framework 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 52.

107.2 Authentication

The authentication phase determines if the initiator is actually the one it says it is. Mechanisms to authenticate always need some information related to the user or the OSGi framework to authenticate an external user. This information can consist of the following:

- A secret known only to the initiator.
- Knowledge about cards that can generate a unique token.
- Public information like certificates of trusted signers.
- Information about the user that can be measured in a trusted way.
- Other specific information.

107.2.1 Repository

The User Admin service offers a repository of Role objects. Each Role object has a unique name and a set of properties that are readable by anyone, and are changeable when the changer has the UserAdminPermission. Additionally, User objects, a sub-interface of Role, also have a set of private protected properties called credentials. Credentials are an extra set of properties that are used to authenticate users and that are protected by UserAdminPermission.

Properties are accessed with the Role.getProperties() method and credentials with the User.getCredentials()method. Both methods return a Dictionary object containing key/value pairs. The keys are String objects and the values of the Dictionary object are limited to String or byte[ ] objects.

This specification does not define any standard keys for the properties or credentials. The keys depend on the implementation of the authentication mechanism and are not formally defined by OSGi specifications.

The repository can be searched for objects that have a unique property (key/value pair) with the method UserAdmin.getUser(String,String). This makes it easy to find a specific user related to a specific authentication mechanism. For example, a secure card mechanism that generates unique tokens could have a serial number identifying the user. The owner of the card could be found with the method

```
User owner = useradmin.getUser(
    "secure-card-serial", "132456712-1212" );
```

If multiple User objects have the same property (key *and* value), a null is returned.

There is a convenience method to verify that a user has a credential without actually getting the credential. This is the User.hasCredential(String,Object) method.

Access to credentials is protected on a name basis by UserAdminPermission. Because properties can be read by anyone with access to a User object, UserAdminPermission only protects change access to properties.

107.2.2   Basic Authentication

The following example shows a very simple authentication algorithm based on passwords.

The vendor of the authentication bundle uses the property "com.acme.basic-id" to contain the name of a user as it logs in. This property is used to locate the User object in the repository. Next, the credential "com.acme.password" contains the password and is compared to the entered password. If the password is correct, the User object is returned. In all other cases a SecurityException is thrown.

```
public User authenticate(
        UserAdmin ua, String name, String pwd )
    throws SecurityException {
    User user = ua.getUser("com.acme.basicid",
        username);
    if (user == null)
        throw new SecurityException( "No such user" );

    if (!user.hasCredential("com.acme.password", pwd))
        throw new SecurityException(
            "Invalid password" );
    return user;
}
```

107.2.3   Certificates

Authentication based on certificates does not require a shared secret. Instead, a certificate contains a name, a public key, and the signature of one or more signers.

The name in the certificate can be used to locate a User object in the repository. Locating a User object, however, only identifies the initiator and does not authenticate it.

1.   The first step to authenticate the initiator is to verify that it has the private key of the certificate.

2. Next, the User Admin service must verify that it has a User object with the right property, for example "com.acme.certificate"="Fudd".

3. The next step is to see if the certificate is signed by a trusted source. The bundle could use a central list of trusted signers and only accept certificates signed by those sources. Alternatively, it could require that the certificate itself is already stored in the repository under a unique key as a byte[] in the credentials.

4. In any case, once the certificate is verified, the associated User object is authenticated.

107.3   Authorization

The User Admin service authorization architecture is a *role-based model*. In this model, every action that can be performed by a bundle is associated with a *role*. Such a role is a Group object (called group from now on) from the User Admin service repository. For example, if a servlet could be used to activate the alarm system, there should be a group named AlarmSystemActivation.

The operator can administrate authorizations by populating the group with User objects (users) and other groups. Groups are used to minimize the amount of administration required. For example, it is easier to create one Administrators group and add administrative roles to it rather than individually administer all users for each role. Such a group requires only one action to remove or add a user as an administrator.

The authorization decision can now be made in two fundamentally different ways:

An initiator could be allowed to carry out an action (represented by a Group object) if it implied any of the Group object's members. For example, the AlarmSystemActivation Group object contains an Administrators and a Family Group object:

```
Administrators        = { Elmer, Pepe,Bugs }
Family                = { Elmer, Pepe, Daffy }

AlarmSystemActivation = { Administrators, Family}
```

Any of the four members Elmer, Pepe, Daffy, or Bugs can activate the alarm system.

Alternatively, an initiator could be allowed to perform an action (represented by a Group object) if it implied *all* the Group object's members. In this case, using the same AlarmSystemActivation group, only Elmer and Pepe would be authorized to activate the alarm system, since Daffy and Bugs are *not* members of *both* the Administrators and Family Group objects.

The User Admin service supports a combination of both strategies by defining both a set of *basic members* (any) and a set of *required members* (all).

```
Administrators        = { Elmer, Pepe, Bugs}
Family                = { Elmer, Pepe, Daffy }

AlarmSystemActivation
    required          = { Administrators }
    basic             = { Family }
```

The difference is made when Role objects are added to the Group object. To add a basic member, use the Group.addMember(Role) method. To add a required member, use the Group.addRequiredMember(Role) method.

Basic members define the set of members that can get access and required members reduce this set by requiring the initiator to *imply* each required member.

A User object implies a Group object if it implies the following:

- *All* of the Group's required members, and
- At *least* one of the Group's basic members

A User object always implies itself.

If only required members are used to qualify the implication, then the standard user Role.USER_ANYONE can be obtained from the User Admin service and added to the Group object. This Role object is implied by anybody and therefore does not affect the required members.

107.3.1 The Authorization Object

The complexity of authorization is hidden in an Authorization class. Normally, the authenticator should retrieve an Authorization object from the User Admin service by passing the authenticated User object as an argument. This Authorization object is then passed to the bundle that performs the action. This bundle checks the authorization with the Authorization.hasRole(String) method. The performing bundle must pass the name of the action as an argument. The Authorization object checks whether the authenticated user implies the Role object, specifically a Group object, with the given name. This is shown in the following example.

```
public void activateAlarm(Authorization auth) {
    if ( auth.hasRole( "AlarmSystemActivation" ) ) {
        // activate the alarm
        ...
    }
    else throw new SecurityException(
        "Not authorized to activate alarm" );
}
```

107.3.2 Authorization Example

This section demonstrates a possible use of the User Admin service. The service has a flexible model and many other schemes are possible.

Assume an Operator installs an OSGi framework. Bundles in this environment have defined the following action groups:

```
AlarmSystemControl
InternetAccess
TemperatureControl
PhotoAlbumEdit
PhotoAlbumView
PortForwarding
```

Installing and uninstalling bundles could potentially extend this set. Therefore, the Operator also defines a number of groups that can be used to contain the different types of system users.

```
Administrators
Buddies
Children
Adults
Residents
```

In a particular instance, the Operator installs it in a household with the following residents and buddies:

```
Residents:      Elmer, Fudd, Marvin, Pepe
Buddies:        Daffy, Foghorn
```

First, the residents and buddies are assigned to the system user groups. Second, the user groups need to be assigned to the action groups.

The following tables show how the groups could be assigned.

Table 107.1   *Example Groups with Basic and Required Members*

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

Table 107.2   *Example Action Groups with their Basic and Required Members*

Groups	Residents	Buddies	Children	Adults	Admin
AlarmSystemControl	Basic	-	-	-	Required
InternetAccess	Basic	-	-	Required	-
TemperatureControl	Basic	-	-	Required	-
PhotoAlbumEdit	Basic	-	Basic	Basic	-
PhotoAlbumView	Basic	Basic	-	-	-
PortForwarding	Basic	-	-	-	Required

107.4   Repository Maintenance

The UserAdmin interface is a straightforward API to maintain a repository of User and Group objects. It contains methods to create new Group and User objects with the createRole(String,int) method. The method is prepared so that the same signature can be used to create new types of roles in the future. The interface also contains a method to remove a Role object.

The existing configuration can be obtained with methods that list all Role objects using a filter argument. This filter, which has the same syntax as the Framework filter, must only return the Role objects for which the filter matches the properties.

Several utility methods simplify getting User objects depending on their properties.

107.5   User Admin Events

Changes in the User Admin service can be determined in real time. Each User Admin service implementation must send a UserAdminEvent object to any service in the Framework service registry that is registered under the UserAdminListener interface. This event must be send asynchronously from the cause of the event. The way events must be delivered is the same as described in *Delivering Events* of *OSGi Core Release 6*.

This procedure is demonstrated in the following code sample.

```
class Listener implements UserAdminListener{
    public void roleChanged( UserAdminEvent event ) {
        ...
    }
}
public class MyActivator
    implements BundleActivator {
    public void start( BundleContext context ) {
        context.registerService(
            UserAdminListener.class.getName(),
```

```
                    new Listener(), null );
        }
        public void stop( BundleContext context ) {}
}
```

It is not necessary to unregister the listener object when the bundle is stopped because the Framework automatically unregisters it. Once registered, the UserAdminListener object must be notified of all changes to the role repository.

107.5.1   Event Admin and User Admin Change Events

User Admin events must be delivered asynchronously to the Event Admin service by the implementation, if present. The topic of a User Admin Event is:

org/osgi/service/useradmin/UserAdmin/<eventtype>

The following event types are supported:

ROLE_CREATED
ROLE_CHANGED
ROLE_REMOVED

All User Admin Events must have the following properties:

- event - (UserAdminEvent) The event that was broadcast by the User Admin service.
- role - (Role) The Role object that was created, modified or removed.
- role.name - (String) The name of the role.
- role.type - (Integer) One of ROLE, USER or GROUP.
- service - (ServiceReference) The Service Reference of the User Admin service.
- service.id - (Long) The User Admin service's ID.
- service.objectClass - (String[]) The User Admin service's object class (which must include org.osgi.service.useradmin.UserAdmin)
- service.pid - (String) The User Admin service's persistent identity

107.6   Security

The User Admin service is related to the security model of the OSGi framework, but is complementary to the [1] *The Java Security Architecture for JDK 1.2*. The final permission of most code should be the intersection of the Java 2 Permissions, which are based on the code that is executing, and the User Admin service authorization, which is based on the user for whom the code runs.

107.6.1   User Admin Permission

The User Admin service defines the UserAdminPermission class that can be used to restrict bundles in accessing credentials. This permission class has the following actions:

- changeProperty - This permission is required to modify properties. The name of the permission is the prefix of the property name.
- changeCredential - This action permits changing credentials. The name of the permission is the prefix of the name of the credential.
- getCredential - This action permits getting credentials. The name of the permission is the prefix of the credential.

If the name of the permission is "admin", it allows the owner to administer the repository. No action is associated with the permission in that case.

Otherwise, the permission name is used to match the property name. This name may end with a ".*" string to indicate a wildcard. For example, `com.acme.*` matches `com.acme.fudd.elmer` and `com.acme.bugs`.

107.7 Relation to JAAS

At a glance, the Java Authorization and Authentication Service (JAAS) seems to be a very suitable model for user administration. The OSGi organization, however, decided to develop an independent User Admin service because JAAS was not deemed applicable. The reasons for this include dependency on Java SE version 1.3 ("JDK 1.3") and existing mechanisms in the previous OSGi Service Gateway 1.0 specification.

107.7.1 JDK 1.3 Dependencies

The authorization component of JAAS relies on the `java.security.DomainCombiner` interface, which provides a means to dynamically update the `ProtectionDomain` objects affiliated with an `AccessControlContext` object.

This interface was added in JDK 1.3. In the context of JAAS, the `SubjectDomainCombiner` object, which implements the `DomainCombiner` interface, is used to update `ProtectionDomain` objects. The permissions of `ProtectionDomain` objects depend on where code came from and who signed it, with permissions based on who is running the code.

Leveraging JAAS would have resulted in user-based access control on the OSGi framework being available only with JDK 1.3, which was not deemed acceptable.

107.7.2 Existing OSGi Mechanism

JAAS provides a pluggable authentication architecture, which enables applications and their underlying authentication services to remain independent from each other.

The Http Service already provides a similar feature by allowing servlet and resource registrations to be supported by an `HttpContext` object, which uses a callback mechanism to perform any required authentication checks before granting access to the servlet or resource. This way, the registering bundle has complete control on a per-servlet and per-resource basis over which authentication protocol to use, how the credentials presented by the remote requester are to be validated, and who should be granted access to the servlet or resource.

107.7.3 Future Road Map

In the future, the main barrier of 1.3 compatibility will be removed. JAAS could then be implemented in an OSGi environment. At that time, the User Admin service will still be needed and will provide complementary services in the following ways:

- The authorization component relies on group membership information to be stored and managed outside JAAS. JAAS does not manage persistent information, so the User Admin service can be a provider of group information when principals are assigned to a `Subject` object.
- The authorization component allows for credentials to be collected and verified, but a repository is needed to actually validate the credentials.

In the future, the User Admin service can act as the back-end database to JAAS. The only aspect JAAS will remove from the User Admin service is the need for the `Authorization` interface.

107.8 org.osgi.service.useradmin

User Admin Package Version 1.1.

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

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

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

107.8.1 Summary

- Authorization - The Authorization interface encapsulates an authorization context on which bundles can base authorization decisions, where appropriate.
- Group - A named grouping of roles (Role objects).
- Role - The base interface for Role objects managed by the User Admin service.
- User - A User role managed by a User Admin service.
- UserAdmin - This interface is used to manage a database of named Role objects, which can be used for authentication and authorization purposes.
- UserAdminEvent - Role change event.
- UserAdminListener - Listener for UserAdminEvents.
- UserAdminPermission - Permission to configure and access the Role objects managed by a User Admin service.

107.8.2 public interface Authorization

The Authorization interface encapsulates an authorization context on which bundles can base authorization decisions, where appropriate.

Bundles associate the privilege to access restricted resources or operations with roles. Before granting access to a restricted resource or operation, a bundle will check if the Authorization object passed to it possess the required role, by calling its hasRole method.

Authorization contexts are instantiated by calling the UserAdmin.getAuthorization(User) method.

Trusting Authorization objects

There are no restrictions regarding the creation of Authorization objects. Hence, a service must only accept Authorization objects from bundles that has been authorized to use the service using code based (or Java 2) permissions.

In some cases it is useful to use ServicePermission to do the code based access control. A service basing user access control on Authorization objects passed to it, will then require that a calling bundle has the ServicePermission to get the service in question. This is the most convenient way. The OSGi environment will do the code based permission check when the calling bundle attempts to get the service from the service registry.

Example: A servlet using a service on a user's behalf. The bundle with the servlet must be given the ServicePermission to get the Http Service.

However, in some cases the code based permission checks need to be more fine-grained. A service might allow all bundles to get it, but require certain code based permissions for some of its methods.

Example: A servlet using a service on a user's behalf, where some service functionality is open to anyone, and some is restricted by code based permissions. When a restricted method is called (e.g., one handing over an Authorization object), the service explicitly checks that the calling bundle has permission to make the call.

No Implement Consumers of this API must not implement this interface

107.8.2.1 **public String getName()**

☐ Gets the name of the User that this Authorization context was created for.

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

107.8.2.2 **public String[] getRoles()**

☐ Gets the names of all roles implied by this Authorization context.

Returns The names of all roles implied by this Authorization context, or null if no roles are in the context. The predefined role user.anyone will not be included in this list.

107.8.2.3 **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 object for each role and adding User objects to it.

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

107.8.3 **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
```

No Implement Consumers of this API must not implement this interface

107.8.3.1 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.

107.8.3.2 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.

107.8.3.3 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.

107.8.3.4 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.

107.8.3.5 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.

107.8.4   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 objects are required to change a Role object's properties.

Role object properties are Dictionary objects. Changes to these objects are propagated to the User Admin service and made persistent.

Every User Admin service contains a set of predefined Role objects that are always present and cannot be removed. All predefined Role objects are of type ROLE. This version of the org.osgi.service.useradmin package defines a single predefined role named "user.anyone", which is inherited by any other role. Other predefined roles may be added in the future. Since "user.anyone" is a Role object that has properties associated with it that can be read and modified. Access to these properties and their use is application specific and is controlled using UserAdminPermission in the same way that properties for other Role objects are.

No Implement   Consumers of this API must not implement this interface

107.8.4.1   public static final int GROUP = 2

The type of a Group role.

The value of GROUP is 2.

107.8.4.2   public static final int ROLE = 0

The type of a predefined role.

The value of ROLE is 0.

107.8.4.3   public static final int USER = 1

The type of a User role.

The value of USER is 1.

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

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

Since   1.1

107.8.4.5   public String getName()

□   Returns the name of this role.

Returns   The role's name.

107.8.4.6   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 to be broadcast to any UserAdminListener objects.

Only objects of type String may be used as property keys, and only objects of type String or byte[] may be used as property values. Any other types will cause an exception of type IllegalArgumentException to be raised.

In order to add, change, or remove a property in the returned Dictionary, a UserAdminPermission named after the property name (or a prefix of it) with action changeProperty is required.

Returns   Dictionary containing the properties of this Role object.

107.8.4.7   public int getType()

□   Returns the type of this role.

Returns   The role's type.

107.8.5   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 class) associated with it. Specific UserAdminPermission objects are required to read or change a User object's credentials.

Credentials are Dictionary objects and have semantics that are similar to the properties in the Role class.

No Implement   Consumers of this API must not implement this interface

107.8.5.1   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 to be broadcast to any UserAdminListeners objects.

Only objects of type String may be used as credential keys, and only objects of type String or of type byte[] may be used as credential values. Any other types will cause an exception of type IllegalArgumentException to be raised.

In order to retrieve a credential from the returned Dictionary object, a UserAdminPermission named after the credential name (or a prefix of it) with action getCredential is required.

In order to add or remove a credential from the returned Dictionary object, a UserAdminPermission named after the credential name (or a prefix of it) with action changeCredential is required.

Returns   Dictionary object containing the credentials of this User object.

107.8.5.2   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.

107.8.6   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 object can be configured with credentials (e.g., a password) and properties (e.g., a street address, phone number, etc.).

A Group object represents an aggregation of User and Group objects. In other words, the members of a Group object are roles themselves.

Every User Admin service manages and maintains its own namespace of Role objects, in which each Role object has a unique name.

No Implement   Consumers of this API must not implement this interface

107.8.6.1   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 type or Role.GROUP type.

□   Creates a Role object with the given name and of the given type.

If a Role object was created, a UserAdminEvent object of type UserAdminEvent.ROLE_CREATED is broadcast to any UserAdminListener object.

Returns   The newly created Role object, or null if a role with the given name already exists.

Throws   IllegalArgumentException– if type is invalid.

SecurityException– If a security manager exists and the caller does not have the UserAdminPermission with name admin.

107.8.6.2   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.

107.8.6.3   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.

107.8.6.4   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.

Throws InvalidSyntaxException– If the filter is not well formed.

107.8.6.5　　**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.

107.8.6.6　　**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 and all groups it is a member of.

If the Role object was removed, a UserAdminEvent object of type UserAdminEvent.ROLE_REMOVED is broadcast to any UserAdminListener object.

Returns true If a Role object with the given name is present in this User Admin service and could be removed, otherwise false.

Throws SecurityException– If a security manager exists and the caller does not have the UserAdminPermission with name admin.

107.8.7　　**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 type, ROLE_CHANGED type, and ROLE_REMOVED type. Additional event types may be defined in the future.

See Also UserAdmin, UserAdminListener

107.8.7.1　　**public static final int ROLE_CHANGED = 2**

A Role object has been modified.

The value of ROLE_CHANGED is 0x00000002.

107.8.7.2　　**public static final int ROLE_CREATED = 1**

A Role object has been created.

The value of ROLE_CREATED is 0x00000001.

107.8.7.3　　**public static final int ROLE_REMOVED = 4**

A Role object has been removed.

The value of ROLE_REMOVED is 0x00000004.

107.8.7.4   **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.

107.8.7.5   **public Role getRole()**

☐   Gets the Role object this event was generated for.

Returns   The Role object this event was generated for.

107.8.7.6   **public ServiceReference getServiceReference()**

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

Returns   The User Admin service's ServiceReference object.

107.8.7.7   **public int getType()**

☐   Returns the type of this event.

The type values are ROLE_CREATED type, ROLE_CHANGED type, and ROLE_REMOVED type.

Returns   The event type.

107.8.8   **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, UserAdminEvent

107.8.8.1   **public void roleChanged(UserAdminEvent event)**

event   The UserAdminEvent object.

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

107.8.9   **public final class UserAdminPermission**
              extends BasicPermission

Permission to configure and access the Role objects managed by a User Admin service.

This class represents access to the Role objects managed by a User Admin service and their properties and credentials (in the case of User objects).

The permission name is the name (or name prefix) of a property or credential. The naming convention follows the hierarchical property naming convention. Also, an asterisk may appear at the end of the name, following a ".", or by itself, to signify a wildcard match. For example: "org.osgi.security.protocol.∗" or "∗" is valid, but "∗protocol" or "a∗b" are not valid.

The UserAdminPermission with the reserved name "admin" represents the permission required for creating and removing Role objects in the User Admin service, as well as adding and removing members in a Group object. This UserAdminPermission does not have any actions associated with it.

The actions to be granted are passed to the constructor in a string containing a list of one or more comma-separated keywords. The possible keywords are: changeProperty,changeCredential, and getCredential. Their meaning is defined as follows:

```
action
changeProperty     Permission to change (i.e., add and remove)
                   Role object properties whose names start with
                   the name argument specified in the constructor.
changeCredential   Permission to change (i.e., add and remove)
                   User object credentials whose names start
                   with the name argument specified in the constructor.
getCredential      Permission to retrieve and check for the
                   existence of User object credentials whose names
                   start with the name argument specified in the
                   constructor.
```

The action string is converted to lowercase before processing.

Following is a PermissionInfo style policy entry which grants a user administration bundle a number of UserAdminPermission object:

```
(org.osgi.service.useradmin.UserAdminPermission "admin")
(org.osgi.service.useradmin.UserAdminPermission "com.foo.*"
            "changeProperty,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.

Concurrency   Thread-safe

107.8.9.1   **public static final String ADMIN = "admin"**

The permission name "admin".

107.8.9.2   **public static final String CHANGE_CREDENTIAL = "changeCredential"**

The action string "changeCredential".

107.8.9.3   **public static final String CHANGE_PROPERTY = "changeProperty"**

The action string "changeProperty".

107.8.9.4   **public static final String GET_CREDENTIAL = "getCredential"**

The action string "getCredential".

107.8.9.5   **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.

107.8.9.6   **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 UserAdmin-Permission object.

107.8.9.7   **public String getActions()**

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

Returns   the canonical string representation of the actions.

107.8.9.8   **public int hashCode()**

☐ Returns the hash code value for this object.

Returns   A hash code value for this object.

107.8.9.9   **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.

107.8.9.10   **public PermissionCollection newPermissionCollection()**

☐ Returns a new PermissionCollection object for storing UserAdminPermission objects.

Returns   a new PermissionCollection object suitable for storing UserAdminPermission objects.

107.8.9.11   **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()

107.9   References

[1]   *The Java Security Architecture for JDK 1.2*
       Version 1.0, Sun Microsystems, October 1998

[2]   *Java Authentication and Authorization Service*
       http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html

110 Initial Provisioning Specification

Version 1.2

110.1 Introduction

To allow freedom regarding the choice of management protocol, the OSGi Specifications assumes an architecture to remotely manage a OSGi framework 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 OSGi framework, 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 OSGi framework by an Operator, and (optionally) to hand over the management of the OSGi framework 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 OSGi framework.

This specification consists of a prologue, in which the principles of the Initial Provisioning are outlined, and a number of mappings to different mechanisms.

110.1.1 Essentials
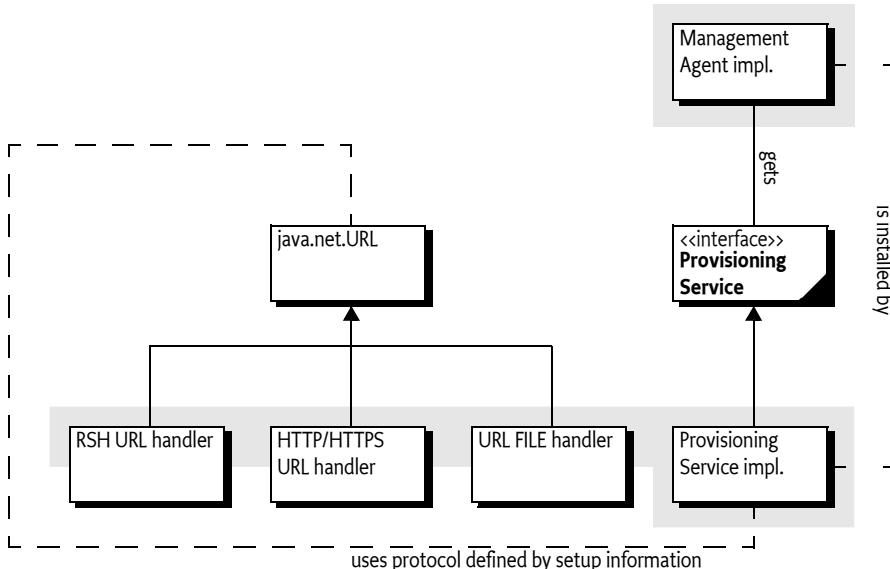
- *Policy Free* - The proposed solution must be business model agnostic; none of the affected parties (Operators, SPS Manufacturers, etc.) should be forced into any particular business model.
- *Inter-operability* - The Initial Provisioning must permit arbitrary inter-operability between management systems and OSGi frameworks. Any compliant Remote Manager should be able to manage any compliant OSGi framework, 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 OSGi framework Servers using a single management system of the Operator's choice. This rule also gives the consumer the greatest choice when selecting an Operator.
- *Flexible* - The management process should be as open as possible, to allow innovation and specialization while still achieving interoperability.

110.1.2 Entities

- *Provisioning Service* - A service registered with the Framework that provides information about the initial provisioning to the Management Agent.
- *Provisioning Dictionary* - A Dictionary object that is filled with information from the ZIP files that are loaded during initial setup.
- *RSH Protocol* - An OSGi specific secure protocol based on HTTP.
- *Management Agent* - A bundle that is responsible for managing a OSGi framework under control of a Remote Manager.

110.2   Procedure

The following procedure should be executed by an OSGi Framework implementation that supports this Initial Provisioning specification.

When the OSGi framework 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 OSGi framework 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 OSGi framework 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 OSGi framework information is appended to the URL. This information includes at least the OSGi framework Identifier, but may also contain proprietary information, as long as the keys for this information do not conflict. Different URL schemes may use different methods of appending parameters; these details are specified in the mappings of this specification to concrete protocols.

The result of the request must be a ZIP file. (The content type should be `application/zip`). It is the responsibility of the underlying protocol to guarantee the integrity and authenticity of this ZIP file.

This ZIP file is unpacked and its entries (except `bundle` and `bundle-url` entries, described in Table 110.2) are placed in a Dictionary object. This `Dictionary` object is called the *Provisioning Dictionary*. It must be made available from the Provisioning Service in the service registry. The names of the entries in the ZIP file must not start with a solidus ('/' \u002F).

The ZIP file may contain only four types of dictionary entries: text, binary, bundle, or bundle-url. The type of an entry can be specified in different ways. An Initial Provisioning service must look in the following places to find the information about an entry's (MIME) type (in the given order):

1. The manifest header InitialProvisioning-Entries of the given ZIP file. This header is defined in *InitialProvisioning-Entries Manifest Header* on page 161. If this header is present, but a given entry's path is not named then try the next step.

2. The extension of the entry path name if one of .txt, .jar, .url extensions. See Table 110.1 on page 159 for the mapping of types, MIME types, and extensions.

3. The entry is assumed to be a binary type

The types can optionally be specified as a MIME type as defined in [7] *MIME Types*. The text and bundle-url entries are translated into a String object from an UTF-8 encoded byte array. All other entries must be stored as a byte[].

Table 110.1   *Content types of provisioning ZIP file*

Type	MIME Type	Ext	Description
text	MIME_STRING text/plain;charset=utf-8	.txt	Must be represented as a String object
binary	MIME_BYTE_ARRAY application/octet-stream	not txt, .url, or .jar	Must be represented as a byte array (byte[]).
bundle	MIME_BUNDLE application/vnd.osgi.bundle MIME_BUNDLE_ALT application/x-osgi-bundle	.jar	Entries must be installed using BundleContext.installBundle(String, InputStream), with the InputStream object constructed from the contents of the ZIP entry. The location must be the name of the ZIP entry without leading solidus ('/' \u002F). This entry must not be stored in the Provisioning Dictionary. If a bundle with this location name is already installed in this system, then this bundle must be updated instead of installed. The MIME_BUNDLE_ALT version is intended for backward compatibility, it specifies the original MIME type for bundles before there was an official IANA MIME type.
bundle-url	MIME_BUNDLE_URL text/x-osgi-bundle-url; charset=utf-8	.url	The content of this entry is a string coded in utf-8. Entries must be installed using BundleContext.installBundle(String, InputStream), with the InputStream object created from the given URL. The location must be the name of the ZIP entry without leading solidus ('/' \u002F). This entry must not be stored in the Provisioning Dictionary. If a bundle with this location URL is already installed in this system, then this bundle must be updated instead of installed.

The Provisioning Service must install (but not start) all entries in the ZIP file that are typed with bundle or bundle-url.

If an entry named PROVISIONING_START_BUNDLE is present in the Provisioning Dictionary, then its content type must be text as defined in Table 110.1. The content of this entry must match the bundle location of a previously loaded bundle. This designated bundle must be given AllPermission and started.

If no PROVISIONING_START_BUNDLE entry is present in the Provisioning Dictionary, the Provisioning Dictionary should contain a reference to another ZIP file under the PROVISIONING_REFERENCE key. If both keys are absent, no further action must take place.

If this PROVISIONING_REFERENCE key is present and holds a String object that can be mapped to a valid URL, then a new ZIP file must be retrieved from this URL. The PROVISIONING_REFERENCE link may be repeated multiple times in successively loaded ZIP files.

Referring to a new ZIP file with such a URL allows a manufacturer to place a fixed reference inside the OSGi framework Server (in a file or smart card) that will provide some platform identifying information and then also immediately load the information from the management system. The PROVISIONING_REFERENCE link may be repeated multiple times in successively loaded ZIP files. The entry PROVISIONING_UPDATE_COUNT must be an Integer object that must be incremented on every iteration.
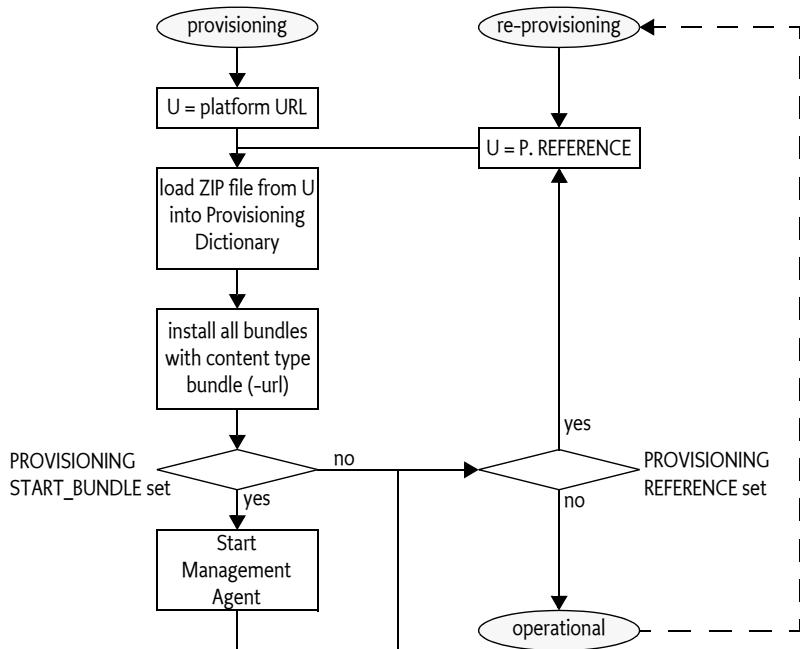
Information retrieved while loading subsequent PROVISIONING_REFERENCE URLs may replace previous key/values in the Provisioning Dictionary, but must not erase unrecognized key/values. For example, if an assignment has assigned the key proprietary-x, with a value '3', then later assignments must not override this value, unless the later loaded ZIP file contains an entry with that name. All these updates to the Provisioning Dictionary must be stored persistently. At the same time, each entry of type bundle or bundle-url (see Table 110.1) must be installed and not started.

Once the Management Agent has been started, the Initial Provisioning service has become operational. In this state, the Initial Provisioning service must react when the Provisioning Dictionary is updated with a new PROVISIONING_REFERENCE property. If this key is set, it should start the cycle again. For example, if the control of a OSGi framework needs to be transferred to another Remote Manager, the Management Agent should set the PROVISIONING_REFERENCE to the location of this new Remote Manager's Initial Provisioning ZIP file. This process is called *re-provisioning*.

If errors occur during this process, the Initial Provisioning service should try to notify the Service User of the problem.

The previous description is depicted in Figure 110.2 as a flow chart.

Figure 110.2   *Flow chart installation Management Agent bundle*

The Management Agent may require configuration data that is specific to the OSGi framework 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 OSGi framework. Transferring the data separately will make it possible to simplify the implementation on the server side, as it is not necessary to create *personalized* OSGi framework 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 OSGi framework Identifier.

110.2.1   InitialProvisioning-Entries Manifest Header

The InitialProvisioning-Entries manifest header optionally specifies the type of the entries in the ZIP file. The syntax for this header is:

```
InitialProvisioning-Entries ::= ip-entry (',' ip-entry ) *
ip-entry                     ::= path ( ';' parameter ) *
```

The entry is the path name of a resource in the ZIP file. This InitialProvisioning-Entries header recognizes the following attribute:

* type - Gives the type of the dictionary entry. The type can have one of the following values: text, binary, bundle, or bundle-url

If the type parameter entry is not specified for an entry, then the type will be inferred from the extension of the entry, as defined in table Table 110.1 on page 159.

110.3   Special Configurations

The next section shows some examples of specially configured types of OSGi framework Servers and how they are treated with the respect to the specifications in this document.

110.3.1   Branded OSGi framework Server

If a OSGi framework Operator is selling OSGi framework Servers branded exclusively for use with their service, the provisioning will most likely be performed prior to shipping the OSGi framework Server to the User. Typically the OSGi framework 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 OSGi framework is probably equipped with necessary security entities, like certificates, to enable secure downloads from the Operator's URL over open networks, if necessary.

110.3.2   Non-connected OSGi framework

Circumstances might exist in which the OSGi framework 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 OSGi framework.

110.4   The Provisioning Service

Provisioning information is conveyed between bundles using the Provisioning Service, as defined in the ProvisioningService interface. The Provisioning Dictionary is retrieved from the ProvisioningService object using the getInformation() method. This is a read-only Dictionary object, any changes to this Dictionary object must throw an UnsupportedOperationException.

The Provisioning Service provides a number of methods to update the Provisioning Dictionary.

- addInformation(Dictionary) - Add all key/value pairs in the given Dictionary object to the Provisioning Dictionary.
- addInformation(ZipInputStream) - It is also possible to add a ZIP file to the Provisioning Service immediately. This will unpack the ZIP file and add the entries to the Provisioning Dictionary. This method must install the bundles contained in the ZIP file as described in *Procedure* on page 158.
- setInformation(Dictionary) - Set a new Provisioning Dictionary. This will remove all existing entries.

Each of these method will increment the PROVISIONING_UPDATE_COUNT entry.

110.5   Management Agent Environment

The Management Agent should be written with great care to minimize dependencies on other packages and services, as *all* services in OSGi are optional. Some OSGi frameworks 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 OSGi framework. Installing other bundles might even involve downloading a more full featured Management Agent to replace the initial Management Agent.

110.6   Mapping To File Scheme

The file: scheme is the simplest and most completely supported scheme which can be used by the Initial Provisioning specification. It can be used to store the configuration data and Management Agent bundle on the OSGi framework Server, and avoids any outside communication.

If the initial request URL has a file scheme, no parameters should be appended, because the file: scheme does not accept parameters.

110.6.1   Example With File Scheme

The manufacturer should prepare a ZIP file containing only one entry named PROVISIONING_START_BUNDLE that contains a location string of an entry of type bundle or bundle-url. For example, the following ZIP file demonstrates this:

```
provisioning.start.bundle  text        agent
agent                      bundle      C0AF0E9B2AB..
```

The bundle may also be specified with a URL:

```
provisioning.start.bundle  text        http://acme.com/a.jar
agent                      bundle-url  http://acme.com/a.jar
```

Upon startup, the framework is provided with the URL with the file: scheme that points to this ZIP file:

```
file:/opt/osgi/ma.zip
```

110.7   Mapping To HTTP(S) Scheme

This section defines how HTTP and HTTPS URLs must be used with the Initial Provisioning specification.

• HTTP - May be used when the data exchange takes place over networks that are secured by other means, such as a Virtual Private Network (VPN) or a physically isolated network. Otherwise, HTTP is not a valid scheme because no authentication takes place.
• HTTPS - May be used if the OSGi framework is equipped with appropriate certificates.

HTTP and HTTPS share the following qualities:

- Both are well known and widely used
- Numerous implementations of the protocols exist
- Caching of the Management Agent will be desired in many implementations where limited bandwidth is an issue. Both HTTP and HTTPS already contain an accepted protocol for caching.

Both HTTP and HTTPS must be used with the GET method. The response is a ZIP file, implying that the response header `Content-Type` header must contain `application/zip`.

110.7.1   HTTPS Certificates

In order to use HTTPS, certificates must be in place. These certificates, that are used to establish trust towards the Operator, may be made available to the OSGi framework using the Provisioning Service. The root certificate should be assigned to the Provisioning Dictionary before the HTTPS provider is used. Additionally, the OSGi framework should be equipped with a OSGi framework certificate that allows the OSGi framework to properly authenticate itself towards the Operator. This specification does not state how this certificate gets installed into the OSGi framework.

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 OSGi framework. 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 164.

110.7.2   Certificate Encoding

Root certificates are X.509 certificates. Each individual certificate is stored as a byte[] object. This byte[] object is encoded in the default Java manner, as follows:

- The original, binary certificate data is DER encoded
- The DER encoded data is encoded into base64 to make it text.
- The base64 encoded data is prefixed with

  ```
  -----BEGIN CERTIFICATE-----
  ```

  and suffixed with:

  ```
  -----END CERTIFICATE-----
  ```

- If a record contains more than one certificate, they are simply appended one after the other, each with a delimiting prefix and suffix.

The decoding of such a certificate may be done with the java.security.cert.CertificateFactory class:

```
InputStream bis = new ByteArrayInputStream(x509);// byte[]
CertificateFactory cf =
    CertificateFactory.getInstance("X.509");
Collection c = cf.generateCertificates(bis);
Iterator i = c.iterator();
while (i.hasNext()) {
    Certificate cert = (Certificate)i.next();
    System.out.println(cert);
}
```

110.7.3   **URL Encoding**

The URL must contain the OSGi framework 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 OSGi framework Identifier.

If the request URL already contains HTTP parameters (if there is a '?' in the request), the `service_platform_id` is appended to this URL as an additional parameter. If, on the other hand, the request URL does not contain any HTTP parameters, the `service_platform_id` will be appended to the URL after a '?', becoming the first HTTP parameter. The following two examples show these two variants:

```
http://server.operator.com/service-x? «
    foo=bar&service_platform_id=VIN:123456789
```

```
http://server.operator.com/service-x? «
    service_platform_id=VIN:123456789
```

Proper URL encoding must be applied when the URL contains characters that are not allowed. See [6] *RFC 2396 - Uniform Resource Identifier (URI)*.

110.8   Mapping To RSH Scheme

The RSH protocol is an OSGi-specific protocol, and is included in this specification because it is optimized for Initial Provisioning. It requires a shared secret between the management system and the OSGi framework 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 OSGi framework and the Operator prior to the start of the protocol execution.

The protocol is based on an ordinary HTTP GET request/response, in which the request must be *signed* and the response must be *encrypted* and *authenticated*. Both the *signature* and *encryption key* are derived from the shared secret using Hashed Message Access Codes (HMAC) functions.

As additional input to the HMAC calculations, one client-generated nonce and one server-generated nonce are used to prevent replay attacks. The nonces are fairly large random numbers that must be generated in relation to each invocation of the protocol, in order to guarantee freshness. These nonces are called `clientfg` (client-generated freshness guarantee) and `serverfg` (server-generated freshness guarantee).

In order to separate the HMAC calculations for authentication and encryption, each is based on a different constant value. These constants are called the *authentication constant* and the *encryption constant*.

From an abstract perspective, the protocol may be described as follows.

- δ - Shared secret, 160 bits or more
- s - Server nonce, called `servercfg`, 128 bits
- c - Client nonce, called `clientfg`, 128 bits
- K_a - Authentication key, 160 bits
- K_e - Encryption key, 192 bits
- r - Response data
- e - Encrypted data
- E - Encryption constant, a `byte[]` of 05, 36, 54, 70, 00 (hex)
- A - Authentication constant, a `byte[]` of 00, 4f, 53, 47, 49 (hex)

- M - Message material, used for K_e calculation.
- m - The calculated message authentication code.
- *3DES* - Triple DES, encryption function, see [8] *3DES*. The bytes of the key must be set to odd parity. CBC mode must be used where the padding method is defined in [9] *RFC 1423 Part III: Algorithms, Modes, and Identifiers*. In [11] *Java Cryptography API (part of Java 1.4)* this is addressed as `PKCS5Padding`.
- *IV* - Initialization vector for 3DES.
- *SHA1* - Secure Hash Algorithm to generate the Hashed Message Authentication Code, see [12] *SHA-1*. The function takes a single parameter, the block to be worked upon.
- *HMAC* - The function that calculates a message authentication code, which must HMAC-SHA1. HMAC-SHA1 is defined in [1] *HMAC: Keyed-Hashing for Message Authentication*. The HMAC function takes a key and a block to be worked upon as arguments. Note that the lower 16 bytes of the result must be used.
- *{}* - Concatenates its arguments
- *[]* - Indicates access to a sub-part of a variable, in bytes. Index starts at one, not zero.

In each step, the emphasized server or client indicates the context of the calculation. If both are used at the same time, each variable will have server or client as a subscript.

1. The *client* generates a random nonce, stores it and denotes it `clientfg`

   $c = nonce$

2. The client sends the request with the `clientfg` to the server.

   $c_{server} \Leftarrow c_{client}$

3. The *server* generates a nonce and denotes it `serverfg`.

   $s = nonce$

4. The *server* calculates an authentication key based on the SHA1 function, the shared secret, the received `clientfg`, the `serverfg` and the authentication constant.

   $K_a \leftarrow SHA1(\{δ, c, s, A\})$

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.

   $M[1, 20] \leftarrow SHA1(\{ δ, c, s, E\})$

   $M[21, 40] \leftarrow SHA1(\{ δ, M[1, 20], c, s, E\})$

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 step 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

$s_{\text{client}} \Leftarrow s_{\text{server}}$

$m_{\text{client}} \Leftarrow m_{\text{server}}$

$e_{\text{client}} \Leftarrow e_{\text{server}}$

The *client* calculates the encryption key K_e the same way the server did in steps 5 and 6, and uses this to decrypt the encrypted response data. The `serverfg` value received in the response is used in the calculation.

$r \leftarrow 3DES(\ K_e,\ IV,\ e)$

9. The *client* performs the calculation of the MAC *m'* in the same way the server did, and checks that the results match the received MAC *m.* If they do not match, further processing is discarded. The `serverfg` value received in the response is used in the calculation.
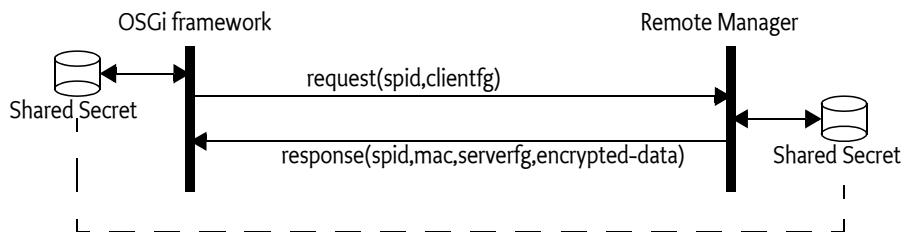
$K_a \leftarrow SHA1(\{\delta,\ c,\ s,\ A\})$

$m' \leftarrow HMAC(\ K_a,\ e)$

$m' = m$

Figure 110.3   *Action Diagram for RSH*

110.8.1   **Shared Secret**

The *shared secret* should be a key of length 160 bits (20 bytes) or more. The length is selected to match the output of the selected hash algorithm [2] *NIST, FIPS PUB 180-1: Secure Hash Standard, April 1995.*.

In some scenarios, the shared secret is generated by the Operator and communicated to the User, who inserts the secret into the OSGi framework through some unspecified means.

The opposite is also possible: the shared secret can be stored within the OSGi framework, extracted from it, and then communicated to the Operator. In this scenario, the source of the shared secret could be either the OSGi framework 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 OSGi framework it is to support. To be able to resolve this issue, the server must typically also have access to the OSGi framework Identifier of the OSGi framework. The normal way for the server to know the OSGi framework 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 OSGi framework device using a mechanism not covered by this specification. Or the device itself may generate the new secret and convey it to the owner of the device using a display device or read-out, which is then communicated to the new operator out-of-band. Additionally, the generation of the new secret may be triggered by some external event, like holding down a button for a specified amount of time.

110.8.2 Request Coding

RSH is mapped to HTTP or HTTPS. Thus, the request parameters are URL encoded as discussed in *URL Encoding* on page 165. 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 168.

110.8.3 Response Coding

The server's response to the client is composed of three parts:

- A header containing the protocol version and the serverfg
- The MAC
- The encrypted response

These three items are packaged into a binary container according to Table 110.2.

Table 110.2   *RSH Header description*

Bytes	Description	Value hex
4	Number of bytes in header	2E
1	Major version number	01
1	Minor version number	00
16	serverfg	...
4	Number of bytes in MAC	10
16	Message Authentication Code	MAC
4	Number of bytes of encrypted ZIP file	N
N	Encrypted ZIP file	...

The response content type is an RSH-specific encrypted ZIP file, implying that the response header Content-Type must be application/x-rsh for the HTTP request. When the content file is decrypted, the content must be a ZIP file.

110.8.4 RSH URL

The RSH URL must be used internally within the OSGi framework to indicate the usage of RSH for initial provisioning. The RSH URL format is identical to the HTTP URL format, except that the scheme is rsh: instead of http:. For example (« means line continues on next line):

```
rsh://server.operator.com/service-x
```

110.8.5 Extensions to the Provisioning Service Dictionary

RSH specifies one additional entry for the Provisioning Dictionary:

PROVISIONING_RSH_SECRET

The value of this entry is a byte[] containing the shared secret used by the RSH protocol.

110.8.6 RSH Transport

RSH is mapped to HTTP or HTTPS and follows the same URL encoding rules, except that the clientfg is additionally appended to the URL. The key in the URL must be clientfg and the value must be encoded in base 64 format:

The clientfg parameter is transported as an HTTP parameter that is appended after the service_platform_id parameter. The second example above would then be:

```
rsh://server.operator.com/service-x
```

Which, when mapped to HTTP, must become:

```
http://server.operator.com/service-x «
    service_platform_id=VIN:123456789& «
    clientfg=AHPmWcw%2FsiWYC37xZNdKvQ%3D%3D
```

110.9   Exception Handling

The Initial Provisioning process is a sensitive process that must run without user supervision. There is therefore a need to handle exceptional cases in a well defined way to simplify trouble shooting.

There are only 2 types of problems that halt the provisioning process. They are:

- IO Exception when reading or writing provisioning information.
- IO Exception when retrieving or processing a provisioning zip file.

Other exceptions can occur and the Provisioning Service must do any attempt to log these events.

In the cases that the provisioning process stops, it is important that the clients of the provisioning service have a way to find out that the process is stopped. The mechanism that is used for this is a special entry in the provisioning dictionary. The name of the entry must be provisioning.error. The value is a String object with the following format:

- Numeric error code
- Space
- A human readable string describing the error.

Permitted error codes are:

- 0 - Unknown error
- 1 - Couldn't load or save provisioning information
- 2 - Malformed URL Exception
- 3 - IO Exception when retrieving document of a URL
- 4 - Corrupted Zip Input Stream

The provisioning.update.count will be incremented as normal when a provisioning.error entry is added to the provisioning information. After, the provisioning service will take no further action.

Some examples:

```
0 SIM card removed
2 "http://www.acme.com/secure/blib/ifa.zip"
```

110.10   Security

The security model for the OSGi framework 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 OSGi framework by inserting the smart card into the OSGi framework.

110.10.1   Concerns

The major security concerns related to the deployment of the Management Agent are:

- The OSGi framework is controlled by the intended Operator
- The Operator controls the intended OSGi framework(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 OSGi framework
- The OSGi framework authenticates itself to the Operator
- The integrity and confidentiality of the Management Agent, certificates, and configuration data are fully protected if they are transported over public transports.

Each mapping of the Initial Provisioning specification to a concrete implementation must describe how these goals are met.

110.10.2   OSGi framework Long-Term Security

Secrets for long-term use may be exchanged during the Initial Provisioning procedures. This way, one or more secrets may be shared securely, assuming that the Provisioning Dictionary assignments used are implemented with the proper security characteristics.

110.10.3   Permissions

The provisioning information may contain sensitive information. Also, the ability to modify provisioning information can have drastic consequences. Thus, only trusted bundles should be allowed to register, or get the Provisioning Service. This restriction can be enforced using ServicePermission[ProvisioningService, GET].

No Permission classes guard reading or modification of the Provisioning Dictionary, so care must be taken not to leak the Dictionary object received from the Provisioning Service to bundles that are not trusted.

Whether message-based or connection-based, the communications used for Initial Provisioning must support mutual authentication and message integrity checking, at a minimum.

By using both server and client authentication in HTTPS, the problem of establishing identity is solved. In addition, HTTPS will encrypt the transmitted data. HTTPS requires a Public Key Infrastructure implementation in order to retrieve the required certificates.

When RSH is used, it is vital that the shared secret is shared only between the Operator and the OSGi framework, and no one else.

110.11   org.osgi.service.provisioning

Provisioning Package Version 1.2.

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

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

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

110.11.1 Summary

- `ProvisioningService` - Service for managing the initial provisioning information.

110.11.2 public interface ProvisioningService

Service for managing the initial provisioning information.

Initial provisioning of an OSGi device is a multi step process that culminates with the installation and execution of the initial management agent. At each step of the process, information is collected for the next step. Multiple bundles may be involved and this service provides a means for these bundles to exchange information. It also provides a means for the initial Management Bundle to get its initial configuration information.

The provisioning information is collected in a `Dictionary` object, called the Provisioning Dictionary. Any bundle that can access the service can get a reference to this object and read and update provisioning information. The key of the dictionary is a `String` object and the value is a `String` or `byte[]` object. The single exception is the PROVISIONING_UPDATE_COUNT value which is an Integer. The `provisioning` prefix is reserved for keys defined by OSGi, other key names may be used for implementation dependent provisioning systems.

Any changes to the provisioning information will be reflected immediately in all the dictionary objects obtained from the Provisioning Service.

Because of the specific application of the Provisioning Service, there should be only one Provisioning Service registered. This restriction will not be enforced by the Framework. Gateway operators or manufactures should ensure that a Provisioning Service bundle is not installed on a device that already has a bundle providing the Provisioning Service.

The provisioning information has the potential to contain sensitive information. Also, the ability to modify provisioning information can have drastic consequences. Thus, only trusted bundles should be allowed to register and get the Provisioning Service. The `ServicePermission` is used to limit the bundles that can gain access to the Provisioning Service. There is no check of `Permission` objects to read or modify the provisioning information, so care must be taken not to leak the Provisioning Dictionary received from `getInformation` method.

No Implement   Consumers of this API must not implement this interface

110.11.2.1 public static final String INITIALPROVISIONING_ENTRIES = "InitialProvisioning-Entries"

Name of the header that specifies the type information for the ZIP file entries.

Since 1.2

110.11.2.2 public static final String MIME_BUNDLE = "application/vnd.osgi.bundle"

MIME type to be stored in the extra field of a `ZipEntry` object for an installable bundle file. Zip entries of this type will be installed in the framework, but not started. The entry will also not be put into the information dictionary.

110.11.2.3 public static final String MIME_BUNDLE_ALT = "application/x-osgi-bundle"

Alternative MIME type to be stored in the extra field of a `ZipEntry` object for an installable bundle file. Zip entries of this type will be installed in the framework, but not started. The entry will also not be put into the information dictionary. This alternative entry is only for backward compatibility, new applications are recommended to use `MIME_BUNDLE`, which is an official IANA MIME type.

Since 1.2

110.11.2.4   **public static final String MIME_BUNDLE_URL = "text/x-osgi-bundle-url"**

MIME type to be stored in the extra field of a ZipEntry for a String that represents a URL for a bundle. Zip entries of this type will be used to install (but not start) a bundle from the URL. The entry will not be put into the information dictionary.

110.11.2.5   **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.

110.11.2.6   **public static final String MIME_STRING = "text/plain;charset=utf-8"**

MIME type to be stored in the extra field of a ZipEntry object for String data.

110.11.2.7   **public static final String PROVISIONING_AGENT_CONFIG = "provisioning.agent.config"**

The key to the provisioning information that contains the initial configuration information of the initial Management Agent. The value will be of type byte[].

110.11.2.8   **public static final String PROVISIONING_REFERENCE = "provisioning.reference"**

The key to the provisioning information that contains the location of the provision data provider. The value must be of type String.

110.11.2.9   **public static final String PROVISIONING_ROOTX509 = "provisioning.rootx509"**

The key to the provisioning information that contains the root X509 certificate used to establish trust with operator when using HTTPS.

110.11.2.10   **public static final String PROVISIONING_RSH_SECRET = "provisioning.rsh.secret"**

The key to the provisioning information that contains the shared secret used in conjunction with the RSH protocol.

110.11.2.11   **public static final String PROVISIONING_SPID = "provisioning.spid"**

The key to the provisioning information that uniquely identifies the Service Platform. The value must be of type String.

110.11.2.12   **public static final String PROVISIONING_START_BUNDLE = "provisioning.start.bundle"**

The key to the provisioning information that contains the location of the bundle to start with AllPermission. The bundle must have be previously installed for this entry to have any effect.

110.11.2.13   **public static final String PROVISIONING_UPDATE_COUNT = "provisioning.update.count"**

The key to the provisioning information that contains the update count of the info data. Each set of changes to the provisioning information must end with this value being incremented. The value must be of type Integer. This key/value pair is also reflected in the properties of the ProvisioningService in the service registry.

110.11.2.14   **public void addInformation(Dictionary info)**

info   the set of Provisioning Information key/value pairs to add to the Provisioning Information dictionary. Any keys are values that are of an invalid type will be silently ignored.

□   Adds the key/value pairs contained in info to the Provisioning Information dictionary. This method causes the PROVISIONING_UPDATE_COUNT to be incremented.

110.11.2.15   **public void addInformation(ZipInputStream zis) throws IOException**

zis   the ZipInputStream that will be used to add key/value pairs to the Provisioning Information dictionary and install and start bundles. If a ZipEntry does not have an Extra field that corresponds to one of the four defined MIME types (MIME_STRING, MIME_BYTE_ARRAY, MIME_BUNDLE, and MIME_BUNDLE_URL) in will be silently ignored.

☐ Processes the ZipInputStream and extracts information to add to the Provisioning Information dictionary, as well as, install/update and start bundles. This method causes the PROVISIONING_UPDATE_COUNT to be incremented.

Throws IOException– if an error occurs while processing the ZipInputStream. No additions will be made to the Provisioning Information dictionary and no bundles must be started or installed.

110.11.2.16   **public Dictionary getInformation()**

☐ Returns a reference to the Provisioning Dictionary. Any change operations (put and remove) to the dictionary will cause an UnsupportedOperationException to be thrown. Changes must be done using the setInformation and addInformation methods of this service.

Returns A reference to the Provisioning Dictionary.

110.11.2.17   **public void setInformation(Dictionary info)**

info the new set of Provisioning Information key/value pairs. Any keys are values that are of an invalid type will be silently ignored.

☐ Replaces the Provisioning Information dictionary with the key/value pairs contained in info. Any key/value pairs not in info will be removed from the Provisioning Information dictionary. This method causes the PROVISIONING_UPDATE_COUNT to be incremented.

110.12   References

[1]   *HMAC: Keyed-Hashing for Message Authentication*
http://www.ietf.org/rfc/rfc2104.txt Krawczyk ,et. al. 1997.

[2]   *NIST, FIPS PUB 180-1: Secure Hash Standard, April 1995.*

[3]   *Hypertext Transfer Protocol - HTTP/1.1*
http://www.ietf.org/rfc/rfc2616.txt Fielding, R., et. al.

[4]   *Rescorla, E., HTTP over TLS, IETF RFC 2818, May 2000*
http://www.ietf.org/rfc/rfc2818.txt.

[5]   *ZIP Archive format*
http://www.pkware.com/support/zip-app-note/archives

[6]   *RFC 2396 - Uniform Resource Identifier (URI)*
http://www.ietf.org/rfc/rfc2396.txt

[7]   *MIME Types*
http://www.ietf.org/rfc/rfc2046.txt
http://www.iana.org/assignments/media-types

[8]   *3DES*
W/ Tuchman, "Hellman Presents No Shortcut Solution to DES," IEEE Spectrum, v. 16, n. 7 July 1979, pp40-41.

[9]   *RFC 1423 Part III: Algorithms, Modes, and Identifiers*
http://www.ietf.org/rfc/rfc1423.txt

[10]   *PKCS 5*
ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2

[11]   *Java Cryptography API (part of Java 1.4)*
http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html/

[12]   *SHA-1*
U.S. Government, Proposed Federal Information Processing Standard for Secure Hash Standard, January 1992

[13]   *Transport Layer Security*
        http://www.ietf.org/rfc/rfc2246.txt, January 1999, The TLS Protocol Version 1.0, T. Dierks & C. Allen.

112   Declarative Services Specification

Version 1.3

112.1   Introduction

The OSGi Framework contains a procedural service model which provides a publish/find/bind model for using *services.* This model is elegant and powerful, it enables the building of applications out of bundles that communicate and collaborate using these services.

This specification addresses some of the complications that arise when the OSGi service model is used for larger systems and wider deployments, such as:

- *Startup Time* - The procedural service model requires a bundle to actively register and acquire its services. This is normally done at startup time, requiring all present bundles to be initialized with a Bundle Activator. In larger systems, this quickly results in unacceptably long startup times.
- *Memory Footprint* - A service registered with the Framework implies that the implementation, and related classes and objects, are loaded in memory. If the service is never used, this memory is unnecessarily occupied. The creation of a class loader may therefore cause significant overhead.
- *Complexity* - Service can come and go at any time. This dynamic behavior makes the service programming model more complex than more traditional models. This complexity negatively influences the adoption of the OSGi service model as well as the robustness and reliability of applications because these applications do not always handle the dynamicity correctly.

The *service component* model uses a declarative model for publishing, finding and binding to OSGi services. This model simplifies the task of authoring OSGi services by performing the work of registering the service and handling service dependencies. This minimizes the amount of code a programmer has to write; it also allows service components to be loaded only when they are needed. As a result, bundles need not provide a `BundleActivator` class to collaborate with others through the service registry.

From a system perspective, the service component model means reduced startup time and potentially a reduction of the memory footprint. From a programmer's point of view the service component model provides a simplified programming model.

The Service Component model makes use of concepts described in [1] *Automating Service Dependency Management in a Service-Oriented Component Model.*

112.1.1   Essentials

- *Backward Compatibility* - The service component model must operate seamlessly with the existing service model.
- *Size Constraints* - The service component model must not require memory and performance intensive subsystems. The model must also be applicable on resource constrained devices.
- *Delayed Activation* - The service component model must allow delayed activation of a service component. Delayed activation allows for delayed class loading and object creation until needed, thereby reducing the overall memory footprint.
- *Simplicity* - The programming model for using declarative services must be very simple and not require the programmer to learn a complicated API or XML sub-language.
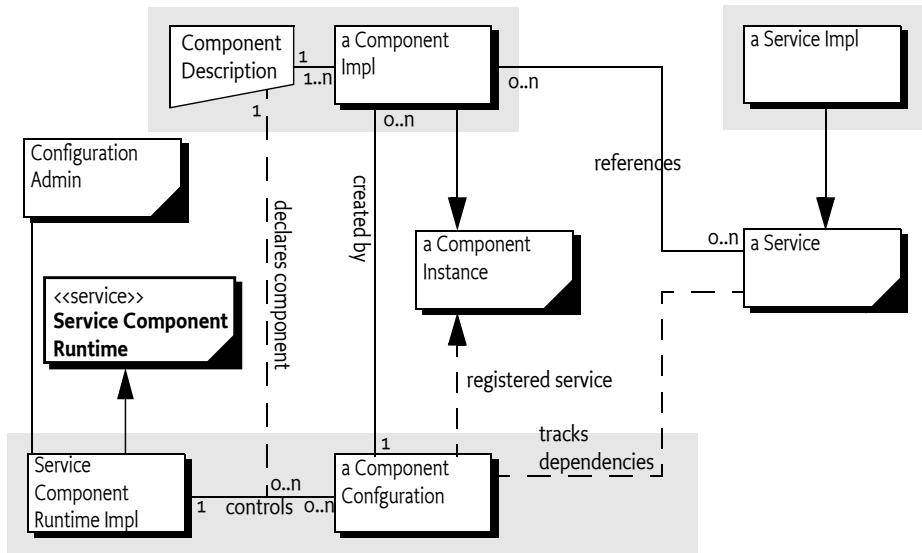
- *Reactive* - It must be possible to react to changes in the external dependencies with different policies.
- *Annotations* - Annotations must be provided that can leverage the type information to create the XML descriptor.
- *Introspection* - It must be possible to introspect the service components.

112.1.2 Entities

- *Service Component* - A service component contains a description that is interpreted at run time to create and dispose objects depending on the availability of other services, the need for such an object, and available configuration data. Such objects can optionally provide a service. This specification also uses the generic term *component* to refer to a service component.
- *Service Component Runtime (SCR)* - The actor that manages the components and their life cycle and allows introspection of the components.
- *Component Description* - The declaration of a service component. It is contained within an XML document in a bundle.
- *Component Properties* - A set of properties which can be specified by the component description, Configuration Admin service and from the component factory.
- *Component Property Type* - A user defined annotation type which defines component properties and is implemented by SCR to provide type safe access to the defined component properties.
- *Component Configuration* - A component configuration represents a component description parameterized by component properties. It is the entity that tracks the component dependencies and manages a component instance. An activated component configuration has a component context.
- *Component Instance* - An instance of the component implementation class. A component instance is created when a component configuration is activated and discarded when the component configuration is deactivated. A component instance is associated with exactly one component configuration.
- *Delayed Component* - A component whose component configurations are activated when their service is requested.
- *Immediate Component* - A component whose component configurations are activated immediately upon becoming satisfied.
- *Factory Component* - A component whose component configurations are created and activated through the component's component factory.
- *Reference* - A specified dependency of a component on a set of target services.
- *Target Services* - The set of services that is defined by the reference interface and target property filter.
- *Bound Services* - The set of target services that are bound to a component configuration.
- *Event methods* - The bind, updated, and unbind methods associated with a Reference.

Figure 112.1   *Service Component Runtime, org.osgi.service.component package*

112.1.3   Synopsis

The Service Component Runtime reads component descriptions from started bundles. These descriptions are in the form of XML documents which define a set of components for a bundle. A component can refer to a number of services that must be available before a component configuration becomes satisfied. These dependencies are defined in the descriptions and the specific target services can be influenced by configuration information in the Configuration Admin service. After a component configuration becomes satisfied, a number of different scenarios can take place depending on the component type:

- *Immediate Component* - The component configuration of an immediate component must be activated immediately after becoming satisfied. Immediate components may provide a service.
- *Delayed Component* - When a component configuration of a delayed component becomes satisfied, SCR will register the service specified by the service element without activating the component configuration. If this service is requested, SCR must activate the component configuration creating an instance of the component implementation class that will be returned as the service object. If the scope attribute of the service element is bundle, then, for each distinct bundle that requests the service object, a different component configuration is created and activated and a new instance of the component implementation class is returned as the service object. If the scope attribute of the service element is prototype, then, for each distinct request for the service object, such as via ServiceObjects, a different component configuration is created and activated and a new instance of the component implementation class is returned as the service object.
- *Factory Component* - If a component's description specifies the factory attribute of the component element, SCR will register a Component Factory service. This service allows client bundles to create and activate multiple component configurations and dispose of them. If the component's description also specifies a service element, then as each component configuration is activated, SCR will register it as a service.

112.1.4   Readers

- *Architects* - The chapter, *Components* on page 178, gives a comprehensive introduction to the capabilities of the component model. It explains the model with a number of examples. The section about *Component Life Cycle* on page 200 provides some deeper insight in the life cycle of components.

- *Service Programmers* - Service programmers should read *Components* on page 178. This chapter should suffice for the most common cases. For the more advanced possibilities, they should consult *Component Description* on page 191 for the details of the XML grammar for component descriptions.
- *Deployers* - Deployers should consult *Deployment* on page 211.

112.2 Components

A component is a normal Java class contained within a bundle. The distinguishing aspect of a component is that it is *declared* in an XML document. Component configurations are activated and deactivated under the full control of SCR. SCR bases its decisions on the information in the component's description. This information consists of basic component information like the name and type, optional services that are implemented by the component, and *references*. References are dependencies that the component has on other services.

SCR must *activate* a component configuration when the component is enabled and the component configuration is satisfied and a component configuration is needed. During the life time of a component configuration, SCR can notify the component of changes in its bound references.

SCR will *deactivate* a previously activated component configuration when the component becomes disabled, the component configuration becomes unsatisfied, or the component configuration is no longer needed.

If an activated component configuration's configuration properties change, SCR must either notify the component configuration of the change, if the component description specifies a method to be notified of such changes, or deactivate the component configuration and then attempt to reactivate the component configuration using the new configuration information.

112.2.1 Declaring a Component

A component requires the following artifacts in the bundle:

- An XML document that contains the component description.
- The Service-Component manifest header which names the XML documents that contain the component descriptions.
- An implementation class that is specified in the component description.

The elements in the component's description are defined in *Component Description* on page 191. The XML grammar for the component declaration is defined by the XML Schema, see *Component Description Schema* on page 224.

112.2.2 Immediate Component

An *immediate component* is activated as soon as its dependencies are satisfied. If an immediate component has no dependencies, it is activated immediately. A component is an immediate component if it is not a factory component and either does not specify a service or specifies a service and the immediate attribute of the component element set to true. If an immediate component configuration is satisfied and specifies a service, SCR must register the component configuration as a service in the service registry and then activate the component configuration.

For example, the bundle entry /OSGI-INF/activator.xml contains:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.activator"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.3.0">
    <implementation class="com.acme.impl.Activator"/>
```

```
</scr:component>
```

The manifest header `Service-Component` must also be specified in the bundle manifest. For example:

```
Service-Component: OSGI-INF/activator.xml
```

An example class for this component could look like:

```
public class Activator {
    public Activator() {...}
    private void activate(BundleContext context) {...}
    private void deactivate() {...}
}
```

This example component is virtually identical to a Bundle Activator. It has no references to other services so it will be satisfied immediately. It publishes no service so SCR will activate a component configuration immediately.

The `activate` method is called when SCR activates the component configuration and the `deactivate` method is called when SCR deactivates the component configuration. If the `activate` method throws an Exception, then the component configuration is not activated and will be discarded.

112.2.3   Delayed Component

A *delayed component* specifies a service, is not specified to be a factory component and does not have the immediate attribute of the component element set to `true`. If a delayed component configuration is satisfied, SCR must register the component configuration as a service in the service registry but the activation of the component configuration is delayed until the registered service is requested. The registered service of a delayed component looks like a normal registered service but does not incur the overhead of an ordinarily registered service that require a service's bundle to be initialized to register the service.

For example, a bundle needs to see events of a specific topic. The Event Admin uses the white board pattern, receiving the events is therefore as simple as registering a Event Handler service. The example XML for the delayed component looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.handler"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.3.0">
    <implementation class="com.acme.impl.HandlerImpl"/>
    <property name="event.topics">some/topic</property>
    <service>
        <provide interface="org.osgi.service.event.EventHandler"/>
    </service>
</scr:component>
```

The associated component class looks like:

```
public class HandlerImpl implements EventHandler{
    public void handleEvent(Event evt ) {
        ...
  }
}
```

The component configuration will only be activated once the Event Admin service requires the service because it has an event to deliver on the topic to which the component subscribed.

112.2.4   **Factory Component**

Certain software patterns require the creation of component configurations on demand. For example, a component could represent an application that can be launched multiple times and each application instance can then quit independently. Such a pattern requires a factory that creates the instances. This pattern is supported with a *factory component.* A factory component is used if the factory attribute of the component element is set to a *factory identifier.* This identifier can be used by a bundle to associate the factory with externally defined information.

SCR must register a Component Factory service on behalf of the component as soon as the component factory is satisfied. The service properties must be:

- component.name - The name of the component.
- component.factory - The factory identifier.

The service properties of the Component Factory service must not include the component properties.

New configurations of the component can be created and activated by calling the newInstance method on this Component Factory service. The newInstance(Dictionary) method has a Dictionary object as argument. This Dictionary object is merged with the component properties as described in *Component Properties* on page 210. If the component specifies a service, then the service is registered after the created component configuration is satisfied with the component properties. Then the component configuration is activated.

For example, a component can provide a connection to a USB device. Such a connection should normally not be shared and should be created each time such a service is needed. The component description to implement this pattern looks like:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.factory"
    factory="usb.connection"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.3.0">
    <implementation class="com.acme.impl.USBConnectionImpl"/>
</scr:component>
```

The component class looks like:

```java
public class USBConnectionImpl implements USBConnection {
    private void activate(Map<String, ?> properties) {
     ...
    }
}
```

A factory component can be associated with a service. In that case, such a service is registered for each component configuration. For example, the previous example could provide a USB Connection service.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.factory"
    factory="usb.connection"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.3.0">
    <implementation class="com.acme.impl.USBConnectionImpl"/>
    <service>
        <provide interface="com.acme.usb.USBConnection"/>
    </service>
</scr:component>
```

The associated component class looks like:

```
public class USBConnectionImpl implements USBConnection {
    private void activate(Map<String, ?> properties) {...}
    public void connect() { ... }
    ...
    public void close() { ... }
}
```

A new service will be registered each time a new component configuration is created and activated with the newInstance method. This allows a bundle other than the one creating the component configuration to utilize the service. If the component configuration is deactivated, the service must be unregistered.

112.3　References to Services

Most bundles will require access to other services from the service registry. The dynamics of the service registry require care and attention of the programmer because referenced services, once acquired, could be unregistered at any moment. The component model simplifies the handling of these service dependencies significantly.

The services that are selected by a reference are called the *target services*. These are the services selected by the BundleContext.getServiceReferences method where the first argument is the reference's interface and the second argument is the reference's target property, which must be a valid filter.

A component configuration becomes *satisfied* when each specified reference is satisfied. A reference is *satisfied* if it specifies optional cardinality or when the number of target services is equal to or more than the minimum cardinality of the reference. An activated component configuration that becomes *unsatisfied* must be deactivated.

During the activation of a component configuration, SCR must bind some or all of the target services of a reference to the component configuration. Any target service that is bound to the component configuration is called a *bound* service. See *Binding Services* on page 204.

112.3.1　Accessing Services

A component instance must be able to use the services that are referenced by the component configuration, that is, the bound services of the references. The following strategies are available for a component instance to acquire these bound services:

- *Event strategy* - SCR calls a method on the component instance when a service becomes bound, when a service becomes unbound, or when its properties are updated. These methods are the bind, updated, and unbind methods specified by the reference. The event strategy is useful if the component needs to be notified of changes to the bound services for a dynamic reference.
- *Field strategy* - SCR modifies a field in the component instance when a service becomes bound, when a service becomes unbound, or when its properties are updated.
- *Lookup strategy* - The component instance can use one of the locateService methods of ComponentContext to locate a bound service. These methods take the name of the reference as a parameter. If the reference has a dynamic policy, it is important to not store returned service objects but look them up every time they are needed.

A component may use multiple strategies to access the bound services of a reference.

112.3.2　Event Methods

When using the event strategy, SCR must callback the component instance at the appropriate time. SCR must callback on the following events:

- bind - The bind method, if specified, is called to bind a new service to the component that match-es the selection criteria. If the policy is dynamic then the bind method of a replacement service can be called before its corresponding unbind method.
- updated - The updated method, if specified, is called when the service properties of a bound ser-vices are modified and the resulting properties do not cause the service to become unbound be-cause it is no longer selected by the target property.
- unbind - The unbind method, if specified, is called when SCR needs to unbind the service.

Each event is associated with an *event method*. The prototype of the event methods is:

```
void <method-name>(<arguments>);
```

An event method can take one or more arguments. Each argument must be of one of the following types:

- <service-type> - The bound service object.
- ServiceReference - A Service Reference for the bound service. This Service Reference may later be passed to the locateService(String,ServiceReference) method to obtain the actual service object. This approach is useful when the service properties need to be examined before accessing the service object. It also allows for the delayed activation of bound services when using the event strategy.
- ComponentServiceObjects - A Component Service Objects for the bound service. This Compo-nent Service Objects can be used to obtain the actual service object or objects. This approach is useful when the referenced service has prototype service scope and the component instance needs multiple service objects for the service.
- Map - An unmodifiable Map containing the service properties of the bound service. This Map must additionally implement Comparable with the compareTo method comparing service prop-erty maps using the same ordering as ServiceReference.compareTo based upon service ranking and service id.

A suitable method is selected using the following priority:

1. The method takes a single argument and the type of the argument is org.osgi.framework.ServiceReference. This method will receive a Service Reference for the bound service.
2. The method takes a single argument and the type of the argument is ComponentServiceOb-jects. This method will receive a Component Service Objects for the bound service.
3. The method takes a single argument and the type of the argument is the type specified by the reference's interface attribute. This method will receive the bound service object.
4. The method takes a single argument and the type of the argument is assignable from the type specified by the reference's interface attribute. If multiple methods match this rule, this implies the method name is overloaded and SCR may choose any of the methods to call. This method will receive the bound service object.
5. The method takes a single argument and the type of the argument is java.util.Map. This method will receive an unmodifiable Map containing the service properties of the bound service.
6. The method takes two or more arguments and the types of the arguments must be one of: the type specified by the reference's interface attribute, a type assignable from the type specified by the reference's interface attribute, org.osgi.framework.ServiceReference, ComponentSer-viceObjects, or java.util.Map. If multiple methods match this rule, this implies the method name is overloaded and SCR may choose any of the methods to call. In the case where the type specified by the reference's interface attribute is org.osgi.framework.ServiceReference, Compo-nentServiceObjects, or java.util.Map, the first argument of that type will receive the bound ser-vice object. If selected event method has more than one argument of that type, the remaining ar-guments of that type will receive a Service Reference for the bound service, a Service Objects for

the bound service, or an unmodifiable Map containing the service properties of the bound service.

When searching for an event method to call, SCR must locate a suitable method as specified in *Locating Component Methods and Fields* on page 220. If no suitable method is located, SCR must log an error message with the Log Service, if present, and there will be no bind, updated, or unbind notification.

The bind and unbind methods must be called once for each bound service. This implies that if the reference has multiple cardinality, then the methods may be called multiple times. The updated method can be called multiple times per service.

In the following examples, a component requires the Log Service. The first example uses the lookup strategy. The reference is declared without any bind, updated, and unbind methods:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.listen"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.3.0">
    <implementation class="com.acme.impl.LogLookupImpl"/>
    <reference name="LOG"
        interface="org.osgi.service.log.LogService"/>
</scr:component>
```

The component implementation class must now lookup the service. This looks like:

```java
public class LogLookupImpl {
    private void activate(ComponentContext ctxt) {
        LogService log = (LogService)
            ctxt.locateService("LOG");
        log.log(LogService.LOG_INFO, "Hello Components!"));
    }
}
```

Alternatively, the component could use the event strategy and ask to be notified with the Log Service by declaring bind, updated, and unbind methods.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.listen"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.3.0">
    <implementation class="com.acme.impl.LogEventImpl"/>
    <reference name="LOG"
        interface="org.osgi.service.log.LogService"
        bind="setLog"
        updated="updatedLog"
        unbind="unsetLog"
    />
</scr:component>
```

The component implementation class looks like:

```java
public class LogEventImpl {
    LogService log;
    Integer    level;
    void setLog( LogService l, Map<String,?> ref ) {
        log = l;
        updatedLog(ref);
    }
```

```
        void updatedLog( LogService l, Map<String,?> ref) {
            level = (Integer) ref.get("level");
        }
        void unsetLog( LogService l ) { log = null; }

        private void activate() {
            log.log(LogService.LOG_INFO, "Hello Components!"));
        }
    }
```

Event methods can be declared private in the component class but are only looked up in the inheritance chain when they are protected, public, or have default access. See *Locating Component Methods and Fields* on page 220.

112.3.3　　Field Strategy

When using the field strategy, SCR must modify fields in the component instance at the appropriate time. SCR must modify the fields on the following events:

- bind - The field is modified to bind a new service to the component that matches the selection criteria.
- updated - For certain field types, the field is modified when the service properties of a bound services are modified and the resulting properties do not cause the service to become unbound because it is no longer selected by the target property.
- unbind - The field is modified when SCR needs to unbind the service.

For a reference with unary cardinality, a field must be of one of the following types:

- <service-type> - The bound service object. The type of the field can be the actual service type or it can be a type that is assignable from the actual service type.
- ServiceReference - A Service Reference for the bound service. This Service Reference may later be passed to the locateService(String,ServiceReference) method to obtain the actual service object. This approach is useful when the service properties need to be examined before accessing the service object. It also allows for the delayed activation of bound services when using field strategy.
- ComponentServiceObjects - A Component Service Objects for the bound service. This Component Service Objects can be used to obtain the actual service object or objects. This approach is useful when the referenced service has prototype service scope and the component instance needs multiple service objects for the service.
- Map - An unmodifiable Map containing the service properties of the bound service. This Map must additionally implement Comparable with the compareTo method comparing service property maps using the same ordering as ServiceReference.compareTo based upon service ranking and service id.
- Map.Entry - An unmodifiable Map.Entry whose key is an unmodifiable Map containing the service properties of the bound service, as above, and whose value is the bound service object. This Map.Entry must additionally implement Comparable with the compareTo method comparing the service property map key using the same ordering as ServiceReference.compareTo based upon service ranking and service id.

If the actual service type is one of ServiceReference, ComponentServiceObjects, Map, or Map.Entry, the field will be set to the service object rather than the object about the service.

For a reference with multiple cardinality, a field must be a collection of one of the following types:

- Collection
- List

- A subtype of Collection - This type can only be used for dynamic references using the update reference field option. The component instance must initialize the field to a collection object in its constructor.

The type of objects set in the collection are specified by the field-collection-type attribute in the component description:

- service - The bound service object. This is the default field collection type.
- reference - A Service Reference for the bound service.
- serviceobjects - A Component Service Objects for the bound service.
- properties - An unmodifiable Map containing the service properties of the bound service. This Map must implement Comparable, as above.
- tuple - An unmodifiable Map.Entry whose key is an unmodifiable Map containing the service properties of the bound service, as above, and whose value is the bound service object. This Map.Entry must implement Comparable, as above.

Only instance fields of the field types above are supported. If a referenced field is declared with the static modifier or has a type other than one of the above, SCR must log an error message with the Log Service, if present, and the field must not be modified.

Care must be taken by the component implementation regarding the field. SCR has no way to know if the component implementation itself may alter the field value. The component implementation should not alter the field value and allow SCR to manage it. SCR must treat the field as if the component implementation does not alter the field value so SCR may retain its own copy of the value set in the field.

In the following examples, a component requires the Log Service.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.listen"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.3.0">
    <implementation class="com.acme.impl.LogEventImpl"/>
    <reference name="LOG"
        interface="org.osgi.service.log.LogService"
        field="log"
    />
</scr:component>
```

The component implementation class looks like:

```
public class LogEventImpl {
    LogService log;
    private void activate() {
        log.log(LogService.LOG_INFO, "Hello Components!"));
    }
}
```

Fields can be declared private in the component class but are only looked up in the inheritance chain when they are protected, public, or have default access. See *Locating Component Methods and Fields* on page 220.

112.3.4   Reference Cardinality

A component implementation is always written with a certain *cardinality* for each reference in mind. The cardinality represents two important concepts:

- *Multiplicity* - Does the component implementation assume a single service or does it explicitly handle multiple services? For example, when a component uses the Log Service, it only needs to

bind to one Log Service to function correctly. Alternatively, when the Configuration Admin uses the Configuration Listener services it needs to bind to all target services present in the service registry to dispatch its events correctly.

- *Optionality* - Can the component function without any bound service present? Some components can still perform useful tasks even when no service is available; other components must bind to at least one service before they can be useful. For example, the Configuration Admin in the previous example must still provide its functionality even if there are no Configuration Listener services present. Alternatively, an application that registers a Servlet with the Http Service has little to do when the Http Service is not present, it should therefore use a reference with a mandatory cardinality.

The cardinality is expressed with the following syntax:

```
cardinality  ::= optionality '..' multiplicity
optionality  ::= '0' | '1'
multiplicity ::= '1' | 'n'
```

The cardinality for a reference can be specified as one of four choices:

- 0..1 - Optional and unary.
- 1..1 - Mandatory and unary (Default) .
- 0..n - Optional and multiple.
- 1..n - Mandatory and multiple.

The *minimum cardinality* is specified by the optionality part of the cardinality. This is either 0 or 1. A minimum cardinality property can be used to raise the minimum cardinality of a reference from this initial value. For example, a 0..n cardinality in the component description can be raised into a 3..n cardinality at runtime by setting the minimum cardinality property for the reference to 3. This would typically be done by a deployer setting the minimum cardinality property in a configuration for the component. The minimum cardinality for a unary cardinality cannot exceed 1. See *Minimum Cardinality Property* on page 211 for more information.

A reference is *satisfied* if the number of target services is equal to or more than the minimum cardinality. The multiplicity is irrelevant for the satisfaction of the reference. The multiplicity only specifies if the component implementation is written to handle being bound to multiple services (n) or requires SCR to select and bind to a single service (1).

When a satisfied component configuration is activated, there must be at most one bound service for each reference with a unary cardinality and at least as many bound services as the minimum cardinality for each reference. If the cardinality constraints cannot be maintained after a component configuration is activated, that is the reference becomes unsatisfied, the component configuration must be deactivated. If the reference has a unary cardinality and there is more than one target service for the reference, then the bound service must be the target service with the highest service ranking as specified by the service.ranking property. If there are multiple target services with the same service ranking, then the bound service must be the target service with the highest service ranking and the lowest service id as specified by the service.id property.

In the following example, a component wants to register a resource with all Http Services that are available. Such a scenario has the cardinality of 0..n. The code must be prepared to handle multiple calls to the bind method for each Http Service in such a case. In this example, the code uses the registerResources method to register a directory for external access.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.listen"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.3.0">
    <implementation class="com.acme.impl.HttpResourceImpl"/>
    <reference name="HTTP"
        interface="org.osgi.service.http.HttpService"
```

```
            cardinality="0..n"
            bind="setPage"
            unbind="unsetPage"
      />
  </scr:component>

  public class HttpResourceImpl {
      private void setPage(HttpService http) {
          http.registerResources("/scr", "scr", null );
      }
      private void unsetPage(HttpService http) {
          http.unregister("/scr");
      }
  }
```

112.3.5 Reference Scope

A component implementation must be written to understand the service scope of referenced services. The *reference scope* defines whether the component expects the bundle to be exposed to a single service object for a bound service or to potentially multiple services objects. The following reference scopes are available:

- *bundle* - For all references to a given bound service, all activated component instances within a bundle must use the same service object. That is, for a given bound service, all component instances within a bundle will be using the same service object. This is the default reference scope.
- *prototype* - For all references to a given bound service, each activated component instance may use a single, distinct service object. That is, for a given bound service, each component instance may use a distinct service object but within a component instance all references to the bound service will use the same service object.
- *prototype_required* - For all references to a given bound service, each activated component instance must use a single, distinct service object. That is, for a given bound service, each component instance will use a distinct service object but within a component instance all references to the bound service will use the same service object.

For a bound service of a reference with bundle reference scope, SCR must get the service object from the OSGi Framework's service registry using the getService method on the component's Bundle Context. If the service object for a bound service has been obtained and the service becomes unbound, SCR must unget the service object using the ungetService method on the component's Bundle Context and discard all references to the service object. This ensures that the bundle will only be exposed to a single instance of the service object at any given time.

For a bound service of a reference with prototype or prototype required reference scope, SCR must use a Service Objects object obtained from the OSGi Framework's service registry using the component's Bundle Context to get any service objects. If service objects for a bound service have been obtained and the service becomes unbound, SCR must unget any unreleased service objects using the Service Objects object obtained from the OSGi Framework's service registry using the component's Bundle Context. This means that if a component instance used a Component Service Objects object to obtain service objects, SCR must track those service objects so that when the service becomes unbound, SCR can unget any unreleased service objects.

Additionally, for a bound service of a reference with prototype required reference scope, only services registered with prototype service scope can be considered as target services. This ensures that each component instance can be exposed to a single, distinct instance of the service object. Using prototype required reference scope effectively adds service.scope=prototype to the target property for the reference. A service that does not use prototype service scope cannot be used as a bound service for a reference with prototype required reference scope since the service cannot provide a distinct service object for each component instance.

112.3.6   Reference Policy

Once all the references of a component are satisfied, a component configuration can be activated and therefore bound to target services. However, the dynamic nature of the OSGi service registry makes it likely that services are registered, modified and unregistered after target services are bound. These changes in the service registry could make one or more bound services no longer a target service thereby making obsolete any object references that the component has to these service objects. Components therefore must specify a *policy* how to handle these changes in the set of bound services. A *policy-option* can further refine how changes affect bound services.

112.3.6.1   Static Reference Policy

The *static policy* is the most simple policy and is the default policy. A reference with a static policy is called a *static reference.* A component instance never sees any of the dynamics of the static reference. The bind method is called and/or the field is set before the component instance is activated. Component configurations are deactivated before any bound service for the static reference becomes unavailable. If a target service is available to replace the bound service which became unavailable, the component configuration must be reactivated and the replacement service is bound to the new component instance.

If the policy-option is reluctant then the registration of an additional target service for a reference must not result in deactivating and reactivating a component configuration. If the policy-option is greedy then the component configuration must be reactivated when new applicable services become available. See Table 112.1 on page 189.

If a static reference specifies an updated method and the bound service's properties change, SCR must call the updated method.

The static policy can be very expensive if it depends on services that frequently unregister and re-register or if the cost of activating and deactivating a component configuration is high. Static policy is usually also not applicable if the cardinality specifies multiple bound services.

112.3.6.2   Dynamic Reference Policy

The *dynamic policy* is slightly more complex since the component implementation must properly handle changes in the set of bound services that can occur on any thread at any time after the component instance is created. A reference with a dynamic policy is called a *dynamic reference*. With the dynamic policy, SCR can change the set of bound services without deactivating a component configuration. If the component uses the event strategy to access services, then the component instance will be notified of changes in the set of bound services by calls to the bind, updated, and unbind methods.

If the policy-option is reluctant then a bound reference is not rebound even if a more suitable service becomes available for a 1..1 or 0..1 reference. If the policy-option is greedy then the component must be unbound and rebound for that reference. See Table 112.1 on page 189.

The previous example with the registering of a resource directory used a static policy. This implied that the component configurations are deactivated when there is a change in the bound set of Http Services. The code in the example can be seen to easily handle the dynamics of Http Services that come and go. The component description can therefore be updated to:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.listen"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.3.0">
    <implementation class="com.acme.impl.HttpResourceImpl"/>
    <reference name="HTTP"
        interface="org.osgi.service.http.HttpService"
        cardinality="0..n"
        policy="dynamic"
        bind="setPage"
```

```
                unbind="unsetPage"
        />
    </scr:component>
```

The code is identical to the previous example.

112.3.7   Reference Policy Option

The reference policy option defines how eager the reference is to rebind when a new, potentially a higher ranking, target service becomes available. The reference policy option can have the following values:

- `reluctant` - Minimize rebinding and reactivating. This is the default reference policy option.
- `greedy` - Maximize the use of the best service by deactivating static references or rebinding dynamic references.

Table 112.1 defines the actions that are taken when a *better* target service becomes available. In this context, better is when the reference is not bound or when the new target service has a higher ranking than the bound service.

Table 112.1   Action taken for policy-option when a new or higher ranking service becomes available

Cardinality	static reluctant	static greedy	dynamic reluctant	dynamic greedy
0..1	Ignore	Reactivate to bind the better target service.	If no service is bound, bind to new target service. Otherwise, ignore new target service.	If no service is bound, bind to better target service. Otherwise, unbind the bound service and bind the better target service.
1..1	Ignore	Reactivate to bind the better target service.	Ignore	Unbind the bound service, then bind the new service.
0..n	Ignore	Reactivate	Bind new target service	Bind new target service
1..n	Ignore	Reactivate	Bind new target service	Bind new target service

112.3.8   Reference Field Option

For a reference using field strategy, the reference field option defines how SCR must manage the field value. The reference field option can have the following values:

- `replace` - SCR must set the field value. Any field value set by the constructor of the component instance is overwritten. This is the default reference field option.
- `update` - SCR must update the collection set in the field. This collection can be set by the constructor of the component instance. This reference field option can only be used for a dynamic reference with multiple cardinality.

For a static reference, the replace option must be used.

For a dynamic reference, the choice of reference field option is influenced by the cardinality of the reference. For unary cardinality, the replace option must be used. For multiple cardinality, either the replace or update option can be used.

If the update option is used when not permitted, SCR must log an error message with the Log Service, if present, and the field must not be modified.

112.3.8.1   Replace Field Option

If the field is declared with the `final` modifier, SCR must log an error message with the Log Service, if present, and the field must not be modified.

For a static reference, SCR must set the value of the field before the component instance is activated and must not change the field while the component is active. This means there is a *happens-before* relationship between setting the field and activating the component instance, so the active component can safely read the field.

For a dynamic reference, the field must be declared with the `volatile` modifier so that field value changes made by SCR are visible to other threads. If the field is not declared with the `volatile` modifier, SCR must log an error message with the Log Service, if present, and the field must not be modified.

For a dynamic reference with unary cardinality, SCR must set the field with initial bound service, if any, before the component instance is activated. When there is a new bound service or the service properties of the bound service are modified and the field holds service properties, SCR must replace the field value. If the reference has optional cardinality and there is no bound service, SCR must set the field value to `null`.

For a dynamic reference with multiple cardinality, the type of the field must be `Collection` or `List`. If the field has a different type, SCR must log an error message with the Log Service, if present, and the field must not be modified. Before the component instance is activated, SCR must set the field with a new mutable collection that must contain the initial set of bound services sorted using the same ordering as `ServiceReference.compareTo` based upon service ranking and service id. The new collection may be empty if the reference has optional cardinality and there are no bound services. When there is a change in the set of bound services or the service properties of a bound service are modified and the collection holds service properties, SCR must replace the field value with a new mutable collection that must contain the updated set of bound services sorted using the same ordering as `ServiceReference.compareTo` based upon service ranking and service id. The new collection may be empty if the reference has optional cardinality and there are no bound services.

112.3.8.2   Update Field Option

The update option can only be used for a dynamic reference with multiple cardinality. The component's constructor can set the field with its choice of collection implementation. In this case, the field can be declared with the `final` modifier. The collection implementation used by the component should use identity rather than `equals` or `hashCode` to manage the elements of the collection. The collection implementation should also be thread-safe since SCR may update the collection from threads different than those used by the component instance.

After constructing the component instance, if the field value is `null`:

- If the type of the field is `Collection` or `List`, SCR will set the field to a new mutable empty collection or list object, respectively. If the field is declared with the `final` modifier, SCR must log an error message with the Log Service, if present, and the field must not be modified.
- Otherwise, SCR must log an error message with the Log Service, if present, and the field must not be modified.

SCR must not change the field value while the component is active and only update the contents of the collection. SCR must update the collection before the component instance is activated by calling `Collection.add` for each bound service. When there is a change to the set of bound services:

- SCR must call `Collection.add` for a newly bound service.
- SCR must call `Collection.remove` for an unbound service.
- If the service properties of a bound service are modified and the collection holds service properties, SCR must call `Collection.add` for the replacement element followed by `Collection.remove` for the old element.

The collection may be empty if the reference has optional cardinality and there are no bound services.

112.3.9 Selecting Target Services

The target services for a reference are constrained by the reference's interface name and target property. By specifying a filter in the target property, the programmer and deployer can constrain the set of services that should be part of the target services.

For example, a component wants to track all Component Factory services that have a factory identification of acme.application. The following component description shows how this can be done.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.listen"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.3.0">
    <implementation class="com.acme.impl.FactoryTracker"/>
    <reference name="FACTORY"
        interface=
            "org.osgi.service.component.ComponentFactory"
        target="(component.factory=acme.application)"
    />
</scr:component>
```

The filter is manifested as a component property called the *target property*. The target property can also be set by property and properties elements, see *Property and Properties Elements* on page 195. The deployer can also set the target property by establishing a configuration for the component which sets the value of the target property. This allows the deployer to override the target property in the component description. See *Target Property* on page 211 for more information.

112.3.10 Circular References

It is possible for a set of component descriptions to create a circular dependency. For example, if component A references a service provided by component B and component B references a service provided by component A then a component configuration of one component cannot be satisfied without accessing a partially activated component instance of the other component. SCR must ensure that a component instance is never accessible to another component instance or as a service until it has been fully activated, that is it has returned from its activate method if it has one.

Circular references must be detected by SCR when it attempts to satisfy component configurations and SCR must fail to satisfy the references involved in the cycle and log an error message with the Log Service, if present. However, if one of the references in the cycle has optional cardinality SCR must break the cycle. The reference with the optional cardinality can be satisfied and bound to zero target services. Therefore the cycle is broken and the other references may be satisfied.

112.4 Component Description

Component descriptions are defined in XML documents contained in a bundle and any attached fragments.

If SCR detects an error when processing a component description, it must log an error message with the Log Service, if present, and ignore the component description. Errors can include XML parsing errors and ill-formed component descriptions.

112.4.1 Annotations

A number of CLASS retention annotations have been provided to allow tools to construct the component description XML from the Java class files. These annotations will be discussed with the appropriate elements and attributes. Since the naming rules between XML and Java differ, some name changes are necessary.

Multi-word element and attribute names that use a minus sign ('-' \u002D) are changed to camel case. For example, the `configuration-pid` attribute in the `component` element is the `configurationPid` member in the `@Component` annotation. The annotation class that corresponds to an element starts with an upper case letter. For example the `component` element is represented by the `@Component` annotation.

Some elements do not have a corresponding annotation since the annotations can be parameterized by the type information in the Java class. For example, the `@Component` annotation synthesizes the `implement` element's `class` attribute from the type it is applied to.

See *Component Annotations* on page 214 for more information.

112.4.2   Service Component Header

XML documents containing component descriptions must be specified by the Service-Component header in the manifest. The value of the header is a comma separated list of paths to XML entries within the bundle.

```
Service-Component ::= header // See Common Header Syntax in Core
```

The Service-Component header has no architected directives or properties. The header can be left empty.

The last component of each path in the Service-Component header may use wildcards so that `Bundle.findEntries` can be used to locate the XML document within the bundle and its fragments. For example:

```
Service-Component: OSGI-INF/*.xml
```

A Service-Component manifest header specified in a fragment is ignored by SCR. However, XML documents referenced by a bundle's Service-Component manifest header may be contained in attached fragments.

SCR must process each XML document specified in this header. If an XML document specified by the header cannot be located in the bundle and its attached fragments, SCR must log an error message with the Log Service, if present, and continue.

112.4.3   XML Document

A component description must be in a well-formed XML document, [4] *Extensible Markup Language (XML) 1.0*, stored in a UTF-8 encoded bundle entry. The namespace for component descriptions is:

```
http://www.osgi.org/xmlns/scr/v1.3.0
```

The recommended prefix for this namespace is `scr`. This prefix is used by examples in this specification. XML documents containing component descriptions may contain a single, root `component` element or one or more `component` elements embedded in a larger document. Use of the namespace for component descriptions is mandatory. The attributes and sub-elements of a `component` element are always unqualified.

If an XML document contains a single, root `component` element which does not specify a namespace, then the `http://www.osgi.org/xmlns/scr/v1.0.0` namespace is assumed. Component descriptions using the `http://www.osgi.org/xmlns/scr/v1.0.0` namespace must be treated according to version 1.0 of this specification.

SCR must parse all `component` elements in the namespace. Elements not in this namespace must be ignored. Ignoring elements that are not recognized allows component descriptions to be embedded in any XML document. For example, an entry can provide additional information about components. These additional elements are parsed by another sub-system.

See *Component Description Schema* on page 224 for component description schema.

112.4.4   Component Element

The component element specifies the component description. The following text defines the structure of the XML grammar using a form that is similar to the normal grammar used in OSGi specifications. In this case the grammar should be mapped to XML elements:

```
<component> ::= (<property> | <properties>)*
                <service>?
                <reference>*
                <implementation>
```

SCR must not require component descriptions to specify the elements in the order listed above and as required by the XML schema. SCR must allow other orderings since arbitrary orderings of these elements do not affect the meaning of the component description. Only the relative ordering of property and properties elements and of reference elements have meaning.

The component element has the attributes and @Component annotations defined in the following table.

Table 112.2   Component Element and Annotations

Attribute	Annotation	Description
name	name	The *name* of a component must be unique within a bundle. The component name is used as a PID to retrieve component properties from the OSGi Configuration Admin service if present, unless a configuration-pid attribute has been defined. See *Deployment* on page 211 for more information. If the component name is used as a PID then it should be unique within the framework. The XML schema allows the use of component names which are not valid PIDs. Care must be taken to use a valid PID for a component name if the component should be configured by the Configuration Admin service. This attribute is optional. The default value of this attribute is the value of the class attribute of the nested implementation element. If multiple component elements in a bundle use the same value for the class attribute of their nested implementation element, then using the default value for this attribute will result in duplicate component names. In this case, this attribute must be specified with a unique value.
enabled	enabled	Controls whether the component is *enabled* when the bundle is started. The default value is true. If enabled is set to false, the component is disabled until the method enableComponent is called on the ComponentContext object. This allows some initialization to be performed by some other component in the bundle before this component can become satisfied. See *Enabled* on page 200.
factory	factory	If set to a non-empty string, it indicates that this component is a *factory component*. SCR must register a Component Factory service for each factory component. See *Factory Component* on page 180.
immediate	immediate	Controls whether component configurations must be immediately activated after becoming satisfied or whether activation should be delayed. The default value is false if the factory attribute or if the service element is specified and true otherwise. If this attribute is specified, its value must be false if the factory attribute is also specified or must be true unless the service element is also specified.

Attribute	Annotation	Description
configura-tion-policy	configurationPol-icy OPTIONAL REQUIRE IGNORE	Controls whether component configurations must be satisfied depending on the presence of a corresponding Configuration object in the OSGi Configuration Admin service. A corresponding configuration is a Configuration object where the PID is the name of the component. • optional - (default) Use the corresponding Configuration object if present but allow the component to be satisfied even if the corresponding Configuration object is not present. • require - There must be a corresponding Configuration object for the component configuration to become satisfied. • ignore - Always allow the component configuration to be satisfied and do not use the corresponding Configuration object even if it is present.
configuration-pid	configurationPid	The configuration PIDs to be used for the component in conjunction with Configuration Admin. Multiple configuration PIDs can be specified by using a whitespace separated list in the attribute. The default value for this attribute is the name of the component. The annotation uses a String[] to specify multiple configuration PIDs. The order in which configuration PIDs are specified must be preserved in the generated component description. The annotation can also use the special configuration PID name "$" to specify the name of the component. This special name must be replaced with the actual name of the component in the generated component description.
activate	Activate	Specifies the name of the method to call when a component configuration is activated. The default value of this attribute is activate. See *Activate Method* on page 204 for more information. The annotation must be applied to the activate method and can only be used once.
deactivate	Deactivate	Specifies the name of the method to call when a component configuration is deactivated. The default value of this attribute is deactivate. See *Deactivate Method* on page 207 for more information. The annotation must be applied to the deactivate method and can only be used once.
modified	Modified	Specifies the name of the method to call when the configuration properties for a component configuration is using a Configuration object from the Configuration Admin service and that Configuration object is modified without causing the component configuration to become unsatisfied. If this attribute is not specified, then the component configuration will become unsatisfied if its configuration properties use a Configuration object that is modified in any way. See *Modified Method* on page 206 for more information. The annotation must be applied to the modified method and can only be used once.

112.4.5   Implementation Element

The implementation element is required and defines the name of the component implementation class. The single attribute is defined in the following table.

Table 112.3   *Implementation Element and Annotations*

Attribute	Annotation	Description
class	Component	The Java fully qualified name of the implementation class.
		The component Component annotation will define the implementation element automatically from the type it is applied to.

The class is retrieved with the loadClass method of the component's bundle. The class must be public and have a public constructor without arguments (this is normally the default constructor) so component instances may be created by SCR with the newInstance method on Class.

If the component description specifies a service, the class must implement all interfaces that are provided by the service.

112.4.6 Property and Properties Elements

A component description can define a number of properties. These can defined inline or from a resource in the bundle. The property and properties elements can occur multiple times and they can be interleaved. This interleaving is relevant because the properties are processed from top to bottom. Later properties override earlier properties that have the same name.

Properties can also be overridden by a Configuration Admin service's Configuration object before they are exposed to the component or used as service properties. This is described in *Component Properties* on page 210 and *Deployment* on page 211.

The property element has the attributes and annotations defined in the following table.

Table 112.4   *Property Element and Annotations*

Attribute	Annotation	Description
name	Component property	The name of the property.
value		The value of the property. This value is parsed according to the property type. If the value attribute is specified, the body of the element is ignored. If the type of the property is not String, parsing of the value is done by the static valueOf(String) method in the given type. For Character types, the conversion must be handled by Integer.valueOf method, a Character is always represented by its Unicode value.
type		The type of the property. Defines how to interpret the value. The type must be one of the following Java types:

- String (default)
- Long
- Double
- Float
- Integer
- Byte
- Character
- Boolean
- Short

Attribute	Annotation	Description
‹body›		If the value attribute is not specified, the body of the property element must contain one or more values. The value of the property is then an array of the specified type. Except for String objects, the result will be translated to an array of primitive types. For example, if the type attribute specifies Integer, then the resulting array must be int[].
		Values must be placed one per line and blank lines are ignored. Parsing of the value is done by the parse methods in the class identified by the type, after trimming the line of any beginning and ending white space. String values are also trimmed of beginning and ending white space before being placed in the array.

For example, a component that needs an array of hosts can use the following property definition:

```
<property name="hosts">
        www.acme.com
        backup.acme.com
</property>
```

This property declaration results in the property hosts, with a value of String[] { "www.acme.com", "backup.acme.com" }.

A property can also be set with the property annotation element of Component. This element is an array of strings that must follow the following syntax:

```
property ::= name ( ':' type )? '=' value
```

In this case name, type, and value parts map to the attributes of the property element. If multiple values must be specified then the same name can be repeated multiple times. For example:

```
@Component(property={"foo:Integer=1", "foo:Integer=2", "foo:Integer=3"})
public class FooImpl {
   ...
}
```

The properties element references an entry in the bundle whose contents conform to a standard [3] *Java Properties File*.

At runtime, SCR reads the entry to obtain the properties and their values. The properties element attributes are defined in the following table.

Table 112.5   *Properties Element and Annotations*

Attribute	Annotation	Description
entry	Component properties	The entry path relative to the root of the bundle

For example, to include vendor identification properties that are stored in the OSGI-INF directory, the following definition could be used:

```
<properties entry="OSGI-INF/vendor.properties"/>
```

The Component properties element can be used to provide the same information, this element consists of an array of strings where each string defines an entry. The order within the array is the order that must be used for the XML. However, the annotations do not support interleaving of the generated property and properties elements.

For example:

```
@Component(properties="OSGI-INF/vendor.properties")
```

See *Ordering of Generated Properties* on page 218 for more information on the ordering of generated properties when using annotations.

112.4.7   Service Element

The service element is optional. It describes the service information to be used when a component configuration is to be registered as a service.

A service element has the following attribute defined in the following table.

Table 112.6   Service Element and Annotations

Attribute	Annotation	Description
scope	Component scope SINGLETON BUNDLE PROTOTYPE	Controls the scope of the provided service. If set to singleton, when the component is registered as a service, it must be registered as a bundle scope service but only a single component configuration must be created and activated and a new instance of the component implementation class of the component must be used for all bundles using the service. If set to bundle, when the component is registered as a service, it must be registered as a bundle scope service and a different component configuration is created and activated and a new instance of the component implementation class must be created for each bundle using the service. If set to prototype, when the component is registered as a service, it must be registered as a prototype scope service and a different component configuration is created and activated and a new instance of the component implementation class must be created for each distinct request for the service, such as via ServiceObjects.

The scope attribute must be singleton if the component is a factory component or an immediate component. This is because SCR is not free to create component configurations as necessary to support non-singleton scoped services. A component description is ill-formed if it specifies that the component is a factory component or an immediate component and scope is not singleton.

The service element must have one or more provide elements that define the service interfaces. The provide element has the attribute defined in the following table.

Table 112.7   Provide Element and Annotations

Attribute	Annotation	Description
interface	Component service	The name of the interface that this service is registered under. This name must be the fully qualified name of a Java class. For example, org.osgi.service.log.LogService. The specified Java class should be an interface rather than a class, however specifying a class is supported. The component implementation class must implement all the specified service interfaces. The Component annotation can specify the provided services, if this element is not specified all directly implemented interfaces on the component's type are defined as service interfaces. Specifying an empty array indicates that no service should be registered.

For example, a component implements an Event Handler service.

```
<service>
    <provide interface=
        "org.osgi.service.eventadmin.EventHandler"/>
</service>
```

This previous example can be generated with the following annotation:

```
@Component
public class Foo implements EventHandler { ... }
```

112.4.8   Reference Element

A *reference* declares a dependency that a component has on a set of target services. A component configuration is not satisfied, unless all its references are satisfied. A reference specifies target services by specifying their interface and an optional target property.

A reference element has the attributes defined in the following table.

Table 112.8   *Reference Element and Annotations*

Attribute	Annotation	Description
name	name	The name of the reference. This name is local to the component and can be used to locate a bound service of this reference with one of the locateService methods of ComponentContext. Each reference element within the component must have a unique name. This name attribute is optional. The default value of this attribute is the value of the interface attribute of this element. If multiple reference elements in the component use the same interface name, then using the default value for this attribute will result in duplicate reference names. In this case, this attribute must be specified with a unique name for the reference to avoid an error.
		The Reference annotation will use the name of the annotated method or field as the default reference name. If the method name begins with bind, set or add, that prefix is removed.
interface	service	Fully qualified name of the class that is used by the component to access the service. The service provided to the component must be type compatible with this class. That is, the component must be able to cast the service object to this class. A service must be registered under this name to be considered for the set of target services.
		The Reference annotation will use the type of the first argument of the annotated method or the type of the annotated field to determine the service value.
cardinality	cardinality MANDATORY OPTIONAL MULTIPLE AT_LEAST_ONE	Specifies if the reference is optional and if the component implementation support a single bound service or multiple bound services. See *Reference Cardinality* on page 185.
policy	policy STATIC DYNAMIC	The policy declares the assumption of the component about dynamicity. See *Reference Policy* on page 188.
policy-option	policyOption RELUCTANT GREEDY	Defines the policy when a better service becomes available. See *Reference Policy* on page 188.
target	target	An optional OSGi Framework filter expression that further constrains the set of target services. The default is no filter, limiting the set of matched services to all service registered under the given reference interface. The value of this attribute is used for the value of the target property of the reference. See *Target Property* on page 211.

Attribute	Annotation	Description
scope	scope BUNDLE PROTOTYPE PROTOTYPE_ REQUIRED	The reference scope for this reference. See *Reference Scope* on page 187.
bind	Reference bind	The name of a method in the component implementation class that is used to notify that a service is bound to the component configuration. For static references, this method is only called before the `activate` method. For dynamic references, this method can also be called while the component configuration is active. See *Accessing Services* on page 181. The Reference annotation will use the name of the method it is applied to as the bind method name.
updated	updated	The name of a method in the component implementation class that is used to notify that a bound service has modified its properties.
unbind	unbind	Same as bind, but is used to notify the component configuration that the service is unbound. For static references, the method is only called after the `deactivate` method. For dynamic references, this method can also be called while the component configuration is active. See *Accessing Services* on page 181.
field	Reference field	The name of a field in the component implementation class that is used to hold service(s) that are bound to the component configuration. For static references, this field is only set before the `activate` method. For dynamic references, this field can modified while the component configuration is active. See *Accessing Services* on page 181. The Reference annotation will use the name of the field it is applied to as the field name.
field-option	fieldOption REPLACE UPDATE	Defines how the field value must be managed. This is ignored if the field attribute is not set. See *Reference Field Option* on page 189.
field-collec- tion-type		Defines the types of elements in the collection referenced by the field value. This is ignored if the field attribute is not set or the cardinality is unary. See *Field Strategy* on page 184 for more information. The Reference annotation can infer the value of the collection elements from the generic type information of the annotated field.

In the generated component description, the reference elements must be ordered in ascending lexicographical order, using String.compareTo, of the names of the references.

The following code demonstrates the use of the Reference annotation for the event strategy.

```
@Component
public class FooImpl implements Foo {
  @Reference(
    policy = DYNAMIC,
    policyOption = GREEDY,
    cardinality = MANDATORY )
  void setLog( LogService log ) { ... }
  void unsetLog( LogService log ) { ... }
  void updatedLog( Map<String,?> ref ) { ... }

  @Activate
```

```
  void open() { ... }
  @Deactivate
  void close() { ... }
}
```

The following code demonstrates the use of the Reference annotation for the field strategy.

```
@Component
public class FooImpl implements Foo {
  @Reference(
    policy = DYNAMIC,
    policyOption = GREEDY,
    cardinality = MANDATORY )
  volatile LogService log;

  @Activate
  void open() { log.log(LOG_INFO, "activated"); }
  @Deactivate
  void close() { log.log(LOG_INFO, "deactivated"); }
}
```

For a reference to be used with the lookup strategy, there are no bind methods or fields to annotate with the Reference annotation. Instead Reference annotations can be specified in the reference element of the Component annotation. When used in this way, the name and service elements must be specified since there is no annotated member from which the name or service can be determined. The following code demonstrates the use of the Reference annotation for the lookup strategy.

```
@Component( reference =
  @Reference( name = "log", service = LogService.class )
)
public class FooImpl implements Foo {
  @Activate
  void open( ComponentContext context ) {
    LogService log = (LogService) context.locateService( "log" );
    ...
  }
  @Deactivate
  void close() { ... }
}
```

112.5 Component Life Cycle

112.5.1 Enabled

A component must first be *enabled* before it can be used. A component cannot be enabled unless the component's bundle is started. See *Starting Bundles* in *OSGi Core Release 6*. All components in a bundle become disabled when the bundle is stopped. So the life cycle of a component is contained within the life cycle of its bundle.

Every component can be enabled or disabled. The initial enabled state of a component is specified in the component description via the enabled attribute of the component element. See *Component Element* on page 193. Component configurations can be created, satisfied and activated only when the component is enabled.

The enabled state of a component can be controlled with the Component Context enableComponent(String) and disableComponent(String) methods. The purpose of later enabling

a component is to be able to decide programmatically when a component can become enabled. For example, an immediate component can perform some initialization work before other components in the bundle are enabled. The component descriptions of all other components in the bundle can be disabled by having enabled set to false in their component descriptions. After any necessary initialization work is complete, the immediate component can call enableComponent to enable the remaining components.

The enableComponent and disableComponent methods must return after changing the enabled state of the named component. Any actions that result from this, such as activating or deactivating a component configuration, must occur asynchronously to the method call. Therefore a component can disable itself.

All components in a bundle can be enabled by passing a null as the argument to enableComponent.

112.5.2 Satisfied

Component configurations can only be activated when the component configuration is *satisfied*. A component configuration becomes satisfied when the following conditions are all satisfied:

- The component is *enabled.*
- If the component description specifies configuration-policy=required, then a Configuration object for the component is present in the Configuration Admin service.
- Using the component properties of the component configuration, all the component's references are satisfied. A reference is satisfied when the reference specifies optional cardinality or the number of target services is equal to or more than the minimum cardinality of the reference.

Once any of the listed conditions are no longer true, the component configuration becomes *unsatisfied.* An activated component configuration that becomes unsatisfied must be deactivated.
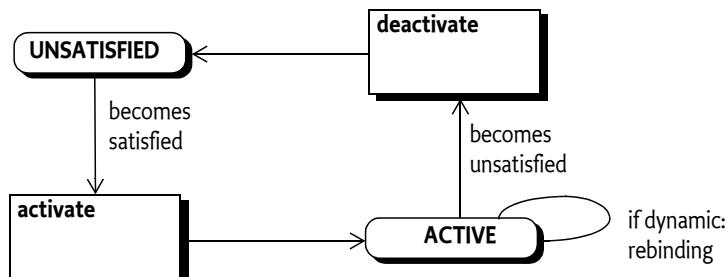
112.5.3 Immediate Component

A component is an immediate component when it must be activated as soon as its dependencies are satisfied. Once the component configuration becomes unsatisfied, the component configuration must be deactivated. If an immediate component configuration is satisfied and specifies a service, SCR must register the component configuration as a service in the service registry and then activate the component configuration. The service properties for this registration consist of the component properties as defined in *Service Properties* on page 210.

The state diagram is shown in Figure 112.2.

Figure 112.2   *Immediate Component Configuration*

112.5.4 Delayed Component

A key attribute of a delayed component is the delaying of class loading and object creation. Therefore, the activation of a delayed component configuration does not occur until there is an actual request for a service object. A component is a delayed component when it specifies a service but it is

not a factory component and does not have the `immediate` attribute of the `component` element set to `true`.

SCR must register a service after the component configuration becomes satisfied. The registration of this service must look to observers of the service registry as if the component's bundle actually registered this service. This strategy makes it possible to register services without creating a class loader for the bundle and loading classes, thereby allowing reduction in initialization time and a delay in memory footprint.

When SCR registers the service on behalf of a component configuration, it must avoid causing a class load to occur from the component's bundle. SCR can ensure this by registering a `ServiceFactory` object with the Framework for that service. By registering a `ServiceFactory` object, the actual service object is not needed until the `ServiceFactory` is called to provide the service object. The service properties for this registration consist of the component properties as defined in *Service Properties* on page 210.
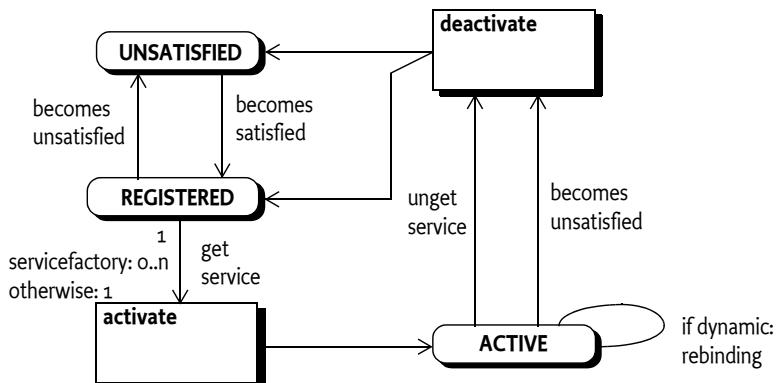
The activation of a component configuration must be delayed until its service is requested. When the service is requested, if the service has the `scope` attribute set to `bundle`, SCR must create and activate a unique component configuration for each bundle requesting the service. If the service has the `scope` attribute set to `prototype`, SCR must create and activate a unique component configuration for each distinct request for the service. Otherwise, if the service has the `scope` attribute set to `singleton`, SCR must activate a single component configuration which is used by all requests for the service. A component instance can determine the bundle it was activated for by calling the `getUsingBundle()` method on the Component Context.

The activation of delayed components is depicted in a state diagram in Figure 112.3. Notice that multiple component configurations can be created from the `REGISTERED` state if a delayed component specifies a service `scope` set to a value other than `singleton`.

If the service registered by a component configuration becomes unused because there are no more bundles using it, then SCR should deactivate that component configuration. This allows SCR implementations to eagerly reclaim activated component configurations.

Figure 112.3   *Delayed Component Configuration*

112.5.5   Factory Component

SCR must register a Component Factory service as soon as the *component factory* becomes satisfied. The component factory is satisfied when the following conditions are all satisfied:

- The component is enabled.
- Using the component properties specified by the component description, all the component's references are satisfied. A reference is satisfied when the reference specifies optional cardinality or there is at least one target service for the reference

The component factory, however, does not use any of the target services and does not bind to them.

Once any of the listed conditions are no longer true, the component factory becomes unsatisfied and the Component Factory service must be unregistered. Any component configurations activated via the component factory are unaffected by the unregistration of the Component Factory service, but may themselves become unsatisfied for the same reason.

The Component Factory service must be registered under the name org.osgi.service.component.ComponentFactory with the following service properties:
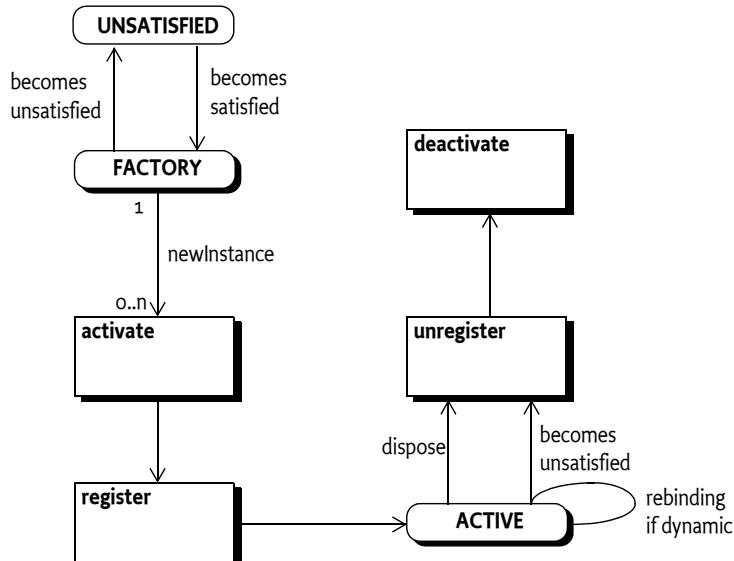
- component.name - The name of the component.
- component.factory - The value of the factory attribute.

The service properties of the Component Factory service must not include the component properties.

New component configurations are created and activated when the newInstance method of the Component Factory service is called. If the component description specifies a service, the component configuration is registered as a service under the provided interfaces. The service properties for this registration consist of the component properties as defined in *Service Properties* on page 210. The service registration must take place before the component configuration is activated. Service unregistration must take place before the component configuration is deactivated.

Figure 112.4   *Factory Component*

A Component Factory service has a single method: newInstance(Dictionary). This method must create, satisfy and activate a new component configuration and register its component instance as a service if the component description specifies a service. It must then return a ComponentInstance object. This ComponentInstance object can be used to get the component instance with the getInstance() method.

SCR must attempt to satisfy the component configuration created by newInstance before activating it. If SCR is unable to satisfy the component configuration given the component properties and the Dictionary argument to newInstance, the newInstance method must throw a ComponentException.

The client of the Component Factory service can also deactivate a component configuration with the dispose() method on the ComponentInstance object. If the component configuration is already

deactivated, or is being deactivated, then this method is ignored. Also, if the component configuration becomes unsatisfied for any reason, it must be deactivated by SCR.

Once a component configuration created by the Component Factory has been deactivated, that component configuration will not be reactivated or used again.

112.5.6   Activation

Activating a component configuration consists of the following steps:

1. Load the component implementation class.
2. Create the component instance and component context.
3. Bind the target services. See *Binding Services* on page 204.
4. Call the activate method, if present. See *Activate Method* on page 204.

Component instances must never be reused. Each time a component configuration is activated, SCR must create a new component instance to use with the activated component configuration. A component instance must complete activation before it can be deactivated. Once the component configuration is deactivated or fails to activate due to an exception, SCR must unbind all the component's bound services and discard all references to the component instance associated with the activation.

112.5.7   Binding Services

When a component configuration's reference is satisfied, there is a set of zero or more target services for that reference. When the component configuration is activated, a subset of the target services for each reference are bound to the component configuration. The subset is chosen by the cardinality of the reference. See *Reference Cardinality* on page 185.

When binding services, the references are processed in the order in which they are specified in the component description. That is, target services from the first specified reference are bound before services from the next specified reference.

Obtaining the service object for a bound service may result in activating a component configuration of the bound service which could result in an exception. If the loss of the bound service due to the exception causes the reference's cardinality constraint to be violated, then activation of this component configuration will fail. Otherwise the bound service which failed to activate will be considered unbound.

If the reference uses the field strategy, the field must be set. Then, if the reference uses the event strategy, the bind method must be called for each bound service of that reference. If a bind method throws an exception, SCR must log an error message containing the exception with the Log Service, if present, but the activation of the component configuration does not fail.

112.5.8   Activate Method

A component instance can have an activate method. The name of the activate method can be specified by the `activate` attribute. See *Component Element* on page 193. If the `activate` attribute is not specified, the default method name of `activate` is used. The prototype of the activate method is:

```
void <method-name>(<arguments>);
```

The activate method can take zero or more arguments. Each argument must be of one of the following types:

- `ComponentContext` - The component instance will be passed the Component Context for the component configuration.
- `BundleContext` - The component instance will be passed the Bundle Context of the component's bundle.

- Map - The component instance will be passed an unmodifiable Map containing the component properties.
- A component property type - The component instance will be passed an instance of the component property type which allows type safe access to component properties defined by the component property type. See *Component Property Types* on page 216.

A suitable method is selected using the following priority:

1. The method takes a single argument and the type of the argument is org.osgi.service.component.ComponentContext.
2. The method takes a single argument and the type of the argument is org.osgi.framework.BundleContext.
3. The method takes a single argument and the type of the argument is a component property type.
4. The method takes a single argument and the type of the argument is java.util.Map.
5. The method takes two or more arguments and the type of each argument must be org.osgi.service.component.ComponentContext, org.osgi.framework.BundleContext, a component property type or java.util.Map. If multiple methods match this rule, this implies the method name is overloaded and SCR may choose any of the methods to call.
6. The method takes zero arguments.

When searching for the activate method to call, SCR must locate a suitable method as specified in *Locating Component Methods and Fields* on page 220. If the activate attribute is specified and no suitable method is located, SCR must log an error message with the Log Service, if present, and the component configuration is not activated.

If an activate method is located, SCR must call this method to complete the activation of the component configuration. If the activate method throws an exception, SCR must log an error message containing the exception with the Log Service, if present, and the component configuration is not activated.

112.5.9   Component Context

The Component Context is made available to a component instance via the activate and deactivate methods. It provides the interface to the execution context of the component, much like the Bundle Context provides a bundle the interface to the Framework. A Component Context should therefore be regarded as a capability and not shared with other components or bundles.

Each distinct component instance receives a unique Component Context. Component Contexts are not reused and must be discarded when the component configuration is deactivated.

112.5.10   Bound Service Replacement

If an active component configuration has a dynamic reference with unary cardinality and the bound service is modified or unregistered and ceases to be a target service, or the policy-option is greedy and a better target service becomes available then SCR must attempt to replace the bound service with a new bound service.

If the reference uses the field strategy, the field must be set for the replacement bound service. Then, if the reference uses the event strategy, SCR must first bind the new bound service and then unbind the outgoing service. This reversed order allows the component to not have to handle the inevitable gap between the unbind and bind methods. However, this means that in the unbind method care must be taken to not overwrite the newly bound service. For example, the following code handles the associated concurrency issues and simplify handling the reverse order.

```
final AtomicReference<LogService> log = new AtomicReference<LogService>();
```

```
void setLogService( LogService log ) {
    this.log.set(log);
}
void unsetLogService( LogService log ) {
    this.log.compareAndSet(log, null);
}
```

If the dynamic reference falls below the minimum cardinality, the component configuration must be deactivated because the cardinality constraints will be violated.

If a component configuration has a static reference and a bound service is modified or unregistered and ceases to be a target service, or the policy-option is greedy and a better target service becomes available then SCR must deactivate the component configuration. Afterwards, SCR must attempt to activate the component configuration again if another target service can be used as a replacement for the outgoing service.

112.5.11 Updated

If an active component is bound to a service that modifies its service properties then the component can be updated. If the reference uses the field strategy and the field holds the service properties, the field must be set for the updated bound service. Then, if the reference uses the event strategy and specifies an updated method, the updated method must be called.

112.5.12 Modification

Modifying a component configuration can occur if the component description specifies the modified attribute and the component properties of the component configuration use a Configuration object from the Configuration Admin service and that Configuration object is modified without causing the component configuration to become unsatisfied. If this occurs, the component instance will be notified of the change in the component properties.

If the modified attribute is not specified, then the component configuration will become unsatisfied if its component properties use a Configuration object and that Configuration object is modified in any way.

Modifying a component configuration consists of the following steps:

1. Update the component context for the component configuration with the modified configuration properties.
2. Call the modified method. See *Modified Method* on page 206.
3. Modify the bound services for the dynamic references if the set of target services changed due to changes in the target properties. See *Bound Service Replacement* on page 205.
4. If the component configuration is registered as a service, modify the service properties.

A component instance must complete activation, or a previous modification, before it can be modified.

See *Configuration Changes* on page 212 for more information.

112.5.13 Modified Method

The name of the modified method is specified by the modified attribute. See *Component Element* on page 193. The prototype and selection priority of the modified method is identical to that of the activate method. See *Activate Method* on page 204.

SCR must locate a suitable method as specified in *Locating Component Methods and Fields* on page 220. If the modified attribute is specified and no suitable method is located, SCR must log an error message with the Log Service, if present, and the component configuration becomes unsatisfied and is deactivated as if the modified attribute was not specified.

If a modified method is located, SCR must call this method to notify the component configuration of changes to the component properties. If the modified method throws an exception, SCR must log an error message containing the exception with the Log Service, if present and continue processing the modification.

112.5.14   Deactivation

Deactivating a component configuration consists of the following steps:

1.   Call the deactivate method, if present. See *Deactivate Method* on page 207.
2.   Unbind any bound services. See *Unbinding* on page 208.
3.   Release all references to the component instance and component context.

A component instance must complete activation or modification before it can be deactivated. A component configuration can be deactivated for a variety of reasons. The deactivation reason can be received by the deactivate method. The following reason values are defined:

• 0 - Unspecified.
• 1 - The component was disabled.
• 2 - A reference became unsatisfied.
• 3 - A configuration was changed.
• 4 - A configuration was deleted.
• 5 - The component was disposed.
• 6 - The bundle was stopped.

Once the component configuration is deactivated, SCR must discard all references to the component instance and component context associated with the activation.

112.5.15   Deactivate Method

A component instance can have a deactivate method. The name of the deactivate method can be specified by the deactivate attribute. See *Component Element* on page 193. If the deactivate attribute is not specified, the default method name of deactivate is used. The prototype of the deactivate method is:

```
void <method-name>(<arguments>);
```

The deactivate method can take zero or more arguments. Each argument must be assignable from one of the following types:

• ComponentContext - The component instance will be passed the Component Context for the component.
• BundleContext - The component instance will be passed the Bundle Context of the component's bundle.
• Map - The component instance will be passed an unmodifiable Map containing the component properties.
• A component property type - The component instance will be passed an instance of the component property type which allows type safe access to component properties defined by the component property type. See *Component Property Types* on page 216.
• int or Integer - The component instance will be passed the reason the component configuration is being deactivated. See *Deactivation* on page 207.

A suitable method is selected using the following priority:

1.   The method takes a single argument and the type of the argument is org.osgi.service.component.ComponentContext.

2. The method takes a single argument and the type of the argument is
   `org.osgi.framework.BundleContext`.

3. The method takes a single argument and the type of the argument is a component property
   type.

4. The method takes a single argument and the type of the argument is `java.util.Map`.

5. The method takes a single argument and the type of the argument is `int`.

6. The method takes a single argument and the type of the argument is `java.lang.Integer`.

7. The method takes two or more arguments and the type of each argument must be
   `org.osgi.service.component.ComponentContext`, `org.osgi.framework.BundleContext`, a com-
   ponent property type, `java.util.Map`, `int` or `java.lang.Integer`. If multiple methods match this
   rule, this implies the method name is overloaded and SCR may choose any of the methods to
   call.

8. The method takes zero arguments.

When searching for the deactivate method to call, SCR must locate a suitable method as specified in
Locating Component Methods and Fields on page 220. If the `deactivate` attribute is specified and no
suitable method is located, SCR must log an error message with the Log Service, if present, and the
deactivation of the component configuration will continue.

If a deactivate method is located, SCR must call this method to commence the deactivation of the
component configuration. If the deactivate method throws an exception, SCR must log an error
message containing the exception with the Log Service, if present, and the deactivation of the com-
ponent configuration will continue.

112.5.16 Unbinding

When a component configuration is deactivated, the bound services are unbound from the compo-
nent configuration.

When unbinding services, the references are processed in the reverse order in which they are spec-
ified in the component description. That is, target services from the last specified reference are un-
bound before services from the previous specified reference.

If the reference uses the event strategy, the unbind method must be called for each bound service of
that reference. If an unbind method throws an exception, SCR must log an error message containing
the exception with the Log Service, if present, and the deactivation of the component configuration
will continue. Then, if the reference uses the field strategy, the field must be set to `null`.

112.5.17 Life Cycle Example

A component could declare a dependency on the Http Service to register some resources.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<scr:component name="example.binding"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.3.0">
    <implementation class="com.acme.impl.Binding"/>
    <reference name="LOG"
        interface="org.osgi.service.log.LogService"
        cardinality="1..1"
        policy="static"
    />
    <reference name="HTTP"
        interface="org.osgi.service.http.HttpService"
        cardinality="0..1"
        policy="dynamic"
        bind="setHttp"
        unbind="unsetHttp"
```

```
    />
</scr:component>
```

The component implementation code looks like:
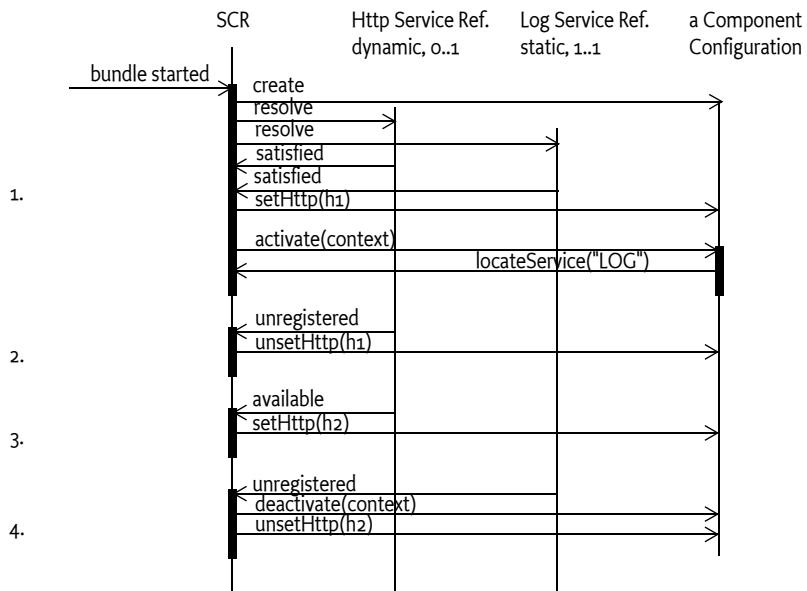
```
public class Binding {
    LogService  log;
    HttpService http;

    private void setHttp(HttpService h) {
        http = h;
        // register servlet
    }
    private void unsetHttp(HttpService h){
        if (http == h)
            http = null;
        // unregister servlet
    }
    private void activate(ComponentContext context ) {
        log = (LogService) context.locateService("LOG");
    }
    private void deactivate(ComponentContext context ) {...}
}
```

This example is depicted in a sequence diagram in Figure 112.5 with the following scenario:

1. A bundle with the example.Binding component is started. At that time there is a Log Service l1 and a Http Service h1 registered.
2. The Http Service h1 is unregistered
3. A new Http Service h2 is registered
4. The Log Service h1 is unregistered.

Figure 112.5   *Sequence Diagram for binding*

112.6 Component Properties

Each component configuration is associated with a set of component properties. The component properties are specified in the following *configuration sources* (in order of precedence):

1. Properties specified in the argument of the ComponentFactory.newInstance method. This is only applicable for factory components.

2. Properties retrieved from the OSGi Configuration Admin service in Configuration objects whose PID matches a *configuration PID*. The configuration PIDs are specified by the configuration-pid attribute of the component element. See *Component Element* on page 193. If no configuration-pid attribute is specified, the component name is used as the default configuration PID. If multiple configuration PIDs are specified, the order of precedence follows the order the configuration PIDs are specified in the component description. That is, the precedence for the configuration for an earlier specified configuration PID is lower than the precedence for the configurations for a later specified configuration PID.

3. Properties specified in the component description. Properties specified later in the component description override properties that have the same name specified earlier. Properties can be specified in the component description in the following ways:

   - target attribute of reference elements - Sets the target property of the reference. See *Target Property* on page 211. The value of the target attribute is used for the value of a target property.

   - property and properties elements - See *Property and Properties Elements* on page 195.

The precedence behavior allows certain default values to be specified in the component description while allowing properties to be replaced and extended by:

- A configuration in Configuration Admin
- The argument to the ComponentFactory.newInstance method

Normally, a property value from a higher precedence configuration source replace a property value from a lower precedence configuration source. However, the service.pid property values receive different treatment. For the service.pid property, if the property appears multiple times in the configuration sources, SCR must aggregate all the values found into a Collection<String> having an iteration order such that the first item in the iteration is the property value from the lowest precedence configuration source and the last item in the iteration is the property value from the highest precedence configuration source. If the component description specifies multiple configuration PIDs, then the order of the service.pid property values from the corresponding configurations matches the order the configuration PIDs are specified in the component description. The values of the service.pid component property are the values as they come from the configuration sources which, for Configuration objects, may be more detailed than the configuration PIDs specified in the component description.

SCR always adds the following component properties, which cannot be overridden:

- component.name - The component name.
- component.id - A unique value (Long) that is larger than all previously assigned values. These values are not persistent across restarts of SCR.

112.6.1 Service Properties

When SCR registers a service on behalf of a component configuration, SCR must follow the recommendations in *Property Propagation* on page 69 and must not propagate private configuration properties. That is, the service properties of the registered service must be all the component properties of the component configuration whose property names do not start with full stop ('.' \u002E).

Component properties whose names start with full stop are available to the component instance but are not available as service properties of the registered service.

112.6.2   Reference Properties

This specification defines some component properties which are associated with specific component references. These are called *reference properties*. The name of a reference property for a reference is the name of the reference appended with a full stop ('.' \u002E) and a suffix unique to the reference property. Reference properties can be set wherever component properties can be set.

All component property names starting with a reference name followed by a full stop ('.' \u002E) are reserved for use by this specification.

Following are the reference properties defined by this specification.

112.6.2.1   Target Property

The *target property* is a reference property which aids in the selection of target services for the reference. See *Selecting Target Services* on page 191. The name of a target property is the name of a reference appended with .target. For example, the target property for a reference with the name http would have the name http.target. The value of a target property is a filter string used to select targets services for the reference.

The target property for a reference can also be set by the target attribute of the reference element. See *Reference Element* on page 198.

112.6.2.2   Minimum Cardinality Property

The initial minimum cardinality of a reference is specified by the optionality: the first part of the cardinality. It is either 0 or 1. The minimum cardinality of a reference cannot exceed the multiplicity: the second part of the cardinality. See *Reference Cardinality* on page 185 for more information on the cardinality of a reference.

The *minimum cardinality property* is a reference property which can be used to raise the minimum cardinality of a reference from its initial value. That is, a 0..1 cardinality can be raised to a 1..1 cardinality by setting the reference's minimum cardinality property to 1, and a 0..n or 1..n cardinality can be raised to a m..n cardinality by setting the reference's minimum cardinality property to m such that m is a positive integer. The minimum cardinality of a reference cannot be lowered. That is, a 1..1 or 1..n cardinality can be lowered to a 0..1 or 0..n cardinality because the component was coded to expect at least one bound service.

The name of a minimum cardinality property is the name of a reference appended with .cardinality.minimum. For example, the minimum cardinality property for a reference with the name http would have the name http.cardinality.minimum. The value of a minimum cardinality property must be a positive integer or a value that can be coerced into a positive integer. See *Coercing Component Property Values* on page 218 for information on coercing property values. If the numerical value of the minimum cardinality property is not valid for the reference's cardinality or the minimum cardinality property value cannot be coerced into a numerical value, then the minimum cardinality property must be ignored.

SCR must support the minimum cardinality property for all components even those with component descriptions in older namespaces.

112.7   Deployment

A component description contains default information to select target services for each reference. However, when a component is deployed, it is often necessary to influence the target service selection in a way that suits the needs of the deployer. Therefore, SCR uses Configuration objects from

Configuration Admin to replace and extend the component properties for a component configuration. That is, through Configuration Admin, a deployer can configure component properties.

A component's configuration PIDs are used as keys for obtaining additional component properties from Configuration Admin. When *matching* a configuration PID to a Configuration object, SCR must use the Configuration object with the best matching PID for the component's bundle. See *Targeted PIDs* on page 65 for more information on targeted PIDs and *Extenders and Targeted PIDs* on page 66 for more information on selecting the Configuration object with the best matching PID.

The following situations can arise when looking for Configuration objects:

- *No Configuration* - If the component's configuration-policy is set to ignore or there are no Configurations with a PID or factory PID matching any of the configuration PIDs, then component configurations will not obtain component properties from Configuration Admin. Only component properties specified in the component description or via the ComponentFactory.newInstance method will be used.
- *Not Satisfied* - If the component's configuration-policy is set to require and, for each configuration PID, there is no Configuration with a matching PID or factory PID, then the component configuration is not satisfied and will not be activated.
- *Single Configuration*s - If none of the configuration PIDs matches a factory PID, then component configurations will obtain additional component properties from Configuration Admin.
- *Factory Configuration* - If one of the configuration PIDs matches a factory PID, with zero or more Configurations, then for each Configuration of the factory PID, a component configuration must be created that will obtain additional component properties from Configuration Admin.

  It is a configuration error if more than one of the configuration PIDs match a factory PID and SCR must log an error message with the Log Service, if present. If the configuration-policy is set to optional, the component configuration must be satisfied without the configurations PIDs which match a factory PID. If the configuration-policy is set to require, the component configuration is not satisfied and will not be activated.

  A factory configuration must not be used if the component is a factory component. This is because SCR is not free to create component configurations as necessary to support multiple Configurations. When SCR detects this condition, it must log an error message with the Log Service, if present, and ignore the component description.

SCR must obtain the Configuration objects from the Configuration Admin service using the Bundle Context of the bundle containing the component.

For example, there is a component named com.acme.client with a reference named HTTP that requires an Http Service which must be bound to a component com.acme.httpserver which provides an Http Service. A deployer can establish the following configuration:

```
[PID=com.acme.client, factoryPID=null]
HTTP.target = (component.name=com.acme.httpserver)
```

112.7.1   Configuration Changes

SCR must track changes in the Configuration objects matching the configuration PIDs of a component description. Changes include the creating, updating and deleting of Configuration objects matching the configuration PIDs. The actions SCR must take when a configuration change for a component configuration occurs are based upon how the configuration-policy and modified attributes are specified in the component description, whether a component configuration becomes satisfied, remains satisfied or becomes unsatisfied and the type and number of matching Configuration objects.

With targeted PIDs, multiple Configuration objects can exist which can match a configuration PID. Creation of a Configuration object with a better matching PID than a Configuration object currently being used by a component configuration results in a configuration change for the component

configuration with the new Configuration object replacing the currently used Configuration object. Deletion of a Configuration object currently being used by a component configuration when there is another Configuration object matching the configuration PID also results in a configuration change for the component configuration with the Configuration object having the best matching PID replacing the currently used, and now deleted, Configuration object.

112.7.1.1 Ignore Configuration Policy

For configuration-policy of ignore, component configurations are unaffected by configuration changes since the component properties do not include properties from Configuration objects.

112.7.1.2 Require Configuration Policy

For configuration-policy of require, component configurations require a Configuration object for each specified configuration PID. With a factory configuration, there can be zero or more matching Configuration objects which will result in a component configuration for each Configuration object of the factory configuration. With a factory component, multiple component configurations can be created all using the matching Configuration objects.

A configuration change can cause a component configuration to become unsatisfied if any of the following occur:

- Each configuration PID of the component description does not have a matching Configuration object.
- A target property change results in a bound service of a static reference ceasing to be a target service.
- A target property change results in unbound target services for a static reference with the greedy policy option.
- A target property change or minimum cardinality property change results in a reference falling below the minimum cardinality.
- The component description does not specify the modified attribute.

112.7.1.3 Optional Configuration Policy

For configuration-policy of optional, component configurations do not require Configuration objects. Since matching Configuration objects are optional, component configurations can be satisfied with zero or more matched configuration PIDs. If a Configuration object is then created which matches a configuration PID, this is a configuration change for the component configurations that are not using the created Configuration object. If a Configuration object is deleted which matches a configuration PID, this is a configuration change for the component configurations using the deleted Configuration object.

Furthermore, with a factory configuration matching a configuration PID, the factory configuration can provide zero or more Configuration objects which will result in a component configuration for each Configuration object or a single component configuration when zero matching Configuration objects are provided. With a factory component, multiple component configurations can be created all using the Configuration objects matching the configuration PIDs.

A configuration change can cause a component configuration to become unsatisfied if any of the following occur:

- A target property change results in a bound service of a static reference ceasing to be a target service.
- A target property change results in unbound target services for a static reference with the greedy policy option.
- A target property change or minimum cardinality property change results in a reference falling below the minimum cardinality.
- The component description does not specify the modified attribute.

Configuration Change Actions

If a component configuration becomes unsatisfied:

- SCR must deactivate the component configuration.
- If the component configuration was not created from a factory component, SCR must attempt to satisfy the component configuration with the current configuration state.

If a component configuration remains satisfied:

- If the component configuration has been activated, the modified method is called to provide the updated component properties. See *Modification* on page 206 for more information.
- If the component configuration is registered as a service, SCR must modify the service properties.

112.8 Annotations

A number of CLASS retention annotations have been provided to allow tools to construct the component description XML from the Java class files. The Component Annotations are intended to be used during build time to generate the component description XML.

Component Property Types, which are user defined annotations, can be used to describe component properties in the component description XML and to access those component properties at runtime in a type safe manner.

112.8.1 Component Annotations

The Component Annotations provide a convenient way to create the component description XML during build time. Since annotations are placed in the source file and can use types, fields, and methods, they can significantly simplify the use of Declarative Services.

The Component Annotations are build time annotations because one of the key aspects of Declarative Services is its laziness. SCR can easily read the component description XML from the bundle, preprocess it, and cache the results between framework invocations. This way it is unnecessary to load a class from the bundle when the bundle is started and/or scan the classes for annotations. Component Annotations are not recognized by SCR at runtime.

The Component Annotations are not inherited, they can only be used on a given class, annotations on its super class hierarchy or interfaces are not taken into account.

The primary annotation is the Component annotation. It indicates that a class is a component. Its defaults create the easiest to use component:

- Its name is the class name
- It registers all of the class's directly implemented interfaces as services
- The instance will be shared by all bundles
- It is enabled
- It is immediate if it has no services, otherwise it is delayed
- It has an optional configuration policy
- The configuration PID is the class name

For example, the following class registers a Speech service that can run on a Macintosh:

```
pubic interface Speech {
  void say(String what) throws Exception;
}


@Component
```

```
public class MacSpeech implements Speech {
    ScriptEngine engine =
        new ScriptEngineManager().getEngineByName("AppleScript");

    public void say(String message) throws Exception {
        engine.eval("say \"" + message.replace('"','\'' + "\"");
    }
}
```

The previous example would be processed at build time into a component description similar to the following XML:

```
<scr:component name="com.example.MacSpeech"
  xmlns:scr="http://www.osgi.org/xmlns/scr/v1.3.0">
  <implementation class="com.acme.impl.MacSpeech"/>
  <service>
    <provide interface="com.acme.service.speech.Speech"/>
  </service>
</scr:component>
```

It is possible to add activate and deactivate methods on the component with the Activate and Deactivate annotations. If the component wants to be updated for changes in the configuration properties than it can also indicated the modified method with the Modified annotation. For example:

```
@Activate
void open(Map<String,?> properties) { ... }

@Deactivate
void close() { ... }

@Modified
void modified(Map<String,?> properties) { ... }
```

If a component has dependencies on other services then they can be referenced with the Reference annotation that can be applied to a bind method or a field. For a bind method, the defaults for the Reference annotation are:

- The name of the bind method or field is used for the name of the reference.
- 1:1 cardinality.
- Static reluctant policy.
- The requested service is the type of the first argument of the bind method.
- It will infer a default unset method and updated method based on the name of the bind method.

For example:

```
@Reference(cardinality=MULTIPLE, policy=DYNAMIC)
void setLogService( LogService log, Map<String,?> props) { ... }
void unsetLogService( LogService log ) {  ... }
void updatedLogService( Map<String,?> map ) { ...}
```

For a field, the defaults for the Reference annotation are:

- The name of the bind method or field is used for the name of the reference.
- 1:1 cardinality if the field is not a collection. 0..n cardinality if the field is a collection.
- Static reluctant policy if the field is not declared volatile. Dynamic reluctant policy if the field is declared volatile.

- The requested service is the type of the field.

For example:

```
@Reference
volatile Collection<LogService> log;
```

112.8.2   Component Property Types

Component properties can be defined and accessed through a user defined annotation type, called a *component property type*, containing the property names, property types and default values. A component property type allows properties to be defined and accessed in a type safe manner. The following example shows the definition of a component property type called Config which defines three properties where the name of the property is the name of the method, the type of the property is the return type of the method and the default value for the property is the default value of the method.

```
@interface Config {
  boolean enabled() default true;
  String[] names() default {"a", "b"};
  String topic() default "default/topic";
}
```

Component property types can be referenced as argument types in the component's life cycle methods, activate, deactivate, and modified, and used in the method implementation to access component property values in a type safe manner. The following example shows the activate method taking the example Config component property type as an argument type and the method implementation accesses component property values by invoking methods on the component property type argument.

```
@Component
public class MyComponent {
  void activate(Config config) {
    if (config.enabled()) {
      // do something
    }
    for (String name:config.names()) {
      // do something with each name
    }
  }
}
```

If a component implementation needs to access component properties which are not represented by a component property type, it can use a life cycle method signature which also receives the properties map in addition to component property types. For example:

```
@Component
public class MyComponent {
  void activate(Config config, Map<String, ?> allProperties) {
    if (config.enabled()) {
      // do something
    }
    if (allProperties.get("other.prop") != null) {
      // do something
    }
  }
}
```

Component property types must be defined as annotation types even though they are not applied as annotations but are rather used as life cycle method argument types. This is done for two reasons. First, the limitations on annotation type definitions make them well suited for component property types. The methods must have no arguments and the return types supported are limited to a set which is well suited for component properties. Second, annotation types support default values which is useful for defining the default value of a component property.

At build time, the component property types must be processed to potentially generate property elements in the component description. See *Ordering of Generated Properties* on page 218.

At runtime, when SCR needs to call a lifecycle method on a component instance which takes an argument whose type is a component property type, SCR must construct an instance of the component property type whose methods are backed by the values of the component properties for the component instance. This object is then passed to the life cycle method which can use the object to obtain the property values in a type safe manner.

112.8.2.1   **Component Property Mapping**

Each method of a configuration property type is mapped to a component property. The property name is derived from the method name. Certain common property name characters, such as full stop ('.' \u002E) are not valid in Java identifiers. So the name of a method must be converted to its corresponding property name as follows:

- A single dollar sign ('$' \u0024) is removed unless it is followed by another dollar sign in which case the two consecutive dollar signs ("$$") are converted to a single dollar sign.
- A single low line ('_' \u005F) is converted into a full stop ('.' \u002E) unless is it followed by another low line in which case the two consecutive low lines ("__") are converted to a single low line.
- All other characters are unchanged.

Table 112.9 contains some name mapping examples.

Table 112.9   *Component Property Name Mapping Examples*

Component Property Type Method Name	Component Property Name
myProperty143	myProperty143
$new	new
my$$prop	my$prop
dot_prop	dot.prop
_secret	.secret
another__prop	another_prop
three___prop	three_.prop
four_$__prop	four._prop
five_$_prop	five..prop

The property type can be directly derived from the type of the method. All types supported for annotation elements can be used except for annotation types. Method types of an annotation type or array thereof are not supported. A tool processing the component property types must declare an error during processing in this case.

If the method type is Class or Class[], then the property type must be String or String[], respectively, whose values are fully qualified class names in the form returned by the Class.getName() method.

If the method type is an enumeration type or an array thereof, then the property type must be String or String[], respectively, whose values are the names of the enum constants in the form returned by the Enum.name() method.

112.8.2.2   **Coercing Component Property Values**

When a component property type is used as an argument for a life cycle method, SCR must create an object that implements the component property type and maps the methods of the component property type to component properties. The name of the method is converted to the property name as described in *Component Property Mapping* on page 217. The property value may need to be coerced to the type of the method. In Table 112.10, the columns are source types, that is, the type of the component property value, and the rows are target types, that is, the method types. The property value is *v*; *number* is a primitive numerical type and *Number* is a wrapper numerical type. An invalid coercion is represented by throw. Such a coercion attempt must result in throwing a Component Exception when the component property type method is called. Any other coercion error, such as parsing a non-numerical string to a number or the inability to coerce a string into a Class or enum object, must be wrapped in a Component Exception and thrown when the component property type method is called.

Table 112.10   *Coercion From Property Value to Method Type*

target \ source	String	Boolean	Character	*Number*	Collection/array
String	*v*	*v*. toString()	*v*. toString()	*v*. toString()	If *v* has no elements, null; otherwise the first element of *v* is coerced.
boolean	Boolean. parse-Boolean(*v*)	*v*. booleanValue()	*v*. charValue() ! = 0	*v*. *number*Value() != 0	If *v* has no elements, false; otherwise the first element of *v* is coerced.
char	*v*. length() > 0 ? *v*. charAt(0) : 0	*v*. booleanValue() ? 1 : 0	*v*. charValue()	(char) *v*. *number*Value()	If *v* has no elements, 0; otherwise the first element of *v* is coerced.
number	*Number*. parse*Number*(*v*)	*v*. booleanValue() ? 1 : 0	(*number*) *v*. charValue()	*v*. *number*Value()	If *v* has no elements, 0; otherwise the first element of *v* is coerced.
Class	Bundle. load-Class(*v*)	throw	throw	throw	If *v* has no elements, null; otherwise the first element of *v* is coerced.
EnumType	*EnumType*. valueOf(*v*)	throw	throw	throw	If *v* has no elements, null; otherwise the first element of *v* is coerced.
annotation type	throw	throw	throw	throw	throw
array	A single element array is created and *v* is coerced into the single element of the new array.				An array the size of *v* is created and each element of *v* is coerced into the corresponding element of the new array.

Component properties whose names do not map to component property type methods are ignored. If there is no corresponding component property for a component property type method, the component property type method must:

- Return 0 for numerical and char method types.
- Return false for boolean method type.
- Return null for String, Class, enum and array method types.
- Throw a ComponentException for annotation method types.

112.8.3   Ordering of Generated Properties

The Component annotation contains two ways to define component properties via the property and properties elements. See *Property and Properties Elements* on page 195. If Component Annotations are used to describe the component, then any component property types referenced in the signatures of the component's life cycle methods must also be processed since component prop-

erty types can be used to define component property values as well. See *Component Property Types* on page 216. A tool processing the Component Annotations and the component property types must write the defined component properties into the generated component description in the following order.

1. Properties defined through component property types used in the signatures of the life cycle methods.

   If any of the referenced component property types have methods with defaults, then the generated component description must include a property element for each such method with the property name mapped from the method name, the property type mapped from the method type, and the property value set to the method's default value. See *Component Property Mapping* on page 217. The generated property elements must be added to the component description by processing the component property types in the following order. First, the component property types used as arguments to the activate method, followed by the component property types used as arguments to the modified method and finally the component property types used as arguments to the deactivate method. If a method has more than one component property type argument, the component property types are processed in the order of the method arguments.

   For component property type methods without a default value, a property element must not be generated.
2. property element of the Component annotation.
3. properties element of the Component annotation.

This means that the properties defined through component property types are declared first in the generated component description, followed by all properties defined through the property element of the Component annotation and finally the properties entries defined through the properties element of the Component annotation.

Since property values defined later in the component description override property values defined earlier in the component description, this means that property values defined in properties element of the Component annotation can override property values defined in property element of the Component annotation which can override values defined by default values in the component property types used in life cycle method signatures.

112.9 Service Component Runtime

Service Component Runtime (SCR) is the actor that manages the components and their life cycle and allows introspection of the components.

112.9.1 Relationship to OSGi Framework

SCR must have access to the Bundle Context of any bundle that contains a component. SCR needs access to the Bundle Context for the following reasons:

- To be able to register and get services on behalf of a bundle with components.
- To interact with the Configuration Admin on behalf of a bundle with components.
- To provide a component its Bundle Context when the Component Context `getBundleContext` method is called.

SCR should use the `Bundle.getBundleContext()` method to obtain the Bundle Context reference.

112.9.2 Starting and Stopping SCR

When SCR is implemented as a bundle, any component configurations activated by SCR must be deactivated when the SCR bundle is stopped. When the SCR bundle is started, it must process any

components that are declared in bundles that are started. This includes bundles which are started and are awaiting lazy activation.

112.9.3 Logging Error Messages

When SCR must log an error message to the Log Service, it must use a Log Service obtained using the component's Bundle Context so that the resulting Log Entry is associated with the component's bundle.

If SCR is unable to obtain, or use, a Log Service using the component's Bundle Context, then SCR must log the error message to a Log Service obtained using SCR's bundle context to ensure the error message is logged.

112.9.4 Locating Component Methods and Fields

SCR will need to locate activate, deactivate, modified, bind, updated, and unbind methods as well as fields in a component instance. These members will be located, and called or modified, using reflection. The declared members of each class in the component implementation class's hierarchy are examined for a suitable member. If a suitable member is found in a class, and it is accessible to the component implementation class, then that member must be used. If suitable members are found in a class but none of the suitable members are accessible by the component implementation class, then the search for suitable members terminates with no suitable member having been located. If no suitable members are found in a class, the search continues in the superclass.

Only members that are accessible to the component implementation class will be used. If the member has the `public` or `protected` access modifier, then access is permitted. Otherwise, if the member has the `private` access modifier, then access is permitted only if the member is declared in the component implementation class. Otherwise, if the member has default access, also known as package private access, then access is permitted only if the member is declared in the component implementation class or if the member is declared in a superclass and all classes in the hierarchy from the component implementation class to the superclass, inclusive, are in the same package and loaded by the same class loader.

It is recommended that these members should not be declared with the `public` access modifier so that they do not appear as public members on the component instance when it is used as a service object. Having these members declared `public` allows any code to call or access the members with reflection, even if a Security Manager is installed. These members are generally intended to only be called or modified by SCR.

112.9.5 Bundle Activator Interaction

A bundle containing components may also declare a Bundle Activator. Such a bundle may also be marked for lazy activation. Since components are activated by SCR and Bundle Activators are called by the OSGi Framework, a bundle using both components and a Bundle Activator must take care. The Bundle Activator's start method must not rely upon SCR having activated any of the bundle's components. However, the components can rely upon the Bundle Activator's start method having been called. That is, there is a *happens-before* relationship between the Bundle Activator's start method being run and the components being activated.

112.9.6 Introspection

SCR provides an introspection API for examining the runtime state of the components in bundles processed by SCR. SCR must register a `ServiceComponentRuntime` service upon startup. The Service Component Runtime service provides methods to inspect the component descriptions and component configurations as well as inspect and modify the enabled state of components. The service uses *Data Transfer Objects (DTO)* as arguments and return values. The rules for Data Transfer Objects are specified in *OSGi Core Release 6*.

The Service Component Runtime service provides the following methods.
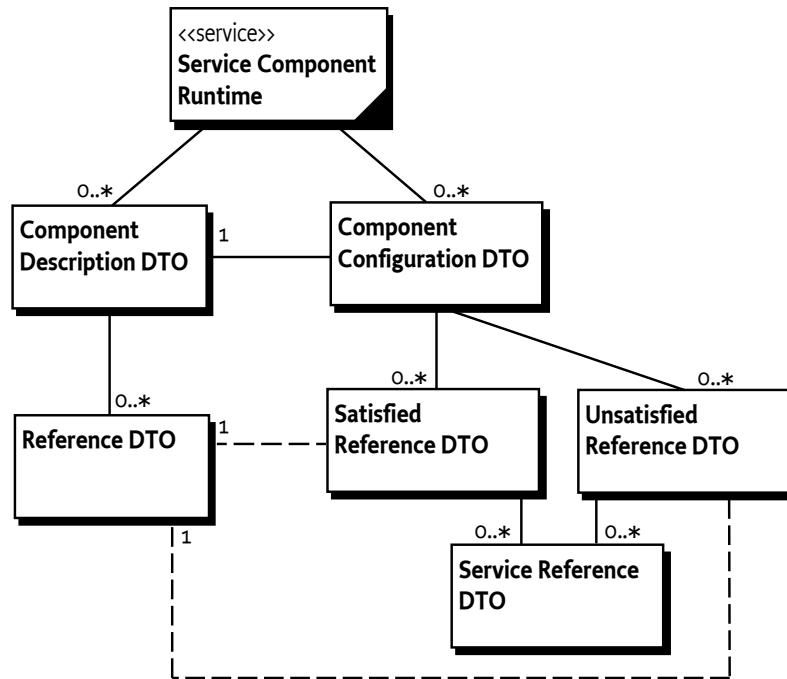
- getComponentDescriptionDTOs(Bundle...) - For each specified bundle, if the bundle is active and processed by SCR, the returned collection will contain a ComponentDescriptionDTO for each valid component description in the bundle.
- getComponentDescriptionDTO(Bundle,String) - If the specified bundle is active and processed by SCR, and the specified bundle contains a valid component description with the specified name, the method will return a ComponentDescriptionDTO for the component description.
- getComponentConfigurationDTOs(ComponentDescriptionDTO) - If the specified Component-DescriptionDTO represents a valid component description from an active bundle processed by SCR, the returned collection will contain a ComponentConfigurationDTO for each component configuration of the component.
- isComponentEnabled(ComponentDescriptionDTO) - Returns true if the specified Component Description DTO represents a valid component description from an active bundle processed by SCR, and the component is enabled. Otherwise, the method returns false.
- enableComponent(ComponentDescriptionDTO) - If the specified Component Description DTO represents a valid component description from an active bundle processed by SCR, the component is enabled. This method must return after changing the enabled state of the specified component. Any actions that result from this, such as activating or deactivating a component configuration, must occur asynchronously to this method call. The method returns a Promise that will be resolved when the actions that result from changing the enabled state of the specified component have completed.
- disableComponent(ComponentDescriptionDTO) - If the specified Component Description DTO represents a valid component description from an active bundle processed by SCR, the component is disabled. This method must return after changing the enabled state of the specified component. Any actions that result from this, such as activating or deactivating a component configuration, must occur asynchronously to this method call. The method returns a Promise that will be resolved when the actions that result from changing the enabled state of the specified component have completed.

The runtime state of the components can change at any time. So any information returned by these methods only provides a snapshot of the state at the time of the method call.

There are a number of DTOs available via the Service Component Runtime service.

Figure 112.6   *Service Component Runtime DTOs*

The two main DTOs are ComponentDescriptionDTO, which represents a component description, and ComponentConfigurationDTO, which represents a component configuration. The Component Description DTO contains an array of ReferenceDTO objects which represent each declared reference in the component description. The Component Configuration DTO contains an array of SatisfiedReferenceDTO objects and an array of UnsatisfiedReferenceDTO objects. A Satisfied Reference DTO represents a satisfied reference of the component configuration and an Unsatisfied Reference DTO represents an unsatisfied reference of the component configuration. The Component Configuration DTO for a satisfied component configuration must contain no Unsatisfied Reference DTOs. The Component Configuration DTO for an unsatisfied component configuration may contain some Satisfied Reference DTOs and some Unsatisfied Reference DTOs. This information can be used to diagnose why the component configuration is not satisfied.

112.9.7   Capabilities

SCR must provide the following capabilities.

- A capability in the osgi.extender namespace declaring an extender with the name COMPONENT_CAPABILITY_NAME. This capability must also declare a uses constraint for the org.osgi.service.component package. For example:

```
Provide-Capability: osgi.extender;
    osgi.extender="osgi.component";
    version:Version="1.3";
    uses:="org.osgi.service.component"
```

This capability must follow the rules defined for the *osgi.extender Namespace* on page 681.

A bundle that contains service components should require the osgi.extender capability from SCR. This requirement will wire the bundle to the SCR implementation and ensure that SCR is using the same org.osgi.service.component package as the bundle if the bundle uses that package.

```
Require-Capability: osgi.extender;
  filter:="(&(osgi.extender=osgi.component)(version>=1.3)(!(version>=2.0)))"
```

SCR must only process a bundle's service components if one of the following is true:

- The bundle's wiring has a required wire for at least one osgi.extender capability with the name osgi.component and the first of these required wires is wired to SCR.
- The bundle's wiring has no required wire for an osgi.extender capability with the name osgi.component.

Otherwise, SCR must not process the bundle's service components.

- A capability in the osgi.service namespace representing the ServiceComponentRuntime service. This capability must also declare a uses constraint for the org.osgi.service.component.runtime package. For example:

```
Provide-Capability: osgi.service;
  objectClass:List<String>=
    "org.osgi.service.component.runtime.ServiceComponentRuntime";
  uses:="org.osgi.service.component.runtime"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 685.

112.10 Security

When Java permissions are enabled, SCR must perform the following security procedures.

112.10.1 Service Permissions

Declarative services are built upon the existing OSGi service infrastructure. This means that Service Permission applies regarding the ability to publish, find or bind services.

If a component specifies a service, then component configurations for the component cannot be satisfied unless the component's bundle has ServicePermission[<provides>, REGISTER] for each provided interface specified for the service.

If a component's reference does not specify optional cardinality, the reference cannot be satisfied unless the component's bundle has ServicePermission[<interface>, GET] for the specified interface in the reference. If the reference specifies optional cardinality but the component's bundle does not have ServicePermission[<interface>, GET] for the specified interface in the reference, no service must be bound for this reference.

If a component is a factory component, then the above Service Permission checks still apply. But the component's bundle is not required to have ServicePermission[ComponentFactory, REGISTER] as the Component Factory service is registered by SCR.

SCR must have ServicePermission[ServiceComponentRuntime, REGISTER] permission to register the ServiceComponentRuntime service. Administrative bundles wishing to use the ServiceComponentRuntime service must have ServicePermission[ServiceComponentRuntime, GET] permission. In general, this permission should only be granted to administrative bundles to limit access to the potentially intrusive methods provided by this service.

112.10.2 Required Admin Permission

SCR requires AdminPermission[*,CONTEXT] because it needs access to the bundle's Bundle Context object with the Bundle.getBundleContext() method.

112.10.3 Using hasPermission

SCR does all publishing, finding and binding of services on behalf of the component using the Bundle Context of the component's bundle. This means that normal stack-based permission checks will check SCR and not the component's bundle. Since SCR is registering and getting services on behalf of a component's bundle, SCR must call the `Bundle.hasPermission` method to validate that a component's bundle has the necessary permission to register or get a service.

112.10.4 Configuration Multi-Locations and Regions

SCR must ensure a bundle has the proper `ConfigurationPermission` for a Configuration used by its components when the Configuration has a multi-location. See *Using Multi-Locations* on page 79 for more information on multi-locations and *Regions* on page 79 for more information on regions. If a bundle does not have the necessary permission for a multi-location Configuration, then SCR must act as if the Configuration does not exist for the bundle.

112.11 Component Description Schema

This XML Schema defines the component description grammar.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:scr="http://www.osgi.org/xmlns/scr/v1.3.0"
    targetNamespace="http://www.osgi.org/xmlns/scr/v1.3.0"
    elementFormDefault="unqualified"
    attributeFormDefault="unqualified"
    version="1.3.0">

    <annotation>
        <documentation xml:lang="en">
            This is the XML Schema for component descriptions used by
            the Service Component Runtime (SCR). Component description
            documents may be embedded in other XML documents. SCR will
            process all XML documents listed in the Service-Component
            manifest header of a bundle. XML documents containing
            component descriptions may contain a single, root component
            element or one or more component elements embedded in a
            larger document. Use of the namespace for component
            descriptions is mandatory. The attributes and subelements
            of a component element are always unqualified.
        </documentation>
    </annotation>
    <element name="component" type="scr:Tcomponent" />
    <complexType name="Tcomponent">
        <sequence>
            <annotation>
                <documentation xml:lang="en">
                    Implementations of SCR must not require component
                    descriptions to specify the subelements of the component
                    element in the order as required by the schema. SCR
                    implementations must allow other orderings since
                    arbitrary orderings do not affect the meaning of the
                    component description. Only the relative ordering of
                    property and properties element have meaning.
                </documentation>
            </annotation>
            <choice minOccurs="0" maxOccurs="unbounded">
                <element name="property" type="scr:Tproperty" />
                <element name="properties" type="scr:Tproperties" />
            </choice>
            <element name="service" type="scr:Tservice" minOccurs="0"
                maxOccurs="1" />
            <element name="reference" type="scr:Treference"
                minOccurs="0" maxOccurs="unbounded" />
            <element name="implementation" type="scr:Timplementation" />
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="enabled" type="boolean" default="true"
```

```
            use="optional" />
        <attribute name="name" type="token" use="optional">
            <annotation>
                <documentation xml:lang="en">
                    The default value of this attribute is the value of
                    the class attribute of the nested implementation
                    element. If multiple component elements use the same
                    value for the class attribute of their nested
                    implementation element, then using the default value
                    for this attribute will result in duplicate names.
                    In this case, this attribute must be specified with
                    a unique value.
                </documentation>
            </annotation>
        </attribute>
        <attribute name="factory" type="string" use="optional" />
        <attribute name="immediate" type="boolean" use="optional" />
        <attribute name="configuration-policy"
            type="scr:Tconfiguration-policy" default="optional" use="optional" />
        <attribute name="activate" type="token" use="optional"
            default="activate" />
        <attribute name="deactivate" type="token" use="optional"
            default="deactivate" />
        <attribute name="modified" type="token" use="optional" />
        <attribute name="configuration-pid" use="optional">
            <annotation>
                <documentation xml:lang="en">
                    The default value of this attribute is the value of
                    the name attribute of this element.
                </documentation>
            </annotation>
            <simpleType>
                <restriction>
                    <simpleType>
                        <list itemType="token" />
                    </simpleType>
                    <minLength value="1" />
                </restriction>
            </simpleType>
        </attribute>
        <anyAttribute processContents="lax" />
    </complexType>
    <complexType name="Timplementation">
        <sequence>
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="class" type="token" use="required" />
        <anyAttribute processContents="lax" />
    </complexType>
    <complexType name="Tproperty">
        <simpleContent>
            <extension base="string">
                <attribute name="name" type="string" use="required" />
                <attribute name="value" type="string" use="optional" />
                <attribute name="type" type="scr:Tproperty_type"
                    default="String" use="optional" />
                <anyAttribute processContents="lax" />
            </extension>
        </simpleContent>
    </complexType>
    <complexType name="Tproperties">
        <sequence>
            <any namespace="##any" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="entry" type="string" use="required" />
        <anyAttribute processContents="lax" />
    </complexType>
    <complexType name="Tservice">
        <sequence>
            <element name="provide" type="scr:Tprovide" minOccurs="1"
                maxOccurs="unbounded" />
            <!-- It is non-deterministic, per W3C XML Schema 1.0:
            http://www.w3.org/TR/xmlschema-1/#cos-nonambig
```

```
            to use name space="##any" below. -->
        <any namespace="##other" processContents="lax" minOccurs="0"
            maxOccurs="unbounded" />
    </sequence>
    <attribute name="scope" type="scr:Tservice_scope" default="singleton"
        use="optional" />
    <anyAttribute processContents="lax" />
</complexType>
<complexType name="Tprovide">
    <sequence>
        <any namespace="##any" processContents="lax" minOccurs="0"
            maxOccurs="unbounded" />
    </sequence>
    <attribute name="interface" type="token" use="required" />
    <anyAttribute processContents="lax" />
</complexType>
<complexType name="Treference">
    <sequence>
        <any namespace="##any" processContents="lax" minOccurs="0"
            maxOccurs="unbounded" />
    </sequence>
    <attribute name="name" type="token" use="optional">
        <annotation>
            <documentation xml:lang="en">
                The default value of this attribute is the value of
                the interface attribute of this element. If multiple
                instances of this element within a component element
                use the same value for the interface attribute, then
                using the default value for this attribute will result
                in duplicate names. In this case, this attribute
                must be specified with a unique value.
            </documentation>
        </annotation>
    </attribute>
    <attribute name="interface" type="token" use="required" />
    <attribute name="cardinality" type="scr:Tcardinality"
        default="1..1" use="optional" />
    <attribute name="policy" type="scr:Tpolicy" default="static"
        use="optional" />
    <attribute name="policy-option" type="scr:Tpolicy-option"
        default="reluctant" use="optional" />
    <attribute name="target" type="string" use="optional" />
    <attribute name="bind" type="token" use="optional" />
    <attribute name="unbind" type="token" use="optional" />
    <attribute name="updated" type="token" use="optional" />
    <attribute name="scope" type="scr:Treference_scope" default="bundle"
        use="optional" />
    <attribute name="field" type="token" use="optional" />
    <attribute name="field-option" type="scr:Tfield-option" default="replace"
        use="optional" />
    <attribute name="field-collection-type" type="scr:Tfield-collection-type"
        default="service" use="optional" />
    <anyAttribute processContents="lax" />
</complexType>
<simpleType name="Tproperty_type">
    <restriction base="string">
        <enumeration value="String" />
        <enumeration value="Long" />
        <enumeration value="Double" />
        <enumeration value="Float" />
        <enumeration value="Integer" />
        <enumeration value="Byte" />
        <enumeration value="Character" />
        <enumeration value="Boolean" />
        <enumeration value="Short" />
    </restriction>
</simpleType>
<simpleType name="Tcardinality">
    <restriction base="string">
        <enumeration value="0..1" />
        <enumeration value="0..n" />
        <enumeration value="1..1" />
        <enumeration value="1..n" />
    </restriction>
</simpleType>
```

```
    <simpleType name="Tpolicy">
        <restriction base="string">
            <enumeration value="static" />
            <enumeration value="dynamic" />
        </restriction>
    </simpleType>
    <simpleType name="Tpolicy-option">
        <restriction base="string">
            <enumeration value="reluctant" />
            <enumeration value="greedy" />
        </restriction>
    </simpleType>
    <simpleType name="Tconfiguration-policy">
        <restriction base="string">
            <enumeration value="optional" />
            <enumeration value="require" />
            <enumeration value="ignore" />
        </restriction>
    </simpleType>
    <simpleType name="Tservice_scope">
        <restriction base="string">
            <enumeration value="singleton" />
            <enumeration value="bundle" />
            <enumeration value="prototype" />
        </restriction>
    </simpleType>
    <simpleType name="Treference_scope">
        <restriction base="string">
            <enumeration value="bundle" />
            <enumeration value="prototype" />
            <enumeration value="prototype_required" />
        </restriction>
    </simpleType>
    <simpleType name="Tfield-option">
        <restriction base="string">
            <enumeration value="replace" />
            <enumeration value="update" />
        </restriction>
    </simpleType>
    <simpleType name="Tfield-collection-type">
        <restriction base="string">
            <enumeration value="service" />
            <enumeration value="properties" />
            <enumeration value="reference" />
            <enumeration value="serviceobjects" />
            <enumeration value="tuple" />
        </restriction>
    </simpleType>
    <attribute name="must-understand" type="boolean">
        <annotation>
            <documentation xml:lang="en">
                This attribute should be used by extensions to documents
                to require that the document consumer understand the
                extension. This attribute must be qualified when used.
            </documentation>
        </annotation>
    </attribute>
</schema>
```

SCR must not require component descriptions to specify the elements in the order required by the schema. SCR must allow other orderings since arbitrary orderings of these elements do not affect the meaning of the component description. Only the relative ordering of property, properties and reference elements have meaning for overriding previously set property values.

The schema is also available in digital form from [5] *OSGi XML Schemas.*

112.12   org.osgi.service.component

Service Component Package Version 1.3.

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

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

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

112.12.1   Summary

- ComponentConstants - Defines standard names for Service Component constants.
- ComponentContext - A Component Context object is used by a component instance to interact with its execution context including locating services by reference name.
- ComponentException - Unchecked exception which may be thrown by Service Component Runtime.
- ComponentFactory - When a component is declared with the factory attribute on its component element, Service Component Runtime will register a Component Factory service to allow new component configurations to be created and activated rather than automatically creating and activating component configuration as necessary.
- ComponentInstance - A ComponentInstance encapsulates a component instance of an activated component configuration.
- ComponentServiceObjects - Allows multiple service objects for a service to be obtained.

112.12.2   public interface ComponentConstants

Defines standard names for Service Component constants.

Provider Type   Consumers of this API must not implement this type

112.12.2.1   public static final String COMPONENT_CAPABILITY_NAME = "osgi.component"

Capability name for Service Component Runtime.

Used in Provide-Capability and Require-Capability manifest headers with the osgi.extender namespace. For example:

```
 Require-Capability: osgi.extender;
  filter:="(&(osgi.extender=osgi.component)(version>=1.3)(!(version>=2.0)))"
```

Since   1.3

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

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

112.12.2.3   public static final String COMPONENT_ID = "component.id"

A component property that contains the generated id for a component configuration. The value of this property must be of type Long.

The value of this property is assigned by Service Component Runtime when a component configuration is created. Service Component Runtime assigns a unique value that is larger than all previously assigned values since Service Component Runtime was started. These values are NOT persistent across restarts of Service Component Runtime.

112.12.2.4   **public static final String COMPONENT_NAME = "component.name"**

A component property for a component configuration that contains the name of the component as specified in the name attribute of the component element. The value of this property must be of type String.

112.12.2.5   **public static final int DEACTIVATION_REASON_BUNDLE_STOPPED = 6**

The component configuration was deactivated because the bundle was stopped.

Since 1.1

112.12.2.6   **public static final int DEACTIVATION_REASON_CONFIGURATION_DELETED = 4**

The component configuration was deactivated because its configuration was deleted.

Since 1.1

112.12.2.7   **public static final int DEACTIVATION_REASON_CONFIGURATION_MODIFIED = 3**

The component configuration was deactivated because its configuration was changed.

Since 1.1

112.12.2.8   **public static final int DEACTIVATION_REASON_DISABLED = 1**

The component configuration was deactivated because the component was disabled.

Since 1.1

112.12.2.9   **public static final int DEACTIVATION_REASON_DISPOSED = 5**

The component configuration was deactivated because the component was disposed.

Since 1.1

112.12.2.10   **public static final int DEACTIVATION_REASON_REFERENCE = 2**

The component configuration was deactivated because a reference became unsatisfied.

Since 1.1

112.12.2.11   **public static final int DEACTIVATION_REASON_UNSPECIFIED = 0**

The reason the component configuration was deactivated is unspecified.

Since 1.1

112.12.2.12   **public static final String REFERENCE_TARGET_SUFFIX = ".target"**

The suffix for reference target properties. These properties contain the filter to select the target services for a reference. The value of this property must be of type String.

112.12.2.13   **public static final String SERVICE_COMPONENT = "Service-Component"**

Manifest header specifying the XML documents within a bundle that contain the bundle's Service Component descriptions.

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

112.12.3   public interface ComponentContext

A Component Context object is used by a component instance to interact with its execution context including locating services by reference name. Each component instance has a unique Component Context.

A component instance may obtain its Component Context object through its activate, modified, and deactivate methods.

Concurrency   Thread-safe

Provider Type   Consumers of this API must not implement this type

112.12.3.1   **public void disableComponent(String name)**

name   The name of a component.

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

This method must return after changing the enabled state of the specified component name. Any actions that result from this, such as activating or deactivating a component configuration, must occur asynchronously to this method call.

112.12.3.2   **public void enableComponent(String name)**

name   The name of a component or null to indicate all components in the bundle.

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

This method must return after changing the enabled state of the specified component name. Any actions that result from this, such as activating or deactivating a component configuration, must occur asynchronously to this method call.

112.12.3.3   **public BundleContext getBundleContext()**

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

Returns   The BundleContext of the bundle containing this component.

112.12.3.4   **public ComponentInstance getComponentInstance()**

□   Returns the Component Instance object for the component instance associated with this Component Context.

Returns   The Component Instance object for the component instance.

112.12.3.5   **public Dictionary<String,Object> getProperties()**

□   Returns the component properties for this Component Context.

Returns   The properties for this Component Context. The Dictionary is read only and cannot be modified.

112.12.3.6   **public ServiceReference<?> getServiceReference()**

□   If the component instance is registered as a service using the service element, then this method returns the service reference of the service provided by this component instance.

This method will return null if the component instance is not registered as a service.

Returns   The ServiceReference object for the component instance or null if the component instance is not registered as a service.

112.12.3.7   **public Bundle getUsingBundle()**

□   If the component instance is registered as a service using the servicescope="bundle" or servicescope="prototype" attribute, then this method returns the bundle using the service provided by the component instance.

This method will return null if:

• The component instance is not a service, then no bundle can be using it as a service.

- The component instance is a service but did not specify the servicescope="bundle" or servicescope="prototype" attribute, then all bundles using the service provided by the component instance will share the same component instance.
- The service provided by the component instance is not currently being used by any bundle.

Returns The bundle using the component instance as a service or null.

112.12.3.8   **public Object locateService(String name)**

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

□ Returns the service object for the specified reference name.

If the cardinality of the reference is 0..n or 1..n and multiple services are bound to the reference, the service with the highest ranking (as specified in its Constants.SERVICE_RANKING property) is returned. If there is a tie in ranking, the service with the lowest service id (as specified in its Constants.SERVICE_ID property); that is, the service that was registered first is returned.

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

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

112.12.3.9   **public S locateService(String name,ServiceReference‹S› reference)**

Type Arguments ‹S›

‹S› Type of Service.

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

reference The ServiceReference to a bound service. This must be a ServiceReference provided to the component via the bind or unbind method for the specified reference name.

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

Returns A service object for the referenced service or null if the specified ServiceReference is not a bound service for the specified reference name.

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

112.12.3.10   **public Object[] locateServices(String name)**

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

□ Returns the service objects for the specified reference name.

Returns An array of service objects for the referenced service or null if the reference cardinality is 0..1 or 0..n and no bound service is available. If the reference cardinality is 0..1 or 1..1 and a bound service is available, the array will have exactly one element.

Throws ComponentException– If Service Component Runtime catches an exception while activating a bound service.

112.12.4   **public class ComponentException**
               extends RuntimeException

Unchecked exception which may be thrown by Service Component Runtime.

112.12.4.1   **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.

112.12.4.2   **public ComponentException(String message)**

message The message for the exception.

        □ Construct a new ComponentException with the specified message.

112.12.4.3   **public ComponentException(Throwable cause)**

cause The cause of the exception. May be null.

        □ Construct a new ComponentException with the specified cause.

112.12.4.4   **public Throwable getCause()**

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

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

112.12.4.5   **public Throwable initCause(Throwable cause)**

cause The cause of this exception.

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

Returns This exception.

Throws IllegalArgumentException– If the specified cause is this exception.

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

112.12.5   public interface ComponentFactory

When a component is declared with the factory attribute on its component element, Service Component Runtime will register a Component Factory service to allow new component configurations to be created and activated rather than automatically creating and activating component configuration as necessary.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

112.12.5.1   **public ComponentInstance newInstance(Dictionary<String,?> properties)**

properties Additional properties for the component configuration or null if there are no additional properties.

        □ Create and activate a new component configuration. Additional properties may be provided for the component configuration.

Returns A ComponentInstance object encapsulating the component instance of the component configuration. The component configuration has been activated and, if the component specifies a service element, the component instance has been registered as a service.

Throws ComponentException– If Service Component Runtime is unable to activate the component configuration.

112.12.6   public interface ComponentInstance

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

ComponentInstances are never reused. A new ComponentInstance object will be created when the component configuration is activated again.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

112.12.6.1 public void dispose()

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

112.12.6.2 public Object getInstance()

☐ Returns the component instance of the activated component configuration.

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

112.12.7 public interface ComponentServiceObjects‹S›

‹S› Type of Service

Allows multiple service objects for a service to be obtained.

A component instance can receive a ComponentServiceObjects object via a reference that is typed ComponentServiceObjects.

For services with prototype scope, multiple service objects for the service can be obtained. For services with singleton or bundle scope, only one, use-counted service object is available.

Any unreleased service objects obtained from this ComponentServiceObjects object are automatically released by Service Component Runtime when the service becomes unbound.

See Also ServiceObjects

Since 1.3

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

112.12.7.1 public S getService()

☐ Returns a service object for the associated service.

This method will always return null when the associated service has been become unbound.

Returns A service object for the associated service or null if the service is unbound, the customized service object returned by a ServiceFactory does not implement the classes under which it was registered or the ServiceFactory threw an exception.

Throws IllegalStateException– If the associated service has been become unbound.

See Also ungetService(Object)

112.12.7.2 public ServiceReference‹S› getServiceReference()

☐ Returns the ServiceReference for the service associated with this ComponentServiceObjects object.

Returns The ServiceReference for the service associated with this ComponentServiceObjects object.

112.12.7.3 public void ungetService(S service)

service A service object previously provided by this ComponentServiceObjects object.

☐ Releases a service object for the associated service.

The specified service object must no longer be used and all references to it should be destroyed after calling this method.

Throws IllegalStateException– If the associated service has been become unbound.

IllegalArgumentException– If the specified service object was not provided by this ComponentServiceObjects object.

See Also   getService()

112.13   org.osgi.service.component.annotations

Service Component Annotations Package Version 1.3.

This package is not used at runtime. Annotated classes are processed by tools to generate Component Descriptions which are used at runtime.

112.13.1   Summary

- Activate - Identify the annotated method as the activate method of a Service Component.
- Component - Identify the annotated class as a Service Component.
- ConfigurationPolicy - Configuration Policy for the Component annotation.
- Deactivate - Identify the annotated method as the deactivate method of a Service Component.
- FieldOption - Field options for the Reference annotation.
- Modified - Identify the annotated method as the modified method of a Service Component.
- Reference - Identify the annotated member as a reference of a Service Component.
- ReferenceCardinality - Cardinality for the Reference annotation.
- ReferencePolicy - Policy for the Reference annotation.
- ReferencePolicyOption - Policy option for the Reference annotation.
- ReferenceScope - Reference scope for the Reference annotation.
- ServiceScope - Service scope for the Component annotation.

112.13.2   @Activate

Identify the annotated method as the activate method of a Service Component.

The annotated method is the activate method of the Component.

This annotation is not processed at runtime by Service Component Runtime. It must be processed by tools and used to add a Component Description to the bundle.

See Also   The activate attribute of the component element of a Component Description.

Since   1.1

Retention   CLASS

Target   METHOD

112.13.3   @Component

Identify the annotated class as a Service Component.

The annotated class is the implementation class of the Component.

This annotation is not processed at runtime by Service Component Runtime. It must be processed by tools and used to add a Component Description to the bundle.

See Also   The component element of a Component Description.

Retention   CLASS

Target   TYPE

112.13.3.1   **String name default ""**

□   The name of this Component.

If not specified, the name of this Component is the fully qualified type name of the class being annotated.

See Also The name attribute of the component element of a Component Description.

112.13.3.2   Class<?>[] service default {}

☐ The types under which to register this Component as a service.

If no service should be registered, the empty value {} must be specified.

If not specified, the service types for this Component are all the *directly* implemented interfaces of the class being annotated.

See Also The service element of a Component Description.

112.13.3.3   String factory default ""

☐ The factory identifier of this Component. Specifying a factory identifier makes this Component a Factory Component.

If not specified, the default is that this Component is not a Factory Component.

See Also The factory attribute of the component element of a Component Description.

112.13.3.4   boolean servicefactory default false

☐ Declares whether this Component uses the OSGi ServiceFactory concept and each bundle using this Component's service will receive a different component instance.

This element is ignored when the scope() element does not have the default value. If true, this Component uses bundle service scope. If false or not specified, this Component uses singleton service scope. If the factory() element is specified or the immediate() element is specified with true, this element can only be specified with false.

See Also The scope attribute of the service element of a Component Description.

Deprecated Since 1.3. Replaced by scope().

112.13.3.5   boolean enabled default true

☐ Declares whether this Component is enabled when the bundle containing it is started.

If true or not specified, this Component is enabled. If false, this Component is disabled.

See Also The enabled attribute of the component element of a Component Description.

112.13.3.6   boolean immediate default false

☐ Declares whether this Component must be immediately activated upon becoming satisfied or whether activation should be delayed.

If true, this Component must be immediately activated upon becoming satisfied. If false, activation of this Component is delayed. If this property is specified, its value must be false if the factory() property is also specified or must be true if the service() property is specified with an empty value.

If not specified, the default is false if the factory() property is specified or the service() property is not specified or specified with a non-empty value and true otherwise.

See Also The immediate attribute of the component element of a Component Description.

112.13.3.7   String[] property default {}

☐ Properties for this Component.

Each property string is specified as "name=value". The type of the property value can be specified in the name as name:type=value. The type must be one of the property types supported by the type attribute of the property element of a Component Description.

To specify a property with multiple values, use multiple name, value pairs. For example, `"foo=bar"`, `"foo=baz"`.

See Also   The property element of a Component Description.

112.13.3.8   String[] properties default {}

□ Property entries for this Component.

Specifies the name of an entry in the bundle whose contents conform to a standard Java Properties File. The entry is read and processed to obtain the properties and their values.

See Also   The properties element of a Component Description.

112.13.3.9   String xmlns default ""

□ The XML name space of the Component Description for this Component.

If not specified, the XML name space of the Component Description for this Component should be the lowest Declarative Services XML name space which supports all the specification features used by this Component.

See Also   The XML name space specified for a Component Description.

112.13.3.10   ConfigurationPolicy configurationPolicy default OPTIONAL

□ The configuration policy of this Component.

Controls whether component configurations must be satisfied depending on the presence of a corresponding Configuration object in the OSGi Configuration Admin service. A corresponding configuration is a Configuration object where the PID equals the name of the component.

If not specified, the configuration policy is based upon whether the component is also annotated with the Meta Type Designate annotation.

- Not annotated with `Designate` - The configuration policy is OPTIONAL.
- Annotated with `Designate(factory=false)` - The configuration policy is OPTIONAL.
- Annotated with `Designate(factory=true)` - The configuration policy is REQUIRE.

See Also   The configuration-policy attribute of the component element of a Component Description.

Since   1.1

112.13.3.11   String[] configurationPid default "$"

□ The configuration PIDs for the configuration of this Component.

Each value specifies a configuration PID for this Component.

If no value is specified, the name of this Component is used as the configuration PID of this Component.

A special string (`"$"`) can be used to specify the name of the component as a configuration PID. The NAME constant holds this special string. For example:

```
@Component(configurationPid={"com.acme.system", Component.NAME})
```

Tools creating a Component Description from this annotation must replace the special string with the actual name of this Component.

See Also   The configuration-pid attribute of the component element of a Component Description.

Since   1.2

112.13.3.12   ServiceScope scope default DEFAULT

□ The service scope for the service of this Component.

If not specified (and the deprecated servicefactory() element is not specified), the singleton service scope is used. If the factory() element is specified or the immediate() element is specified with true, this element can only be specified with the singleton service scope.

See Also　The scope attribute of the service element of a Component Description.

Since　1.3

112.13.3.13　Reference[] reference default {}

□　The lookup strategy references of this Component.

To access references using the lookup strategy, Reference annotations are specified naming the reference and declaring the type of the referenced service. The referenced service can be accessed using one of the locateService methods of ComponentContext.

To access references using the event strategy, bind methods are annotated with Reference. To access references using the field strategy, fields are annotated with Reference.

See Also　The reference element of a Component Description.

Since　1.3

112.13.3.14　String NAME = "$"

Special string representing the name of this Component.

This string can be used in configurationPid() to specify the name of the component as a configuration PID. For example:

```
@Component(configurationPid={"com.acme.system", Component.NAME})
```

Tools creating a Component Description from this annotation must replace the special string with the actual name of this Component.

Since　1.3

112.13.4　enum ConfigurationPolicy

Configuration Policy for the Component annotation.

Controls whether component configurations must be satisfied depending on the presence of a corresponding Configuration object in the OSGi Configuration Admin service. A corresponding configuration is a Configuration object where the PID is the name of the component.

Since　1.1

112.13.4.1　OPTIONAL

Use the corresponding Configuration object if present but allow the component to be satisfied even if the corresponding Configuration object is not present.

112.13.4.2　REQUIRE

There must be a corresponding Configuration object for the component configuration to become satisfied.

112.13.4.3　IGNORE

Always allow the component configuration to be satisfied and do not use the corresponding Configuration object even if it is present.

112.13.5　@Deactivate

Identify the annotated method as the deactivate method of a Service Component.

The annotated method is the deactivate method of the Component.

This annotation is not processed at runtime by Service Component Runtime. It must be processed by tools and used to add a Component Description to the bundle.

See Also   The deactivate attribute of the component element of a Component Description.

Since   1.1

Retention   CLASS

Target   METHOD

112.13.6   enum FieldOption

Field options for the Reference annotation.

Since   1.3

112.13.6.1   UPDATE

The update field option is used to update the collection referenced by the field when there are changes to the bound services.

This field option can only be used when the field reference has dynamic policy and multiple cardinality.

112.13.6.2   REPLACE

The replace field option is used to replace the field value with a new value when there are changes to the bound services.

112.13.7   @Modified

Identify the annotated method as the modified method of a Service Component.

The annotated method is the modified method of the Component.

This annotation is not processed at runtime by Service Component Runtime. It must be processed by tools and used to add a Component Description to the bundle.

See Also   The modified attribute of the component element of a Component Description.

Since   1.1

Retention   CLASS

Target   METHOD

112.13.8   @Reference

Identify the annotated member as a reference of a Service Component.

When the annotation is applied to a method, the method is the bind method of the reference. When the annotation is applied to a field, the field will contain the bound service(s) of the reference.

This annotation is not processed at runtime by Service Component Runtime. It must be processed by tools and used to add a Component Description to the bundle.

In the generated Component Description for a component, the references must be ordered in ascending lexicographical order (using String.compareTo) of the reference names.

See Also   The reference element of a Component Description.

Retention   CLASS

Target   METHOD, FIELD

112.13.8.1   String name default ""

  □ The name of this reference.

The name of this reference must be specified when using this annotation in the
Component.reference() element since there is no annotated member from which the name can be
determined. If not specified, the name of this reference is based upon how this annotation is used:

- Annotated method - If the method name begins with bind, set or add, that prefix is removed to
  create the name of the reference. Otherwise, the name of the reference is the method name.
- Annotated field - The name of the reference is the field name.

See Also   The name attribute of the reference element of a Component Description.

112.13.8.2   **Class<?> service default Object.class**

□ The type of the service for this reference.

The type of the service for this reference must be specified when using this annotation in the
Component.reference() element since there is no annotated member from which the type of the ser-
vice can be determined.

If not specified, the type of the service for this reference is based upon how this annotation is used:

- Annotated method - The type of the service is the type of the first argument of the method.
- Annotated field - The type of the service is based upon the type of the field being annotated and
  the cardinality of the reference. If the cardinality is either 0..n, or 1..n, the type of the field must
  be one of java.util.Collection, java.util.List, or a subtype of java.util.Collection so the type of the
  service is the generic type of the collection. Otherwise, the type of the service is the type of the
  field.

See Also   The interface attribute of the reference element of a Component Description.

112.13.8.3   **ReferenceCardinality cardinality default MANDATORY**

□ The cardinality of this reference.

If not specified, the cardinality of this reference is based upon how this annotation is used:

- Annotated method - The cardinality is 1..1.
- Annotated field - The cardinality is based on the type of the field. If the type is either
  java.util.Collection, java.util.List, or a subtype of java.util.Collection, the cardinality is 0..n. Oth-
  erwise the cardinality is 1..1.
- Component.reference() element - The cardinality is 1..1.

See Also   The cardinality attribute of the reference element of a Component Description.

112.13.8.4   **ReferencePolicy policy default STATIC**

□ The policy for this reference.

If not specified, the policy of this reference is based upon how this annotation is used:

- Annotated method - The policy is STATIC.
- Annotated field - The policy is based on the modifiers of the field. If the field is declared volatile,
  the policy is ReferencePolicy.DYNAMIC. Otherwise the policy is STATIC.
- Component.reference() element - The policy is STATIC.

See Also   The policy attribute of the reference element of a Component Description.

112.13.8.5   **String target default ""**

□ The target property for this reference.

If not specified, no target property is set.

See Also   The target attribute of the reference element of a Component Description.

112.13.8.6   **ReferencePolicyOption policyOption default RELUCTANT**

□ The policy option for this reference.

If not specified, the RELUCTANT reference policy option is used.

See Also The policy-option attribute of the reference element of a Component Description.

Since 1.2

112.13.8.7   **ReferenceScope scope default BUNDLE**

□ The reference scope for this reference.

If not specified, the bundle reference scope is used.

See Also The scope attribute of the reference element of a Component Description.

Since 1.3

112.13.8.8   **String bind default ""**

□ The name of the bind method for this reference.

If specified and this reference annotates a method, the specified name must match the name of the annotated method.

If not specified, the name of the bind method is based upon how this annotation is used:

· Annotated method - The name of the annotated method is the name of the bind method.
· Annotated field - There is no bind method name.
· Component.reference() element - There is no bind method name.

If there is a bind method name, the component must contain a method with that name.

See Also The bind attribute of the reference element of a Component Description.

Since 1.3

112.13.8.9   **String updated default ""**

□ The name of the updated method for this reference.

If not specified, the name of the updated method is based upon how this annotation is used:

· Annotated method - The name of the updated method is created from the name of the annotated method. If the name of the annotated method begins with bind, set or add, that prefix is replaced with updated to create the name candidate for the updated method. Otherwise, updated is pre-fixed to the name of the annotated method to create the name candidate for the updated method. If the component type contains a method with the candidate name, the candidate name is used as the name of the updated method. To declare no updated method when the component type contains a method with the candidate name, the value "-" must be used.
· Annotated field - There is no updated method name.
· Component.reference() element - There is no updated method name.

If there is an updated method name, the component must contain a method with that name.

See Also The updated attribute of the reference element of a Component Description.

Since 1.2

112.13.8.10   **String unbind default ""**

□ The name of the unbind method for this reference.

If not specified, the name of the unbind method is based upon how this annotation is used:

- Annotated method - The name of the unbind method is created from the name of the annotated method. If the name of the annotated method begins with bind, set or add, that prefix is replaced with unbind, unset or remove, respectively, to create the name candidate for the unbind method. Otherwise, un is prefixed to the name of the annotated method to create the name candidate for the unbind method. If the component type contains a method with the candidate name, the candidate name is used as the name of the unbind method. To declare no unbind method when the component type contains a method with the candidate name, the value "-" must be used.
- Annotated field - There is no unbind method name.
- Component.reference() element - There is no unbind method name.

If there is an unbind method name, the component must contain a method with that name.

See Also The unbind attribute of the reference element of a Component Description.

112.13.8.11   String field default ""

☐ The name of the field for this reference.

If specified and this reference annotates a field, the specified name must match the name of the annotated field.

If not specified, the name of the field is based upon how this annotation is used:

- Annotated method - There is no field name.
- Annotated field - The name of the annotated field is the name of the field.
- Component.reference() element - There is no field name.

If there is a field name, the component must contain a field with that name.

See Also The field attribute of the reference element of a Component Description.

Since 1.3

112.13.8.12   FieldOption fieldOption default REPLACE

☐ The field option for this reference.

If not specified, the field option is based upon how this annotation is used:

- Annotated method - There is no field option.
- Annotated field - The field option is based upon the policy and cardinality of the reference and the modifiers of the field. If the policy is ReferencePolicy.DYNAMIC, the cardinality is 0..n or 1..n, and the field is declared final, the field option is FieldOption.UPDATE. Otherwise, the field option is FieldOption.REPLACE
- Component.reference() element - There is no field option.

See Also The field-option attribute of the reference element of a Component Description.

Since 1.3

112.13.9   enum ReferenceCardinality

Cardinality for the Reference annotation.

Specifies if the reference is optional and if the component implementation support a single bound service or multiple bound services.

112.13.9.1   OPTIONAL

The reference is optional and unary. That is, the reference has a cardinality of 0..1.

112.13.9.2   MANDATORY

The reference is mandatory and unary. That is, the reference has a cardinality of 1..1.

112.13.9.3   MULTIPLE

The reference is optional and multiple. That is, the reference has a cardinality of 0..n.

112.13.9.4   AT_LEAST_ONE

The reference is mandatory and multiple. That is, the reference has a cardinality of 1..n.

112.13.10   enum ReferencePolicy

Policy for the Reference annotation.

112.13.10.1   STATIC

The static policy is the most simple policy and is the default policy. A component instance never sees any of the dynamics. Component configurations are deactivated before any bound service for a reference having a static policy becomes unavailable. If a target service is available to replace the bound service which became unavailable, the component configuration must be reactivated and bound to the replacement service.

112.13.10.2   DYNAMIC

The dynamic policy is slightly more complex since the component implementation must properly handle changes in the set of bound services. With the dynamic policy, SCR can change the set of bound services without deactivating a component configuration. If the component uses the event strategy to access services, then the component instance will be notified of changes in the set of bound services by calls to the bind and unbind methods.

112.13.11   enum ReferencePolicyOption

Policy option for the Reference annotation.

Since 1.2

112.13.11.1   RELUCTANT

The reluctant policy option is the default policy option for both static and dynamic reference policies. When a new target service for a reference becomes available, references having the reluctant policy option for the static policy or the dynamic policy with a unary cardinality will ignore the new target service. References having the dynamic policy with a multiple cardinality will bind the new target service.

112.13.11.2   GREEDY

The greedy policy option is a valid policy option for both static and dynamic reference policies. When a new target service for a reference becomes available, references having the greedy policy option will bind the new target service.

112.13.12   enum ReferenceScope

Reference scope for the Reference annotation.

Since 1.3

112.13.12.1   BUNDLE

A single service object is used for all references to the service in this bundle.

112.13.12.2   PROTOTYPE

If the bound service has prototype service scope, then each instance of the component with this reference can receive a unique instance of the service. If the bound service does not have prototype service scope, then this reference scope behaves the same as BUNDLE.

112.13.12.3   **PROTOTYPE_REQUIRED**

Bound services must have prototype service scope. Each instance of the component with this reference can receive a unique instance of the service.

112.13.13   enum ServiceScope

Service scope for the Component annotation.

Since   1.3

112.13.13.1   **SINGLETON**

When the component is registered as a service, it must be registered as a bundle scope service but only a single instance of the component must be used for all bundles using the service.

112.13.13.2   **BUNDLE**

When the component is registered as a service, it must be registered as a bundle scope service and an instance of the component must be created for each bundle using the service.

112.13.13.3   **PROTOTYPE**

When the component is registered as a service, it must be registered as a prototype scope service and an instance of the component must be created for each distinct request for the service.

112.13.13.4   **DEFAULT**

Default element value for annotation. This is used to distinguish the default value for an element and should not otherwise be used.

112.14   org.osgi.service.component.runtime

Service Component Runtime Package Version 1.3.

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

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

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

112.14.1   Summary

- `ServiceComponentRuntime` - The `ServiceComponentRuntime` service represents the Declarative Services actor, known as Service Component Runtime (SCR), that manages the service components and their life cycle.

112.14.2   public interface ServiceComponentRuntime

The `ServiceComponentRuntime` service represents the Declarative Services actor, known as Service Component Runtime (SCR), that manages the service components and their life cycle. The `ServiceComponentRuntime` service allows introspection of the components managed by Service Component Runtime.

This service differentiates between a ComponentDescriptionDTO and a ComponentConfigurationDTO. A ComponentDescriptionDTO is a representation of a declared component description. A

ComponentConfigurationDTO is a representation of an actual instance of a declared component description parameterized by component properties.

Access to this service requires the ServicePermission[ServiceComponentRuntime, GET] permission. It is intended that only administrative bundles should be granted this permission to limit access to the potentially intrusive methods provided by this service.

Since 1.3

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

112.14.2.1 **public Promise<Void> disableComponent(ComponentDescriptionDTO description)**

description The component description to disable. Must not be null.

□ Disables the specified component description.

If the specified component description is currently disabled, this method has no effect.

This method must return after changing the enabled state of the specified component description. Any actions that result from this, such as activating or deactivating a component configuration, must occur asynchronously to this method call.

Returns A promise that will be resolved when the actions that result from changing the enabled state of the specified component have completed.

See Also isComponentEnabled(ComponentDescriptionDTO)

112.14.2.2 **public Promise<Void> enableComponent(ComponentDescriptionDTO description)**

description The component description to enable. Must not be null.

□ Enables the specified component description.

If the specified component description is currently enabled, this method has no effect.

This method must return after changing the enabled state of the specified component description. Any actions that result from this, such as activating or deactivating a component configuration, must occur asynchronously to this method call.

Returns A promise that will be resolved when the actions that result from changing the enabled state of the specified component have completed.

See Also isComponentEnabled(ComponentDescriptionDTO)

112.14.2.3 **public Collection<ComponentConfigurationDTO>**
getComponentConfigurationDTOs(ComponentDescriptionDTO description)

description The component description. Must not be null.

□ Returns the component configurations for the specified component description.

Returns A collection containing a snapshot of the current component configurations for the specified component description. An empty collection is returned if there are none.

112.14.2.4 **public ComponentDescriptionDTO getComponentDescriptionDTO(Bundle bundle,String name)**

bundle The bundle declaring the component description. Must not be null.

name The name of the component description. Must not be null.

□ Returns the ComponentDescriptionDTO declared with the specified name by the specified bundle.

Only component descriptions from active bundles are returned. null if no such component is declared by the given bundle or the bundle is not active.

Returns The declared component description or null if the specified bundle is not active or does not declare a component description with the specified name.

112.14.2.5   **public Collection<ComponentDescriptionDTO> getComponentDescriptionDTOs(Bundle ... bundles)**

bundles The bundles whose declared component descriptions are to be returned. Specifying no bundles, or the equivalent of an empty Bundle array, will return the declared component descriptions from all active bundles.

☐ Returns the component descriptions declared by the specified active bundles.

Only component descriptions from active bundles are returned. If the specified bundles have no declared components or are not active, an empty collection is returned.

Returns The declared component descriptions of the specified active bundles. An empty collection is returned if there are no component descriptions for the specified active bundles.

112.14.2.6   **public boolean isComponentEnabled(ComponentDescriptionDTO description)**

description The component description. Must not be null.

☐ Returns whether the specified component description is currently enabled.

The enabled state of a component description is initially set by the enabled attribute of the component description.

Returns true if the specified component description is currently enabled. Otherwise, false.

See Also enableComponent(ComponentDescriptionDTO), disableComponent(ComponentDescriptionDTO), ComponentContext.disableComponent(String), ComponentContext.enableComponent(String)

112.15   org.osgi.service.component.runtime.dto

Service Component Runtime Data Transfer Objects Package Version 1.3.

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

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

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

112.15.1   Summary

- ComponentConfigurationDTO - A representation of an actual instance of a declared component description parameterized by component properties.
- ComponentDescriptionDTO - A representation of a declared component description.
- ReferenceDTO - A representation of a declared reference to a service.
- SatisfiedReferenceDTO - A representation of a satisfied reference.
- UnsatisfiedReferenceDTO - A representation of an unsatisfied reference.

112.15.2   public class ComponentConfigurationDTO
extends DTO

A representation of an actual instance of a declared component description parameterized by component properties.

Since 1.3

Concurrency Not Thread-safe

112.15.2.1   **public static final int ACTIVE = 8**

The component configuration is active.

This is the normal operational state of a component configuration.

112.15.2.2   **public ComponentDescriptionDTO description**

The representation of the component configuration's component description.

112.15.2.3   **public long id**

The id of the component configuration.

The id is a non-persistent, unique value assigned at runtime. The id is also available as the
component.id component property. The value of this field is unspecified if the state of this compo-
nent configuration is unsatisfied.

112.15.2.4   **public Map<String,Object> properties**

The component properties for the component configuration.

See Also ComponentContext.getProperties()

112.15.2.5   **public static final int SATISFIED = 4**

The component configuration is satisfied.

Any services declared by the component description are registered.

112.15.2.6   **public SatisfiedReferenceDTO[] satisfiedReferences**

The satisfied references.

Each SatisfiedReferenceDTO in the array represents a satisfied reference of the component configu-
ration. The array must be empty if the component configuration has no satisfied references.

112.15.2.7   **public int state**

The current state of the component configuration.

This is one of UNSATISFIED_CONFIGURATION, UNSATISFIED_REFERENCE, SATISFIED or AC-
TIVE.

112.15.2.8   **public static final int UNSATISFIED_CONFIGURATION = 1**

The component configuration is unsatisfied due to a missing required configuration.

112.15.2.9   **public static final int UNSATISFIED_REFERENCE = 2**

The component configuration is unsatisfied due to an unsatisfied reference.

112.15.2.10   **public UnsatisfiedReferenceDTO[] unsatisfiedReferences**

The unsatisfied references.

Each UnsatisfiedReferenceDTO in the array represents an unsatisfied reference of the component
configuration. The array must be empty if the component configuration has no unsatisfied refer-
ences.

112.15.2.11   **public ComponentConfigurationDTO()**

112.15.3   public class ComponentDescriptionDTO
extends DTO

A representation of a declared component description.

Since 1.3

Concurrency Not Thread-safe

112.15.3.1   public String activate

The name of the activate method.

This is declared in the activate attribute of the component element. This must be null if the component description does not declare an activate method name.

112.15.3.2   public BundleDTO bundle

The bundle declaring the component description.

112.15.3.3   public String[] configurationPid

The configuration pids.

These are declared in the configuration-pid attribute of the component element. This must contain the default configuration pid if the component description does not declare a configuration pid.

112.15.3.4   public String configurationPolicy

The configuration policy.

This is declared in the configuration-policy attribute of the component element. This must be the default configuration policy if the component description does not declare a configuration policy.

112.15.3.5   public String deactivate

The name of the deactivate method.

This is declared in the deactivate attribute of the component element. This must be null if the component description does not declare a deactivate method name.

112.15.3.6   public boolean defaultEnabled

The initial enabled state.

This is declared in the enabled attribute of the component element.

112.15.3.7   public String factory

The component factory name.

This is declared in the factory attribute of the component element. This must be null if the component description is not declared as a component factory.

112.15.3.8   public boolean immediate

The immediate state.

This is declared in the immediate attribute of the component element.

112.15.3.9   public String implementationClass

The fully qualified name of the implementation class.

This is declared in the class attribute of the implementation element.

112.15.3.10   public String modified

The name of the modified method.

This is declared in the modified attribute of the component element. This must be null if the component description does not declare a modified method name.

112.15.3.11   **public String name**

The name of the component.

This is declared in the name attribute of the component element. This must be the default name if the component description does not declare a name.

112.15.3.12   **public Map<String,Object> properties**

The declared component properties.

These are declared in the property and properties elements.

112.15.3.13   **public ReferenceDTO[] references**

The referenced services.

These are declared in the reference elements. The array must be empty if the component description does not declare references to any services.

112.15.3.14   **public String scope**

The service scope.

This is declared in the scope attribute of the service element. This must be null if the component description does not declare any service interfaces.

112.15.3.15   **public String[] serviceInterfaces**

The fully qualified names of the service interfaces.

These are declared in the interface attribute of the provide elements. The array must be empty if the component description does not declare any service interfaces.

112.15.3.16   **public ComponentDescriptionDTO()**

112.15.4   public class ReferenceDTO
extends DTO

A representation of a declared reference to a service.

Since   1.3

Concurrency   Not Thread-safe

112.15.4.1   **public String bind**

The name of the bind method of the reference.

This is declared in the bind attribute of the reference element. This must be null if the component description does not declare a bind method for the reference.

112.15.4.2   **public String cardinality**

The cardinality of the reference.

This is declared in the cardinality attribute of the reference element. This must be the default cardinality if the component description does not declare a cardinality for the reference.

112.15.4.3   **public String field**

The name of the field of the reference.

This is declared in the field attribute of the reference element. This must be null if the component description does not declare a field for the reference.

112.15.4.4   **public String fieldOption**

The field option of the reference.

This is declared in the field-option attribute of the reference element. This must be null if the component description does not declare a field for the reference.

112.15.4.5   **public String interfaceName**

The service interface of the reference.

This is declared in the interface attribute of the reference element.

112.15.4.6   **public String name**

The name of the reference.

This is declared in the name attribute of the reference element. This must be the default name if the component description does not declare a name for the reference.

112.15.4.7   **public String policy**

The policy of the reference.

This is declared in the policy attribute of the reference element. This must be the default policy if the component description does not declare a policy for the reference.

112.15.4.8   **public String policyOption**

The policy option of the reference.

This is declared in the policy-option attribute of the reference element. This must be the default policy option if the component description does not declare a policy option for the reference.

112.15.4.9   **public String scope**

The scope of the reference.

This is declared in the scope attribute of the reference element. This must be the default scope if the component description does not declare a scope for the reference.

112.15.4.10   **public String target**

The target of the reference.

This is declared in the target attribute of the reference element. This must be null if the component description does not declare a target for the reference.

112.15.4.11   **public String unbind**

The name of the unbind method of the reference.

This is declared in the unbind attribute of the reference element. This must be null if the component description does not declare an unbind method for the reference.

112.15.4.12   **public String updated**

The name of the updated method of the reference.

This is declared in the updated attribute of the reference element. This must be null if the component description does not declare an updated method for the reference.

112.15.4.13   **public ReferenceDTO()**

112.15.5   public class SatisfiedReferenceDTO
extends DTO

A representation of a satisfied reference.

Since   1.3

Concurrency   Not Thread-safe

112.15.5.1   public ServiceReferenceDTO[] boundServices

The bound services.

Each ServiceReferenceDTO in the array represents a service bound to the satisfied reference. The array must be empty if there are no bound services.

112.15.5.2   public String name

The name of the declared reference.

This is declared in the name attribute of the reference element of the component description.

See Also   ReferenceDTO.name

112.15.5.3   public String target

The target property of the satisfied reference.

This is the value of the component property whose name is the concatenation of the declared reference name and ".target". This must be null if no target property is set for the reference.

112.15.5.4   public SatisfiedReferenceDTO()

112.15.6   public class UnsatisfiedReferenceDTO
extends DTO

A representation of an unsatisfied reference.

Since   1.3

Concurrency   Not Thread-safe

112.15.6.1   public String name

The name of the declared reference.

This is declared in the name attribute of the reference element of the component description.

See Also   ReferenceDTO.name

112.15.6.2   public String target

The target property of the unsatisfied reference.

This is the value of the component property whose name is the concatenation of the declared reference name and ".target". This must be null if no target property is set for the reference.

112.15.6.3   public ServiceReferenceDTO[] targetServices

The target services.

Each ServiceReferenceDTO in the array represents a target service for the reference. The array must be empty if there are no target services. The upper bound on the number of target services in the array is the upper bound on the cardinality of the reference.

112.15.6.4   public UnsatisfiedReferenceDTO()

112.16   References

[1] *Automating Service Dependency Management in a Service-Oriented Component Model*
Humberto Cervantes, Richard S. Hall, Proceedings of the Sixth Component-Based Software Engineering Workshop, May 2003, pp. 91-96
http://www-adele.imag.fr/Les.Publications/intConferences/CBSE2003Cer.pdf

[2] *Service Binder*
Humberto Cervantes, Richard S. Hall
http://gravity.sourceforge.net/servicebinder

[3] *Java Properties File*
http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html

[4] *Extensible Markup Language (XML) 1.0*
http://www.w3.org/TR/REC-xml/

[5] *OSGi XML Schemas*
http://www.osgi.org/Specifications

112.17   Changes

- *Configuration Changes* on page 212 has been rewritten to more accurately state the effects of configuration changes on component configurations.
- New introspection API. See *Introspection* on page 220.
- Defined capabilities SCR must provide. See *Capabilities* on page 222.
- Expanded the allowed signatures and argument types for Event Methods. See *Event Methods* on page 181.
- New scope attribute for the service element. See *Service Element* on page 197.
- New scope attribute for the reference element. See *Reference Scope* on page 187.
- New reference element for Component annotation. See *Reference Element* on page 198.
- New minimum cardinality property. See *Minimum Cardinality Property* on page 211.
- Added support for a component using multiple configuration PIDs. See *Component Element* on page 193, *Component Properties* on page 210, and *Deployment* on page 211.
- service.pid property values are aggregated across configuration sources. See *Component Properties* on page 210.
- SCR must check ConfigurationPermission for multi-location Configurations. See *Configuration Multi-Locations and Regions* on page 224.
- Added support for targeted PIDs for Configurations. See *Deployment* on page 211.
- New component property types can be used for type safe component property definition and access. See *Component Property Types* on page 216.
- Specified the ordering of property information component descriptions generated from annotations. See *Ordering of Generated Properties* on page 218.
- Maps of service properties must also implement Comparable. See *Event Methods* on page 181.
- Support for field strategy is added. See *Field Strategy* on page 184. This change affects many areas of this specification.

113   Event Admin Service Specification

Version 1.3

113.1   Introduction

Nearly all the bundles in an OSGi framework must deal with events, either as an event publisher or as an event handler. So far, the preferred mechanism to disperse those events have been the service interface mechanism.

Dispatching events for a design related to X, usually involves a service of type XListener. However, this model does not scale well for fine grained events that must be dispatched to many different handlers. Additionally, the dynamic nature of the OSGi environment introduces several complexities because both event publishers and event handlers can appear and disappear at any time.

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

This specification defines the details for the participants in this event model.

113.1.1   Essentials

- *Simplifications* - The model must significantly simplify the process of programming an event source and an event handler.
- *Dependencies* - Handle the myriad of dependencies between event sources and event handlers for proper cleanup.
- *Synchronicity* - It must be possible to deliver events asynchronously or synchronously with the caller.
- *Event Window* - Only event handlers that are active when an event is published must receive this event, handlers that register later must not see the event.
- *Performance* - The event mechanism must impose minimal overhead in delivering events.
- *Selectivity* - Event listeners must only receive notifications for the event types for which they are interested
- *Reliability* - The Event Admin must ensure that events continue to be delivered regardless the quality of the event handlers.
- *Security* - Publishing and receiving events are sensitive operations that must be protected per event type.
- *Extensibility* - It must be possible to define new event types with their own data types.
- *Native Code* - Events must be able to be passed to native code or come from native code.
- *OSGi Events* - The OSGi Framework, as well as a number of OSGi services, already have number of its own events defined. For uniformity of processing, these have to be mapped into generic event types.
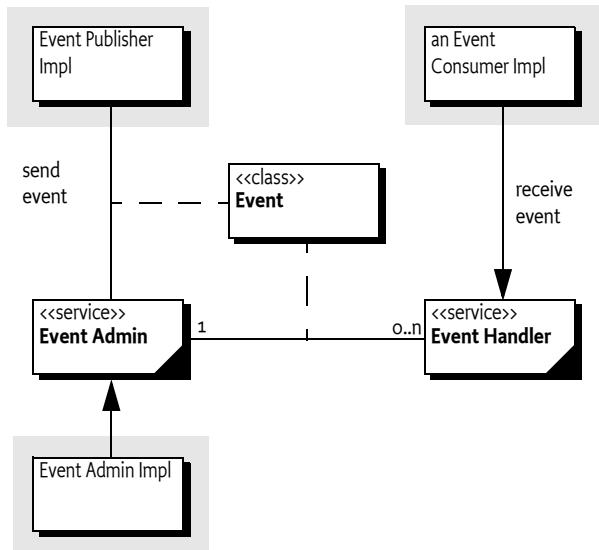
113.1.2   Entities

- *Event* - An Event object has a topic and a Dictionary object that contains the event properties. It is an immutable object.
- *Event Admin* - The service that provides the publish and subscribe model to Event Handlers and Event Publishers.

- *Event Handler* - A service that receives and handles Event objects.
- *Event Publisher* - A bundle that sends event through the Event Admin service.
- *Event Subscriber* - Another name for an Event Handler.
- *Topic* - The name of an Event type.
- *Event Properties* - The set of properties that is associated with an Event.

Figure 113.1   *The Event Admin service org.osgi.service.event package*

113.1.3   Synopsis

The Event Admin service provides a place for bundles to publish events, regardless of their destination. It is also used by Event Handlers to subscribe to specific types of events.

Events are published under a topic, together with a number of event properties. Event Handlers can specify a filter to control the Events they receive on a very fine grained basis.
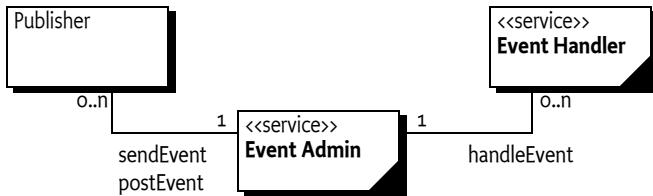
113.1.4   What To Read

- *Architects* - The *Event Admin Architecture* on page 254 provides an overview of the Event Admin service.
- *Event Publishers* - The *Event Publisher* on page 257 provides an introduction of how to write an Event Publisher. The *Event Admin Architecture* on page 254 provides a good overview of the design.
- *Event Subscribers/Handlers* - The *Event Handler* on page 256 provides the rules on how to subscribe and handle events.

113.2   Event Admin Architecture

The Event Admin is based on the *Publish-Subscribe* pattern. This pattern decouples sources from their handlers by interposing an *event channel* between them. The publisher posts events to the channel, which identifies which handlers need to be notified and then takes care of the notification process. This model is depicted in Figure 113.2.

In this model, the event source and event handler are completely decoupled because neither has any direct knowledge of the other. The complicated logic of monitoring changes in the event publishers and event handlers is completely contained within the event channel. This is highly advantageous in an OSGi environment because it simplifies the process of both sending and receiving events.

113.3 The Event

Events have the following attributes:

- *Topic* - A topic that defines what happened. For example, when a bundle is started an event is published that has a topic of `org/osgi/framework/BundleEvent/STARTED`.
- *Properties* - Zero or more properties that contain additional information about the event. For example, the previous example event has a property of `bundle.id` which is set to a `Long` object, among other properties.

113.3.1 Topics

The topic of an event defines the *type* of the event. It is fairly granular in order to give handlers the opportunity to register for just the events they are interested in. When a topic is designed, its name should not include any other information, such as the publisher of the event or the data associated with the event, those parts are intended to be stored in the event properties.

The topic is intended to serve as a first-level filter for determining which handlers should receive the event. Event Admin service implementations use the structure of the topic to optimize the dispatching of the events to the handlers.

Topics are arranged in a hierarchical namespace. Each level is defined by a token and levels are separated by solidi ('/' \u002F). More precisely, the topic must conform to the following grammar:

```
topic ::= token ( '/' token ) *    // See General Syntax Definitions in Core
```

Topics should be designed to become more specific when going from left to right. Handlers can provide a prefix that matches a topic, using the preferred order allows a handler to minimize the number of prefixes it needs to register.

Topics are case-sensitive. As a convention, topics should follow the reverse domain name scheme used by Java packages to guarantee uniqueness. The separator must be a solidus ('/' \u002F) instead of the full stop ('.' \u002E).

This specification uses the convention `fully/qualified/package/ClassName/ACTION`. If necessary, a pseudo-class-name is used.

113.3.2 Properties

Information about the actual event is provided as properties. The property name is a case-sensitive string and the value can be any object. Although any Java object can be used as a property value, only `String` objects and the eight primitive types (plus their wrappers) should be used. Other types cannot be passed to handlers that reside external from the Java VM.

Another reason that arbitrary classes should not be used is the mutability of objects. If the values are not immutable, then any handler that receives the event could change the value. Any handlers that received the event subsequently would see the altered value and not the value as it was when the event was sent.

The topic of the event is available as a property with the key EVENT_TOPIC. This allows filters to include the topic as a condition if necessary.

113.3.3   High Performance

An event processing system can become a bottleneck in large systems. One expensive aspect of the Event object is its properties and its immutability. This combination requires the Event object to create a copy of the properties for each object. There are many situations where the same properties are dispatched through Event Admin, the topic is then used to signal the information. Creating the copy of the properties can therefore take unnecessary CPU time and memory. However, the immutability of the Event object requires the properties to be immutable.

For this reason, this specification also provides an immutable Map with the Event Properties class. This class implements an immutable map that is recognized and trusted by the Event object to not mutate. Using an Event Properties object allows a client to create many different Event objects with different topics but sharing the same properties object.

The following example shows how an event poster can limit the copying of the properties.

```
void foo(EventAdmin eventAdmin) {
    Map<String,Object> props = new HashMap<String,Object>();
    props.put("foo", 1);
    EventProperties eventProps = new EventProperties( props);

    for ( int i=0; i<1000; i++)
        eventAdmin.postEvent( new Event( "my/topic/" + i, eventProps));
}
```

113.4   Event Handler

Event handlers must be registered as services with the OSGi framework under the object class org.osgi.service.event.EventHandler.

Event handlers should be registered with a property (constant from the EventConstants class) EVENT_TOPIC. The value being a String, String[] or Collection<String> object that describes which *topics* the handler is interested in. A wildcard asterisk ('*' \u002A) may be used as the last token of a topic name, for example com/action/*. This matches any topic that shares the same first tokens. For example, com/action/* matches com/action/listen.

Event Handlers which have not specified the EVENT_TOPIC service property must not receive events.

The value of each entry in the EVENT_TOPIC service registration property must conform to the following grammar:

```
topic-scope ::= '*' | ( topic '/*'?  )
```

Event handlers can also be registered with a service property named EVENT_FILTER. The value of this property must be a string containing a Framework filter specification. Any of the event's properties can be used in the filter expression.

```
event-filter ::= filter           //  See Filter Syntax in Core
```

Each Event Handler is notified for any event which belongs to the topics the handler has expressed an interest in. If the handler has defined a EVENT_FILTER service property then the event properties must also match the filter expression. If the filter is an error, then the Event Admin service should log a warning and further ignore the Event Handler.

For example, a bundle wants to see all Log Service events with a level of WARNING or ERROR, but it must ignore the INFO and DEBUG events. Additionally, the only events of interest are when the bundle symbolic name starts with com.acme.

```
public AcmeWatchDog implements BundleActivator,
        EventHandler {
    final static String [] topics = new String[] {
        "org/osgi/service/log/LogEntry/LOG_WARNING",
        "org/osgi/service/log/LogEntry/LOG_ERROR" };

    public void start(BundleContext context) {
        Dictionary d = new Hashtable();
        d.put(EventConstants.EVENT_TOPIC, topics );
        d.put(EventConstants.EVENT_FILTER,
            "(bundle.symbolicName=com.acme.*)" );
        context.registerService( EventHandler.class.getName(),
            this, d );
    }
    public void stop( BundleContext context) {}

    public void handleEvent(Event event ) {
        //...
    }
}
```

If there are multiple Event Admin services registered with the Framework then all Event Admin services must send their published events to all registered Event Handlers.

113.4.1   Ordering

In the default case, an Event Handler will receive posted (asynchronous) events from a single thread in the same order as they were posted. Maintaining this ordering guarantee requires the Event Admin to serialize the delivery of events instead of, for example, delivering the events on different worker threads. There are many scenarios where this ordering is not really required. For this reason, an Event Handler can signal to the Event Admin that events can be delivered out of order. This is notified with the EVENT_DELIVERY service property. This service property can be used in the following way:

- Not set or set to both - The Event Admin must deliver the events in the proper order.
- DELIVERY_ASYNC_ORDERED - Events must be delivered in order.
- DELIVERY_ASYNC_UNORDERED - Allow the events to be delivered in any order.

113.5   Event Publisher

To fire an event, the event source must retrieve the Event Admin service from the OSGi service registry. Then it creates the event object and calls one of the Event Admin service's methods to fire the event either synchronously or asynchronously.

The following example is a class that publishes a time event every 60 seconds.

```
public class TimerEvent extends Thread
```

```
implements BundleActivator {
Hashtable      time = new Hashtable();
ServiceTracker tracker;

public TimerEvent() { super("TimerEvent"); }

public void start(BundleContext context ) {
    tracker = new ServiceTracker(context,
        EventAdmin.class.getName(), null );
    tracker.open();
    start();
}

public void stop( BundleContext context ) {
    interrupt();
    tracker.close();
}

public void run() {
    while ( ! Thread.interrupted() ) try {
        Calendar c = Calendar.getInstance();
        set(c,Calendar.MINUTE,"minutes");
        set(c,Calendar.HOUR,"hours");
        set(c,Calendar.DAY_OF_MONTH,"day");
        set(c,Calendar.MONTH,"month");
        set(c,Calendar.YEAR,"year");

        EventAdmin ea =
            (EventAdmin) tracker.getService();
        if ( ea != null )
            ea.sendEvent(new Event("com/acme/timer",
                time ));
        Thread.sleep(60000-c.get(Calendar.SECOND)*1000);
    } catch( InterruptedException e ) {
        return;
    }
}

void set( Calendar c, int field, String key ) {
    time.put( key, new Integer(c.get(field)) );
}
}
```

113.6   Specific Events

113.6.1   General Conventions

Some handlers are more interested in the contents of an event rather than what actually happened. For example, a handler wants to be notified whenever an Exception is thrown anywhere in the system. Both Framework Events and Log Entry events may contain an exception that would be of interest to this hypothetical handler. If both Framework Events and Log Entries use the same property names then the handler can access the Exception in exactly the same way. If some future event type follows the same conventions then the handler can receive and process the new event type even though it had no knowledge of it when it was compiled.

The following properties are suggested as conventions. When new event types are defined they should use these names with the corresponding types and values where appropriate. These values should be set only if they are not null

A list of these property names can be found in the following table.

Table 113.1   *General property names for events*

Name	Type	Notes
BUNDLE_SIGNER	String | Collection <String>	A bundle's signers DN
BUNDLE_VERSION	Version	A bundle's version
BUNDLE_SYMBOLICNAME	String	A bundle's symbolic name
EVENT	Object	The actual event object. Used when rebroadcasting an event that was sent via some other event mechanism
EXCEPTION	Throwable	An exception or error
EXCEPTION_MESSAGE	String	Must be equal to exception.getMessage().
EXCEPTION_CLASS	String	Must be equal to the name of the Exception class.
MESSAGE	String	A human-readable message that is usually not localized.
SERVICE	Service Reference	A Service Reference
SERVICE_ID	Long	A service's id
SERVICE_OBJECTCLASS	String[]	A service's objectClass
SERVICE_PID	String | Collection <String>	A service's persistent identity. A PID that is specified with a String[] must be coerced into a Collection<String>.
TIMESTAMP	Long	The time when the event occurred, as reported by System.currentTimeMillis()

The topic of an OSGi event is constructed by taking the fully qualified name of the event class, substituting a solidus ('/' \u002F) for every full stop, and appending a solidus followed by the name of the constant that defines the event type. For example, the topic of

BundleEvent. STARTED

Event becomes

org/osgi/framework/BundleEvent/STARTED

If a type code for the event is unknown then the event must be ignored.

113.6.2   OSGi Events

In order to present a consistent view of all the events occurring in the system, the existing Framework-level events are mapped to the Event Admin's publish-subscribe model. This allows event subscribers to treat framework events exactly the same as other events.

It is the responsibility of the Event Admin service implementation to map these Framework events to its queue.

The properties associated with the event depends on its class as outlined in the following sections.

113.6.3   Framework Event

Framework Events must be delivered asynchronously with a topic of:

```
org/osgi/framework/FrameworkEvent/<eventtype>
```

The following event types are supported:

```
STARTED
ERROR
PACKAGES_REFRESHED
STARTLEVEL_CHANGED
WARNING
INFO
```

Other events are ignored, no event will be send by the Event Admin. The following event properties must be set for a Framework Event.

•   event - (FrameworkEvent) The original event object.

If the FrameworkEvent getBundle method returns a non-null value, the following fields must be set:

•   bundle.id – (Long) The source's bundle id.
•   bundle.symbolicName - (String) The source bundle's symbolic name. Only set if the bundle's symbolic name is not null.
•   bundle.version - (Version) The version of the bundle, if set.
•   bundle.signer - (String|Collection<String>) The DNs of the signers.
•   bundle - (Bundle) The source bundle.

If the FrameworkEvent getThrowable method returns a non-null value:

•   exception.class - (String) The fully-qualified class name of the attached Exception.
•   exception.message -(String) The message of the attached exception. Only set if the Exception message is not null.
•   exception - (Throwable) The Exception returned by the getThrowable method.

113.6.4   Bundle Event

Framework Events must be delivered asynchronously with a topic of:

```
org/osgi/framework/BundleEvent/<event type>
```

The following event types are supported:

```
INSTALLED
STARTED
STOPPED
```

```
UPDATED
UNINSTALLED
RESOLVED
UNRESOLVED
```

Unknown events must be ignored.

The following event properties must be set for a Bundle Event. If listeners require synchronous delivery then they should register a Synchronous Bundle Listener with the Framework.

- event - (BundleEvent) The original event object.
- bundle.id - (Long) The source's bundle id.
- bundle.symbolicName - (String) The source bundle's symbolic name. Only set if the bundle's symbolic name is not null.
- bundle.version - (Version) The version of the bundle, if set.
- bundle.signer - (String|Collection<String>) The DNs of the signers.
- bundle - (Bundle) The source bundle.

113.6.5   Service Event

Service Events must be delivered asynchronously with the topic:

```
org/osgi/framework/ServiceEvent/<eventtype>
```

The following event types are supported:

```
REGISTERED
MODIFIED
UNREGISTERING
```

Unknown events must be ignored.

- event - (ServiceEvent) The original Service Event object.
- service - (ServiceReference) The result of the getServiceReference method
- service.id - (Long) The service's ID.
- service.pid - (String or Collection<String>) The service's persistent identity. Only set if not null. If the PID is specified as a String[] then it must be coerced into a Collection<String>.
- service.objectClass - (String[]) The service's object class.

113.6.6   Other Event Sources

Several OSGi service specifications define their own event model. It is the responsibility of these services to map their events to Event Admin events. Event Admin is seen as a core service that will be present in most devices. However, if there is no Event Admin service present, applications are not mandated to buffer events.

113.7   Event Admin Service

The Event Admin service must be registered as a service with the object class org.osgi.service.event.EventAdmin. Multiple Event Admin services can be registered. Publishers should publish their event on the Event Admin service with the highest value for the SERVICE_RANKING service property. This is the service selected by the getServiceReference method.

The Event Admin service is responsible for tracking the registered handlers, handling event notifications and providing at least one thread for asynchronous event delivery.

113.7.1 Synchronous Event Delivery

Synchronous event delivery is initiated by the sendEvent method. When this method is invoked, the Event Admin service determines which handlers must be notified of the event and then notifies each one in turn. The handlers can be notified in the caller's thread or in an event-delivery thread, depending on the implementation. In either case, all notifications must be completely handled before the sendEvent method returns to the caller.

Synchronous event delivery is significantly more expensive than asynchronous delivery. All things considered equal, the asynchronous delivery should be preferred over the synchronous delivery.

Callers of this method will need to be coded defensively and assume that synchronous event notifications could be handled in a separate thread. That entails that they must not be holding any monitors when they invoke the sendEvent method. Otherwise they significantly increase the likelihood of deadlocks because Java monitors are not reentrant from another thread by definition. Not holding monitors is good practice even when the event is dispatched in the same thread.

113.7.2 Asynchronous Event Delivery

Asynchronous event delivery is initiated by the postEvent method. When this method is invoked, the Event Admin service must determine which handlers are interested in the event. By collecting this list of handlers during the method invocation, the Event Admin service ensures that only handlers that were registered at the time the event was posted will receive the event notification. This is the same as described in *Delivering Events* of *OSGi Core Release 6*.

The Event Admin service can use more than one thread to deliver events. If it does then it must guarantee that each handler receives the events in the same order as the events were posted, unless this handler allows unordered deliver, see *Ordering* on page 257. This ensures that handlers see events in their expected order. For example, for some handlers it would be an error to see a destroyed event before the corresponding created event.

Before notifying each handler, the event delivery thread must ensure that the handler is still registered in the service registry. If it has been unregistered then the handler must not be notified.

113.7.3 Order of Event Delivery

Asynchronous events are delivered in the order in which they arrive in the event queue. Thus if two events are posted by the same thread then they will be delivered in the same order (though other events may come between them). However, if two or more events are posted by different threads then the order in which they arrive in the queue (and therefore the order in which they are delivered) will depend very much on subtle timing issues. The event delivery system cannot make any guarantees in this case. An Event Handler can indicate that the ordering is not relevant, allowing the Event Admin to more aggressively parallelize the event deliver, see *Ordering* on page 257.

Synchronous events are delivered as soon as they are sent. If two events are sent by the same thread, one after the other, then they must be guaranteed to be processed serially and in the same order. However, if two events are sent by different threads then no guarantees can be made. The events can be processed in parallel or serially, depending on whether or not the Event Admin service dispatches synchronous events in the caller's thread or in a separate thread.

Note that if the actions of a handler trigger a synchronous event, then the delivery of the first event will be paused and delivery of the second event will begin. Once delivery of the second event has completed, delivery of the first event will resume. Thus some handlers may observe the second event before they observe the first one.

113.8   Reliability

113.8.1   Exceptions in callbacks

If a handler throws an Exception during delivery of an event, it must be caught by the Event Admin service and handled in some implementation specific way. If a Log Service is available the exception should be logged. Once the exception has been caught and dealt with, the event delivery must continue with the next handlers to be notified, if any.

As the Log Service can also forward events through the Event Admin service there is a potential for a loop when an event is reported to the Log Service.

113.8.2   Dealing with Stalled Handlers

Event handlers should not spend too long in the handleEvent method. Doing so will prevent other handlers in the system from being notified. If a handler needs to do something that can take a while, it should do it in a different thread.

An event admin implementation can attempt to detect stalled or deadlocked handlers and deal with them appropriately. Exactly how it deals with this situation is left as implementation specific. One allowed implementation is to mark the current event delivery thread as invalid and spawn a new event delivery thread. Event delivery must resume with the next handler to be notified.

Implementations can choose to blacklist any handlers that they determine are misbehaving. Black-listed handlers must not be notified of any events. If a handler is blacklisted, the event admin should log a message that explains the reason for it.

113.9   Inter-operability with Native Applications

Implementations of the Event Admin service can support passing events to, and/or receiving events from native applications.

If the implementation supports native inter-operability, it must be able to pass the topic of the event and its properties to/from native code. Implementations must be able to support property values of the following types:

- String objects, including full Unicode support
- Integer, Long, Byte, Short, Float, Double, Boolean, Character objects
- Single-dimension arrays of the above types (including String)
- Single-dimension arrays of Java's eight primitive types (int, long, byte, short, float, double, boolean, char)

Implementations can support additional types. Property values of unsupported types must be silently discarded.

113.10   Security

113.10.1   Topic Permission

The TopicPermission class allows fine-grained control over which bundles may post events to a given topic and which bundles may receive those events.

The target parameter for the permission is the topic name. TopicPermission classes uses a wildcard matching algorithm similar to the BasicPermission class, except that solidi ('/' \u002F) are used as

separators instead of full stop characters. For example, a name of a/b/∗ implies a/b/c but not x/y/z or a/b.

There are two available actions: PUBLISH and SUBSCRIBE. These control a bundle's ability to either publish or receive events, respectively. Neither one implies the other.

113.10.2   Required Permissions

Bundles that need to register an event handler must be granted ServicePermission[org.osgi.service.event.EventHandler, REGISTER]. In addition, handlers require TopicPermission[<topic>, SUBSCRIBE] for each topic they want to be notified about.

Bundles that need to publish an event must be granted ServicePermission[org.osgi.service.event.EventAdmin, GET] so that they may retrieve the Event Admin service and use it. In addition, event sources require TopicPermission[<topic>, PUBLISH] for each topic they want to send events to.

Bundles that need to iterate the handlers registered with the system must be granted ServicePermission[org.osgi.service.event.EventHandler, GET] to retrieve the event handlers from the service registry.

Only a bundle that contains an Event Admin service implementation should be granted ServicePermission[org.osgi.service.event.EventAdmin, REGISTER] to register the event channel admin service.

113.10.3   Security Context During Event Callbacks

During an event notification, the Event Admin service's Protection Domain will be on the stack above the handler's Protection Domain. In the case of a synchronous event, the event publisher's protection domain can also be on the stack.

Therefore, if a handler needs to perform a secure operation using its own privileges, it must invoke the doPrivileged method to isolate its security context from that of its caller.

The event delivery mechanism must not wrap event notifications in a doPrivileged call.

113.11   org.osgi.service.event

Event Admin Package Version 1.3.

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

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

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

113.11.1   Summary

- Event - An event.
- EventAdmin - The Event Admin service.
- EventConstants - Defines standard names for EventHandler properties.
- EventHandler - Listener for Events.
- EventProperties - The properties for an Event.

- TopicPermission - A bundle's authority to publish or subscribe to event on a topic.

113.11.2 public class Event

An event. Event objects are delivered to EventHandler services which subscribe to the topic of the event.

Concurrency Immutable

113.11.2.1 public Event(String topic,Map<String,?> properties)

topic The topic of the event.

properties The event's properties (may be null). A property whose key is not of type String will be ignored. If the specified properties is an EventProperties object, then it will be directly used. Otherwise, a copy of the specified properties is made.

☐ Constructs an event.

Throws IllegalArgumentException– If topic is not a valid topic name.

Since 1.2

113.11.2.2 public Event(String topic,Dictionary<String,?> properties)

topic The topic of the event.

properties The event's properties (may be null). A property whose key is not of type String will be ignored. A copy of the specified properties is made.

☐ Constructs an event.

Throws IllegalArgumentException– If topic is not a valid topic name.

113.11.2.3 public final boolean containsProperty(String name)

name The name of the property.

☐ Indicate the presence of an event property. The event topic is present using the property name "event.topics".

Returns true if a property with the specified name is in the event. This property may have a null value. false otherwise.

Since 1.3

113.11.2.4 public boolean equals(Object object)

object The Event object to be compared.

☐ Compares this Event object to another object.

An event is considered to be **equal to** another event if the topic is equal and the properties are equal. The properties are compared using the java.util.Map.equals() rules which includes identity comparison for array values.

Returns true if object is a Event and is equal to this object; false otherwise.

113.11.2.5 public final Object getProperty(String name)

name The name of the property to retrieve.

☐ Retrieve the value of an event property. The event topic may be retrieved with the property name "event.topics".

Returns The value of the property, or null if not found.

113.11.2.6   **public final String[] getPropertyNames()**

☐ Returns a list of this event's property names. The list will include the event topic property name
"event.topics".

Returns A non-empty array with one element per property.

113.11.2.7   **public final String getTopic()**

☐ Returns the topic of this event.

Returns The topic of this event.

113.11.2.8   **public int hashCode()**

☐ Returns a hash code value for this object.

Returns An integer which is a hash code value for this object.

113.11.2.9   **public final boolean matches(Filter filter)**

filter The filter to test.

☐ Tests this event's properties against the given filter using a case sensitive match.

Returns true If this event's properties match the filter, false otherwise.

113.11.2.10   **public String toString()**

☐ Returns the string representation of this event.

Returns The string representation of this event.

113.11.3   public interface EventAdmin

The Event Admin service. Bundles wishing to publish events must obtain the Event Admin service
and call one of the event delivery methods.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

113.11.3.1   **public void postEvent(Event event)**

event The event to send to all listeners which subscribe to the topic of the event.

☐ Initiate asynchronous, ordered delivery of an event. This method returns to the caller before de-
livery of the event is completed. Events are delivered in the order that they are received by this
method.

Throws SecurityException– If the caller does not have TopicPermission[topic,PUBLISH] for the topic speci-
fied in the event.

113.11.3.2   **public void sendEvent(Event event)**

event The event to send to all listeners which subscribe to the topic of the event.

☐ Initiate synchronous delivery of an event. This method does not return to the caller until delivery of
the event is completed.

Throws SecurityException– If the caller does not have TopicPermission[topic,PUBLISH] for the topic speci-
fied in the event.

113.11.4   public interface EventConstants

Defines standard names for EventHandler properties.

Provider Type Consumers of this API must not implement this type

113.11.4.1   **public static final String BUNDLE = "bundle"**

The Bundle object of the bundle relevant to the event. The type of the value for this event property is Bundle.

Since 1.1

113.11.4.2   **public static final String BUNDLE_ID = "bundle.id"**

The Bundle id of the bundle relevant to the event. The type of the value for this event property is Long.

Since 1.1

113.11.4.3   **public static final String BUNDLE_SIGNER = "bundle.signer"**

The Distinguished Names of the signers of the bundle relevant to the event. The type of the value for this event property is String or Collection of String.

113.11.4.4   **public static final String BUNDLE_SYMBOLICNAME = "bundle.symbolicName"**

The Bundle Symbolic Name of the bundle relevant to the event. The type of the value for this event property is String.

113.11.4.5   **public static final String BUNDLE_VERSION = "bundle.version"**

The version of the bundle relevant to the event. The type of the value for this event property is Version.

Since 1.2

113.11.4.6   **public static final String DELIVERY_ASYNC_ORDERED = "async.ordered"**

Event Handler delivery quality value specifying the Event Handler requires asynchronously delivered events be delivered in order. Ordered delivery is the default for asynchronously delivered events.

This delivery quality value is mutually exclusive with DELIVERY_ASYNC_UNORDERED. However, if both this value and DELIVERY_ASYNC_UNORDERED are specified for an event handler, this value takes precedence.

See Also EVENT_DELIVERY

Since 1.3

113.11.4.7   **public static final String DELIVERY_ASYNC_UNORDERED = "async.unordered"**

Event Handler delivery quality value specifying the Event Handler does not require asynchronously delivered events be delivered in order. This may allow an Event Admin implementation to optimize asynchronous event delivery by relaxing ordering requirements.

This delivery quality value is mutually exclusive with DELIVERY_ASYNC_ORDERED. However, if both this value and DELIVERY_ASYNC_ORDERED are specified for an event handler, DELIVERY_ASYNC_ORDERED takes precedence.

See Also EVENT_DELIVERY

Since 1.3

113.11.4.8   **public static final String EVENT = "event"**

The forwarded event object. Used when rebroadcasting an event that was sent via some other event mechanism. The type of the value for this event property is Object.

113.11.4.9   **public static final String EVENT_DELIVERY = "event.delivery"**

Service Registration property specifying the delivery qualities requested by an Event Handler service.

Event handlers MAY be registered with this property. Each value of this property is a string specifying a delivery quality for the Event handler.

The value of this property must be of type String, String[], or Collection<String>.

See Also   DELIVERY_ASYNC_ORDERED, DELIVERY_ASYNC_UNORDERED

Since   1.3

113.11.4.10   **public static final String EVENT_FILTER = "event.filter"**

Service Registration property specifying a filter to further select Event s of interest to an Event Handler service.

Event handlers MAY be registered with this property. The value of this property is a string containing an LDAP-style filter specification. Any of the event's properties may be used in the filter expression. Each event handler is notified for any event which belongs to the topics in which the handler has expressed an interest. If the event handler is also registered with this service property, then the properties of the event must also match the filter for the event to be delivered to the event handler.

If the filter syntax is invalid, then the Event Handler must be ignored and a warning should be logged.

The value of this property must be of type String.

See Also   Event, Filter

113.11.4.11   **public static final String EVENT_TOPIC = "event.topics"**

Service registration property specifying the Event topics of interest to an Event Handler service.

Event handlers SHOULD be registered with this property. Each value of this property is a string that describe the topics in which the handler is interested. An asterisk ('∗') may be used as a trailing wildcard. Event Handlers which do not have a value for this property must not receive events. More precisely, the value of each string must conform to the following grammar:

```
topic-description := '*' | topic ( '/*' )?
topic := token ( '/' token )*
```

The value of this property must be of type String, String[], or Collection<String>.

See Also   Event

113.11.4.12   **public static final String EXCEPTION = "exception"**

An exception or error. The type of the value for this event property is Throwable.

113.11.4.13   **public static final String EXCEPTION_CLASS = "exception.class"**

The name of the exception type. Must be equal to the name of the class of the exception in the event property EXCEPTION. The type of the value for this event property is String.

Since   1.1

113.11.4.14   **public static final String EXCEPTION_MESSAGE = "exception.message"**

The exception message. Must be equal to the result of calling getMessage() on the exception in the event property EXCEPTION. The type of the value for this event property is String.

113.11.4.15   **public static final String EXECPTION_CLASS = "exception.class"**

This constant was released with an incorrectly spelled name. It has been replaced by EXCEPTION_CLASS

Deprecated   As of 1.1, replaced by EXCEPTION_CLASS

113.11.4.16   **public static final String MESSAGE = "message"**

A human-readable message that is usually not localized. The type of the value for this event property is String.

113.11.4.17   **public static final String SERVICE = "service"**

A service reference. The type of the value for this event property is ServiceReference.

113.11.4.18   **public static final String SERVICE_ID = "service.id"**

A service's id. The type of the value for this event property is Long.

113.11.4.19   **public static final String SERVICE_OBJECTCLASS = "service.objectClass"**

A service's objectClass. The type of the value for this event property is String[].

113.11.4.20   **public static final String SERVICE_PID = "service.pid"**

A service's persistent identity. The type of the value for this event property is String or Collection of String.

113.11.4.21   **public static final String TIMESTAMP = "timestamp"**

The time when the event occurred, as reported by System.currentTimeMillis(). The type of the value for this event property is Long.

113.11.5   public interface EventHandler

Listener for Events.

EventHandler objects are registered with the Framework service registry and are notified with an Event object when an event is sent or posted.

EventHandler objects can inspect the received Event object to determine its topic and properties.

EventHandler objects must be registered with a service property EventConstants.EVENT_TOPIC whose value is the list of topics in which the event handler is interested.

For example:

```
 String[] topics = new String[] {"com/isv/*"};
 Hashtable ht = new Hashtable();
 ht.put(EventConstants.EVENT_TOPIC, topics);
 context.registerService(EventHandler.class.getName(), this, ht);
```

Event Handler services can also be registered with an EventConstants.EVENT_FILTER service property to further filter the events. If the syntax of this filter is invalid, then the Event Handler must be ignored by the Event Admin service. The Event Admin service should log a warning.

Security Considerations. Bundles wishing to monitor Event objects will require ServicePermission[EventHandler,REGISTER] to register an EventHandler service. The bundle must also have TopicPermission[topic,SUBSCRIBE] for the topic specified in the event in order to receive the event.

See Also   Event

Concurrency   Thread-safe

113.11.5.1   **public void handleEvent(Event event)**

event   The event that occurred.

  □ Called by the EventAdmin service to notify the listener of an event.

113.11.6   **public class EventProperties
                       implements Map<String,Object>**

The properties for an Event. An event source can create an EventProperties object if it needs to reuse the same event properties for multiple events.

The keys are all of type String. The values are of type Object. The key "event.topics" is ignored as event topics can only be set when an Event is constructed.

Once constructed, an EventProperties object is unmodifiable. However, the values of the map used to construct an EventProperties object are still subject to modification as they are not deeply copied.

Since   1.3

Concurrency   Immutable

113.11.6.1   **public EventProperties(Map<String,?> properties)**

properties   The properties to use for this EventProperties object (may be null).

  □ Create an EventProperties from the specified properties.

The specified properties will be copied into this EventProperties. Properties whose key is not of type String will be ignored. A property with the key "event.topics" will be ignored.

113.11.6.2   **public void clear()**

  □ This method throws UnsupportedOperationException.

Throws   UnsupportedOperationException– if called.

113.11.6.3   **public boolean containsKey(Object name)**

name   The property name.

  □ Indicates if the specified property is present.

Returns   true If the property is present, false otherwise.

113.11.6.4   **public boolean containsValue(Object value)**

value   The property value.

  □ Indicates if the specified value is present.

Returns   true If the value is present, false otherwise.

113.11.6.5   **public Set<Map.Entry<String,Object>> entrySet()**

  □ Return the property entries.

Returns   A set containing the property name/value pairs.

113.11.6.6   **public boolean equals(Object object)**

object   The EventProperties object to be compared.

  □ Compares this EventProperties object to another object.

The properties are compared using the java.util.Map.equals() rules which includes identity comparison for array values.

Returns   true if object is a EventProperties and is equal to this object; false otherwise.

113.11.6.7   **public Object get(Object name)**

name   The name of the specified property.

☐   Return the value of the specified property.

Returns   The value of the specified property.

113.11.6.8   **public int hashCode()**

☐   Returns a hash code value for this object.

Returns   An integer which is a hash code value for this object.

113.11.6.9   **public boolean isEmpty()**

☐   Indicate if this properties is empty.

Returns   true If this properties is empty, false otherwise.

113.11.6.10   **public Set<String> keySet()**

☐   Return the names of the properties.

Returns   The names of the properties.

113.11.6.11   **public Object put(String key,Object value)**

☐   This method throws UnsupportedOperationException.

Throws   UnsupportedOperationException– if called.

113.11.6.12   **public void putAll(Map<? extends String,? extends Object> map)**

☐   This method throws UnsupportedOperationException.

Throws   UnsupportedOperationException– if called.

113.11.6.13   **public Object remove(Object key)**

☐   This method throws UnsupportedOperationException.

Throws   UnsupportedOperationException– if called.

113.11.6.14   **public int size()**

☐   Return the number of properties.

Returns   The number of properties.

113.11.6.15   **public String toString()**

☐   Returns the string representation of this object.

Returns   The string representation of this object.

113.11.6.16   **public Collection<Object> values()**

☐   Return the properties values.

Returns   The values of the properties.

113.11.7   **public final class TopicPermission
extends Permission**

A bundle's authority to publish or subscribe to event on a topic.

A topic is a slash-separated string that defines a topic.

For example:

    org/osgi/service/foo/FooEvent/ACTION

TopicPermission has two actions: publish and subscribe.

Concurrency Thread-safe

113.11.7.1   **public static final String PUBLISH = "publish"**

The action string publish.

113.11.7.2   **public static final String SUBSCRIBE = "subscribe"**

The action string subscribe.

113.11.7.3   **public TopicPermission(String name,String actions)**

name Topic name.

actions publish,subscribe (canonical order).

□ Defines the authority to publish and/or subscribe to a topic within the EventAdmin service.

The name is specified as a slash-separated string. Wildcards may be used. For example:

        org/osgi/service/fooFooEvent/ACTION
        com/isv/*
        *

A bundle that needs to publish events on a topic must have the appropriate TopicPermission for that topic; similarly, a bundle that needs to subscribe to events on a topic must have the appropriate TopicPermssion for that topic.

113.11.7.4   **public boolean equals(Object obj)**

obj The object to test for equality with this TopicPermission object.

□ Determines the equality of two TopicPermission objects. This method checks that specified TopicPermission has the same topic name and actions as this TopicPermission object.

Returns true if obj is a TopicPermission, and has the same topic name and actions as this TopicPermission object; false otherwise.

113.11.7.5   **public String getActions()**

□ Returns the canonical string representation of the TopicPermission actions.

Always returns present TopicPermission actions in the following order: publish,subscribe.

Returns Canonical string representation of the TopicPermission actions.

113.11.7.6   **public int hashCode()**

□ Returns the hash code value for this object.

Returns A hash code value for this object.

113.11.7.7   **public boolean implies(Permission p)**

p The target permission to interrogate.

□ Determines if the specified permission is implied by this object.

This method checks that the topic name of the target is implied by the topic name of this object. The list of TopicPermission actions must either match or allow for the list of the target object to imply the target TopicPermission action.

```
x/y/*,"publish" -> x/y/z,"publish" is true
*,"subscribe" -> x/y,"subscribe"   is true
*,"publish" -> x/y,"subscribe"     is false
x/y,"publish" -> x/y/z,"publish"   is false
```

Returns true if the specified TopicPermission action is implied by this object; false otherwise.

113.11.7.8   **public PermissionCollection newPermissionCollection()**

□ Returns a new PermissionCollection object suitable for storing TopicPermission objects.

Returns A new PermissionCollection object.

121 Blueprint Container Specification

Version 1.0

121.1 Introduction

One of the great promises of object oriented languages was the greater reuse it should enable. However, over time it turned out that reuse was still hard. One of the key reasons was *coupling*. Trying to reuse a few classes usually ended up in dragging in many more classes, that in their turn dragged in even more classes, *ad nauseum*.

One of the key innovations in the Java language to address this coupling issue were *interfaces*. Interfaces significantly could minimize coupling because they were void of any implementation details. Any class can use an interface, where that interface can be implemented by any other class. However, coupling was still necessary because objects need to be created, and for creating an object its concrete class is necessary.

One of the most successful insights in the software industry of late has been *inversion of control*, or more specific *dependency injection*. With dependency injection, an object is given the collaborators that it needs to work with. By not creating these dependencies itself, the object is not coupled to the concrete type of these implementations and their transitive implementation dependencies. However, these objects are not useful on their own, they can only function when an external party provides these objects with their collaborating objects.

An injection framework creates these objects, and also their concrete dependencies, and wires them together. Injection frameworks can significantly increase reuse and provide increased flexibility. For example, during testing it is possible to inject mocked up objects instead of the actual objects.

There exists a number of these injection frameworks in the market, for example [1] *Spring Framework*, [3] *Guice*, and [4] *Picocontainer*. These containers are configured with XML, Java annotations, or provide automatic configuration based on types.

Decoupling is one of the primary drivers for the OSGi specifications. The module layer provides many mechanisms to hide implementation details and explicitly defines any dependencies. The service layer provides a mechanism to collaborate with other bundles without caring about who that other bundle is. However, using the OSGi APIs to construct an application out of services and objects also implies coupling to these OSGi APIs.

This specification therefore defines a dependency injection framework, specifically for OSGi bundles, that understands the unique dynamic nature of services. It provides an OSGi bundle programming model with minimal implementation dependencies and virtually no accidental complexity in the Java code. Bundles in this programming model contain a number of XML definition resources which are used by the Blueprint Container to wire the application together and start it when the bundle is active.

This Blueprint Container specification is derived from the [2] *Spring Dynamic Modules* project.

121.1.1 Essentials

- *Dependency Injection Framework* - Provide an advanced dependency injection framework for bundles that can create and wire objects and services together into an application.
- *Inversion of Control* - (IOC) A pattern in which a framework/library provides the control over the component instances instead of the other way around. Dependency injection is a form of IOC.

- *Extender Model* - Enable the configuration of components inside a bundle based on configuration data provided by the bundle developer. The life cycle of these components is controlled by the extender based on the extended bundle's state.
- *Unencumbered* - Do not require any special bundle activator or other code to be written inside the bundle in order to have components instantiated and configured.
- *Services* - Enable the usage of OSGi services as injected dependencies.
- *Dependencies* - Allow components to depend on other components like services and beans as well as register as services, with the full breadth of the OSGi capabilities.
- *Dynamicity* - Minimize the complexity of using the dynamicity of services
- *Business Logic* - A focus on writing business logic in regular Java classes that are not required to implement certain framework APIs or contracts in order to integrate with a container.
- *Declarative* - This facilitates independent testing of components and reduces environment dependencies.
- *Familiarity* - Familiar to enterprise Java developers.
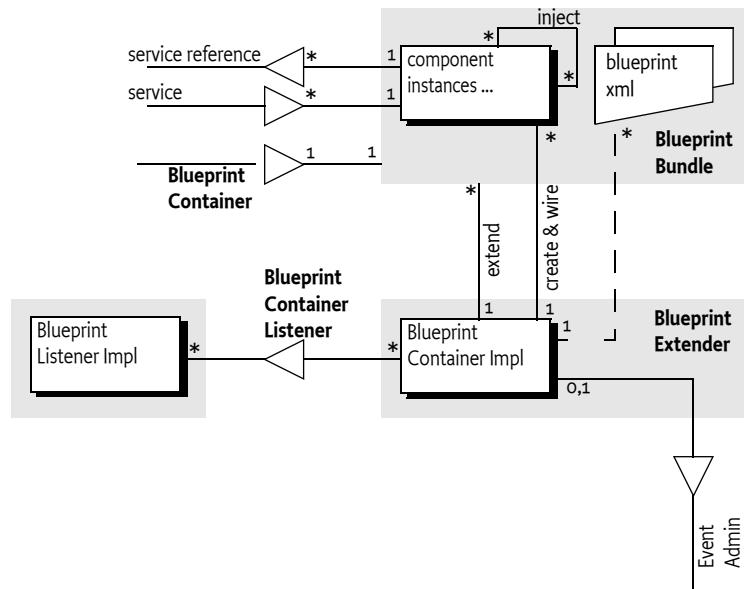
121.1.2   Entities

- *Blueprint Extender* - The bundle that creates and injects component instances for a Blueprint bundle as configured in that Blueprint bundle's XML definition resources.
- *Blueprint Container* - Represents the activities of the Blueprint Extender for a specific Blueprint Bundle.
- *Blueprint Bundle* - A bundle that is being constructed by the Blueprint Container because it has a Bundle-Blueprint header or it contains XML resources in the OSGI-INF/blueprint directory.
- *Manager* - A manager is responsible for the life cycle of all *component instances* for one *component definition.* There are the following types of managers. A manager is a *bean manager*, a *service reference manager*, or a *service manager*. A manager can have *explicit* and *implicit* dependencies on other manager. During instantiation and runtime, a manager can *provide* a component instance to be injected or used in other ways.
- *Component* - A loosely defined term for the application building blocks and their infrastructure. Components are instantiated into *component instances* by a *manager* that is configured with a *Component Metadata* subclass that is derived from a *Component Definition.*
- *Component Instance* - An object that is part of the application. Component Instances are created and managed by their component *manager.*
- *Component Definition* - Configuration data used by a manager to construct and manage component instances. This configuration data is represented in Metadata, an interface hierarchy starting with the *Metadata* interface.
- *Bean Manager* - A manager that has metadata for creating Java objects and injecting them with objects and component instances that come from other managers it implicitly depends on.
- *Service Manager* - A manager that handles the registration of a service object that is provided by a component instance.
- *Service Reference Manager* - The general name for the reference and reference-list managers.
- *Reference Manager* - A manager that handles the dependency on a single OSGi service.
- *Reference-list Manager* - A manager that handles the dependency on a list of OSGi services.
- *Environment Manager* - A manager that can provide information from the Bundle's environment. For example, the BlueprintContainer object is made available through an environment manager.
- *Target* - A manager type useful in a callback context. These are the ref (which is an indirection to), a reference, and a bean manager.
- *Property* - A conceptual instance variable of a component instance provided by a bean manager that is set on the component instance with a corresponding set<Name> method.
- *Argument* - Metadata for an argument in a constructor or method.

- *Type Converter* - A component instance defined, or referenced, in the type-converters section implementing the Converter interface.

Figure 121.1   *Blueprint Class and Service Overview*

121.1.3   Synopsis

The Blueprint Extender bundle waits for Blueprint bundles. These are bundles that contain Blueprint XML resources called the definitions. These XML resources can be found in a fixed location or pointed to from a manifest header. When a Blueprint extender bundle detects that a Blueprint bundle is ready, it creates a Blueprint Container to manage that Blueprint bundle.

The Blueprint Container then parses the definitions into metadata objects. All top-level elements in the definitions are ComponentMetadata objects and are registered in the Blueprint Container by their id.

For each of the ComponentMetadata objects, the Blueprint Container has a corresponding component manager. For example, a BeanMetadata object relates to a Bean Manager instance. There are the following types of managers:

- *Bean Managers* - Can provide general objects that are properly constructed and configured
- *Service Managers* - Can register services
- *Service Reference Managers* - Provide proxies to one or more services. There are two sub-types: reference-list and reference.
- *Environment Managers* - Holding environment values like the Blueprint Bundle object

After creation, all managers are not yet activated. A manager is activated on demand when it has to provide a component instance for the first time.

All service reference managers track services in the service registry in order to determine if they are satisfied or not. If not, the Blueprint Container can optionally start a *grace* period. During the grace period, the Blueprint Container waits for all mandatory service reference managers to become satisfied. If this does not happen during the grace period, the Blueprint Container must abort the initialization.

From now on, the Blueprint Container is ready to provide component instances. Whenever a manager is asked to provide a component instance for the first time, the manager is activated. This acti-

vation will first request all its dependencies to provide a component instance, activating these managers if not already activated, recursively.

However, the activation needs a trigger to start. There are two triggers.

- *Service Request* - All service managers must have a Service Factory registered with the OSGi service registry whenever that service manager is enabled, see *Enabled* on page 308.
- *Eager Managers* - To kick start the application in the bundle, the Blueprint Container must ask all eager managers to provide a component instance, thereby activating these managers, see *Eager Instantiation* on page 290.

Service references must actuate their reference listeners when they are activated.

Bean managers have a scope. This scope can be `singleton`, where the manager always provides the same object, or `prototype`, where the manager creates a new object for each request.

Service reference managers provide proxies to the actual service objects and fetch the service object lazily. They provide a constant reference that dampen the dynamics of the underlying service objects.

If the Blueprint Container has successfully activated the eager managers, it will register a Blueprint Container service.

When the Blueprint Container must be destroyed because: the Blueprint bundle has stopped, there is a failure, or the Blueprint extender is stopped, then the Blueprint Container service is unregistered and all managers are deactivated. This will unregister any services and disable listeners, which release the component instances. Then all component instances are destroyed in reverse dependency order. That is, a component instance is destroyed when no other component instances depend on it.

121.2 Managers

The key feature of the Blueprint Container specification is to let the application in the bundle be constructed in the proper order from objects that are not required to be aware of Blueprint, OSGi, or even each other. These objects are called *component instances*. The active entity that orchestrates the life cycle of the bundle application is the *Blueprint Container*. It is configured by XML resources in the Blueprint bundle. The Blueprint Container is responsible for construction and configuration of the component instances as well as the interaction with the service registry.

Inside the Blueprint Container, component instances are managed by a *manager*. A manager is configured with one Component Definition, for example a bean definition, and can then provide one or more component instances. Such a configured manager instance is also loosely called a *component*.

A manager can have additional behavior associated with it. This behavior is controlled by the manager's *type*. This specification defines a number of manager types: bean, service, environment, reference, and reference-list. These types are further defined in the next section.

These managers are conceptual, they are not visible in the API of this specification. That is, an implementation is free to implement the specification without these objects as long as the externally observable behavior is the same.

As an example, a trivial echo service:

```
<blueprint>
   <service id="echoService"
            interface="com.acme.Echo" ref="echo"/>
   <bean id="echo" class="com.acme.EchoImpl">
     <property name="message" value="Echo: "/>
   </bean>
```

```
</blueprint>

public interface Echo {
  public String echo(String m);
}
public class EchoImpl implements Echo {
  String message;
  public void setMessage(String m) {
    this.message= m;
  }
  public void echo(String s) { return message + s; }
}
```

The example defines two *top-level* managers: echoService and echo. The echoService manager is of type *service*, and the echo manager is of type *bean*. The service manager is responsible for registering an OSGi service, where the service object will be the component instance provided by the echo manager. The echo component instance gets a message injected.

As seen from the example, managers can use component instances from other managers to construct their component instances. The use of other managers creates an *implicit dependency*. Managers can also declare *explicit dependencies*. Dependencies are transitive, see *Manager Dependencies* on page 281 for more information. In the previous example, the echoService service manager depends on the echo manager, this is an implicit dependency.

Managers have their own life cycle. They are conceptually created after the Blueprint Container has decided to run the application, see *Blueprint Life-Cycle* on page 285. However, the intention of this specification is to allow the bundle application to lazily *activate*. That is, no application code is used until there is a trigger like a service request or a service manager has an explicit dependency. A manager must always be atomically activated before it provides its first component instance. During activation, listeners are actuated and notified, service objects are requested, etc. The details are described in the appropriate manager's type description.

Each manager type has an associated *component metadata* type. Component Metadata is used to configure a manager. XML definition resources in the bundle define the source for this Metadata. In the previous example, the service and bean XML element are translated to a ServiceMetadata and BeanMetadata object respectively.

The Blueprint Container maintains a registry of managers by their *id*. These are the managers that are called the *top-level* managers. Top level managers are managers defined as child elements of the top XML blueprint element or bean managers in the type-converters element. Their Metadata is registered under their id (or calculated id) in the Blueprint Container. All top level managers share a single namespace. That is, it is an error if the same id is used multiple times or attempts to override the built-in environment managers.
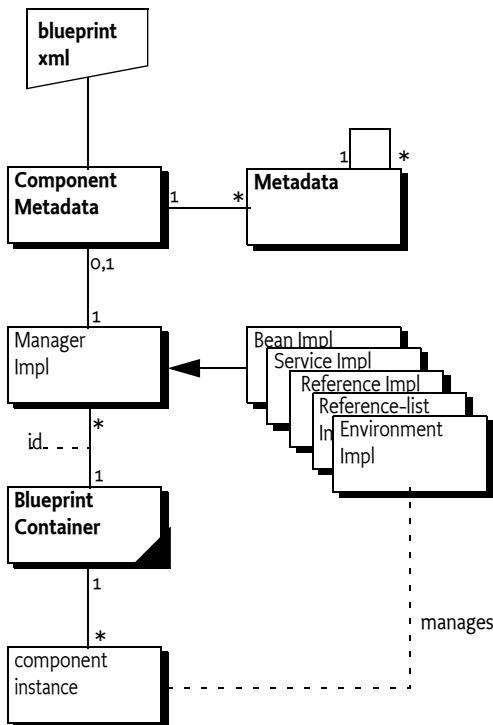
Top level managers can depend on other top level managers but there are many places where a manager can depend on an *inlined* manager. In these places, a complete manager can be defined inside another manager. Such inlined managers are always *anonymous*: they must not have an id and must not be registered as a top-level manager. Inlined beans are further constrained to always have prototype scope. That is, every time they are asked to provide a component instance, they must return a different object.

When the Blueprint Container must be *destroyed*, all singleton component instances that have been created must be destroyed. This must first *deactivate* all activated managers. All these managers must release their dependencies on any component instances they hold. Then the Blueprint Container must destroy all singleton component instances. The order of this destruction must be such that a component instance is only destroyed if there are no other component instances depending on it. See *Reverse Dependency Order* on page 282.

The relations between manager types, component instances, metadata and the Blueprint Container is schematically depicted in Figure 121.2 on page 280.

Figure 121.2   *Managers and Metadata*

121.2.1   Manager Types

Blueprint only supports a fixed set of the following *manager types*:

- *Bean* - A bean manager provides regular Java objects as component instances. It has the following features:
  - Construction via class name, static factory method, or a factory method on a *target*. A *target* is a reference to a top level manager of type bean or service reference, or a referral to a top level manager of those types.
  - Can have *arguments* for a constructor or factory method.
  - Can have *properties* that are injected.
  - Manages a singleton or creates objects on demand depending on its *scope*.
  - Life cycle callbacks for end of initialization and destruction.

  See *Bean Manager* on page 296 for more details.
- *Reference* - Reference managers track a service in the OSGi service registry. When activated, they provide a proxy to a service object. See *Service Reference Managers* on page 308 for more details. A reference is satisfied when its selection matches a service in the registry.
- *Reference-list* - Reference-list managers track multiple services. A reference-list is satisfied when its selection matches one or more services in the registry. See *Service Reference Managers* on page 308 for more details.
- *Service* - Service managers maintain the registration of an OSGi service object. Service managers provide a proxied ServiceRegistration object so that the application code has a constant reference, even if the service is unregistered and then registered again. A service manager is *enabled*

if all the mandatory service references in its dependencies are satisfied. See *Service Manager* on page 302.

- *Environment* - Environment managers provide access to the environment of the Blueprint bundle, for example its Bundle Context. See *Blueprint Container* on page 330 for more details.

121.2.2   Metadata Objects

`Metadata` objects hold the configuration information (from the Component Definition) for the managers. These metadata objects represent the element structure found in the XML definitions in canonical form. Each element in the XML has a corresponding Metadata sub-type that has a name that maps directly to the element. For example, the `bean` element represents the bean manager that has its configuration data defined in the `BeanMetadata` interface.

There are `Metadata` interfaces for all the manager types, except the environment type. Some dependency injections require the construction of arrays, maps, properties, simple objects, etc. For these type of objects, additional Metadata sub-interfaces are defined; these interfaces provide the information to construct the basic programming types. For example, the `CollectionMetadata` interface contains the information to construct an Array or Collection of a given type, where its member values are defined by other `Metadata` objects.

The set of Metadata types is fixed in this specification, just like the set of manager types. It is impossible to extend this set with user defined Metadata types. For more information about Metadata, see *Metadata* on page 335.

121.2.3   Activation and Deactivation

Managers are created after all the definitions are parsed. Some managers can already show some activity, for example service managers always activate explicit dependencies and register a Service Factory with the OSGi service registry. However, in this state a manager should attempt to not use any resources from the Blueprint bundle until it is activated itself.

A manager must be atomically activated when it has to provide its first component instance. During activation it can perform a manager specific initialization that will actually consume resources from the Blueprint bundle. This activation must be atomic. That is, if a manager is being activated then other threads must block until the activation is completed.

Deactivation only happens during the destruction of the Blueprint Container. During deactivation, a manager must release any dependencies on resources of the Blueprint bundle. No components instances are destroyed during deactivation because the singleton component instance destruction must happen after all managers are deactivated.

Each manager type has a dedicated section that describes what must happen during its activation and deactivation.

121.2.4   Manager Dependencies

Managers that refer to other managers depend on these managers transitively. For example, a service manager depends directly on the manager that provides the service object. In its turn, that service object could depend on any provided objects that were used to construct and inject this service object, and so on. This transitive set of dependencies are called *implicit dependencies* because these dependencies are implicitly created by the use of other managers in the Component Definitions.

Managers can also be configured with *explicit dependencies*. The XML definitions for all managers have a `depends-on` attribute with a whitespace delimited list of manager ids. Each of these `depends-on` managers must provide an object, that will be ignored. The timing of activation of dependencies depends on the specific managers but in general should happen before any observable behavior.

There is no ordering guarantee between independent sets of dependencies. The dependency graph is based on the managers, not the component instances. For example, the following definition:

```
<blueprint default-activation='eager'>
  <bean id='A'...>  <argument ref='B'> </bean>
  <bean id='B' depends-on='C E'...>
    <argument ref='C'>
  </bean>
  <bean id='C' scope='prototype' ...>
    <argument ref='D'>
  </bean>
  <bean id='D' .../>
  <bean id='E' ...> <argument ref='C'/> </bean>
  <bean id='F' depends-on='B' activation="lazy"/>
</blueprint>
```
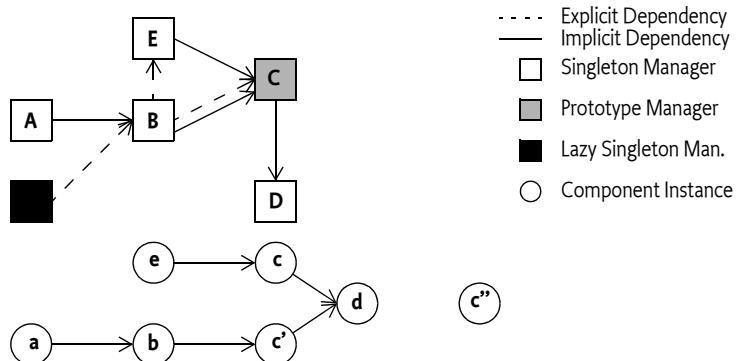
After initialization, there will be the following component instances: a, b, d, e, and three c's. Lower case names are used for instances, the corresponding upper case is its manager. The ordering guarantee is that manager D is activated before manager C, manager C is activated before manager E and B, manager E is activated before manager B, and manager B is activated before manager A. There will be no component instance f created because F is a lazy manager. There are three c's because manager E and B have an implicit dependency on C and manager B has an additional explicit dependency, totaling 3 dependencies. One of these c's is an orphan and will be garbage collected over time because it is not referred to by any component instance.

The example is depicted in Figure 121.3 on page 282.

Figure 121.3   *Dependency Graph after initialization*

121.2.5   Reverse Dependency Order

The destruction of component instances must be done in *reverse dependency order*. This concept is defined as only destroying a singleton component instance (in a manager specific way) when no other activated singleton component instance has an implicit or explicit dependency on it. That is, a component instance has no more field references to other component instances. A component that never was activated does not have any dependencies.

This strategy will ensure that a component instance cannot have an instance field that refers to an component instance that has been destroyed.

Deactivating the manager will release its dependencies, which then frees up other component instances until all component instances are destroyed, or there are cyclic references. In the case of cyclic dependencies, the order of destruction is undefined.

In the example depicted in Figure 121.3 on page 282, the previous rules imply that component instance a can be immediately destroyed first because it has no clients. After component instance a is destroyed, component instance b becomes free because no other component instances refer to it. The explicit dependency from manager F to manager B was never activated, so it is not taken in-

to account. The destruction of component instance b frees up component instance e and c because now the explicit dependency from manager B to manager E and manager B to manager C have been released. Manager C is deactivated but no component instances are destructed because it has prototype scope; these managers do not destroy their component instances. Then component instance d can be destructed.

121.2.6 Cyclic Dependencies

The implicit and explicit dependencies of a component form a dependency graph. In the ideal case, this graph should be free from *cycles*. A cycle occurs when a set of one or more managers find themselves in their own implicit or explicit dependencies. For example:

```
public class A { public A(B b); }
public class B { public void setA(A a); }

<bean id="a" class="A"> <argument ref="b"/> </bean>
<bean id="b" class="B"> <property name="a" ref="a"/> </bean>
```

In this example, the cycle is the set {a,b}. Managers can be part of multiple cycles.

When a member of a cycle is requested to provide a component instance, the Blueprint Container must break the cycle by finding one *breaking member* in the cycle's members. A breaking member must be a singleton bean and use property injection for the dependency that causes the cycle. The Blueprint Container can pick any suitable member of the cycle for breaking member, if no such member can be found, then initialization fails or the getComponentInstance method must throw a Component Definition Exception.

In the previous example, manager b can be a breaking member because it uses the property injection for the cyclic dependency on manager a. Manager a cannot be a breaking member because the cyclic dependency is caused by a constructor argument, a breaking member must use property injection for the cyclic dependency to be broken.

A breaking member must return a partially initialized component instance when it is asked to provide an object. A partially initialized object has done all possible initialization but has not yet been called with the initMethod (if specified) nor has it been injected any of the properties that causes a cycle. The *finalization* of the partially initialized component instance must be delayed until the breaking member has been injected in all referring members of the cycles. Finalization means injecting any remaining unset properties and calling of the initMethod, if specified.

The consequence of partially initialized component instances is that they can be used before they have all properties set, applications must be aware of this.

All partially initialized component instances must be finalized before the Blueprint Container enters the Runtime phase and before a call to the getComponentInstance method returns a component instance.

All detected cycles should be logged.

Consider the following example:

```
public class A {
 public A(B b) {}
}
public class B {
 public B(A a) {}
}
```

And the configuration:

```
<bean id="a" class="A"> <argument ref="b"/> </bean>
```

```
<bean id="b" class="B"> <argument ref="a"/> </bean>
```

In this case, the cycle cannot be broken because neither manager qualifies as breaking manager because they have a constructor/factory argument dependency. That is, it is impossible to construct an object without using the dependency. However, consider the following example:

```
public class A {
 public A(B b) {}
}
public class B {
 public B(C c) {}
}
public class C {
  public void setA(A a) {}
}
```

And the configuration:

```
<bean id="a" class="A"> <argument ref="b"/> </bean>
<bean id="b" class="B"> <argument ref="c"/> </bean>
<bean id="c" class="C" init-method="done">
    <property name="a" ref="a"/>
</bean>
```
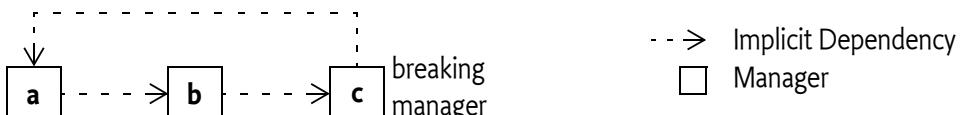
This configuration is depicted in Figure 121.4 on page 284. This cycle {a,b,c} can be broken by selecting manager c as the breaking member. If manager a is requested to provide a component instance for the first time, then the following sequence takes place:

```
activate a
  activate b
    activate c
      c = new C()
    b = new B(c)
  a = new A(b)
  c.seta(a)
  c.done()
return a
```

Figure 121.4   *Cyclic Dependency*

Cycles must be broken, if possible, both for singleton managers as well as prototype beans, although a breaking manager must always be a singleton bean because a prototype bean must always return a new object, making it impossible to break the cycle by returning a partially initialized component instance. That is, the following definition is not allowed to attempt to create an infinite loop:

```
<bean id="a" scope="singleton" class="A">
  <property name="a" ref="a">
</bean>
```

The previous definition must create an A object that refers to itself. However, if the example had used a prototype scope, it would be an unbreakable cycle.

121.2.7 Eager Managers

The Blueprint Container can force the activation of the application in the Blueprint bundle with *eager* managers. An eager manager is a manager that has the activation set to eager. A bean manager can only be eager if it has singleton scope.

Eager managers are explicitly activated by asking them to provide a component instance after all other initialization is done. A bundle that wants to be lazily initialized should not define any eager managers.

121.3 Blueprint Life-Cycle

A bundle is a *Blueprint bundle* if it contains one or more blueprint XML *definition* resources in the OSGI-INF/blueprint directory or it contains the Bundle-Blueprint manifest header referring to existing resources.

A *Blueprint extender* is an implementation of this specification and must track blueprint bundles that are *type compatible* for the Blueprint packages and initialize them appropriately. The timing and ordering of the initialization process is detailed in the following section.

There should be only one Blueprint extender present in an OSGi framework because this specification does not specify a way to resolve the conflicts that arise when two Blueprint extenders extend the same Blueprint bundle.

121.3.1 Class Space Compatibility

A Blueprint extender must not manage a Blueprint bundle if there is a class space incompatibility for the org.osgi.service.blueprint packages. For example, if the Blueprint bundle uses the Blueprint-Container class, then it must import the org.osgi.service.blueprint.container package. The Blueprint extender and the Blueprint bundle must then share the same class space for this package. Type compatibility can be verified by loading a class from the blueprint packages via the Blueprint extender bundle and the Blueprint bundle's loadClass methods. If the Blueprint bundle cannot load the class or the class is identical to the class loaded from the extender, then the two bundles are compatible for the given package. If the Blueprint extender is not class space compatible with the Blueprint bundle, then Blueprint extender must not start to manage the Blueprint bundle.

121.3.2 Initialization of a Blueprint Container

A Blueprint extender manages the application life cycle of Blueprint bundles based on:

- The Blueprint bundle state,
- The Blueprint definitions,
- The Blueprint extender's bundle state
- The class space compatibility

All activities on behalf of the Blueprint bundle must use the Bundle Context of the Blueprint bundle. All dynamic class loads must use the Blueprint bundle's Bundle loadClass method.

The following sections describe a linear process that handles one Blueprint bundle as if it was managed by a special thread, that is, waits are specified if the thread waits. Implementations are likely to use a state machine instead for each managed Blueprint bundle, the linear description is only used for simplicity.

In the following description of the initialization steps, the Blueprint Container will update its state. State changes are broadcast as events, see *Events* on page 331.

If any failure occurs during initialization, or the Blueprint bundle or Blueprint extender bundle is stopped, the Blueprint Container must be destroyed, see *Failure* on page 286. These checks are not indicated in the normal flow for clarity.

121.3.2.1 Initialization Steps

The initialization process of a Blueprint Container is defined in the following steps:

1. Wait until a blueprint bundle is *ready*. A blueprint bundle is ready when it is in the ACTIVE state, and for blueprint bundles that have a lazy activation policy, also in the STARTING state.
2. Prepare, verify if this Blueprint bundle must be managed, see *Preparing* on page 288.
3. State = CREATING
4. Parse the XML definition resources.
5. Service reference managers must start tracking their satisfiability without actually activating. See *Tracking References* on page 289.
6. If all mandatory service references are satisfied, or the blueprint.graceperiod is false, then go to step 9.
7. State = GRACE_PERIOD
8. Perform the *grace period*. This period waits until all mandatory service references are satisfied. See *Grace Period* on page 289. This step fails if the mandatory dependencies are not satisfied at the end of the grace period.
9. The Blueprint Container is now ready to provide component instances.
10. Service managers must initialize their explicit dependencies and have a Service Factory registered during the periods that they are enabled. See *Service Registration* on page 289.
11. Ask all eager managers to provide a component instance. See *Eager Instantiation* on page 290.
12. State = CREATED
13. Register the Blueprint Container
14. The components are now active and perform their function until the Blueprint bundle or the Blueprint extender bundle are stopped.
15. State = DESTROYING
16. Perform the Destroy phase, see *Destroy the Blueprint Container* on page 290.
17. State = DESTROYED

121.3.2.2 Failure

If at any time there is a failure, the Blueprint Container must:
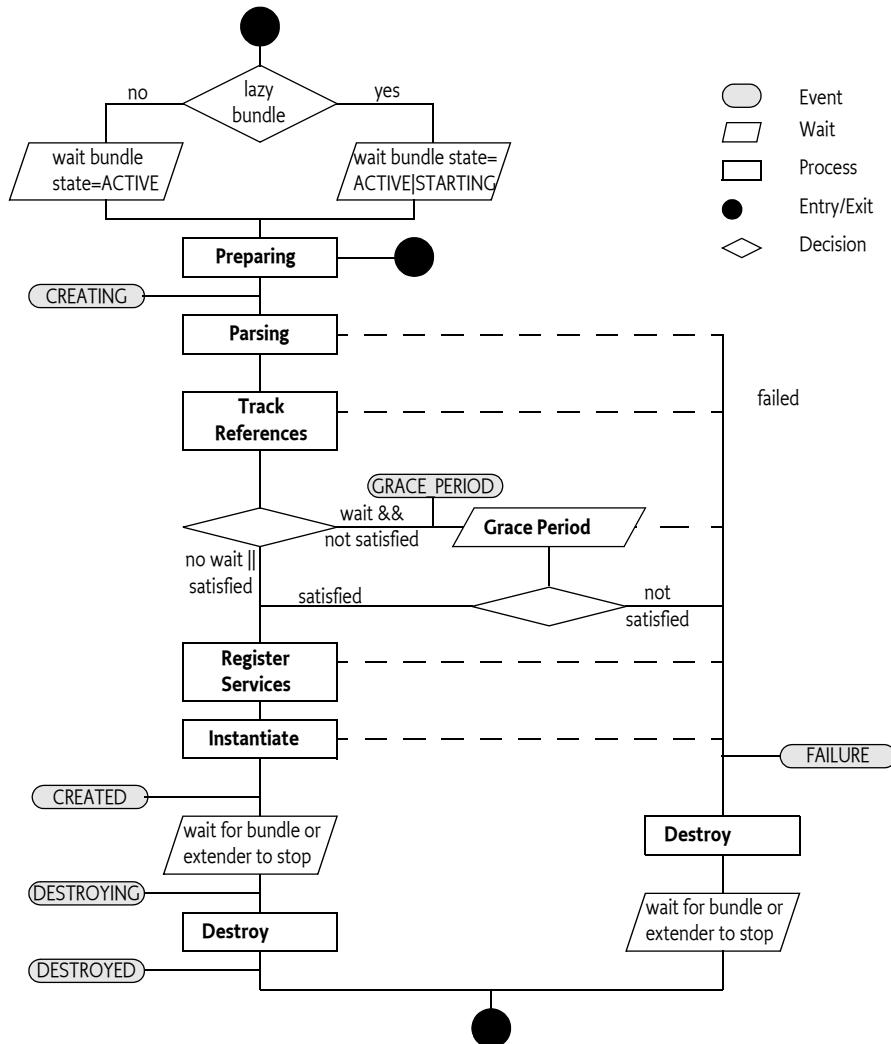
1. State = FAILURE
2. Unregister the Blueprint Container service.
3. Destroy the Blueprint Container.
4. Wait for the Blueprint bundle to be stopped.

121.3.2.3 Diagram

This initialization process is depicted in Figure 121.5 on page 287.

Figure 121.5   *Blueprint Bundle Initialization*

121.3.3   Extensions

A compliant implementation of this specification must follow the rules as outlined. However, implementations can provide functional extensions by including attributes or elements of other namespaces. For example, a Blueprint extender implementation that supports proxying of certain classes and a number of additional type converters could include a http://www.acme.com/extensions namespace that adds an extensions attribute on the blueprint element:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<blueprint
 xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
 xmlns:ext="http://www.acme.com/extensions"

  ext:extensions="proxyClasses"
>
  ...
</blueprint>
```

Blueprint extenders that detect the use of an unrecognized namespace must fail to signal a portability problem.

121.3.4 Preparing

Blueprint definitions are stored as resources in the Blueprint bundle. If a Bundle-Blueprint manifest header is defined, then this header contains a list of paths. The Bundle-Blueprint header has the following syntax:

```
Bundle-Blueprint ::= header
                     // See Common Header Syntax in Core
```

This specification does not define any attributes or directives for this header. Implementations can provide proprietary parameters that should be registered with the OSGi Alliance to prevent name collisions. The non-localized version of the header must be used.

The last component of each path in the Bundle-Blueprint header may use wildcards so that Bundle.findEntries can be used to locate the XML document within the bundle and its fragments. The findEndtries method must always be used in the non-recursive mode. Valid paths in the header have one of the following forms:

- *absolute path* - The path to a resource in the fragment or directory, this resource must exist. For example cnf/start.xml.
- *directory* - The path to directory in a fragment or main bundle, the path must end in a solidus ('/' \u002F). The pattern used in the findEntries method must then be *.xml. The directory is allowed to be empty.
- *pattern* - The last component of the path specifies a filename with optional wildcards. The part before is the path of directory in the bundle or one of its fragments. These two parts specify the parameter to findEntries. It is allowed to have no matching resources. An example of a pattern is: cnf/*.xml.

If no resources can be found, then the Blueprint bundle will not be managed and the initialization exits.

For example, the following header will read the resources /lib/account.xml, /security.bp, and all resources which path ends in .xml in the /cnf directory:

```
Bundle-Blueprint: lib/account.xml, security.bp,cnf/*.xml
```

If the Bundle-Blueprint header is not defined, then its default value is:

```
OSGI-INF/blueprint/*.xml
```

A Bundle-Blueprint manifest header specified in a fragment is ignored by the Blueprint Container. However, XML documents referenced by a bundle's Bundle-Blueprint manifest header, or its default, may be contained in attached fragments, as defined by the findEntries method.

If the Bundle-Blueprint header is specified but empty, then the Blueprint bundle must not be managed. This can be used to temporarily disable a Blueprint bundle.

121.3.5 Parsing

The Blueprint Container must parse the XML definitions into the Blueprint Container's metadata registry. Parsing fails if:

- A path from the Bundle-Blueprint header cannot be found in the bundle or any of its fragments.
- An XML definition does not validate against its schema.
- The XML elements do not meet one or more of their constraints
- Any errors occur

For failure, see *Failure* on page 291.

121.3.6 Tracking References

Service reference managers must track the service registry to see if they are satisfied or not. These managers must not be activated to register these service listeners nor must they activate any dependencies until they are activated. That is, no component instances for the reference listeners are obtained until the service reference manager is activated.

121.3.7 Grace Period

A Blueprint Container by default will wait for its dependencies in the *grace period*. However, this can be overridden with a directive on the Bundle-SymbolicName header of the Blueprint bundle:

- `blueprint.graceperiod` (true|false) - If set to `true`, then the Blueprint Container must enter the grace period and wait for dependencies, this is the default. Otherwise, it must skip the grace period and progress to the next phase regardless if there are any unsatisfied service references.

The purpose of the grace period is to handle the initialization of multiple bundles *gracefully*. The grace period will first wait a configurable time for all mandatory service references to become satisfied, or for the bundle to stop. If these mandatory services are satisfied, then the grace period succeeds, otherwise it will fail. If the bundle is stopped during the grace period, then the Blueprint Container must be destroyed.

During the waiting period services can come and go. Each time such a service event takes place that involves any of the mandatory service references, the Blueprint Container must send out another `GRACE_PERIOD` event if that event does not result in ending the grace period. The event contains the complete filters of the unsatisfied service references, see *Blueprint Event* on page 332.

The wait time for the grace period is defined in a directive on the Bundle-SymbolicName header of the Blueprint bundle:

- `blueprint.timeout` (Integer >= 0) - The time to wait in the grace period for dependencies to become satisfied in milliseconds. The default is 300000, which is 5 minutes. If the `timeout` is 0, an indefinite wait will take place.

OSGi services are dynamic, therefore the grace period does not guarantee that all mandatory service references are still available. It only guarantees that at one moment in time they were available. A mandatory reference can become *unsatisfied* at any moment in time when a service is not available. See the *Service Dynamics* on page 328 for a description of how this is handled.

For example, the following header will make the bundle wait a maximum of 10 seconds for its mandatory service references to be satisfied. These dependencies must be satisfied, or a failure occurs.

```
Bundle-SymbolicName: com.acme.foo;
    blueprint.graceperiod:=true;
    blueprint.timeout:= 10000
```

121.3.8 Service Registration

A service manager must first activate all its explicit dependencies but it must not activate. It must then ensure that a Service Factory object is registered as a service when that service is *enabled*. Enabled means that all of the mandatory service references in its dependencies are satisfied.

Once the Service Factory is registered, any bundle can get the corresponding service object. Such a request must activate the service manager, if it is not already activated. Activation of a service manager must obtain a component instance from the Blueprint Container for the service object and any registration listeners. The registration listeners are then actuated and notified of the initial state.

121.3.9   Eager Instantiation

After all initialization is done, the Blueprint Container is ready. It is now possible to request component instances. If a bundle needs immediate startup because they cannot wait until they are triggered, then it should set the `activation` of its bean managers to `eager`. The Blueprint Container must request all eager managers to provide a component instance in this instantiation phase, see also *Lazy and Eager* on page 295.

121.3.10   Runtime Phase

The Blueprint Container must be registered as a service with the following service properties:

- `osgi.blueprint.container.symbolicname` - The bundle symbolic name of the Blueprint bundle
- `osgi.blueprint.container.version` - The version of the Blueprint bundle

The Blueprint Container service must only be available during the runtime phase when initialization has succeeded.

As long as the Blueprint extender and the Blueprint bundle are active, the application is in the runtime phase. The component instances perform their requested functionality in collaboration. The Blueprint Container can be used to provide objects from the defined managers, get information about the configuration, and general state information, see *Blueprint Container* on page 330.

121.3.11   Destroy the Blueprint Container

The Blueprint Container must be destroyed when any of the following conditions becomes true:

- The Blueprint bundle is stopped, that is, it is no longer ready.
- The Blueprint extender is stopped
- One of the initialization phases failed.

Destroying the Blueprint Container must occur synchronously with the Bundle STOPPING event if that caused any of the previous conditions. For example, if the Blueprint extender is stopped, it must synchronously destroy all Blueprint Containers it has created.

Destroying the Blueprint Container means:

1. Unregistering the Blueprint Container service
2. Deactivating all managers.
3. Destroying all component instances in reverse dependency order, see *Reverse Dependency Order* on page 282.

A Blueprint Container must continue to follow the destruction even when component instances throw exceptions or other problems occur. These errors should be logged.

If the Blueprint extender is stopped, then all its active Blueprint Containers must be destroyed in an orderly fashion, synchronously with the stopping of the Blueprint extender bundle. Blueprint Containers must use the following algorithm to destroy multiple Blueprint Containers:

1. Destroy Blueprint Containers that do not have any services registered that are in use by other bundles. More recently installed bundles must be destroyed before later installed bundles, that is, reverse bundle id order.
2. The previous step can have released services, therefore, repeat step 1 until no more Blueprint Containers can be destroyed.
3. If there are still Blueprint Containers that are not destroyed, then destroy the Blueprint Container with:
   - The registered service that is in use with the lowest ranking number, or if a tie
   - The highest registered service id

If there are still Bundle Containers to be destroyed, retry step 1

During the shutting down of an OSGi framework, it is likely that many bundles are stopped near si‐ multaneously. The Blueprint extender should be able to handle this case, without deadlock, when the stop of a Blueprint bundle overlaps with the stop of the Blueprint extender bundle.

121.3.12   Failure

If a failure occurs during the initialization of the Blueprint bundle, then first a FAILURE event must be posted, see *Events* on page 331. Then the Blueprint Container should be destroyed, ensuring that no uninitialized or half initialized objects are destroyed. Failures should be logged if a Log Ser‐ vice is present.

121.3.13   Lazy

The Blueprint Container specification specifically allows lazy initialization of the application in the Blueprint bundle. No component instances are created until an eager manager is activated, or a ser‐ vice request comes in.

If no eager managers are defined and no service has explicit dependencies, then no component in‐ stances are provided until an external trigger occurs. This trigger can be a service request or a call to the getComponentInstance method of the Blueprint Container, which is registered as a service. This allows a Blueprint bundle to not create component instances, and thereby load classes, until they are really needed. This can significantly reduce startup time.

Some features of the component definitions can only be verified by inspecting a class. This class loading can break the lazy initialization of a Blueprint bundle. It is therefore allowed to delay this kind of verification until the activation of a manager.

This lazy behavior is independent of the bundle's lazy activation policy. Though the Blueprint ex‐ tender recognizes this policy to detect when the bundle is ready (for a lazy activated bundle the STARTING state is like the ACTIVE state), it is further ignored. That is, the relation between a Bundle Activator that is lazily activated and the Blueprint Container is not defined.

121.4   Blueprint Definitions

The Blueprint XML resources in a bundle are the *definitions*. Each definition can include multiple namespaces. Implementations of the Blueprint core namespace must strictly follow this specifica‐ tion, if they add additional behavior they must add additional namespaces that are actually used in the definitions to signal the deviation from this specification.

The namespace for the core Blueprint definition resources is:

```
http://www.osgi.org/xmlns/blueprint/v1.0.0
```

Blueprint resources that use this core specification must have as top the blueprint element. The fol‐ lowing example shows the body of a Blueprint definition:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint
 xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  ...
</blueprint>
```

The recommended prefix for the Blueprint core namespace is bp.

All elements in the Blueprint namespace are prepared for future extensions and provide a descrip‐ tion child element in most positions.

121.4.1 XML

In the following sections, the XML is explained using the normal syntax notation used for headers. There is, however, one addition to the normal usage specific to XML, and that is the use of the angled brackets (‹ ›). A term enclosed in angled brackets, indicates the use of a real element. Without the angled brackets it is the definition of a term that is expanded later to a one or more other terms or elements. For example:

```
people    ::= ‹person› *
person    ::= ‹child›* address
address   ::= ‹fr› | ‹us› | ‹nl›
```

Describes for example the following XML:

```
‹people›
    ‹person id="mieke"›
        ‹child name="mischa"/›
        ‹child name="thomas"/›
        ‹fr zip="34160"/›
    ‹/person›
‹/people›
```

Attributes are described in tables that define how they map to their corresponding Metadata. As a rule, the XML elements and attributes are expressed directly in the Metadata.

The text in the following sections is a normative description of the semantics of the schema. However, the structure information is illustrative. For example, all description elements have been ignored for brevity. The exact structure is described by the XML schema, see *Blueprint XML Schema* on page 336.

There are a number of convenient XML types used in the following sections. There schema types are defined here:

- qname - A fully qualified Java class name in dotted form, for example java.lang.String.
- method - A valid Java method name, for example setFoo.
- NCName - A string syntax for names defined in [8] *XML Schema*.
- ID - A string syntax for ids defined in [8] *XML Schema*.
- type - A name of a Java type including arrays, see the next section *Syntax for Java types* on page 292.
- target - An inline bean, reference, or ref, see *Target* on page 295.
- object - An object value, see *Object Values* on page 315

In several cases, the actual syntax depends on the type conversion. This type of syntax is indicated with ‹‹type›› indicates that the syntax of the string depends on the type conversion, where ten type is usually given as a parameter on the same Metadata.

121.4.2 Syntax for Java types

A number of elements can refer to a Java type, for example the value element has a type attribute and a map element has a key-type attribute. The syntax for these types is as follows:

```
type    ::= qname array
array   ::= '[]' *
```

Where qname is the fully qualified name of a Java class or interface, or the name of a primitive type.

For example:

```
‹value type="java.lang.String[]"/›
```

It is not possible to specify generic information in this syntax.

121.4.3 XML and Metadata

The Blueprint Container parses the XML into Metadata objects, see *Metadata* on page 335. During parsing, the XML parser validates against the detailed Blueprint schema and will therefore catch many errors. However, the XML schema and the Metadata type are not equivalent. The XML contains many conveniences that the Blueprint Container must convert to the canonical type in the Metadata. A number of general rules apply for this conversion:

- An absent attribute will result in null, unless the schema element provides a default value. In that case, the default must be returned from the Metadata object. That is, a default is indistinguishable from a specifically set value.
- Defaults from the blueprint element are filled in the Metadata objects, they are not available in any other way.
- Strings are trimmed from extraneous whitespace, as described in XML normalization.
- Child elements are represented by List objects, in the order of their definition. If no child elements are specified, the list will be empty.

For example, the activation feature reflects the total of default-activation and activation attributes but does not reflect that a prototype scope always makes a bean lazy. That is, even if activation is eager, the bean must still have lazy activation when it has prototype scope.

121.4.4 <blueprint>
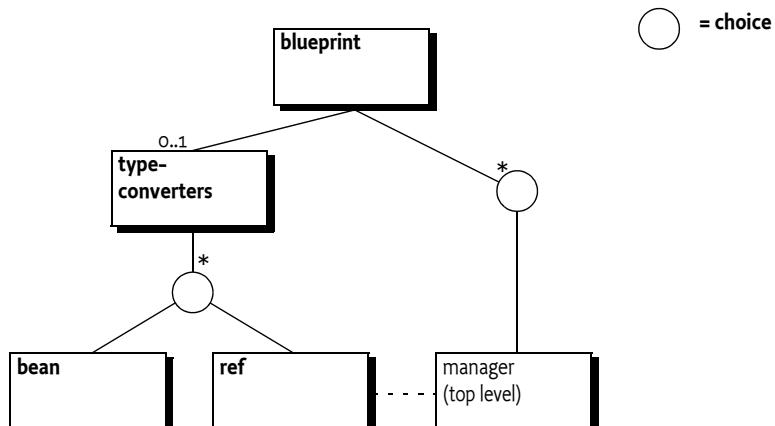
The blueprint element is the top element. The definitions consist of two sections: the type-converter section and the managers section.

```
blueprint          ::= <type-converters>manager*
manager            ::= <bean> | <service>
                         | service-reference
service-reference  ::= <reference> | <reference-list>
type-converters    ::= <bean> | <ref>
```

In this specification, the reference and reference-list managers are referred to as *service reference*s when their differences are irrelevant. The blueprint element structure is visualized in Figure 121.6.

Figure 121.6 *Managers (bold = element name, plain=base type)*

121.4.5 Metadata

The blueprint element has no corresponding Metadata class.

121.4.6 Defaults

The blueprint element supports the setting of the diverse defaults for the current definition resource with the following attributes:

- default-activation - Controls the default for the activation attribute on a manager. See *Lazy and Eager* on page 295. The default for this attribute is eager.
- default-availability - The default availability of the service reference elements, see *Service Reference Managers* on page 308. The default for this attribute is mandatory.
- default-timeout - The default for the reference element timeout attribute, see *Service Reference Managers* on page 308. The default for this attribute is 30000, or 5 minutes.

These defaults are specific for one definition resource, they apply only to elements enclosed to any depth in the blueprint element. These defaults are not visible in the Metadata.

121.4.7 ‹type-converters›

The Blueprint definitions are text based but the component instances require actual classes for their construction and dependency injection. Component instances are injected with general objects the target type is not always compatible with the source type. This specification therefore allows for *type conversion.* Type conversion rules are specified in *Type Conversion* on page 323. This section provides beans, or referrals to beans, that can be used in this type conversion process. They are listed in a separate section so they can be registered as a type converter, pre-instantiated, and preventing dependencies that easily become cyclic. Beans defined in the type-converters element must be registered as top-level managers.

The structure of the type-converters element is:

```
type-converters      ::= ( ‹bean› | ‹ref› )*
```

Type converters defined with the ref element can refer to bean managers or reference managers. Type converters must have ids distinct from any other manager and are available through the Blueprint Container's getComponentInstance method.

121.4.8 manager

The component XML schema type is the base type of the bean, service, reference-list, and reference elements. All manager sub-types share the following attributes:

- id - The manager and its Metadata are identified by its id as defined in its Component Definition. In general this id is therefore referred to as the *component id.* This is an optional attribute. If it is not defined, a default calculated unique id will be assigned to it for top-level managers. For in-lined managers, the id attribute cannot be set, their Metadata must return null. All top level manager ids must be unique in a Blueprint Container.

  The id attribute must be of type ID as defined in XML Schema, see [8] *XML Schema.* The syntax for an id is therefore:

  ```
  id  ::=    ID            // See ID in [8] XML Schema
  ```

  Ids generally use camel case, like myComponent, and they are case sensitive. That is, component id madHatter and madhatter are distinct ids. Applications should not use ids starting with the prefix blueprint.

  Ids are not required, if no component id is specified, the Blueprint Container must assign a unique id when it is a configured in a top level element. This calculated id must start with a full stop ('.' \u002E).
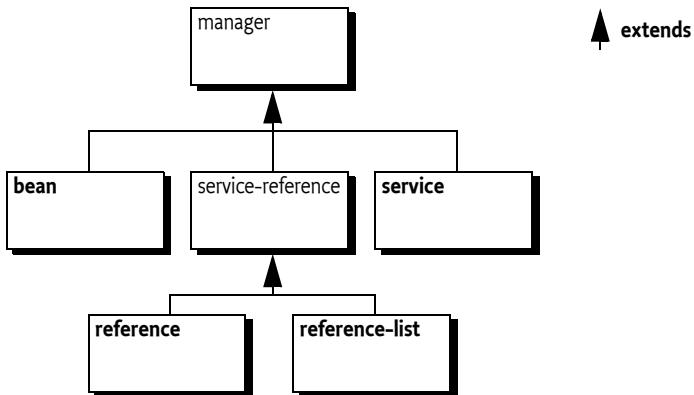
- activation - Defines the activation mode to be lazy or eager. See *Eager Instantiation* on page 290.

- dependsOn - The list of explicit dependencies that must be activated. See *Explicit Dependencies* on page 295.

The Metadata interface of top level managers will be a sub-interface of ComponentMetadata and is available from the Blueprint Container by its component id.

Figure 121.7   *Inheritance hierarchy for managers*

121.4.9   Explicit Dependencies

The dependsOn list contains the ids of the top-level managers the bean explicitly depends on. Unless stated otherwise in the specific manager description, explicit dependencies must be activated before their manager is activated.

For example:

```
<bean id="alice" class="com.acme.MadHatter"
      depends-on="cheshire rabbit queen"/>
```

This example will ask the top level managers cheshire, rabbit, and queen to provide an object before alice is activated. For a discussion about dependencies see *Manager Dependencies* on page 281.

121.4.10   Lazy and Eager

During initialization, all *eager* top level managers are requested to provide a component instance. Applications can use this request as an indication to start providing their intended functionality.

Managers that are *lazy*, that is, not singleton scope, activation is lazy, or inlined, are activated when they are first asked to provide a component instance. Therefore, even lazy managers can activate during initialization when they happen to be a dependency of another manager that activates its dependencies.

Services and service references can also have lazy or eager activation. The eager activation will ensure that all listeners are properly actuated during the corresponding activation. For services, the service object is then also requested at startup.

The following example defines an eager bean by making it a singleton and setting the activation to eager:

```
<bean id="eager" scope="singleton"
      class="com.acme.FooImpl" activation="eager"/>
```

121.4.11   Target

In several places in the Blueprint schema it is necessary to refer to a *target*. A target is a:

- ref - Must reference one of the following managers
- reference - An inlined reference manager
- bean - An inlined bean manager

The target type is normally used for listeners, service objects, and other places where a general application component instance is required.

121.5   Bean Manager

A bean manager provides an arbitrary Java object. It constructs this object from a given class or factory and then configures the object by injecting its *properties* with other component instances or more general object values.

The provided component instance can be a singleton or a new object can be returned on every invocation (prototype), this behavior is defined with the scope attribute, see *Scope* on page 299.

The provided object can optionally be notified when all of its properties have been injected, and when the providing bean manager will be deactivated, see *Life Cycle Callbacks* on page 301.
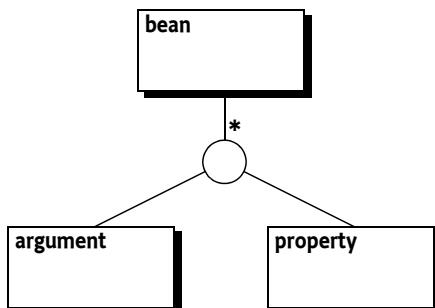
121.5.1   Bean Component XML

The structure of a bean element is:

```
bean    ::= ( <argument> | <property>)*
```

Figure 121.8   *Bean Structure*

121.5.2   <bean>

The Metadata for a bean manager is represented in the BeanMetadata interface, which extends ComponentMetadata. Table 121.1 provides an overview of the related XML definitions and the BeanMetadata interface. The table only provides a summary, the sometimes subtle interactions between the different features are discussed in later sections.

Table 121.1   *Bean Manager Features*

Attribute or Element	Syntax	Bean Metadata	Description
id	ID	id : String	The id of a top level manager, must be unique in the Blueprint Container. All inlined managers must return null for their id.

Attribute or Element	Syntax	Bean Metadata	Description
activation	lazy | eager	activation : int	Defines if this bean is lazily or eagerly activated. If not explicitly set, the blueprint element's value for the default-activation attributes is used. If this is also not set, the value is eager. See *Lazy and Eager* on page 295.
depends-on	NCName*	dependsOn : List<String>	Explicit list of ids that are the dependencies. These referred managers must be activated before this bean can provide an object. See *Explicit Dependencies* on page 295. This is a white-space separated list.
class	qname	className : String	Class name of the object to be provided or the class name for a static factory. See *Construction* on page 300.
scope	singleton | prototype	scope : String	The scope defines the construction strategy for the component instance. The default is singleton except for inlined bean managers, where it is prototype. There is no schema default, so if it is not explicitly set, the Metadata will be null. See *Scope* on page 299.
init-method	method	initMethod : String	The name of a method to invoke when a provided object has been injected with all its properties. If this is not set, it is null. See *Life Cycle Callbacks* on page 301.
destroy-method	method	destroyMethod : String	A name of a method to invoke on the provided objects with singleton scope when the Blueprint Container is destroyed. If this is not set, it is null. See *Life Cycle Callbacks* on page 301.
factory-method	method	factoryMethod : String	The name of the method on a static or component instance factory. See *Construction* on page 300.
factory-ref	NCName	factoryComponent : String	A reference to a manager that acts as the factory. See *Construction* on page 300.

Attribute or Element	Syntax	Bean Metadata	Description
<argument>	Table	arguments : List<BeanArgument>	Defined as sub-elements of the bean element. A BeanArgument object contains the value of an argument in the factory method or constructor. The order of the arguments is declaration order. See *Construction* on page 300.
<property>	Table	properties : List<BeanProperties>	Defined as sub-elements of the bean element. A BeanProperty object provides the property name and injection value. See *Properties* on page 300.

The bean element has the following constraints that are not enforced by the schema but must be enforced by the Blueprint Container:

- The destroyMethod must not be set when the scope is prototype.
- The activation must not be set to eager if the bean also has prototype scope.
- The following combinations of arguments are valid, all other combinations are invalid:
  - className
  - className, factory-method
  - factory-ref, factory-method

121.5.3   <argument>

The argument element holds a value for a constructor or factory method's parameters.

Table 121.2   *Bean Argument Features*

Attribute or Element	Syntax	Bean Argument	Description
index	int >= 0	index : int	The index of the argument in the constructor or factory-method signature. If this is not set, the Blueprint Container must use the type information to calculate it to match the disambiguation algorithm. The index will be -1 when not explicitly set.
type	qname	valueType : String	The fully qualified class name of a Java type to match the argument to the signature against.
ref	NCName	value : RefMetadata	A reference to a top level manager that provides the value for the argument.
value	<<type>>	value : ValueMetadata	The Value Metadata based on the value property.
<...>	object	value : Metadata	An inlined value.

The argument element has the following additional constraints:

- Either all arguments have a specified index or none have a specified index.
- If indexes are specified, they must be unique and run from 0..(n-1), where n is the number of arguments.
- The following attributes and elements are mutually exclusive:
  - ref
  - value
  - An inlined object value

121.5.4 <property>

The property element holds the information to inject a bean property with an object value.

Table 121.3   Bean Property Features

Attribute or Element	Syntax	Bean Property	Description
name	method ('.' method)*	name : String	The property name, for example foo. The method name can consist of full stop separated method names, indicating nested property access.
ref	NCName	value : RefMetadata	A reference to a top level manager.
value	<<type>>	value : ValueMetadata	A Value Metadata where the type is null.
<...>	object	value : Metadata	An inlined object value.

The argument element has the following additional constraints:

- The following attributes/elements are mutually exclusive
  - ref
  - value
  - An inlined object value

121.5.5 Scope

A bean manager has a recipe for the construction and injection of an object value. However, there can be different strategies in constructing its component instance, this strategy is reflected in the scope. The following scopes are architected for this specification:

- singleton - The bean manager only holds a single component instance. This object is created and set when the bean is activated. Subsequent requests must provide the same instance. Singleton is the default scope. It is usually used for core component instances as well as stateless services.
- prototype - The object is created and configured anew each time the bean is requested to provide a component instance, that is, every call to getComponentInstance must result in a new component instance. This is usually the only possible scope for stateful objects. All inlined beans are always prototype scope.

Implementations can provide additional scope types. However, these types must only be allowed when a defining namespace is included in the definitions and is actually used in the definitions to specify the dependency on this feature.

121.5.6   Construction

The Blueprint specification supports a number of ways for a bean manager to construct an object. Each possibility is a combination of the following Metadata properties:

- className - Defines the fully qualified name of a class to construct, or the name of a class with a static factory method. The class must be loadable from the Blueprint bundle loadClass method.
- factoryMethod - A static or instance factory method name that corresponds to a publicly accessible method on the given class or factory manager.
- factoryComponent - The id of a top-level target manager in the Blueprint Container that is an instance factory.

The Bean manager can have a number of BeanArgument objects that specify arguments for the constructor or for the factory class/object method. The matching constructor or method must be publicly accessible. The argument's valueType can be used to disambiguate between multiple signatures of constructors or methods. See *Signature Disambiguation* on page 321.

The value of the argument is always a Metadata object. Such an object can be converted into a general object value, see *Object Values* on page 315.

The construction properties can be used in a rather large number of combinations, however, not all combinations are valid. Table 121.4 shows the different valid combinations. If none of the combinations matches, then the Bean Metadata is erroneous.

In Table 121.4, a variation of the following bean definition is assumed:

```
<bean class="C" factory-method="f" factory-ref="fc">
    <argument value="1"/>
    <argument value="2"/>
</bean>
```

This definition is invalid because it specifies an invalid combination of metadata properties. The only valid combinations are subsets, they are all specified in the following table.

Table 121.4   Component Attributes and Construction

className	factory-method	factory-ref	argument	Corresponding Java Code
C				new C
C	f			C.f()
C			1,2	new C(1,2)
C	f		1,2	C.f(1,2)
	f	$fc		$fc.f()
	f	$fc	1,2	$fc.f(1,2)
*	*	*	*	failure

The object created this way will be the provided object of the bean after any properties are injected. If the factoryMethod returns a primitive type, then this primitive must be converted to the corresponding wrapper type before any usage.

121.5.7   Properties

Dependency injection configures a constructed object with the help of the properties, which is a List of BeanProperty objects. A Bean Property has the following features:

- name - The name of the bean property. This name refers to the set method on the constructed object as specified in the design pattern for beans getters and setters, see [5] *Java Beans Specification*. For example, if the property name is foo, then the public method setFoo(arg) will be used to set the value. There should only be one set method with a single argument for a specific property. If overloaded properties are encountered, the chosen set method is unspecified.

Nested property names are allowed when setting bean properties, as long as all parts of the path, except the property that is set, result in a `non-null` value. The parts of the path are separated with a full stop ('.' \u002E). For example:

```
<property name="foo.bar.baz" value="42"/>
```

This example gets the `foo` property, from the constructed object, it then gets the `bar` property and then sets the `baz` property on that object with the given value.

- `value` - The value of the property is always a Metadata object. This `Metadata` object can be converted to a value object, see *Object Values* on page 315.

After the Metadata object is converted to an object value, it must be injected into the property. If the value object is not directly assignable to the property type (as defined by its only set method and the rules in *Type Compatibility* on page 322), then the Blueprint Container must use the type conversion mechanism to create a new object that matches the desired type, or fail. See *Dependency Injection* on page 321 for more information about dependency injection.

For example, the following bean creates an instance and then injects a three into a the `foo` property that it gets from the `bar` property. The string that holds the three is converted to a `double`:

```
<bean id="foo" class="com.acme.Foo">
  <property name="bar.foo" value="3"/>
</bean>

// Classes
package com.acme;
public class Bar {
    double v;
    public void setFoo(double v) { this.v = v; }
}
public class Foo {
    Bar bar = new Bar();
    public void getBar() { return bar; }
}

// Corresponding Java code
Foo foo = new Foo();
foo.getBar().setFoo(3.0);
```

121.5.8 Life Cycle Callbacks

The `bean` element provides two attributes that define the *callback* method names for initialization and destruction. A callback must be implemented as a publicly accessible method without any arguments. The callback method names must exist as `void()` methods.

The `initMethod` specifies the name of an initialization method that is called after all properties have been injected. The `destroyMethod` specifies the name of a destroy method that is called when the Blueprint Container has destroyed a component instance. Only bean managers with `singleton` scope support the `destroyMethod`. The destroy callback cannot be used for beans that have `prototype` scope, the responsibility for destroying those instances lies with the application.

121.5.9 Activation and Deactivation

A singleton bean manager must construct its single object during activation and then callback its `initMethod` method. Prototype scoped beans are created after activation and also have their `initMethod` invoked. The destroy method is called during the destruction of all the beans in `singleton` scope, this happens after deactivation.

A prototype bean manager has no special activities for deactivation.

121.6   Service Manager

The service manager defined by a service element is responsible for registering a service object with the service registry. It must ensure that this service is only registered when it is *enabled*. Where enabled means that all its mandatory service reference managers in its dependencies are satisfied.

121.6.1   ‹service›

The XML structure of the ‹service› manager is:

```
service               ::= <interfaces>
                          <service-properties>
                          <registration-listener>*
                          target
interfaces            ::= <value>+
service-properties    ::= <entry>+
registration-listener ::= target
```

The service manager has the features outlined in Table 121.5 on page 302. The following additional constraints apply:

- The interface attribute and interfaces element are mutually exclusive.
- If the auto-export attribute is set to anything else but disabled, neither the interface attribute nor the interfaces element must be used.
- The ref attribute and inlined element are mutually exclusive

Table 121.5   *Service Manager Features*

Attribute or Element	Type	Service Metadata	Description
id	ID	id : String	Optional component id of the manager, if it is a top level manager.
activation	lazy | eager	activation : int	Defines if this service is lazily or eagerly initialized. If not explicitly set, the blueprint element's value for the default-activation attributes is used. If this is also not set, the value is eager. See also *Lazy and Eager* on page 295.
depends-on	NCName*	dependsOn : List‹String›	Explicit list of ids that are the dependencies. These managers must be activated at the start of the registration phase. See *Explicit Dependencies* on page 295. This is a whitespace separated list.
interface	qname	interfaces : List‹String›	Name of the interface under which this service should be registered. See *Service Interfaces* on page 304.

Attribute or Element	Type	Service Metadata	Description
auto-export	disabled | interfaces | class- hierarchy | all-classes	autoExport : int	Defines the way the class must be analyzed to find the interfaces under which the service must be registered. The schema default is disabled. See *Service Interfaces* on page 304
ranking	int	ranking : int	The service.ranking value. The schema default is 0, which implies no service property. See *Ranking* on page 306.
ref	NCName	value : RefMetadata	Reference to the manager that provides the service object. See *Service Object* on page 306.
‹service-properties›	See ‹*map*› on page 318.	serviceProperties : List‹MapEntry›	The service properties for this service. See *Service Properties* on page 305.
‹registration-listener›	See Table 121.6.	registrationListeners : List‹Registration Listener›	The registration listeners. See *Registration Listener* on page 306.

‹interfaces›	‹value›*	interfaces
		: List‹String›
			Names of interfaces under which this service should be registered. Each interface name must be listed as a child value element. This value element has no attributes. For example:
			
			‹interfaces›
			  ‹value›com.a.Foo‹/value›
			  ‹value›com.a.Bar‹/value›
			‹/interfaces›
			
			The value element must only hold a string value. See *Service Interfaces* on page 304

‹...›	target	value : Target	An inlined target manager that is used for the service object. See *Service Object* on page 306

121.6.2   ‹registration-listener›

The service element can contain zero or more registration-listener elements, that define registration listeners to be notified of service registration and unregistration events. This element has the following structure:

```
registration-listener   ::= target*
```

The registration-listener element defines the callback methods for registration and unregistration.

Table 121.6   *Registration Listener Features*

Attribute or Element	Type	Registration Listener	Description
ref	NCName	registrationListener : Target	A reference to a top level manager.

Attribute or Element	Type	Registration Listener	Description
registration-method	method	registrationMethod : String	The name of the method to call after the service has been registered. See *Registration Listener* on page 306.
unregistration-method	method	unregistrationMethod : String	The name of the method to call before the service will be unregistered. See *Registration Listener* on page 306.
<...>	target	registrationListener : Target	An inlined target manager

The additional constraint is:

- The ref attribute and the inlined manager are mutually exclusive.
- Either or both of the registrationMethod and unregistrationMethod must be set.
- For each method name set, there must be at least one method matching the possible prototypes in the registration listener object, see *Registration Listener* on page 306.

121.6.3 Explicit Dependencies

A service manager must initialize any explicit dependencies in the start of its registration phase, even before it tracks its enabled state. The presence of explicit dependencies will not activate the service manager.

121.6.4 Provided Object

A service manager provides a proxy to a ServiceRegistration object. If this proxy is used when the dependencies are not met, and the service is therefore unregistered, an Illegal State Exception must be thrown. In all other cases, the proxy acts as if it was the ServiceRegistration object associated with the registration of its service object.

The unregister method on the returned object must not be used. If the application code calls unregister then this must result in an Unsupported Operation Exception.

121.6.5 Service Interfaces

Each service object is registered under one or more interface names. The list of interface names is provided by interfaces or autoExport.

The autoExport tells the Blueprint Container to calculate the interface(s) from the type of the service object. The autoExport can have the following values:

- disabled - No auto-detection of service interface names is undertaken, the interface names must be found in interfaces. This is the default mode.
- interfaces - The service object will be registered using all of its implemented public Java interface types, including any interfaces implemented by super classes.
- class-hierarchy - The service object will be registered using its actual type and any public super-types up to the Object class (not included).
- all-classes - The service object will be registered using its actual type, all public super-types up to the Object class (not including), as well as all public interfaces implemented by the service object and any of its super classes.

The autoExport requires the actual class object for introspection for all its modes except disabled, which can cause a bundle with a lazy activation policy to activate because a class will be loaded from the Blueprint bundle.

As an example:

```
<bean id="fooImpl" class="FooImpl"/>
```

```
public class FooImpl implements Foo { ... }
```

Then the following service definitions are equivalent:

```
<service id="foo">
    <interfaces>
        <value>com.acme.Foo</value>
    </interface>
</service>
<service id="foo" interface="com.acme.Foo" ref="fooImpl"/>
<service id="foo" auto-export="interfaces" ref="fooImpl"/>
```

121.6.6   Service Properties

Each service can optionally be registered with *service properties*. The serviceProperties is a list of MapEntry, see *‹entry›* on page 319. This metadata must be used to create the service properties. Service properties creation can have side effects because they can use component instances. The service properties must therefore be created once before the first time the first time the service is registered.

The service manager adds the following automatic service properties that cannot be overridden. When these properties are explicitly set, they must be ignored.

- osgi.service.blueprint.compname - This will reflect the id of the manager that provides the service object, unless it is inlined. Inlined beans are always anonymous and must not have this property set.
- service.ranking - If the ranking attribute is not zero, this property will be set and hold an Integer object with the given value, see *Ranking* on page 306.

For example, the following definition is followed by equivalent Java code needed to register the service:

```
<service ref="fooImpl" interface="com.acme.Foo">
  <service-properties>
    <entry key="size" value="42"/>
  </service-properties>
</service>
```

```
Dictionary d = new Hashtable();
d.put("size", "42");
d.put("osgi.service.blueprint.compname", "fooImpl");
ServiceRegistration sr =
    bundleContext.registerService("com.acme.Foo",
        blueprintContainer.getComponentInstance("fooImpl"),
      d);
```

Service properties should specify the valueType of the entry unless the value to be registered needs to be a String object. The service property types should be one of:

- *Primitives Number* - int, long, float, double, byte, short, char, boolean
- *Scalar* - String, Integer, Long, Float, Double, Byte, Short, Character, Boolean.
- *Array* - An array of either the allowable primitive or scalar types.
- *Collection* - An object implementing the Collection interface that contains scalar types.

See ‹*entry*› on page 319 types for information how to create these types.

121.6.7   Service Object

The service manager must not request the Blueprint Container for the service object until it is actually needed because a bundle requests it. The service object is represented in the value. This is a Metadata object that can be used to construct an object value, see *Object Values* on page 315.

For example:

```
‹service id="fooService" ref="fooImpl".../›

‹service id="fooService" ... ›
   ‹bean class="com.acme.fooImpl"/›
‹/service›
```

The scope of the beans is ignored for the manager that provides the service object. Its value will only be created once the first time it is needed for the service.

121.6.8   Scope

A service manager must always register a Service Factory as service object and then dispatch the service requests to the service object. A service manager must obtain a single component instance as service object. This component instance is shared between all bundles. That is, even if the service object comes from a prototype scoped manager, only one instance is ever created per service manager.

If this component instance implements Service Factory, then all incoming service requests are forwarded to this single component instance.

121.6.9   Ranking

When registering a service with the service registry, an optional *service ranking* can be specified that orders service references. The service ranking is registered as the SERVICE_RANKING property defined in the OSGi service layer. When a bundle looks up a service in the service registry, given two or more matching services, then the one with the highest number will be returned. The default ranking value for the OSGi service registry is zero, therefore, this property must not be registered when ranking is zero, which is also the default value.

For example:

```
‹service ref="fooImpl" interface="com.acme.FooImpl"
         ranking="900" /›
```

This will result in the following service property:

```
service.ranking=new Integer(900)
```

121.6.10   Registration Listener

The registrationListeners represent the objects that need to be called back after the service has been registered and just before it will be unregistered.

The listenerComponent must be a Target object; it is the target for the following callbacks:

- registrationMethod - The name of the notification method that is called after this service has been registered.
- unregistrationMethod - This method is called when this service will be unregistered.

The signatures for the callback methods depend on the scope and if the service object implements the ServiceFactory interface. The different possibilities are outlined in the following table.

Table 121.7   *Interaction scopes and types for callback signature.*

Scope	Type	Signature	Comment
singleton	ServiceFactory	void(ServiceFactory,Map)	All service requests are handled by the component instance.
singleton	T	void(super T,Map)	T is assignable from the service object's type.
prototype	ServiceFactory	void(ServiceFactory,Map)	All service requests are handled by the first component instance.
prototype	T	void(,Map)	The first argument must be null because for prototype service objects, the component instance is created when a bundle requests the service. Therefore, at registration time there is no service object available.

If multiple signatures match, then all methods must be called in indeterminate order. At least one method must match.

The service manager must provide the registration listener with the current registration state when the listener is registered. This initial notification must take place before any other callback methods are called on this listener on other threads. That is, if the service is registered at that time, it must call the registration method and otherwise the unregistration method.

The following example shows two registration listeners, one with a referred bean and another one with an inlined bean.

```
<service ref="fooImpl" interface="com.acme.Foo">
   <registration-listener registration-method="reg"
           unregistration-method="unreg">
       <bean class="com.acme.FooListener"/>
   </registration-listener>
</service>

<service ref="fooImpl" interface="com.acme.Foo">
   <registration-listener registration-method="reg"
           unregistration-method="unreg" ref="fooListener"/>
</service>
<bean id="fooListener" class="com.acme.FooListener"/>

package com.acme;
public class FooListener {
  public void reg( Foo foo, Map properties ) { ... }
  public void unreg( Foo foo, Map properties ) { ... }
}
```

The manager that provides the registration listener object is an implicit dependency of the enclosing service manager. However, the registration listener component instance is specifically allowed to use to the service manager though this is technically a cyclic dependency. Therefore, a bean is allowed to be both be injected with a ServiceRegistration object from the service manager as well as being a registered listener to the same service manager.

In the following example, the foo service manager uses manager main, both as a registration listener as well as top-level bean main being injected with reference foo.

```
<service id="foo" interface="com.acme.Foo"ref="main">
  <registration-listener
        registration-method="register" ref="main"/>
</service>

<bean id="main" class="com.acme.Main" init-method="done">
    <property name="foo" ref="foo"/>
</bean>
```

121.6.11   Enabled

A service manager needs a service object that is referred to by the valueMetadata property. This value can in its turn depend on other managers transitively. If any of these managers are service reference managers, then they can be satisfied or not. If these service reference managers are marked to be mandatory, then they influence the *enabled* state of the first service manager. Only if all of these mandatory service reference managers in the dependency graph are satisfied, then the first service manager is enabled.

A service manager must have a Service Factory registered with the OSGi service registry after the primary initialization of the Blueprint Container has been done until the Blueprint Container is destroyed while it is enabled. See see *Service Registration* on page 289.

121.6.12   Activation and Deactivation

When a service manager is activated, it must actuate its registration listeners. Each registration listener must be called back during its actuation with the current service registration state as described in the *Registration Listener* on page 306. Normally, this will also request the container for a service object but this can be further delayed in certain circumstances. See *Service Object* on page 306 for more details.

During deactivation, a service manager must disable any registration listeners and release any dependencies it has on these component instances.

121.7   Service Reference Managers

The reference, and reference-list elements are all *service references*. They select a number of services in the service registry. The structure of these elements is as follows:
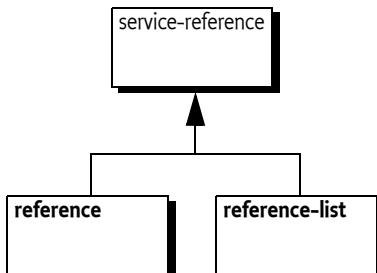
```
reference      ::= <reference-listener>*
reference-list ::= <reference-listener>*
```

The inheritance hierarchy for service references is depicted in Figure 121.9 on page 308.

Figure 121.9   *Inheritance hierarchy for service references*

121.7.1   Service Reference

The service reference managers have almost identical Metadata and share most behavior. The only schema differences between a reference manager and a reference-list manager are:

- `timeout` - A reference manager supports a timeout.
- `memberType` - The reference-list can define its member-type

The features of the service references are explained in the following table.

Table 121.8   *Service Reference Manager Features*

Attribute or Element	Type	ServiceReference-Meta-data	Description
id	ID	id : String	The component id of a top level manager
activation	lazy | eager	activation : int	Defines if this service reference is lazily of eagerly initialized. If not explicitly set, the blueprint element's value for the default-activation attributes is used. If this is also not set, the value is eager. See also *Lazy and Eager* on page 295.
depends-on	NCName*	dependsOn : List<String>	Explicit list of component ids that are the dependencies. These managers must be activated before this service reference's activation. See *Explicit Dependencies* on page 295. This is a whitespace separated List.
availability	mandatory | optional	availability : int	Defines if a service reference is mandatory or optional. The default for the availability attribute is defined by the default-availability attribute in the blueprint element. If the default-availability attribute is not defined, the value is mandatory.
interface	qname	interface : String	A single name of an interface class. It is allowed to not specify an interface name.
component-name	NCName	componentName : String	Points to another manager in another Blueprint Container registered in the service registry. If set, the component name must be part of the effective filter.
filter	filter	filter : String	The given filter string, can be null.

Attribute or Element	Type	ServiceReference-Meta-data	Description
<reference-listener>	See *<reference-listener>* on page 310.	referenceListeners : List<Listener>	The Metadata of the reference listeners

The additional constraints for service references are:

- The interface, if set, must refer to a public interface.

121.7.2 <reference>

A reference manager, selecting a single service, has the additional feature explained in the following table.

Table 121.9   *Reference Features*

Attribute or Element	Type	Reference Metadata	Description
timeout	long >= 0	timeout : long	The timeout in ms. Zero is indefinite.

An additional constraint on the reference is:

- The timeout must be equal or larger than zero.

121.7.3 <reference-list>

A reference-list manager, selecting multiple services, has the additional feature explained in the following table.

Table 121.10   *Reference-list Features*

Attribute or Element	Type	Reference List Metadata	Description
member-type	service-object | service-reference	memberType : int	Defines if the members of the list are ServiceReference objects or the proxies to the actual service objects.

121.7.4 <reference-listener>

The reference element can notify reference listeners of the service selection changes with the referenceListeners. The reference-listener element has the following structure:

```
reference-listener  ::= target*
```

The reference-listener element defines the callback methods for binding and unbinding a service.

Table 121.11   *Reference Listener Features*

Attribute or Element	Type	Reference Listener	Description
ref	NCName	listenerComponent : Target	A reference to a top level target manager.
bind-method	method	bindMethod : String	The name of the method to call after the service has been bound. See *Reference Listeners* on page 313.

Attribute or Element	Type	Reference Listener	Description
unbind-method	method	unbindMethod : String	The name of the method to call before the service will be un-bound. See *Reference Listeners* on page 313.
‹...›	target	listenerComponent : Target	An inlined target manager

The additional constraints are:
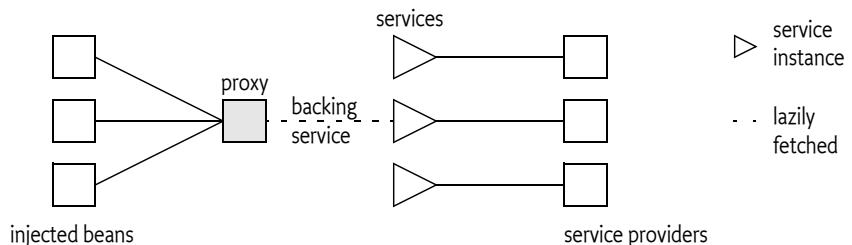
- The ref attribute and the inlined manager are mutually exclusive.
- Either or both bindMethod and unbindMethod must be specified.
- At least one specified method must exist with each given method name, see *Reference Listeners* on page 313.

121.7.5 Provided Object For a Reference

The provided object for a service reference manager is a *proxy* backed by a service object from the service registry. Therefore, even though the injected object will remain constant, it can change its reference to a backing service at any time, implying it can only be used with stateful services if reference listeners are used. If use when no suitable backing service is available, it will wait until it times out. See *Service Dynamics* on page 328 for more details. The model is depicted in Figure 121.10.

Figure 121.10   *Constant references with dynamic selection*

The following example shows how a property can be set to the service object.

```
public class C {
    public void setProxy(T ref) { ... }
}
<reference id="p" interface="T"/>
<bean id="c" class="C">
    <property name="proxy" ref="p"/>
</bean>
```
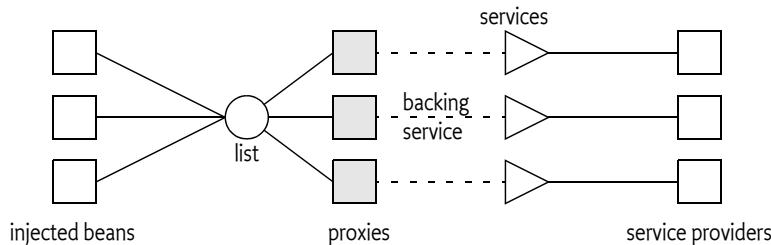
121.7.6 Provided Object For a Reference-list

The reference-list provided object implements the List interface; this List contains proxies to the backing services. These proxies do not have a timeout. That is, when a proxy from a reference-list is used, it must not wait when the backing service is no longer available but it must immediately throw a Service Unavailable Exception.

Changes to the list are dynamic. When a backing service is unregistered, the corresponding proxy is removed from the list synchronously with the service event. When a new service enters the selection, it is added synchronously with the service event. Proxies to newly discovered services must be added at the end of the list. The structure is depicted in Figure 121.11.

Figure 121.11   *Constant reference to list with dynamic selection*

The member type of the list depends on the `memberType`. If this is set to:

- `service-object` - Inject a List of service objects, this is the default.
- `service-reference` - Inject a list of `ServiceReference` objects

If generics information is available, then it is an error if the generic member type of the target list is not assignable with the `memberType`. If the member target type is in itself specified with generic arguments, like List<T<U>>, then the assignment must fail because this would require conversion and no conversion can take place for this assignment. For information about generics, see *Generics* on page 327.

121.7.7 Read Only Lists

The list is a read-only view on the actual set of proxies to the service objects. This `List` object must only support the following methods:

```
contains(Object)
containsAll(Collection)
equals(Object)
get(int)
hashCode()
indexOf(Object)
isEmpty()
iterator()           // no remove method
lastIndexOf(Object)
listIterator()       // not supported
listIterator(int)    // not supported
size()
subList(int, int)    // same list type as parent
toArray()
toArray(T[])
```

All other methods must throw an Unsupported Operation Exception. The List Iterator is not supported for these lists.

121.7.8 Selection

A service reference must provide a *selection* of services from the service registry. The Blueprint Container must logically use a filter for the selection that is the and (&) of the following assertions:

- The `interface`, if specified
- If `componentName` is not null, a filter that asserts `osgi.blueprint.compname=$componentName` This is a convenience function to easily refer to managers in other Blueprint Containers. Registered Blueprint services will automatically get this property set to their blueprint name.
- If `filter` is not null, the filter

The selection is defined as the set of Service References selected by the given filter.

121.7.9 Availability

A service reference is *satisfied* when one or more services match the selection. The availability is used to specify whether a service reference needs to be satisfied before initialization, see *Grace Period* on page 289, or if it controls the registration state of any service managers that depend on this service reference manager (explicit and implicit), see *Mandatory Dependencies* on page 330. The availability can have the following values:

- mandatory - Mandatory indicates that the service reference needs to be satisfied.
- optional - Optional indicates that the satisfaction of this reference is not relevant for any registered services, or for the grace period.

It is an error to declare a mandatory reference to a service that is registered by the same bundle. Such a definition could cause either deadlock or a timeout.

The fact that Blueprint specification has mandatory service references gives no guarantee that a valid service object is available when the service reference is used, in the dynamic world of OSGi, services can get unregistered at any time.

The following example declares a mandatory service reference for a single service. The usage of the reference can stall a maximum of 5 seconds if no service matches the selection.

```
<reference
    id          ="log"
    interface   ="org.osgi.service.log.LogService"
    availability="mandatory"
    timeout     ="5000" />
```

121.7.10 Reference Listeners

The referenceListeners are represented as ReferenceListener objects. They define the following callbacks:

- bindMethod - Called after a service is selected by the service reference manager. For a reference manager, this method can be called repeatedly without an intermediate unbind callback. This happens when a service is unregistered but a replacement can be found immediately.
- unbindMethod - Called when the service is no longer used by the service reference manager but before it has been returned to the service registry with the unget method. For a reference manager, no unbind method is called when the service can immediately be replaced with an alternative service when the service goes away.

A reference listener callback can have any of the following signatures:

- public void(ServiceReference) - Provide the ServiceReference object associated with this service reference. This callback type provides access to the service's properties without actually getting the service.
- public void(super T) - Provide the proxy to the service object, where T is on of the types implemented by the service object proxy.
- public void (super T,Map) - Provide the proxy to the service object. T is a type that is assignable from the service object. The Map object provides the service properties of the corresponding ServiceReference object.

All signatures must be supported regardless of the value of memberType that was specified in the reference-list. The service object given to the reference listeners must be the proxy to the service object.

The callbacks must be made synchronously with the corresponding OSGi service event. For reference-list callbacks, the service proxy is guaranteed to be available in the collection before a bind callback is invoked, and to remain in the collection until after an unbind callback has completed.

If a service listener defines multiple overloaded methods for a callback, then every method with a matching signature is invoked in an undefined order.

For example, the following definition will result in calling all the setLog methods on a FooImpl object:

```
<reference id="log"
        interface="org.osgi.service.log.LogService">
  <reference-listener
        bind-method="setLog">
        <bean class="com.acme.FooImpl"/>
    </reference-listener>
</reference>

public class FooImpl {
    public void setLog(Object o, Map m) { ... }
    public void setLog(LogService l, Map m) { ... }
    public void setLog(ServiceReference ref) { ... }
}
```

The manager that provides the reference listener object is treated as an implicit dependency of the enclosing service reference. This manager is specifically allowed to use to the service reference in a property injection or constructor argument, though this is technically a cyclic dependency. Therefore, a bean must be allowed to both be injected with a reference as well as listening to the bind and unbind callbacks of that same reference.

In the following example, the foo reference manager uses manager main, both as a reference listener as well as manager main being injected with reference foo.

```
<reference id="foo" interface="com.acme.Foo">
  <reference-listener bind-method="setL" ref="main"/>
</reference>
<bean id="main" class="com.acme.Main">
    <property name="r" ref="foo"/>
</bean>
```

121.7.11 Service Proxies

The Blueprint extender must generate proxies for the service reference managers. Reference managers provide proxies that dynamically select a *backing* service, which can change over time. A reference-list provides a list of proxies that have a fixed backing service, these proxies are added and removed from the list. Based on the selection, they do not have a time-out.

The backing service for a reference proxy must not be gotten from the OSGi service registry until an actual service object is needed, that is, when an actual method is called on the proxy. If the backing service becomes unregistered, then the proxy must unget the reference to the backing service (if it had gotten it) and get another service object the next time a method on the proxy is called. If a replacement can be found immediately, the reference listener's bind method must be called without calling the unbind method. Other threads that need the same service object must block until the service object has become available or times out.

The proxies must implement all the methods that are defined in the interface. The interface must refer to an interface, not a class. The proxy must only support the methods in the given interface. That is, it must not proxy methods available on the service object that are not available in the given

interface. If no interface is defined, the proxy must be implemented as if the interface had no methods defined.

Blueprint bundles must ensure that the proper semantics are maintained for `hashCode` and `equals` methods. If these methods are not defined in the interface, then the proxy must use the default semantics of the `Object` class for `equals` and `hashCode` methods.

121.7.12 Activation and Deactivation

Service reference managers are active before activation because they must handle the enable status of service managers.

During activation, a service reference must actuate its listeners and provide these listeners with the initial state of the reference. For a reference, if there is a selected object, the bind method must be called with the proxy object, otherwise the unbind method must be called with a `null` as proxy object. For a reference-list, the bind method must be called for each member of the list. If the list is empty, the unbind method must be called with a `null` as proxy object.

During deactivation, the listeners must be disabled.

121.8 Object Values

Top-level managers can use *object values* in different places. These object values are defined with XML elements and attributes. After parsing, they are all converted to sub-interfaces of the `Metadata` interface, transitively reachable from top-level managers. For example, the following definition creates a bean that is injected with the byte array: byte[] {7,42}:

```
<bean class="com.acme.FooImpl">
    <property name="array">
        <array value-type="byte">
            <value>7</value>
            <value>42</value>
        </array>
    </property>
</bean>
```

This definition provides the configuration data for an *array value*, which is represented by the CollectionMetadata interface. A Metadata object can be used to construct its object value during runtime whenever a new object must be constructed.

In most places where an object value can be used, it can be anything, including objects provided by a managers and even `null`. However, maps require non-null keys. The object values are therefore split in `value` and `nonNullValue` types.

The syntax for object values has the following structure:

```
nonNullValue ::= <ref>
               | <idref>
               | <value>
               | <map>
               | <props>
               | collection
               | manager // see manager on page 294
value        ::= nonNullValue | <null>
collection   ::= <list> | <set> | <array>
```

Object values also include inlined managers. The use of an inlined manager for an object value means that manager will provide a value every time the object value is constructed. Each of the ob-

ject values is created anew and the types are mutable, except for the service references. The use of managers in object values must create an implicit dependency between the top level managers and any transitively reachable manager from their Metadata.

121.8.1 <ref>

The ref element is a reference to a top-level manager in the same Blueprint Container. The ref element has a single attribute component-id.

Table 121.12   *Ref Features*

Attribute	Type	Ref Metadata	Description
component-id	NCName	componentId : String	A reference to a top level manager.

For example, the following definition uses the foo manager to instantiate the service object.

```
<service id="fooService" interface="com.acme.Foo">
    <ref component-id="fooImpl"/>
</service>
<bean id="fooImpl" class="com.acme.FooImpl"/>

public class FooImpl implements Foo { }
```

121.8.2 <idref>

The idref element provides the component id of another manager in the same Blueprint Container. This reference can then be used by the application to look up a manager in the Blueprint Container during runtime. The idref element is a safe way to provide a component id because the Blueprint Container will verify that the component id exists, thereby showing errors early. The idref does not create an implicit dependency on the given manager.

Table 121.13   *IdRef Features*

Attribute	Type	Id Ref Metadata	Description
component-id	NCName	componentId : String	A reference to a top level manager.

The following example provides the foo object with the reference to the database.

```
<bean id="foo" class="com.acme.FooImpl">
    <property name="db">
        <idref component-id="jdbc"/>
    </property>
</bean>

<bean id="jdbc" ... />
```

The following definition is equivalent to except that a non existent component id will not be detected until the foo object access the Blueprint Container. In the previous example this was detected directly after the definitions were parsed.

```
<bean id="foo" class="com.acme.FooImpl">
    <property name="db" value="jdbc"/>
</bean>
```

```
<bean id="jdbc" ... />
```

121.8.3   <value>

A value element represents an object that can directly be constructed from a string formed by its text contents.

Table 121.14   *Value Features*

Attribute, Element	Type	Value Metadata	Description
type	type	type : String	The optional type name to be used in type converting the given string to a target type. This type can commit the conversion to a specific choice. If this type is not set, then it must return null. For the type syntax, see *Syntax for Java types* on page 292.
...	<<type>>	stringValue : String	The string value that must be converted to the target type, if set.

If a value element is used as a member in a list, map, array, or set then the enclosing collection can define a default value for the type attribute of its value elements.

The following example creates a list of two OSGi version objects.

```
<list value-type="org.osgi.framework.Version">
    <value>1.3.4</value>
    <value>5.6.2.v200911121020</value>
</list>
```

The corresponding Java code is:

```
Arrays.asList( new Version("1.3.4"),
    new Version("5.6.2.v200911121020") )
```

121.8.4   <null>

A null element results in a Java null. It has no attributes and no elements. It corresponds to Null Metadata.

121.8.5   <list>, <set>, <array>

Lists, sets, and arrays are referred to as *collections*. List and array are ordered sequences of objects, where equal objects can occur multiple times. A set discards equal objects.

The structure of a collection element is:

```
collection  ::=  value *
```

Table 121.15   *Collection Features*

Attribute or Element	Type	Collection Metadata	Description
value-type	type	valueType : String	Optionally set the type for ValueMetadata children.

Attribute or Element	Type	Collection Metadata	Description
		collectionClass : Class< List | Set | Object[] >	The actual collection class to be used, derived from the appropriate definition.
<...>	object*	values : List<Metadata>	The Metadata for the children of the collection

The valueType sets the default for any contained ValueMetadata objects. The result of a collection element is an object that implements the given collection interface or is an Object[]. That is, the resulting object is mutable and can be used by the application. However, type conversion can create a copy of this list.

The following example creates a List of Lists of 2x2 of int values:

```
<list>
    <list value-type="int">
        <value>2</value>
        <value>7</value>
    </list>
    <list value-type="int">
        <value>9</value>
        <value>5</value>
    </list>
</list>
```

The corresponding Java code is:

```
Arrays.asList(
    new int[] {2,7},
    new int[]{9,5},
)
```

121.8.6 <map>

A map is a sequence of associations between a *key* and some object., this association is called an *entry*. The structure of a map element is therefore:

```
map ::= <entry> *
```

Table 121.16   *Map Features*

Attribute or Element	Type	Map Metadata	Description
key-type	type	keyType : String	Optional default type for keys. For the syntax see *Syntax for Java types* on page 292.
value-type	type	valueType : String	Optional default type for values. For the syntax see *Syntax for Java types* on page 292.
<entry>	See *<entry>* on page 319.	values : List<MapEntry>	The MapEntry object for the children of the map or properties.

There are no additional constraints.

121.8.7   <entry>

The entry element provides an association between a key and a value. The structure of the element is:

```
entry   ::= <key> object
key     ::= nonNullValue
```

Table 121.17   *Entry Features*

Attribute	Type	Map Entry	Description
key	<<type>>	key : NonNullMetadata	Specify the key of the entry.
key-ref	NCName	key : NonNullMetadata	Reference to a top-level manager
<key>	nonNull-Value	key : NonNullMetadata	Contains an inlined value that is never null.
value	<<type>>	value : Metadata	Specify the value directly, this will be a string type.
value-ref	NCName	value : RefMetadata	A reference to a top-level manager
<...>	object	value : Metadata	An inlined manager

Additional constraints:

- key, key-ref attributes and key element are mutually exclusive.
- value, value-ref attributes and value element are mutually exclusive.
- The resulting object of a key must not be a primitive type.

The following example shows the different way an entry can get its key. In this case the value is always a string.

```
<map>
    <entry key="bar"     value="..."/>     // 1
    <entry key-ref="bar" value="..."/>     // 2
    <entry value="...">                    // 3
        <key>
            <value type="org.osgi.framework.Version">
                2.71
            </value>
        </key>
    </entry>
</map>
```

The previous example is equivalent to the following Java code:

```
Map m = new HashMap();
m.put( "bar", "..." );
m.put( container.getComponentInstance("bar"), "..." );
m.put( new Version("2.71"), "..." );
```

The following examples shows the different ways a value of an entry can be defined.

```
<map>
    <entry key="1" value="1"/>
    <entry key="2" value-ref="foo"/>
    <entry key="3">
        <value type="org.osgi.framework.Version">3.14</value>
    </entry>
</map>
```

The previous code is equivalent to the following Java code.

```
Map m = new HashMap()
m.put("1", "1");
m.put("2", container.getComponentInstance("foo"))
m.put("3", new Version("3.14"));
```

121.8.8 <props>

The props element specifies a Properties object. The structure of a props element is as follows:

```
 props  ::= prop *
```

Each prop element is an association between two strings. It defines the following attributes:

- key - A string specifying the property key. This attribute is required.
- value - A string specifying the property value.

The following example initializes the same Properties object in two s ways.

```
<props>
    <prop key="1">one</prop>
    <prop key="2">two</prop>
</props>

<props>
    <prop key="1" value="one"/>
    <prop key="2" value="two"/>
</props>
```

This is equivalent to the following Java code:

```
Properties p = new Properties();
p.setProperty( "1", "one");
p.setProperty( "2", "two");
```

121.8.9 Manager as Value

Each manager can be the provider of component instances that act as object values. When a manager is used in an object value, then that is the manager asked to provide a component instance. The managers are specified in *manager* on page 294. The simple example is a bean. Any inlined bean can act as an object value. For example:

```
<list>
    <bean class="com.acme.FooImpl"/>
</list>
```

Some managers have side effects when they are instantiated. For example, a service manager will result in a ServiceRegistration object but it will also register a service.

```
<map>
    <entry key="foo">
        <service interface="com.acme.Foo">
            <bean class="com.acme.FooImpl"/>
        </service>
    </entry>
</map>
```

121.9   Dependency Injection

A bean has a recipe for constructing a component instance with a constructor or factory and then providing it with its *properties*. These properties are then injected with *object values*, see *Object Values* on page 315.

The following types of dependencies can be injected:

- *Constructor arguments* - The arguments specify the parameters for a constructor.
- *Static Factory arguments* - The arguments specify the parameters for a static method.
- *Instance Factory arguments* - The arguments specify the parameters for a method on an object provided by another manager.
- *Properties* - The value of the Bean Property specifies the single parameter for the property's set method.

In all the previous cases, the Blueprint Container must find an appropriate method or constructor to inject the dependent objects into the bean. The process of selecting the correct method or constructor is described in the following section, which assumes a Bean Argument as context, where a Bean Property acts as a Bean Argument without an index or type set.

121.9.1   Signature Disambiguation

Constructors, factory methods, and property set methods are described with Metadata. The Blueprint Container must map these descriptions to an actual method or constructor. In practice, there can be multiple methods/constructors that could potentially map to the same description. It is therefore necessary to disambiguate this selection. Both factory methods and constructors have the same concept of *signatures*. A signature consists of an ordered sequence of zero or more types. For methods, only publicly accessible methods with the appropriate name are considered. For constructors, all publicly accessible constructors are considered. The disambiguation process described here is valid for all constructors and methods because the signature concept applies to both of them.

1. Discard any signatures that have the wrong cardinality
2. Find the list of signatures that have *assignable* types for each argument in their corresponding positions. Assignable is defined in *Type Compatibility* on page 322. If a type was specified for an argument, then this type must match the name of the corresponding reified type in the signature exactly.
3. If this result list has one element, then this element is the answer. If this list has more than one element, then the disambiguation fails.
4. Otherwise, find the list of signatures that have *compatible* types for each argument in their corresponding positions. Compatibility is defined in *Type Compatibility* on page 322.
5. If this result list has one element, then this element is the answer. If the list has more than one element, then the disambiguation fails.
6. If the arguments cannot be reordered (the index of the argument is used and is thus not -1, or there are less than two arguments) then the disambiguation fails.
7. Find all signatures that match a re-ordered combination of the arguments. Reordering must begin with the first argument and match this argument against the first *assignable* types in a sig-

nature, going from position 0 to n. If the type is assignable from the argument, then it is locked in that position. If the argument has a `type`, then it must exactly match the name of the selected signature type. The same is done for the subsequent arguments. If all arguments can find an exclusive position in the signature this way, than the signature is added to the result.

8. If the result list contains one signature, then this is the resulting signature. If the list has more than one element, then the disambiguation fails.

9. Repeat step 6, but now look for *compatible* types instead of assignable types.

10. If the result list contains one signature, then this is the resulting signature.

11. Otherwise, the disambiguation fails

An example elucidates how the disambiguation works. Assuming the following definition and classes:

```
<bean ...>
    <argument>
        <bean class="Bar"/>
    </argument>
    <argument>
        <bean class="Foo"/>
    </argument>
<bean>


public class Bar extends Foo {}
public class Foo {}
```

The following bullets provide examples how signatures are matched against the previous definition.

- (Bar,Foo) - The arguments will be in the given order and the ordered match will succeed. This is the normal case.
- (Foo,Bar) - This will not match because in the re-ordered match, the Bar argument (which is a Foo sub-type) is matched against the first argument. The second Foo argument can then no longer find a compatible type because that slot is taken by the Bar instance.
- (Object,Object) - This will be called with (aBar,aFoo).

Multiple constructors on a class can require disambiguation with the arguments type. In the following example, the Multiple class has two constructors that would both match the constructor arguments because a String object can be converted to both a File object and a URL object.

```
public class Multiple {
    public Multiple(URL a);
    public Multiple(File a);
}
```

An attempt to configure a Multiple object without the type will fail, because it is not possible to determine the correct constructor. Therefore, the type should be set to disambiguate this:

```
<bean class="Multiple">
  <argument type="java.net.URL" value="http://www.acme.us"/>
</bean>
```

121.9.2 Type Compatibility

During injection, it is necessary to decide about type *assignability* or type *compatibility* in several places. If generics are present, a type must be *reified* in its class, see *Generics* on page 327. In this specification, the canonical representation for a type is T<P1..Pn>, where n is zero for a non-parame-

terized type, which is always true in a VM less than Java 5. The `ReifiedType` class models this kind of type.

If type T or S is primitive, then they are treated as their corresponding wrapper class for deciding assignability and compatibility. Therefore, a type T<P1..Pn> (target) is *assignable* from an object s of type S (source) when the following is true:

- `n == 0`, and
- `T.isAssignableFrom(S)`

T<P1..Pn> is *compatible* with an object s of type S when it is assignable or it can be converted using the Blueprint built-in type converter. The convertability must be verified with the canConvert(s,T<P1..Pn>) method. That is, type compatibility is defined as:

- `assignable(T<P1..Pn>,S)`, and
- `cs.canConvert(s,T<P1..Pn>)` returns `true`

Where `cs` is the Blueprint built in type converter that also uses the custom type converters.

121.9.3 Type Conversion

Strings in Blueprint definitions, object values, and component instances must be made compatible with the type expected by an injection target (method or constructor argument, or property) before being injected, which can require *type conversion.* The Blueprint Container supports a number of built-in type conversions, and provides an extension mechanism for configuring additional type converters. Custom type converters have priority over built-in converters.

The goal of the type conversion is to convert a source object s with type S to a target type T<P1..Pn>. The conversion of the Blueprint built-in type converter must take place in the following order:

1. If T<P1..Pn> is assignable from S, which implies n=0, then no conversion is necessary, except that primitives must be converted to their wrapper types.
2. Try all type converters in declaration order with the canConvert(s,T<P1..Pn>) method, exceptions are ignored and logged. The first converter that returns `true` is considered the converter, its result is obtained by calling convert(s,T<P1..Pn>). Exceptions in this method must be treated as an error.
3. If T is an array, then S must be an array or it must implement `Collection`, otherwise the conversion fails. Each member of array s must be type converted to the component type of T using the generics information if available, see the `getComponentType` method on `Class`. This is a recursive process. The result must be stored in an array of type T.
4. If T implements `Collection`, then S must be an array or implement `Collection`, otherwise the conversion fails. If the platform supports generics, the members of object s must be converted to the member type of the collection if this is available from the generics information, or to `Object` otherwise. The Blueprint Container must create a target collection and add all members of s to this new object in the iteration order of s. The target collection depends on type T:
   - If T is one of the interfaces listed in *Concrete Types for Interfaces* on page 326, then the target collection must be the corresponding concrete class.
   - T must represent a public concrete class with an empty publicly accessible constructor, the target collection is then a new instance of T.
   - Otherwise T represents an interface and the conversion must fail.
5. If T implements `Map` or extends `Dictionary`, then S must implement `Map` or extend `Dictionary` as well, otherwise the conversion fails. If the platform supports generics, the members of map s must be converted to the key and value type of the target map. This is a recursive process. Without generics, the members are not converted and put as is.

   The target map depends on T:

- If T is a public concrete class (not interface) with an empty publicly accessible constructor then the target map must be a new instance of T.
- If T is one of the Map interfaces or Dictionary listed in *Concrete Types for Interfaces* on page 326, then the target map must be the corresponding concrete class.
- Otherwise, the conversion fails.

6. If T is one of the primitive types (byte, char, short, int, long, float, double, boolean) then treat T as the corresponding wrapper class.

7. If T extends class Number and S extends also class Number then convert the source to a number of type T. If the target type cannot hold the value then the conversion fails. However, precision may be lost if a double or float is converted to one of the integer types.

8. If source type S is not class String, then the conversion fails.

9. The conversion is attempted based on the target type T from the string s. The following target types are supported:
   - boolean or Boolean - Construct the appropriate boolean type while accepting the following additional values for true and false respectively:
     - yes, no
     - on, off
   - Character - The string s must have a length of 1, this single character is then converted to a Character object.
   - Locale - The string s is converted to a Locale using the following syntax (no spaces are allowed between terms).

     ```
     locale  ::= <java language-code> ( '_' country)+
     country ::= <java country-code> ('_' <java variant-code>)+
     ```

   - Pattern - Create the Pattern object with Pattern.compile(String).
   - Properties - Create a new Properties object and load the properties from the string. The string must follow the format described with the Properties.load method.
   - Enum subclass - Convert the string s to the appropriate member of the given enum with the Enum.valueOf method. If the string is not one of the enum values, then the conversion must fail.
   - Class - The string s must conform to the syntax in *Syntax for Java types* on page 292. This type must be loaded through the Bundle's loadClass method. The resulting class must match any generic constraints on T. If this fails, the conversion fails.

10. If target type T has a constructor (String), then use this constructor to create an instance with the source string s. This convention caters for many of the built-in Java types such as BigDecimal, BigInteger, File, URL, and so on, as well as for custom types.

If none of the above steps has found a proper conversion than the conversion fails. Failing a conversion must end with throwing an Illegal Argument Exception.

121.9.4   Type Converters

A type converter converts a source type to a target type. The source type for a type converter is not constrained. A type converter must support the following methods:

- canConvert(Object,ReifiedType) - A light weight method that inspects the object and returns true if it can convert it to the given Reified Type, false otherwise. Converters normally can convert a type S to a type T<...>. However, converters can convert to multiple types and the value of the source object can influence the returned type. For example, a converter could convert a string to a type based on its content.
- convert(Object,ReifiedType) - The actual conversion method. This method should not fail if the canConvert method has returned true.

The ReifiedType class provides access to the target class. In a Java 1.4 environment, the ReifiedType object will provide a Class object for conversion and no type arguments. In a Java 5 environment, the ReifiedType object provides access to the reified class as well as the type arguments. Generics and reified types are described in *Generics* on page 327.

Type converters are normal managers with some limitations due to the dependency handling. If they depend on general managers or services then there is a change that cyclic dependencies are created.

Converters must be defined in the type-converters element, see ⟨*type-converters*⟩ on page 294, to be registered as a converter. Component instances of managers in this section must implement the Converter interface. Converters must also only transitively depend on built-in converters. It must be possible to initialize all converters before any of them are used. Type converters should not use the type conversion before all type converters are fully configured.

Converters are ordered within one definition resource but there is no resource ordering, so the overall ordering is not defined, making it a good practice to concentrate all converters in a single XML definition. The definition ordering is used during type conversion. That is, converters are not ordered by their specialization, a converter that is earlier can convert a more general type will override a converter that is later in the list but could have converted to a more specific type.

Converters must always use the type arguments of the given Reified Type, even if they are running on Java 1.4. The default behavior of the Reified Type will automatically work.

The following example demonstrates how a converter can use generics to use an AtomicReference⟨T⟩ whenever type T is supported. Such a type could be for a property like:

```
public void setInteger( AtomicReference<Integer>atomic );
```

The Atomic Converter uses the generic argument to convert a source object to an Integer and then creates an AtomicReference with this converted object. The definition of the type converter looks like:

```
<type-converters>
  <bean class="AtomicConverter">
    <argument ref="blueprintConverter"/>
  </bean>
</type-converters>
```

The Blueprint converter is injected in the constructor of the AtomicInteger class, in order to allow the conversion of the generic arguments. The Blueprint built-in type converter must not be used before all type converters are registered because a needed type converter might not have been registered yet. This is the reason type converters should not require type conversion in their initialization because the state of this converter is not well defined at this time.

The conversion class looks like:

```
public class AtomicConverter {
  Converter bpc;
  public AtomicConverter(Converter bpc) { this.bpc=bpc; }

  public boolean canConvert(Object s,ReifiedType T) {
    return T.getRawClass() == AtomicReference.class
    && bpc.canConvert(s, T.getActualTypeArgument(0));
  }

  public Object convert( Object s, ReifiedType T )
      throws Exception {
    Object obj = bpc.convert(
```

```
                    s,T.getActualTypeArgument(0) );

        return new AtomicReference<Object>(obj);
    }
}
```

Any injection that now targets an `AtomicReference<T>` value will automatically be converted into an `AtomicReference` of the appropriate type because of the example converter. The following definitions test this behavior:

```
public class Foo<T extends Integer> {
  public Foo( AtomicReference<T> v) {}
}
```

```
<bean id="foo" class="Foo"> <argument value="6"/> </bean>
```

This definition will create an foo object with the `Foo(AtomicReference<T>)` constructor. The source type is a string and there is no assignability for an Atomic Reference, so the registered type converters are consulted. The Atomic Converter recognizes that the target T is an `AtomicReference` class and indicates it can convert. The convert method then uses the generic argument information, which is an `Integer` object in the example, to convert the string "6" to an Integer object and return the appropriate `AtomicReference` object.

121.9.5 Built-in Converter

A Blueprint Container must contain an environment manager called `blueprintConverter`. The related component instance must implement the `Converter` interface.

The built-in Converter provides access to the provided type converters as well as the built in types. This service provides the type conversion as defined in *Type Conversion* on page 323.

Injecting a reference to the `blueprintConverter` environment manager into a bean provides access to all the type conversions that the Blueprint Container and registered type converters are able to perform. However, if this converter is injected in a type converter, then by definition, not all custom type converters are yet registered with the built-in converter. Type converters should therefore in general not rely on type conversion during their construction.

121.9.6 Concrete Types for Interfaces

The Blueprint extender can choose an implementation class when it provides an instance during conversion to an interface as well as when it natively provides an object. The actual implementation class can make a noticeable difference in disambiguation, type conversion, and general behavior. Therefore this sections describe the concrete types an implementation must use for specific interfaces if the platform allows this.

Table 121.18   *Implementation types for interfaces*

	Interface/Abstract class	Implementation class
	Collection	ArrayList
	List	ArrayList
Java 5	Queue	LinkedList
	Set	LinkedHashSet
	SortedSet	TreeSet
	Map	LinkedHashMap
	SortedMap	TreeMap
Java 5	ConcurrentMap	ConcurrentHashMap

Interface/Abstract class	Implementation class
Dictionary	Hashtable

If possible, the instances of these types must preserve the definition ordering.

121.9.7 Generics

Java 5 introduced the concept of *generics*. Before Java 5, a *type*, was simply a class or interface, both represented by the Class object. Generics augment these classes and interfaces with additional *type constraints*. These type constraints are not available on an instance because an instance always references a raw Class. For an instance all generic type constraints are *erased*. That is, a List<Integer> object is indistinguishable from a List<String> object, which are indistinguishable from a List object. Objects always refer to a raw Class object, this is the one returned from the getClass method. This Class object is shared between all instances and can therefore not have the actual type constraints (like String, Integer in the list examples).

When a class is used the compiler captures the type constraints and associates them with the specific use and encodes them in a Type object. For example, a field declaration captures the full generic type information:

```
List<String> strings;
```

A field has a getGenericType method that provides access to a Type object, which is a super interface for all type information in the Java 5 and later runtime. In the previous example, this would be a Parameterized Type that has a raw class of List and a type argument that is the String class. These constraints are reflectively available for:

- A superclass
- Implemented interfaces
- Fields
- For each method or constructor:
    - Return type
    - Exception types
    - Parameter types

Generics influence the type conversion rules because most of the time the Blueprint extender knows the actual Type object for an injection. Therefore, conversion must take place to a type like T<P1..Pn>, where T is a raw Class object and P1..Pn form the available type parameters. For a non-parametrized class and for other VMs than 1.4, n is always zero, that is no type arguments are available. The P arguments are in itself instances of Type. The form T<P1..Pn> is called the *reified* form. It can be constructed by traversing the Type graph and calculating a class that matches the constraints. For example < extends List<T>> defines a *wild card* constraint, that has a List<T> as reified type, where T is a Type Variable defined elsewhere that can have additional constraints. The resulting type must be an instance of List<T>. A reified type will use an object implementing List for such an example because that is the only class that is guaranteed to be compatible. The rules to reify the different Type interfaces are:

- Class - A Class represents unparameterized raw type and is reified into T<>. For example:

    ```
    String string;
    ```

- ParameterizedType - A Parameterized Type defines a raw type and 1..n typed parameters. The raw type of the Parameterized Type is also reified and represents T. The arguments map directly to the arguments of the reified form. An example of a Parameterized Type is:

    ```
    Map<String,Object> map;
    ```

- `TypeVariable` - Represents a Type Variable. A type variable is listed in a generics type declaration, for example in `Map<K,V>`, the `K` and `V` are the type variables. A type variable is bounded by a number of types because it is possible to declare a bounded type like: `<A extends Readable&Closeable>`. A Type Variable is reified by taking its first bound in reified form, this is the same as in Java 5 where the first bounds is the erasure type. However, this can fail if multiple bounds are present. An example of a Type Variable is:

```
public <T extends ServiceTracker> void setMap(T st) {}
```

  In this example, the parameter `st` will have a reified type of `ServiceTracker`.
- `WildcardType` - A Wildcard Type constrains a type to a set of lower bounds and a set of upper bounds, at least in the reflective API. In the Java 5 and later syntax a Wildcard Type can only specify 0 or one lower and one upper bound, for example `<T extends Number>` constraints the Type Variable `T` to at least extend the `Number` class. A Wildcard Type is reified into its reified upper bound when no lower bound is set, and otherwise it is reified into its reified lower bound. An example of a Wildcard Type is seen in the example of a Type Variable.
- `GenericArrayType` - A Generic Array Type represents an array. Its component type is reified and then converted to an array. The Reified Type will have the array class as reified class and the type arguments reflect the type arguments of the component type. For example:

```
public void setLists(List<String>[] lists) {}
```

  This example will have a Reified Type of `List[]<String>`.

This specification is written to allow Java 1.4 implementations and clients, the API therefore has no generics. Therefore, the Type class in Java 5 and later cannot be used in the API. However, even if it could use the `Type` class, using the type classes to create the reified form is non-trivial and error prone. The API therefore provides a concrete class that gives convenient access to the reified form without requiring the usage of the `Type` class.

The `ReifiedType` class provides access to the reified form of `Class`, which is itself and has no type arguments. However, Blueprint extender implementations that recognize Java 5 generics should subclass the `ReifiedType` class and use this in the conversion process. The subclass can calculate the reified form of any `Type` subclasses.

121.10   Service Dynamics

The Blueprint Container specification handles the complexities of the dynamic nature of OSGi by *hiding* the dynamic behavior of the OSGi service registry, at least temporarily. This dynamic behavior is caused by service references that select one or more services that can come and go at runtime.

The Blueprint Container must handle the dynamics in the following way:

- *Proxied references* - Service reference managers must provide a proxy implementing the specified interfaces, instead of the actual service object it refers to. The proxy must fetch the real service lazily. For reference managers, when a proxy is used, and no candidate is available, a candidate must be found within a limited time. If no candidate service is available during this time, a Service Unavailable Exception must be thrown. The reference-list manager also maintains proxies but these proxies must throw a Service Unavailable Exception immediately when the proxy is used and the backing service is no longer available.

  When proxied references are used with stateful services, then the application code must register a reference listener to perform the necessary initialization and cleanup when a new backing service is bound.
- *Conditional Service Registrations* - The service manager is responsible for registering a service with the OSGi service registry. A service manager is statically dependent on the transitive set of man-

agers that it depends on. If these static dependencies contain mandatory service references, then the manager's service must not be registered when any of these mandatory service references is unsatisfied, see *Enabled* on page 308.

121.10.1 Damping

When an operation is invoked on an unsatisfied proxy from a reference manager (either optional or mandatory), the invocation must block until either the reference becomes satisfied or a time-out expires (whichever comes first). During this wait, a WAITING event must be broadcast, see *Events* on page 331.

The default timeout for service invocations is 5 minutes. The optional timeout of the reference element specifies an alternate timeout (in milliseconds). If no matching service becomes available within the timeout, then a Service Unavailable Exception must be thrown. A timeout of zero means infinite and a negative timeout is an error.

For example:

```
<reference id="logService"
        interface="org.osgi.service.log.LogService"
      timeout="100000" />


<bean id="bar" class="BarImpl">
  <property name="log" ref="logService"/>
</bean>
```

When this Blueprint Container is instantiated, the reference manager provides a proxy for the Log Service, which gets injected in the log property. If no Log Service is available, then the proxy will have no backing service. If the bar object attempts to log, it will block and if the timeout expires the proxy must throw a Service Unavailable Exception.

If at some later point in time, a Log Service is registered then it becomes satisfied again. If bar now logs a message, the proxy will get the service object again and forward the method invocation to the actual Log Service implementation.

The damping ensures that a mandatory service reference that becomes unsatisfied does not cause the Blueprint Container to be destroyed. Temporary absences of mandatory services are tolerated to allow for administrative operations and continuous operation of as much of the system as possible.

A reference-list manager does not provide damping. It only removes the service proxy from the collection if its service goes away. Using a collection reference manager will never block, it will just have no members if its selection is empty. A timeout attribute is therefore not supported by the reference-list elements. However, the elements are proxied and it is possible that they throw a Service Unavailable Exception when used and the backing service has disappeared. The exceptions for a reference-list proxy will be thrown immediately when the proxy is used.

121.10.2 Iteration

The provided object of a reference-list manager implements the List interface. Depending on the memberType or the optional generics information, it provides a collection that contains the member objects, that is, either proxies to the service object, or ServiceReference objects. These collections are read-only for the receiver, however, their contents can dynamically change due to changes in the selection. The access to these collections with iterators must give a number of guarantees:

- *Safe* - All iterators of reference-list managers must be safe to traverse according to the Iterator interface contract, even while the underlying collection is being modified locally or in another thread. If the hasNext method returns true, the iterator must return a member object on the subsequent next method invocation. If there is no longer a service object available when requested, then a dummy proxy must be returned that throws a Service Unavailable Exception whenever it is used.

- *Visibility* - All the changes made to the collection that affect member objects not yet returned by the iterator must be visible in the iteration. Proxies for new services must be added at the end of the List. Proxies already returned can be affected by changes in the service registry after the iterator has returned them.

After the iterator has returned false for the hasNext method, no more objects can be obtained from it. A List Iterator must not be supported.

121.10.3 Mandatory Dependencies

A service manager can have mandatory service reference managers in its transitive dependencies. Such a service manager must ensure that the service object is registered with the OSGi service registry during the runtime phase when all its mandatory service references that it depends on are satisfied. This called *tracking* the dependency. A service manager is *enabled* when all its mandatory references in its dependencies are satisfied.

This tracking only works for dependencies declared directly in the definitions; dependencies established during runtime by calling the getComponentInstance method are not tracked.

In the following example, service manager S has a transitive dependency on the mandatory reference manager M, which means the Blueprint Container must ensure that the service object provided by bean A is registered when reference manager M is satisfied.

```
<service id="S" ref="A" interface="com.acme.Foo"/>
<bean id="A" class="com.acme.FooImpl">
    <property name="bar" ref="m"/>
</bean>
<reference id="M" interface="com.acme.Bar"
    availability="mandatory"/>
```

However, if the dependency from manager A on manager M is not declared but created through code that manipulates the Blueprint Container then the dependency is not tracked.

121.11 Blueprint Container

The Blueprint Container has a registry where all top-level managers, as well as *environment* managers, are registered by their component id. The Blueprint Container can be injected in application code with the environment blueprintContainer manager. For example:

```
<bean class="com.acme.FooImpl">
    <property name="container" ref="blueprintContainer"/>
</bean>
```

The Blueprint Container allows application code to get objects that are provided by the top-level managers through the getComponentInstance method. However, the Blueprint Container should not be required to get a component instance; the proper way to use Blueprint is to inject them. This declarative approach makes the Blueprint Container aware of any dependencies; one of the primary goals of a dependency injection framework. The Blueprint Container's introspective features are commonly used for management and other non-application purposes.

The Blueprint Container is registered as a service during the runtime phase so that other bundles can use it for these, and other, purposes.

121.11.1 Environment Managers

The Blueprint Container provides a number of *environment managers*. These managers have defined names and provide convenient access to information about the environment. Environment man-

agers cannot be overridden by explicitly defined managers because it is invalid to define a manager with an existing component id. All component ids starting with blueprint are reserved for this specification and future incarnations.

There is no XML definition for environment managers but their Metadata must be provided as ComponentMetadata objects.

The following ids are used for the environment managers:

- blueprintContainer - The Blueprint Container.
- blueprintBundle - A manager that provides the Blueprint bundle's Bundle object.
- blueprintBundleContext - A manager that provides the Blueprint bundle's BundleContext object.
- blueprintConverter - A manager that provides an object implementing the Converter interface. This represents the built-in conversion facility that the Blueprint Container uses to convert objects. See *Built-in Converter* on page 326.

121.11.2 Component Instances

The Blueprint Container provides access to the component instances that the top level managers can provide, as well as their Metadata. The Blueprint Container has the following methods for requesting a component instance and to find out what managers are available:

- getComponentInstance(String) - This method will provide a component instance from the component id. If the manager has not been activated yet, it must atomically activate and ensure its explicit and implicit dependencies are activated transitively.
- getComponentIds() - Returns a set of component ids in this Blueprint Container. These ids must consist of all top level managers (including calculated ids) and environment managers.

121.11.3 Access to Component Metadata

Each of the manager types has specific Component Metadata subtypes associated with it, except Environment managers that use Component Metadata. The Blueprint Container provides access by component id to the Component Metadata of the top level managers. However, managers can also be defined inline, in which case they do not have a component id. Therefore, the Blueprint Container can also enumerate all the managers that are represented by a Metadata sub-interface.

- getComponentMetadata(String) - Answer the Component Metadata sub-type for the given component id. Environment managers will return a ComponentMetadata object, the other managers each have their own specific Metadata type.
- getMetadata(Class) - Answer a collection with the Metadata of the given type, regardless if it is defined as/in a top-level or inlined manager. For example, getMetadata(ServiceMetadata.class) returns all Service Metadata in the Blueprint container. This includes all top level managers as well as any inlined managers. For Environment Managers, this method returns a ComponentMetadata object.

121.11.4 Concurrency

A Blueprint Container must be thread safe. Each method must handle the case when multiple threads access the underlying registry of managers. Activation of managers must be atomic. That is, other threads must be blocked until a manager is completely activated.

The Blueprint Container must handle reentrant calls.

121.12 Events

The Blueprint Container must track all Blueprint Listener services and keep these listeners updated of the progress or failure of all its managed bundles. The Blueprint Listener is kept informed by

sending it events synchronously. These events are therefore normally delivered in order but in exceptional cases this can be seen out of order for a listener when new events are initiated synchronously from within a callback. Therefore, Blueprint Listener services should see the event as a notification, where actual work should be processed on another thread.

Blueprint Events must be sent to each registered Blueprint Listener service. This service has the following method:

- blueprintEvent(BlueprintEvent) - Notify the listener of a new Blueprint Event. These events are send synchronously with their cause. That is, all listeners must be notified before the Blueprint Container continues to the next step.

The events must be delivered as BlueprintEvent objects. The event types that they represent, and the data that these objects carry, is further described in *Blueprint Event* on page 332.

A Blueprint Listener services must be given the initial state of all managed bundles before normal processing starts, see *Replay* on page 332.

Blueprint Listener services that throw Exceptions or do not return in a reasonable time as judged by the Blueprint extender implementation, should be logged, if possible, and further ignored.

121.12.1 Blueprint Event

The Blueprint Event supports the following event types:

- CREATING - The Blueprint extender has started creating a Blueprint Container for the bundle.
- GRACE_PERIOD - The Blueprint Container enters the grace period. This event can be repeated multiple times when the list of dependencies changes due to changes in the service registry.
- CREATED - The Blueprint Container is ready. The application is now running.
- WAITING - A service reference is blocking because of unsatisfied mandatory dependencies. This event can happen multiple times in a row.
- DESTROYING - The Blueprint Container is being destroyed because the Blueprint bundle or Blueprint extender has stopped.
- DESTROYED - The Blueprint Container is completely destroyed.
- FAILURE - An error occurred during the creation of the Blueprint Container.

The Blueprint Event provides the following methods:

- getBundle() - The Blueprint bundle
- getCause() - Any occurred exception or null
- getDependencies() - A list of filters that specify the unsatisfied mandatory references.
- getExtenderBundle() - The Blueprint extender bundle.
- getTimestamp() - The time the event occurred
- getType() - The type of the event.
- isReplay() - Indicates if the event is a replay (true) or if it is a new event (false), see *Replay* on page 332.

121.12.2 Replay

The Blueprint Extender must remember the last Blueprint Event for each ready bundle that it manages, see *Initialization Steps* on page 286. During the (synchronous) service registration event of a Blueprint Listener service, the Blueprint extender must inform the Blueprint Listener service about all its managed bundles by sending it the last known event for each bundle the Blueprint extender manages. This initial event is called the *replay* event, and is marked as such.

The replay event must be delivered to the Blueprint Listener service as the first event, before any other event is delivered, during the registration of the Blueprint Listener service. That is, the blue-

printEvent method must have returned before the first non-replay event can be delivered and no events must be lost. The replay events must be sent every time a Blueprint Listener service is registered.

The set of managed bundles is defined by bundles that are active and are managed by the Blueprint extender, even if their initialization ended in failure.

The BlueprintEvent object for a replay event must return true for the isReplay() method in this situation, and false in all other situations.

121.12.3 Event Admin Mapping

When the Event Admin service is present, the Blueprint extender must create an Event Admin event for each defined Blueprint Event. This Event Admin event must be asynchronously given to the Event Admin service with the postEvent method.

The topic of the Event Admin event is derived from the Blueprint event type with a fixed prefix. All topics must have the prefix of:

TOPIC_BLUEPRINT_EVENTS

After this prefix, the name of the Blueprint Event type must be used as the suffix. That is, CREATING, GRACE_PERIOD, etc. For example, org/osgi/service/blueprint/container/GRACE_PERIOD.

For each Blueprint event the following properties must be included:

- TYPE - The type of the Event, see *Blueprint Event* on page 332.
- BUNDLE - (Bundle) The Bundle object of the Blueprint bundle
- BUNDLE_ID - (Long) The id of the Blueprint bundle.
- BUNDLE_SYMBOLICNAME - (String) The Bundle Symbolic Name of the Blueprint bundle.
- BUNDLE_VERSION - (Version) The version of the Blueprint bundle.
- EXTENDER_BUNDLE - (Bundle) the Bundle object of the Blueprint extender bundle.
- EXTENDER_BUNDLE_ID - (Long) The id of the Blueprint extender bundle
- EXTENDER_BUNDLE_SYMBOLICNAME - (String) The Bundle Symbolic Name of the Blueprint extender bundle.
- EXTENDER_BUNDLE_VERSION - (Version) The version of the Blueprint extender bundle
- TIMESTAMP - (Long) The time when the event occurred
- CAUSE - (Throwable) The failure cause, only included for a FAILURE event.
- DEPENDENCIES - (String[]) The filter of an unsatisfied service reference. Can only appear in a GRACE_PERIOD, WAITING or FAILURE event caused by a time-out.
- EVENT - (BlueprintEvent) The BlueprintEvent object that caused this event.

The property names for Blueprint Listener events may be conveniently referenced using the constants defined in the org.osgi.service.event.EventConstants and EventConstants interfaces.

The Event Admin events do not follow the replay model in use for Blueprint Listener services. That is, the Event Admin must only be kept informed about events as they occur.

121.13 Class Loading

The module layer in OSGi provides advanced class loading rules that potentially can cause bundles to live in different class spaces. This means that not all bundles can collaborate because the classes involved in the collaboration can come from different class loaders, which results in confusing Class Cast Exceptions on classes with the same name. It is therefore crucial that the Blueprint Container uses the Bundle Context and the bundle class loader of the Blueprint bundle for all actions that are made on behalf of the Blueprint bundle. Especially, access to the OSGi service registry must

use the Bundle Context of the Blueprint bundle. Any dynamic class loading must use the Blueprint bundle's loadClass method. The normal OSGi mechanics will then ensure class space consistency for resolved bundles.

121.13.1   Blueprint Extender and Bundle Compatibility

For many Blueprint bundles, there is no class space compatibility issue. These bundles do not use any Blueprint classes and are therefore by definition compatible with any extender. However, if the Blueprint bundle uses some of the Blueprint packages, it must import these packages. Blueprint Containers must verify that they are *type compatible* with the Blueprint bundle before they attempt to manage it. See *Type Compatibility* on page 334.

121.13.2   XML and Class Loading

The Blueprint definition resources contain textual references to classes. These textual references will be loaded with the class loader of the Blueprint bundle. This implies that all the classes of provided component instances must be either imported or available from the bundle.

The Blueprint specification has the following attributes and elements that can cause imports:

- class
- value-type
- interface
- interfaces
- type
- key-type

All these attributes and elements are defined with the Tclass and Ttype XML Schema type for the Blueprint namespace. The Tclass defines simple class names, and Ttype defines types defined in *Syntax for Java types* on page 292.

121.13.3   Foreign Bundle Context

When using the Blueprint Container in its Blueprint bundle, the types that the managers provide are guaranteed to be compatible with the caller.

When using a Blueprint Container service in another bundle (for example, getting it as a service) then there is no guarantee of type compatibility or even visibility between the versions of the types of the returned managers, and the versions of the types visible to the caller. Care must therefore be taken when casting the return value of the getComponentInstance method to a more specific type.

121.13.4   Converters and Class Loading

A converter is closely coupled to its target class. If the converter comes from another bundle, then the converter bundle must ensure class space consistency between the converter implementation and the target class. This can be achieved by specifying the target class in the uses directive.

For example:

```
Export-Package:
    com.converters.ac;uses:="com.converters.dc"
```

A bundle that references a type converter defined in the Blueprint bundle does not need to export that type. When creating a Blueprint Container, the extender bundle uses the class loader of the Blueprint bundle.

121.13.5   Type Compatibility

Two bundles are type compatible for a given class if they both load the same class object, or if either bundle cannot load the given class.

To mitigate type incompatibility problems, a Blueprint extender must export the
`org.osgi.service.blueprint` package. In the `uses:` directive, it should list any packages of classes that
can be shared between the Blueprint extender and the Blueprint bundle. Blueprint bundles should
import this package.

121.13.6 Visibility and Accessibility

The Blueprint Container must load any classes it needs through the Blueprint bundle's `loadClass`
method. If a class can not be loaded, then the initialization fails. Class loading issues are further dis-
cussed in *Class Loading* on page 333.

The Blueprint Container must respect the accessibility of the class and any of its members. That is,
the Blueprint Container must not use the `setAccessibility` method. All classes and reflected mem-
bers must therefore be declared `public` or be implicitly `public` like the default constructor.

121.14 Metadata

An important aspect of the Blueprint specification is the so called *metadata* interfaces. These inter-
faces are used in the Blueprint Container to enable programmatic access to the XML definitions.
During the parsing phase the Blueprint Container reads the XML and converts it to an object imple-
menting the appropriate interface.
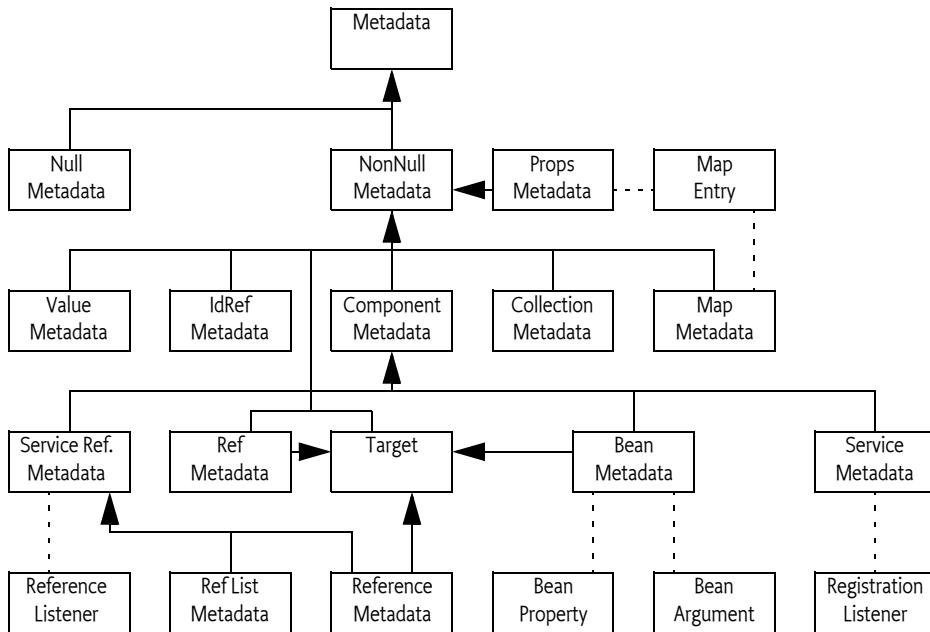
The XML elements and XML Schema types map to the `Metadata` interfaces. For example, `<bean>`
maps to `BeanMetadata`. However, in several cases, the attributes and/or sub-elements in the `Meta-
data` interfaces are merged when possible. For example, the `interface` attribute and `interfaces` ele-
ment in the service element are merged in the `ServiceMetadata` class' `getInterfaces()` method.

The interfaces are arranged in a comprehensive hierarchy that reflects their usage and constraints.
This hierarchy is depicted in Figure 121.12 on page 336.

The hierarchy can roughly be divided in two parts. The first part is the sub-interfaces of the `Compo-
nentMetadata` interface. These interfaces are defining the configuration data of the top-level and in-
lined managers. The manager's component instance(s) are injected with values during runtime. The
configuration of how to create a specific value is also described with Metadata interfaces. For exam-
ple, a `Map` object is described with configuration information in the `MapMetadata` interface. The hi-
erarchy makes it clear that Component Metadata is also a value that can be injected. Keys in maps or
properties can not be `null`. This is the reason the hierarchy is split at the top into a `null` value branch
and a branch that can only generates non-null values.

The `Target` interface describes managers that can be used as the target for the reference listener or
the registration listener, or a ref.

Figure 121.12   *Metadata Interfaces Hierarchy*

121.15   **Blueprint XML Schema**

The Blueprint schema included in this specification can be found in digital form at [9] *OSGi XML Schemas.*

```
<xsd:schema xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    elementFormDefault="qualified" attributeFormDefault="unqualified"
    version="1.0.1">

    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[
                This is the XML Schema for the OSGi Blueprint service 1.0.0
                development descriptor.  Blueprint configuration files
                using this schema must indicate the schema using the
                blueprint/v1.0.0 namespace.  For example,

                <blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

                if used as a qualified namespace, "bp" is the recommended
                namespace prefix.
            ]]>
        </xsd:documentation>
    </xsd:annotation>

    <!-- Schema elements for core component declarations -->

    <xsd:complexType name="Tcomponent" abstract="true">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                    The Tcomponent type is the base type for top-level
                    Blueprint components.  The <bean> <reference>, <service>,
                    and <reference-list> elements are all derived from
                    the Tcomponent type.  This type defines an id attribute
                    that is used create references between different components.
                    Component elements can also be inlined within other component
                    definitions.  The id attribute is not valid when inlined.
```

```
            ]]>
        </xsd:documentation>
    </xsd:annotation>

    <xsd:attribute name="id" type="xsd:ID" />

    <xsd:attribute name="activation" type="Tactivation">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                The activation attribute for this component.  This can either
                be "eager" or "lazy".  If not specified, it
                defaults to default-activation attribute of the enclosing
                <blueprint> element.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>

    <xsd:attribute name="depends-on" type="TdependsOn">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                depends-on identifies (by id) other components that this component
                depends on.  The component only be activated after the
                depends-on components are successfully activated.  Also, if there
                are <reference> or <reference-list> elements with unstatisfied
                manadatory references, then the depends-on relationship will also
                be used to determine whether this service is enabled or not.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
</xsd:complexType>

<xsd:element name="blueprint" type="Tblueprint">
    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[
            The <blueprint> element is the root element for a blueprint
            configuration file.  A blueprint configuration has two sections.
            The first section (contained within the <type-converters> element)
            identifies components that are used for converting values into
            different target types.  The type converters are optional, so
            the file does not need to specify a type converter section.

            Following the type converters are the component definitions.
            Components are <bean>, <service>, <reference>, and
            <reference-list> elements that identify the bundle components that will
            be managed by the blueprint service.
            ]]>
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>

<xsd:complexType name="Tblueprint">
    <xsd:sequence>
        <xsd:element name="description" type="Tdescription"
            minOccurs="0" />
        <xsd:element name="type-converters" type="Ttype-converters"
            minOccurs="0" maxOccurs="1" />
        <!-- top-level components -->
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="service" type="Tservice" />
            <xsd:element name="reference-list" type="Treference-list" />
            <xsd:element name="bean" type="Tbean" />
            <xsd:element name="reference" type="Treference" />
            <xsd:any namespace="##other" processContents="lax" />
        </xsd:choice>
    </xsd:sequence>

    <!-- Defaults-->
    <xsd:attribute name="default-activation" default="eager"
        type="Tactivation">
        <xsd:annotation>
```

```
                    <xsd:documentation>
                        <![CDATA[
                        Specifies the default activation setting that will be defined
                        for components.  If not specified, the global default is "eager".
                        Individual components may override the default value.
                        ]]>
                    </xsd:documentation>
                </xsd:annotation>
            </xsd:attribute>
            <xsd:attribute name="default-timeout" type="Ttimeout"
                default="300000">
                <xsd:annotation>
                    <xsd:documentation>
                        <![CDATA[
                        Specifies the default timeout value to be used when operations
                        are invoked on unstatisfied service references.  If the
                        reference does not change to a satisfied state within the timeout
                        window, an error is raised on the method invocation.  The
                        default timeout value is 300000 milliseconds and individual
                        <reference> element can override the specified configuration
                        default.
                        ]]>
                    </xsd:documentation>
                </xsd:annotation>
            </xsd:attribute>
            <xsd:attribute name="default-availability" type="Tavailability"
                default="mandatory">
                <xsd:annotation>
                    <xsd:documentation>
                        <![CDATA[
                        Specifies the default availability value to be used for
                        <reference>, and <reference-list> components.  The
                        normal default is "mandatory", and can be changed by individual
                        service reference components.
                        ]]>
                    </xsd:documentation>
                </xsd:annotation>
            </xsd:attribute>
            <xsd:anyAttribute namespace="##other"
                processContents="lax" />
    </xsd:complexType>

    <xsd:complexType name="Ttype-converters">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                The type used for the <type-converters> element.  The
                <type-converters> section is a set of <bean>, <ref>, or
                <reference> elements that identify the type converter components.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="bean" type="Tbean" />
            <xsd:element name="reference" type="Treference" />
            <xsd:element name="ref" type="Tref" />
            <xsd:any namespace="##other" processContents="lax" />
        </xsd:choice>
    </xsd:complexType>

    <!--
        Components that provide a reasonable target for injection used for
        listeners, etc.
    -->

    <xsd:group name="GtargetComponent">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                A target component is one that can be a target for a
                listener, registration-listener or service elements.
                This is used in contexts where the requirement is a single
                provided object that will implement a particular interface.
                The provided object is obtained either from a <ref> element
                or an inlined <bean> or <reference>.
```

```
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:choice>
            <xsd:element name="bean" type="Tinlined-bean" />
            <xsd:element name="reference" type="Tinlined-reference" />
            <xsd:element name="ref" type="Tref" />
            <xsd:any namespace="##other" processContents="lax" />
        </xsd:choice>
    </xsd:group>

    <xsd:group name="GallComponents">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                An all components is used in contexts where all component element
                types are values.  The set of component elements contains
                <bean>, <service>, <reference>, <reference-list> and <ref>.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:choice>
            <xsd:element name="service" type="Tinlined-service" />
            <xsd:element name="reference-list" type="Tinlined-reference-list" />
            <xsd:group ref="GtargetComponent" />
        </xsd:choice>
    </xsd:group>

    <xsd:group name="GbeanElements">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                A bean elements is a reusable definition of the elements allowed on
                a <bean> element.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
            <xsd:element name="description" type="Tdescription"
                minOccurs="0" />
            <xsd:choice minOccurs="0" maxOccurs="unbounded">
                <xsd:element name="argument" type="Targument" />
                <xsd:element name="property" type="Tproperty" />
                <xsd:any namespace="##other" processContents="lax" />
            </xsd:choice>
        </xsd:sequence>
    </xsd:group>

    <xsd:complexType name="Tbean">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                The type definition for a <bean> component.  The <bean>
                attributes provide the characteristics for how to create a
                bean instance.  Constructor arguments and injected properties
                are specified via child <argument> and <property> elements.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="Tcomponent">
                <xsd:group ref="GbeanElements" />
                <xsd:attribute name="class" type="Tclass" />
                <xsd:attribute name="init-method" type="Tmethod" />
                <xsd:attribute name="destroy-method" type="Tmethod" />
                <xsd:attribute name="factory-method" type="Tmethod" />
                <xsd:attribute name="factory-ref" type="Tidref" />
                <xsd:attribute name="scope" type="Tscope" />
                <xsd:anyAttribute namespace="##other"
                    processContents="lax" />
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="Tinlined-bean">
```

```
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                The Tinlined-bean type is used for inlined (i.e. non top level)
                <bean> elements.  Those elements have some restrictions on
                the attributes that can be used to define them.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:restriction base="Tbean">
                <xsd:group ref="GbeanElements" />
                <xsd:attribute name="id" use="prohibited" />
                <xsd:attribute name="depends-on" type="TdependsOn" />
                <xsd:attribute name="activation" use="prohibited"
                    fixed="lazy" />
                <xsd:attribute name="class" type="Tclass" />
                <xsd:attribute name="init-method" type="Tmethod" />
                <xsd:attribute name="destroy-method" use="prohibited" />
                <xsd:attribute name="factory-method" type="Tmethod" />
                <xsd:attribute name="factory-ref" type="Tidref" />
                <xsd:attribute name="scope" use="prohibited" />
                <xsd:anyAttribute namespace="##other"
                    processContents="lax" />
            </xsd:restriction>
        </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="Targument">
    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[
            An argument used to create an object defined by a <bean>
            component.  The <argument> elements are the arguments for the
            bean class constructor or passed to the bean factory method.

            The type, if specified, is used to disambiguate the constructor
            or method signature.  Arguments may also be matched up with
            arguments by explicitly specifying the index position.  If the
            index is used, then all <argument> elements for the bean must
            also specify the index.

            The value and ref attributes are convenience shortcuts to make
            the <argument> tag easier to code.  A fuller set of injected
            values and types can be specified using one of the "value"
            type elements.
            ]]>
        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
        <xsd:element name="description" type="Tdescription"
            minOccurs="0" />
        <xsd:group ref="Gvalue" minOccurs="0" />
    </xsd:sequence>
    <xsd:attribute name="index" type="xsd:nonNegativeInteger" />
    <xsd:attribute name="type" type="Ttype" />
    <xsd:attribute name="ref" type="Tidref" />
    <xsd:attribute name="value" type="TstringValue" />
</xsd:complexType>

<xsd:complexType name="Tproperty">
    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[
            A property that will be injected into a created <bean>
            component.  The <property> elements correspond to named
            JavaBean setting methods for a created bean object.

            The value and ref attributes are convenience shortcuts to make
            the <argument> tag easier to code.  A fuller set of injected
            values and types can be specified using one of the "value"
            type elements.
            ]]>
        </xsd:documentation>
    </xsd:annotation>
```

```
        <xsd:sequence>
            <xsd:element name="description" type="Tdescription"
                minOccurs="0" />
            <xsd:group ref="Gvalue" minOccurs="0" />
        </xsd:sequence>
        <xsd:attribute name="name" type="Tmethod" use="required" />
        <xsd:attribute name="ref" type="Tidref" />
        <xsd:attribute name="value" type="TstringValue" />
    </xsd:complexType>

    <xsd:complexType name="Tkey">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                The Tkey type defines the element types that are permitted
                for Map key situations.  These can be any of the "value"
                types other than the <null> element.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:group ref="GnonNullValue" />
    </xsd:complexType>

    <!-- reference -->
    <xsd:complexType name="Treference">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                The Treference type defines the <reference> element.  These
                are instances of the TserviceReference type, with the addition
                of a timeout attribute.  If the timeout is not specified,
                the default-timeout value is inherited from the encapsulating
                <blueprint> definition.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="TserviceReference">
                <xsd:sequence>
                    <xsd:any namespace="##other" minOccurs="0" maxOccurs="unbounded"
                        processContents="lax" />
                </xsd:sequence>
                <xsd:attribute name="timeout" type="Ttimeout" />
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="Tinlined-reference">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                The Tinlined-reference type is used for inlined (i.e. non top level)
                <reference> elements.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:restriction base="Treference">
                <xsd:sequence>
                    <xsd:group ref="GserviceReferenceElements" />
                    <xsd:any namespace="##other" minOccurs="0" maxOccurs="unbounded"
                        processContents="lax" />
                </xsd:sequence>
                <xsd:attribute name="id" use="prohibited" />
                <xsd:attribute name="depends-on" type="TdependsOn" />
                <xsd:attribute name="activation" use="prohibited"
                    fixed="lazy" />
                <xsd:attribute name="interface" type="Tclass" />
                <xsd:attribute name="filter" type="xsd:normalizedString" />
                <xsd:attribute name="component-name" type="Tidref" />
                <xsd:attribute name="availability" type="Tavailability" />
                <xsd:attribute name="timeout" type="Ttimeout" />
                <xsd:anyAttribute namespace="##other"
                    processContents="lax" />
            </xsd:restriction>
```

```
            </xsd:complexContent>
        </xsd:complexType>

        <!-- reference-list -->
        <xsd:complexType name="Treference-list">
            <xsd:annotation>
                <xsd:documentation>
                    <![CDATA[
                    The Treference-list builds in the characteristics of the
                    TserviceReference type to define characteristics of the
                    <reference-list>.  This adds in the characteristics that
                    only apply to collections of references (e.g., member-type).
                    ]]>
                </xsd:documentation>
            </xsd:annotation>
            <xsd:complexContent>
                <xsd:extension base="TserviceReference">
                    <xsd:sequence>
                        <xsd:any namespace="##other" minOccurs="0" maxOccurs="unbounded"
                            processContents="lax" />
                    </xsd:sequence>
                    <xsd:attribute name="member-type" type="Tservice-use"
                        default="service-object">
                    </xsd:attribute>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>

        <xsd:complexType name="Tinlined-reference-list">
            <xsd:annotation>
                <xsd:documentation>
                    <![CDATA[
                    The Tinlined-reference-list type is used for inlined (i.e. non top level)
                    <reference-list> elements.
                    ]]>
                </xsd:documentation>
            </xsd:annotation>
            <xsd:complexContent>
                <xsd:restriction base="Treference-list">
                    <xsd:sequence>
                        <xsd:group ref="GserviceReferenceElements" />
                        <xsd:any namespace="##other" minOccurs="0" maxOccurs="unbounded"
                            processContents="lax" />
                    </xsd:sequence>
                    <xsd:attribute name="id" use="prohibited" />
                    <xsd:attribute name="depends-on" type="TdependsOn" />
                    <xsd:attribute name="activation" use="prohibited"
                        fixed="lazy" />
                    <xsd:attribute name="interface" type="Tclass" />
                    <xsd:attribute name="filter" type="xsd:normalizedString" />
                    <xsd:attribute name="component-name" type="Tidref" />
                    <xsd:attribute name="availability" type="Tavailability" />
                    <xsd:attribute name="member-type" type="Tservice-use"
                        default="service-object" />
                    <xsd:anyAttribute namespace="##other"
                        processContents="lax" />
                </xsd:restriction>
            </xsd:complexContent>
        </xsd:complexType>

        <!-- Reference base class -->
        <xsd:complexType name="TserviceReference">
            <xsd:annotation>
                <xsd:documentation>
                    <![CDATA[
                    TserviceReference is the base element type used for <reference>
                    and <reference-list> elements.  This type defines all of the
                    characteristics common to both sorts of references.
                    ]]>
                </xsd:documentation>
            </xsd:annotation>
            <xsd:complexContent>
                <xsd:extension base="Tcomponent">
                    <xsd:sequence>
                        <xsd:group ref="GserviceReferenceElements" />
```

```
                        </xsd:sequence>

                        <xsd:attribute name="interface" type="Tclass">
                            <xsd:annotation>
                                <xsd:documentation>
                                    <![CDATA[
                                    The interface that the OSGi service must implement and that will be
                                    implemented by the proxy object.
                                    This attribute is optional.
                                    ]]>
                                </xsd:documentation>
                            </xsd:annotation>
                        </xsd:attribute>
                        <xsd:attribute name="filter" type="xsd:normalizedString">
                            <xsd:annotation>
                                <xsd:documentation>
                                    <![CDATA[
                                    A filter string used to narrow the search for a matching service
                                    reference.
                                    ]]>
                                </xsd:documentation>
                            </xsd:annotation>
                        </xsd:attribute>
                        <xsd:attribute name="component-name" type="Tidref">
                            <xsd:annotation>
                                <xsd:documentation>
                                    <![CDATA[
                                    An optional specifier that can be used to match a service definition
                                    to one created by a specific blueprint component.
                                    ]]>
                                </xsd:documentation>
                            </xsd:annotation>
                        </xsd:attribute>
                        <xsd:attribute name="availability" type="Tavailability">
                            <xsd:annotation>
                                <xsd:documentation>
                                    <![CDATA[
                                    Use to control the initial processing of service references at
                                    blueprint context startup.  "mandatory" indicates the context
                                    should not start unless the service is available within the
                                    specified context startup period.  "optional" indicates availability
                                    of this service is not a requirement at bundle startup.

                                    NOTE:  No default is specified because this can be overridden
                                    by the default-availability attribute of the <blueprint> element.
                                    ]]>
                                </xsd:documentation>
                            </xsd:annotation>
                        </xsd:attribute>
                        <xsd:anyAttribute namespace="##other"
                            processContents="lax" />
                    </xsd:extension>
                </xsd:complexContent>
            </xsd:complexType>

            <xsd:group name="GserviceReferenceElements">
                <xsd:sequence>
                    <xsd:element name="description" type="Tdescription"
                        minOccurs="0" />
                    <!-- listener -->
                    <xsd:element name="reference-listener" type="TreferenceListener"
                        minOccurs="0" maxOccurs="unbounded">
                        <xsd:annotation>
                            <xsd:documentation>
                                <![CDATA[
                                A definition of a listener that will watch for bind/unbind events
                                associated with the service reference.  The targetted listener can
                                be a <ref> to a <bean> or <reference> element, or an inline
                                <bean> or <reference>.
                                ]]>
                            </xsd:documentation>
                        </xsd:annotation>
                    </xsd:element>
                </xsd:sequence>
            </xsd:group>
```

```
<xsd:complexType name="TreferenceListener">
    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[
            TReferenceListener defines a reference listener that is attached
            to a <reference> or <reference-list> element.  The listener
            object can be specified as a <ref> or as an inline <bean> or
            <reference> component.  Listener events are mapped to the indicated
            bind or unbind methods.
            ]]>
        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
        <xsd:group ref="GtargetComponent" minOccurs="0" />
    </xsd:sequence>
    <xsd:attribute name="ref" type="Tidref" />
    <xsd:attribute name="bind-method" type="Tmethod" />
    <xsd:attribute name="unbind-method" type="Tmethod" />
</xsd:complexType>

<xsd:simpleType name="Tactivation">
    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[
            Tactivation defines the activation type for components.  This is used in this
            schema by the <blueprint> default-activation attribute and the
            activation attribute.
            ]]>
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:NMTOKEN">
        <xsd:enumeration value="eager" />
        <xsd:enumeration value="lazy" />
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="Tavailability">
    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[
            Tavailability defines an availability attribute type.  This is used in this
            schema by the <blueprint> default-availability attribute and the
            <reference> and <reference-list> availability attribute.
            ]]>
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:NMTOKEN">
        <xsd:enumeration value="mandatory" />
        <xsd:enumeration value="optional" />
    </xsd:restriction>
</xsd:simpleType>

<!-- service -->

<xsd:complexType name="Tservice">
    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[
            Tservice is the type for services exported by this blueprint bundle.
            Services are sourced by either a <ref> to a <bean> component or an
            <inline> bean component.
            ]]>
        </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
        <xsd:extension base="Tcomponent">
            <xsd:sequence>
                <xsd:group ref="GserviceElements" />
            </xsd:sequence>
            <xsd:attribute name="interface" type="Tclass">
                <xsd:annotation>
                    <xsd:documentation>
                        <![CDATA[
                        The interface that this OSGi service will provide.
```

```
                                    ]]>
                                </xsd:documentation>
                            </xsd:annotation>
                        </xsd:attribute>
                        <xsd:attribute name="ref" type="Tidref">
                            <xsd:annotation>
                                <xsd:documentation>
                                    <![CDATA[
                                    The ref attribute can be used to specify the component that provides
                                    the object exported as an OSGi service.
                                    ]]>
                                </xsd:documentation>
                            </xsd:annotation>
                        </xsd:attribute>
                        <xsd:attribute name="auto-export" type="TautoExportModes"
                            default="disabled">
                            <xsd:annotation>
                                <xsd:documentation>
                                    <![CDATA[
                                    If set to a value different from "disabled", the Blueprint Container
                                    will introspect the target to discover the set of interfaces or classes
                                    that the service will be registered under.
                                    ]]>
                                </xsd:documentation>
                            </xsd:annotation>
                        </xsd:attribute>
                        <xsd:attribute name="ranking" type="xsd:int" default="0">
                            <xsd:annotation>
                                <xsd:documentation>
                                    <![CDATA[
                                    A service ranking value that is added to the service properties
                                    the service will be published with.
                                    ]]>
                                </xsd:documentation>
                            </xsd:annotation>
                        </xsd:attribute>
                        <xsd:anyAttribute namespace="##other"
                            processContents="lax" />
                    </xsd:extension>
                </xsd:complexContent>
            </xsd:complexType>

            <xsd:complexType name="Tinlined-service">
                <xsd:annotation>
                    <xsd:documentation>
                        <![CDATA[
                        The Tinlined-service type is used for inlined (i.e. non top level)
                        <service> elements.
                        ]]>
                    </xsd:documentation>
                </xsd:annotation>
                <xsd:complexContent>
                    <xsd:restriction base="Tservice">
                        <xsd:sequence>
                            <xsd:group ref="GserviceElements" />
                        </xsd:sequence>
                        <xsd:attribute name="id" use="prohibited" />
                        <xsd:attribute name="depends-on" type="TdependsOn" />
                        <xsd:attribute name="activation" use="prohibited"
                            fixed="lazy" />
                        <xsd:attribute name="interface" type="Tclass" />
                        <xsd:attribute name="ref" type="Tidref" />
                        <xsd:attribute name="auto-export" type="TautoExportModes"
                            default="disabled" />
                        <xsd:attribute name="ranking" type="xsd:int" default="0" />
                        <xsd:anyAttribute namespace="##other"
                            processContents="lax" />
                    </xsd:restriction>
                </xsd:complexContent>
            </xsd:complexType>

            <xsd:group name="GbaseServiceElements">
                <xsd:sequence>
                    <xsd:element name="description" type="Tdescription"
                        minOccurs="0" />
```

```
        <xsd:element name="interfaces" type="Tinterfaces"
            minOccurs="0">
            <xsd:annotation>
                <xsd:documentation>
                    <![CDATA[
                    A collection of one or more interface class names this service
                    will be registered under.  The <service> element also has
                    a shortcut interface attribute for the usual case of just
                    a single interface being used.  This also cannot be used if
                    the auto-export attribute is used.
                    ]]>
                </xsd:documentation>
            </xsd:annotation>
        </xsd:element>

        <xsd:element name="service-properties" type="TserviceProperties"
            minOccurs="0">
            <xsd:annotation>
                <xsd:documentation>
                    <![CDATA[
                    The service provided when the service is registered.  The service
                    properties are similar to map elements, but the keys must always
                    be strings, and the values are required to be in a narrower range.
                    ]]>
                </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
        <xsd:element name="registration-listener" type="TregistrationListener"
            minOccurs="0" maxOccurs="unbounded">
            <xsd:annotation>
                <xsd:documentation>
                    <![CDATA[
                    A set of 0 or more registration listeners attached to this service
                    component.  The registration listeners will be notified whenever the
                    service is registered or unregistered from the framework service
                    registry.
                    ]]>
                </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
    </xsd:sequence>
</xsd:group>

<xsd:group name="GserviceElements">
    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[
            A set of service elements.
            ]]>
        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
        <xsd:group ref="GbaseServiceElements" />
        <xsd:group ref="GtargetComponent" minOccurs="0">
            <xsd:annotation>
                <xsd:documentation>
                    <![CDATA[
                    A service definition can use any of the target types as an inline element
                    as well.
                    ]]>
                </xsd:documentation>
            </xsd:annotation>
        </xsd:group>
    </xsd:sequence>
</xsd:group>

<xsd:complexType name="TregistrationListener">
    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[
            A registration listener definition.  The target registration listener
            can be either a <ref> to a <bean> or <service> component, or an inline
            <bean> or <service> component definition.  The registration-method and
            unregistration-method attributes define the methods that will be called
            for the respective events.
```

```
                    For the very common case of using a <ref> to a listener component, the
                    ref attribute may also be used as a shortcut.
                    ]]>
                </xsd:documentation>
            </xsd:annotation>
            <xsd:sequence>
                <xsd:group ref="GtargetComponent" minOccurs="0" />
            </xsd:sequence>
            <xsd:attribute name="ref" type="Tidref" />
            <xsd:attribute name="registration-method" type="Tmethod" />
            <xsd:attribute name="unregistration-method" type="Tmethod" />
        </xsd:complexType>

        <!-- Values -->

        <xsd:group name="Gvalue">
            <xsd:annotation>
                <xsd:documentation>
                    <![CDATA[
                    The set of "value" types that can be used in any place a value
                    can be specified.  This set includes the <ref> and <idref> elements, any of the
                    component types (<bean>, <service>, etc.) as inline components, the
                    generic <value> element for types sourced from string values, any of the
                    collection types (<set>, <list>, <array>, <map>, <props>), and the
                    <null> type to inject a null value.
                    ]]>
                </xsd:documentation>
            </xsd:annotation>
            <xsd:choice>
                <xsd:group ref="GnonNullValue" />
                <xsd:element name="null" type="Tnull" />
            </xsd:choice>
        </xsd:group>

        <xsd:complexType name="Tnull">
            <xsd:annotation>
                <xsd:documentation>
                    <![CDATA[
                    The definition for a <null> value type.
                    ]]>
                </xsd:documentation>
            </xsd:annotation>
        </xsd:complexType>

        <xsd:group name="GnonNullValue">
            <xsd:annotation>
                <xsd:documentation>
                    <![CDATA[
                    The set of "value" types that can be used in any place a non-null value
                    can be specified.  This set includes the <ref> and <idref> elements, any of the
                    component types (<bean>, <service>, etc.) as inline components, the
                    generic <value> element for types sourced from string values, and any of the
                    collection types (<set>, <list>, <array>, <map>, <props>).

                    The <null> type is NOT a member of this group.
                    ]]>
                </xsd:documentation>
            </xsd:annotation>
            <xsd:choice>
                <xsd:group ref="GallComponents" />
                <xsd:element name="idref" type="Tref" />
                <xsd:element name="value" type="Tvalue" />
                <xsd:element name="list" type="Tcollection" />
                <xsd:element name="set" type="Tcollection" />
                <xsd:element name="map" type="Tmap" />
                <xsd:element name="array" type="Tcollection" />
                <xsd:element name="props" type="Tprops" />
            </xsd:choice>
        </xsd:group>

        <xsd:complexType name="Tref">
            <xsd:annotation>
                <xsd:documentation>
                    <![CDATA[
```

```
                Tref is the type used for <ref> elements.  This specifies a required
                component id for the reference component.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:attribute name="component-id" type="Tidref" use="required" />
    </xsd:complexType>

    <xsd:complexType name="Tvalue" mixed="true">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                Tvalue is the type used for <value> elements.  The <value> element
                is used for types that can be created from a single string value.
                The string value is the data value for the element.  The optional
                type attribute allows a target conversion value to be explicitly
                specified.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:attribute name="type" type="Ttype" />
    </xsd:complexType>

    <!-- Collection Values -->

    <xsd:complexType name="TtypedCollection">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                TtypeCollection defines comment attributes shared among different
                collection types that allow a default value type to be specified.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:attribute name="value-type" type="Ttype" />
    </xsd:complexType>

    <xsd:complexType name="Tcollection">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                Tcollection is the base schema type for different ordered collection
                types.  This is shared between the <array>, <list>, and <set> elements.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="TtypedCollection">
                <xsd:group ref="Gvalue" minOccurs="0" maxOccurs="unbounded" />
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="Tprops">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                Tprops is the type used by the <props> value element.  The prop elements
                are pairs of string-valued keys and values.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
            <xsd:element name="prop" type="Tprop" minOccurs="0"
                maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="Tprop" mixed="true">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                Tprop is a single property element for a <props> value type.  The property
                value can be specified using either the attribute, or as value data for
                the property element.
```

```
                    ]]>
                </xsd:documentation>
            </xsd:annotation>
            <xsd:attribute name="key" type="TstringValue" use="required" />
            <xsd:attribute name="value" type="TstringValue" />
        </xsd:complexType>


    <!-- 'map' element type -->
    <xsd:complexType name="Tmap">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                Tmap is the base type used for <map> elements.  A map may have a
                default value type specified, so it inherits from the TtypeCollection
                type.  A key type can also be specified, and the map members are
                created from the entry elements, which require a key/value pair.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="TtypedCollection">
                <xsd:sequence>
                    <xsd:element name="entry" type="TmapEntry" minOccurs="0"
                        maxOccurs="unbounded" />
                </xsd:sequence>
                <xsd:attribute name="key-type" type="Ttype" />
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>


    <!-- 'entry' element type -->
    <xsd:complexType name="TmapEntry">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                TmapEntry is used for <entry> elements nested inside of a <map> element.
                Each <entry> instance defines a key/value pair that will be added to the
                Map.  Both the keys and values may be arbitrary types.  Keys must not
                be <null> but <null> is permitted for entry values.  A default type
                can be specified for both the keys and the values, but individual keys
                or values can override the default.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
            <xsd:element name="key" type="Tkey" minOccurs="0" />
            <xsd:group ref="Gvalue" minOccurs="0" />
        </xsd:sequence>
        <xsd:attribute name="key" type="TstringValue" />
        <xsd:attribute name="key-ref" type="Tidref" />
        <xsd:attribute name="value" type="TstringValue" />
        <xsd:attribute name="value-ref" type="Tidref" />
    </xsd:complexType>

    <!-- 'service property' element type -->
    <xsd:complexType name="TserviceProperties">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                TserviceProperty is used for <service-properties> elements.
                The syntax is similar to what is defined for <map>, but keys must be
                string values and there are no type defaults that can be specified.
                created from the entry elements, which require a key/value pair.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
            <xsd:element name="entry" type="TservicePropertyEntry"
                minOccurs="0" maxOccurs="unbounded" />
            <xsd:any namespace="##other" processContents="lax"
                minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>


    <!-- 'entry' element type -->
```

```
<xsd:complexType name="TservicePropertyEntry">
    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[
            TservicePropertyEntry is an entry value used for the <service-properties>
            element.  This does not allow a child <key> element and there are no
            key-ref or value-ref attributes.
            ]]>
        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
        <xsd:group ref="Gvalue" minOccurs="0" />
    </xsd:sequence>
    <xsd:attribute name="key" type="TstringValue" use="required" />
    <xsd:attribute name="value" type="TstringValue" />
</xsd:complexType>

<!-- General types -->

<xsd:complexType name="Tdescription" mixed="true">
    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[
            A generic <description> element type to allow documentation to added to the
            blueprint configuration.
            ]]>
        </xsd:documentation>
    </xsd:annotation>
    <xsd:choice minOccurs="0" maxOccurs="unbounded" />
</xsd:complexType>

<xsd:complexType name="Tinterfaces">
    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[
            The type definition for the <interfaces> element used for <service>
            ]]>
        </xsd:documentation>
    </xsd:annotation>
    <xsd:choice minOccurs="1" maxOccurs="unbounded">
        <xsd:element name="value" type="TinterfaceValue" />
    </xsd:choice>
</xsd:complexType>

<xsd:simpleType name="TinterfaceValue">
    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[
            TinterfaceValue is used for subelements of the <interfaces> element.
            This is just a <value>xxxxx</value> element where the contained
            value is the name of an interface class.
            ]]>
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="Tclass" />
</xsd:simpleType>

<xsd:simpleType name="Tclass">
    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[
            Tclass is a base type that should be used for all attributes that
            refer to java class names.
            ]]>
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:NCName" />
</xsd:simpleType>

<xsd:simpleType name="Ttype">
    <xsd:annotation>
        <xsd:documentation>
            <![CDATA[
            Ttype is a base type that refer to java types such as classes or
            arrays.
```

```
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:restriction base="xsd:token">
            <xsd:pattern value="[\i-[:]][\c-[:]]*(\[\])*" />
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="Tmethod">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                Tmethod is a base type that should be used for all attributes that
                refer to java method names.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:restriction base="xsd:NCName" />
    </xsd:simpleType>

    <!--
    Should be used for all attributes and elements that refer to method
    names
    -->
    <xsd:simpleType name="Tidref">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                Tidref is a base type that should be used for all attributes that
                refer to component ids.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:restriction base="xsd:NCName" />
    </xsd:simpleType>

    <xsd:simpleType name="TstringValue">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                TstringValue is a base type that should be used for all attributes that
                refer to raw string values
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:restriction base="xsd:normalizedString" />
    </xsd:simpleType>

    <xsd:simpleType name="TautoExportModes">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                TautoExportModes is a base type that should be used for export-mode
                attributes.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:restriction base="xsd:NMTOKEN">
            <xsd:enumeration value="disabled" />
            <xsd:enumeration value="interfaces" />
            <xsd:enumeration value="class-hierarchy" />
            <xsd:enumeration value="all-classes" />
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="Ttimeout">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                Ttimeout is a base type that should be used for all attributes that
                specify timeout values
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:restriction base="xsd:unsignedLong" />
```

```
        </xsd:simpleType>

    <xsd:simpleType name="TdependsOn">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                TdependsOn is a base type that should be used for all attributes that
                specify depends-on relationships
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:restriction>
            <xsd:simpleType>
                <xsd:list itemType="Tidref" />
            </xsd:simpleType>
            <xsd:minLength value="1" />
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="Tscope">
        <xsd:union>
            <xsd:simpleType>
                <xsd:restriction base="xsd:NMTOKEN">
                    <xsd:enumeration value="singleton" />
                    <xsd:enumeration value="prototype" />
                </xsd:restriction>
            </xsd:simpleType>
            <xsd:simpleType>
                <xsd:restriction base="xsd:QName">
                    <xsd:pattern value=".+:.+" />
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:union>
    </xsd:simpleType>

    <xsd:simpleType name="Tservice-use">
        <xsd:annotation>
            <xsd:documentation>
                <![CDATA[
                Indicates the type of object that will be placed within the
                reference collection.  "service-object" indicates the
                collection contains blueprint proxies for imported services.
                "service-reference" indicates the collection contains
                ServiceReference objects matching the target service type.
                ]]>
            </xsd:documentation>
        </xsd:annotation>
        <xsd:restriction base="xsd:NMTOKEN">
            <xsd:enumeration value="service-object" />
            <xsd:enumeration value="service-reference" />
        </xsd:restriction>
    </xsd:simpleType>

</xsd:schema>
```

121.16   Security

121.16.1   Blueprint Extender

A Blueprint Extender must use the Bundle Context of the Blueprint bundle. This will ensure that much of the resources allocated will be used on behalf of the Blueprint bundle. However, most Java 2 permissions will also verify the stack and this will inevitably include the Blueprint extender's code. Therefore, the Blueprint extender will require the combined set of permissions needed by all Blueprint bundles. It is therefore likely that in practical situations the Blueprint extender requires All Permission.

The Blueprint bundle requires permission for all actions that are done by the Blueprint Container on behalf of this bundle. That is, the Blueprint Container must not give any extra permissions to the Blueprint bundle because it is being extended.

A Blueprint Container must therefore use a `doPriviliged` block around all actions that execute code on behalf of the Blueprint bundle. This `doPrivileged` block must use an Access Control Context that represents the permissions of the Blueprint bundle.

For example, if a Blueprint bundle defines the following bean:

```
<bean class="java.lang.System" factory-method="exit">
    <argument value="1"/>
</bean>
```

Then the Blueprint bundle must have the proper permission to exit the system or the Blueprint bundle must fail when the bean is constructed. At the same time, a Blueprint bundle must not be required to have any permission needed by the Blueprint Container to performs its tasks.

A Blueprint Container must never use the `setAccessibility` method on a returned member. Only publicly accessible members must be used. Using a non-publicly accessible member must initiate failure, resulting in the destruction of the container.

121.16.2   Blueprint Bundle

A Blueprint Bundle must have all the permissions required by its code. There is one additional permission required for the Blueprint Bundle. The Blueprint extender will register a Blueprint Container service on behalf of the Blueprint bundle, and the Blueprint bundle must therefore have:

```
ServicePermission(...BlueprintContainer, [REGISTER])
```

121.17   org.osgi.service.blueprint.container

Blueprint Container Package Version 1.0.

This package defines the primary interface to a Blueprint Container, `BlueprintContainer`. An instance of this type is available inside a Blueprint Container as an implicitly defined component with the name "blueprintContainer".

This package also declares the supporting exception types, listener, and constants for working with a Blueprint Container.

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

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.blueprint.container; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.blueprint.container; version="[1.0,1.1)"

121.17.1 Summary

- BlueprintContainer - A Blueprint Container represents the managed state of a Blueprint bundle.
- BlueprintEvent - A Blueprint Event.
- BlueprintListener - A BlueprintEvent Listener.
- ComponentDefinitionException - A Blueprint exception indicating that a component definition is in error.
- Converter - Type converter to convert an object to a target type.
- EventConstants - Event property names used in Event Admin events published by a Blueprint Container.
- NoSuchComponentException - A Blueprint exception indicating that a component does not exist in a Blueprint Container.
- ReifiedType - Provides access to a concrete type and its optional generic type parameters.
- ServiceUnavailableException - A Blueprint exception indicating that a service is unavailable.

121.17.2 public interface BlueprintContainer

A Blueprint Container represents the managed state of a Blueprint bundle. A Blueprint Container provides access to all managed components. These are the beans, services, and service references. Only bundles in the ACTIVE state (and also the STARTING state for bundles awaiting lazy activation) can have an associated Blueprint Container. A given Bundle Context has at most one associated Blueprint Container. A Blueprint Container can be obtained by injecting the predefined "blueprint-Container" component id. The Blueprint Container is also registered as a service and its managed components can be queried.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

121.17.2.1 public Set<String> getComponentIds()

☐ Returns the set of component ids managed by this Blueprint Container.

Returns An immutable Set of Strings, containing the ids of all of the components managed within this Blueprint Container.

121.17.2.2 public Object getComponentInstance(String id)

id The component id for the requested component instance.

☐ Return the component instance for the specified component id. If the component's manager has not yet been activated, calling this operation will atomically activate it. If the component has singleton scope, the activation will cause the component instance to be created and initialized. If the component has prototype scope, then each call to this method will return a new component instance.

Returns A component instance for the component with the specified component id.

Throws NoSuchComponentException– If no component with the specified component id is managed by this Blueprint Container.

121.17.2.3 public ComponentMetadata getComponentMetadata(String id)

id The component id for the requested Component Metadata.

☐ Return the Component Metadata object for the component with the specified component id.

Returns   The Component Metadata object for the component with the specified component id.

Throws   `NoSuchComponentException`– If no component with the specified component id is managed by this Blueprint Container.

121.17.2.4   **public Collection<T> getMetadata(Class<T> type)**

Type Arguments   `<T extends ComponentMetadata>`

⟨T⟩   Type of Component Metadata.

type   The super type or type of the requested Component Metadata objects.

☐   Return all ComponentMetadata objects of the specified Component Metadata type. The supported Component Metadata types are ComponentMetadata (which returns the Component Metadata for all defined manager types), BeanMetadata , ServiceReferenceMetadata (which returns both ReferenceMetadata and ReferenceListMetadata objects), and ServiceMetadata. The collection will include all Component Metadata objects of the requested type, including components that are declared inline.

Returns   An immutable collection of Component Metadata objects of the specified type.

121.17.3   **public class BlueprintEvent**

A Blueprint Event.

`BlueprintEvent` objects are delivered to all registered BlueprintListener services. Blueprint Events must be asynchronously delivered in chronological order with respect to each listener.

In addition, after a Blueprint Listener is registered, the Blueprint extender will synchronously send to this Blueprint Listener the last Blueprint Event for each ready Blueprint bundle managed by this extender. This *replay* of Blueprint Events is designed so that the new Blueprint Listener can be informed of the state of each Blueprint bundle. Blueprint Events sent during this replay will have the isReplay() flag set. The Blueprint extender must ensure that this replay phase does not interfere with new Blueprint Events so that the chronological order of all Blueprint Events received by the Blueprint Listener is preserved. If the last Blueprint Event for a given Blueprint bundle is DESTROYED, the extender must not send it during this replay phase.

A type code is used to identify the type of event. The following event types are defined:

- CREATING
- CREATED
- DESTROYING
- DESTROYED
- FAILURE
- GRACE_PERIOD
- WAITING

In addition to calling the registered BlueprintListener services, the Blueprint extender must also send those events to the Event Admin service, if it is available.

See Also   BlueprintListener, EventConstants

Concurrency   Immutable

121.17.3.1   **public static final int CREATED = 2**

The Blueprint extender has created a Blueprint Container for the bundle. This event is sent after the Blueprint Container has been registered as a service.

121.17.3.2   **public static final int CREATING = 1**

The Blueprint extender has started creating a Blueprint Container for the bundle.

121.17.3.3   **public static final int DESTROYED = 4**

The Blueprint Container for the bundle has been completely destroyed. This event is sent after the Blueprint Container has been unregistered as a service.

121.17.3.4   **public static final int DESTROYING = 3**

The Blueprint extender has started destroying the Blueprint Container for the bundle.

121.17.3.5   **public static final int FAILURE = 5**

The Blueprint Container creation for the bundle has failed. If this event is sent after a timeout in the Grace Period, the getDependencies() method must return an array of missing mandatory dependencies. The event must also contain the cause of the failure as a `Throwable` through the getCause() method.

121.17.3.6   **public static final int GRACE_PERIOD = 6**

The Blueprint Container has entered the grace period. The list of missing dependencies must be made available through the getDependencies() method. During the grace period, a GRACE_PERIOD event is sent each time the set of unsatisfied dependencies changes.

121.17.3.7   **public static final int WAITING = 7**

The Blueprint Container is waiting on the availability of a service to satisfy an invocation on a referenced service. The missing dependency must be made available through the getDependencies() method which will return an array containing one filter object as a String.

121.17.3.8   **public BlueprintEvent(int type,Bundle bundle,Bundle extenderBundle)**

type   The type of this event.

bundle   The Blueprint bundle associated with this event. This parameter must not be null.

extenderBundle   The Blueprint extender bundle that is generating this event. This parameter must not be null.

☐   Create a simple `BlueprintEvent` object.

121.17.3.9   **public BlueprintEvent(int type,Bundle bundle,Bundle extenderBundle,String[] dependencies)**

type   The type of this event.

bundle   The Blueprint bundle associated with this event. This parameter must not be null.

extenderBundle   The Blueprint extender bundle that is generating this event. This parameter must not be null.

dependencies   An array of `String` filters for each dependency associated with this event. Must be a non-empty array for event types GRACE_PERIOD and WAITING. It is optional for event type FAILURE. Must be null for other event types.

☐   Create a `BlueprintEvent` object associated with a set of dependencies.

121.17.3.10   **public BlueprintEvent(int type,Bundle bundle,Bundle extenderBundle,Throwable cause)**

type   The type of this event.

bundle   The Blueprint bundle associated with this event. This parameter must not be null.

extenderBundle   The Blueprint extender bundle that is generating this event. This parameter must not be null.

cause   A `Throwable` object describing the root cause of the event. May be null.

☐   Create a `BlueprintEvent` object associated with a failure cause.

121.17.3.11   **public BlueprintEvent(int type,Bundle bundle,Bundle extenderBundle,String[] dependencies,Throwable cause)**

type   The type of this event.

bundle The Blueprint bundle associated with this event. This parameter must not be null.

extenderBundle The Blueprint extender bundle that is generating this event. This parameter must not be null.

dependencies An array of String filters for each dependency associated with this event. Must be a non-empty array for event types GRACE_PERIOD and WAITING. It is optional for event type FAILURE. Must be null for other event types.

cause A Throwable object describing the root cause of this event. May be null.

☐ Create a BlueprintEvent object associated with a failure cause and a set of dependencies.

121.17.3.12 **public BlueprintEvent(BlueprintEvent event,boolean replay)**

event The original BlueprintEvent to copy. Must not be null.

replay true if this event should be used as a replay event.

☐ Create a new BlueprintEvent from the specified BlueprintEvent. The timestamp property will be copied from the original event and only the replay property will be overridden with the given value.

121.17.3.13 **public Bundle getBundle()**

☐ Return the Blueprint bundle associated with this event.

Returns The Blueprint bundle associated with this event.

121.17.3.14 **public Throwable getCause()**

☐ Return the cause for this FAILURE event.

Returns The cause of the failure for this event. May be null.

121.17.3.15 **public String[] getDependencies()**

☐ Return the filters identifying the missing dependencies that caused this event.

Returns The filters identifying the missing dependencies that caused this event if the event type is one of WAITING, GRACE_PERIOD or FAILURE or null for the other event types.

121.17.3.16 **public Bundle getExtenderBundle()**

☐ Return the Blueprint extender bundle that is generating this event.

Returns The Blueprint extender bundle that is generating this event.

121.17.3.17 **public long getTimestamp()**

☐ Return the time at which this event was created.

Returns The time at which this event was created.

121.17.3.18 **public int getType()**

☐ Return the type of this event.

The type values are:

- CREATING
- CREATED
- DESTROYING
- DESTROYED
- FAILURE
- GRACE_PERIOD
- WAITING

Returns The type of this event.

121.17.3.19 **public boolean isReplay()**

☐ Return whether this event is a replay event.

Returns true if this event is a replay event and false otherwise.

121.17.4 **public interface BlueprintListener**

A BlueprintEvent Listener.

To receive Blueprint Events, a bundle must register a Blueprint Listener service. After a Blueprint Listener is registered, the Blueprint extender must synchronously send to this Blueprint Listener the last Blueprint Event for each ready Blueprint bundle managed by this extender. This replay of Blueprint Events is designed so that the new Blueprint Listener can be informed of the state of each Blueprint bundle. Blueprint Events sent during this replay will have the isReplay() flag set. The Blueprint extender must ensure that this replay phase does not interfere with new Blueprint Events so that the chronological order of all Blueprint Events received by the Blueprint Listener is preserved. If the last Blueprint Event for a given Blueprint bundle is DESTROYED, the extender must not send it during this replay phase.

See Also BlueprintEvent

Concurrency Thread-safe

121.17.4.1 **public void blueprintEvent(BlueprintEvent event)**

event The BlueprintEvent.

☐ Receives notifications of a Blueprint Event. Implementers should quickly process the event and return.

121.17.5 **public class ComponentDefinitionException**
extends RuntimeException

A Blueprint exception indicating that a component definition is in error. This exception is thrown when a configuration-related error occurs during creation of a Blueprint Container.

121.17.5.1 **public ComponentDefinitionException()**

☐ Creates a Component Definition Exception with no message or exception cause.

121.17.5.2 **public ComponentDefinitionException(String explanation)**

explanation The associated message.

☐ Creates a Component Definition Exception with the specified message

121.17.5.3 **public ComponentDefinitionException(String explanation,Throwable cause)**

explanation The associated message.

cause The cause of this exception.

☐ Creates a Component Definition Exception with the specified message and exception cause.

121.17.5.4 **public ComponentDefinitionException(Throwable cause)**

cause The cause of this exception.

☐ Creates a Component Definition Exception with the exception cause.

121.17.6 **public interface Converter**

Type converter to convert an object to a target type.

Concurrency Thread-safe

121.17.6.1   **public boolean canConvert(Object sourceObject,ReifiedType targetType)**

sourceObject   The source object s to convert.

targetType   The target type T.

□   Return if this converter is able to convert the specified object to the specified type.

Returns   true if the conversion is possible, false otherwise.

121.17.6.2   **public Object convert(Object sourceObject,ReifiedType targetType) throws Exception**

sourceObject   The source object s to convert.

targetType   The target type T.

□   Convert the specified object to an instance of the specified type.

Returns   An instance with a type that is assignable from targetType's raw class

Throws   Exception– If the conversion cannot succeed. This exception should not be thrown when the can-Convert method has returned true.

121.17.7   public class EventConstants

Event property names used in Event Admin events published by a Blueprint Container.

Each type of event is sent to a different topic:

org/osgi/service/blueprint/container/ ⟨*event-type*⟩

where ⟨*event-type*⟩ can have the values CREATING, CREATED, DESTROYING, DESTROYED, FAILURE, GRACE_PERIOD, or WAITING.

Such events have the following properties:

- type
- event
- timestamp
- bundle
- bundle.symbolicName
- bundle.id
- bundle.version
- extender.bundle.symbolicName
- extender.bundle.id
- extender.bundle.version
- dependencies
- cause

Concurrency   Immutable

121.17.7.1   **public static final String BUNDLE = "bundle"**

The Blueprint bundle associated with this event. This property is of type Bundle.

121.17.7.2   **public static final String BUNDLE_ID = "bundle.id"**

The bundle id of the Blueprint bundle associated with this event. This property is of type Long.

121.17.7.3   **public static final String BUNDLE_SYMBOLICNAME = "bundle.symbolicName"**

The bundle symbolic name of the Blueprint bundle associated with this event. This property is of type String.

121.17.7.4   **public static final String BUNDLE_VERSION = "bundle.version"**

The bundle version of the Blueprint bundle associated with this event. This property is of type Version.

121.17.7.5   **public static final String CAUSE = "cause"**

The cause for a FAILURE event. This property is of type Throwable.

121.17.7.6   **public static final String DEPENDENCIES = "dependencies"**

The filters identifying the missing dependencies that caused this event for a FAILURE, GRACE_PERIOD, or WAITING event. This property type is an array of String.

121.17.7.7   **public static final String EVENT = "event"**

The BlueprintEvent object that caused this event. This property is of type BlueprintEvent.

121.17.7.8   **public static final String EXTENDER_BUNDLE = "extender.bundle"**

The Blueprint extender bundle that is generating this event. This property is of type Bundle.

121.17.7.9   **public static final String EXTENDER_BUNDLE_ID = "extender.bundle.id"**

The bundle id of the Blueprint extender bundle that is generating this event. This property is of type Long.

121.17.7.10   **public static final String EXTENDER_BUNDLE_SYMBOLICNAME = "extender.bundle.symbolicName"**

The bundle symbolic of the Blueprint extender bundle that is generating this event. This property is of type String.

121.17.7.11   **public static final String EXTENDER_BUNDLE_VERSION = "extender.bundle.version"**

The bundle version of the Blueprint extender bundle that is generating this event. This property is of type Version.

121.17.7.12   **public static final String TIMESTAMP = "timestamp"**

The time the event was created. This property is of type Long.

121.17.7.13   **public static final String TOPIC_BLUEPRINT_EVENTS = "org/osgi/service/blueprint/container"**

Topic prefix for all events issued by the Blueprint Container

121.17.7.14   **public static final String TOPIC_CREATED = "org/osgi/service/blueprint/container/CREATED"**

Topic for Blueprint Container CREATED events

121.17.7.15   **public static final String TOPIC_CREATING = "org/osgi/service/blueprint/container/CREATING"**

Topic for Blueprint Container CREATING events

121.17.7.16   **public static final String TOPIC_DESTROYED = "org/osgi/service/blueprint/container/DESTROYED"**

Topic for Blueprint Container DESTROYED events

121.17.7.17   **public static final String TOPIC_DESTROYING = "org/osgi/service/blueprint/container/DESTROYING"**

Topic for Blueprint Container DESTROYING events

121.17.7.18   **public static final String TOPIC_FAILURE = "org/osgi/service/blueprint/container/FAILURE"**

Topic for Blueprint Container FAILURE events

121.17.7.19   **public static final String TOPIC_GRACE_PERIOD = "org/osgi/service/blueprint/container/GRACE_PERIOD"**

Topic for Blueprint Container GRACE_PERIOD events

121.17.7.20   **public static final String TOPIC_WAITING = "org/osgi/service/blueprint/container/WAITING"**

Topic for Blueprint Container WAITING events

121.17.7.21   **public static final String TYPE = "type"**

The type of the event that has been issued. This property is of type Integer and can take one of the values defined in BlueprintEvent.

121.17.8   public class NoSuchComponentException extends RuntimeException

A Blueprint exception indicating that a component does not exist in a Blueprint Container. This exception is thrown when an attempt is made to create a component instance or lookup Component Metadata using a component id that does not exist in the Blueprint Container.

121.17.8.1   **public NoSuchComponentException(String msg,String id)**

msg   The associated message.

id   The id of the non-existent component.

☐   Create a No Such Component Exception for a non-existent component.

121.17.8.2   **public NoSuchComponentException(String id)**

id   The id of the non-existent component.

☐   Create a No Such Component Exception for a non-existent component.

121.17.8.3   **public String getComponentId()**

☐   Returns the id of the non-existent component.

Returns   The id of the non-existent component.

121.17.9   public class ReifiedType

Provides access to a concrete type and its optional generic type parameters.

Java 5 and later support generic types. These types consist of a raw class with type parameters. This class models such a Type class but ensures that the type is *reified*. Reification means that the Type graph associated with a Java 5 Type instance is traversed until the type becomes a concrete class. This class is available with the getRawClass() method. The optional type parameters are recursively represented as Reified Types.

In Java 1.4, a class has by definition no type parameters. This class implementation provides the Reified Type for Java 1.4 by making the raw class the Java 1.4 class and using a Reified Type based on the Object class for any requested type parameter.

A Blueprint extender implementations can subclass this class and provide access to the generic type parameter graph for conversion. Such a subclass must *reify* the different Java 5 Type instances into the reified form. That is, a form where the raw Class is available with its optional type parameters as Reified Types.

Concurrency   Immutable

121.17.9.1   **public ReifiedType(Class<?> clazz)**

clazz   The raw class of the Reified Type.

◻ Create a Reified Type for a raw Java class without any generic type parameters. Subclasses can provide the optional generic type parameter information. Without subclassing, this instance has no type parameters.

121.17.9.2   **public ReifiedType getActualTypeArgument(int i)**

i   The zero-based index of the requested type parameter.

◻ Return a type parameter for this type. The type parameter refers to a parameter in a generic type declaration given by the zero-based index i. For example, in the following example:

```
Map<String, ? extends Metadata>
```

type parameter 0 is String, and type parameter 1 is Metadata.

This implementation returns a Reified Type that has Object as class. Any object is assignable to Object and therefore no conversion is then necessary. This is compatible with versions of Java language prior to Java 5. This method should be overridden by a subclass that provides access to the generic type parameter information for Java 5 and later.

Returns   The ReifiedType for the generic type parameter at the specified index.

121.17.9.3   **public Class<?> getRawClass()**

◻ Return the raw class represented by this type. The raw class represents the concrete class that is associated with a type declaration. This class could have been deduced from the generics type parameter graph of the declaration. For example, in the following example:

```
Map<String, ? extends Metadata>
```

The raw class is the Map class.

Returns   The raw class represented by this type.

121.17.9.4   **public int size()**

◻ Return the number of type parameters for this type.

This implementation returns 0. This method should be overridden by a subclass that provides access to the generic type parameter information for Java 5 and later.

Returns   The number of type parameters for this type.

121.17.10   public class ServiceUnavailableException
extends ServiceException

A Blueprint exception indicating that a service is unavailable. This exception is thrown when an invocation is made on a service reference and a backing service is not available.

121.17.10.1   **public ServiceUnavailableException(String message,String filter)**

message   The associated message.

filter   The filter used for the service lookup.

◻ Creates a Service Unavailable Exception with the specified message.

121.17.10.2   **public ServiceUnavailableException(String message,String filter,Throwable cause)**

message   The associated message.

filter   The filter used for the service lookup.

cause   The cause of this exception.

       ☐ Creates a Service Unavailable Exception with the specified message and exception cause.

121.17.10.3   **public String getFilter()**

       ☐ Returns the filter expression that a service would have needed to satisfy in order for the invocation to proceed.

Returns The failing filter.

121.18   org.osgi.service.blueprint.reflect

Blueprint Reflection Package Version 1.0.

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

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.blueprint.reflect; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.blueprint.reflect; version="[1.0,1.1)"

121.18.1   Summary

- `BeanArgument` - Metadata for a factory method or constructor argument of a bean.
- `BeanMetadata` - Metadata for a Bean component.
- `BeanProperty` - Metadata for a property to be injected into a bean.
- `CollectionMetadata` - Metadata for a collection based value.
- `ComponentMetadata` - Metadata for managed components.
- `IdRefMetadata` - Metadata for the verified id of another component managed by the Blueprint Container.
- `MapEntry` - Metadata for a map entry.
- `MapMetadata` - Metadata for a Map based value.
- `Metadata` - Top level Metadata type.
- `NonNullMetadata` - Metadata for a value that cannot null.
- `NullMetadata` - Metadata for a value specified to be null via the ‹null› element.
- `PropsMetadata` - Metadata for a java.util.Properties based value.
- `ReferenceListener` - Metadata for a reference listener interested in the reference bind and unbind events for a service reference.
- `ReferenceListMetadata` - Metadata for a list of service references.
- `ReferenceMetadata` - Metadata for a reference that will bind to a single matching service in the service registry.
- `RefMetadata` - Metadata for a reference to another component managed by the Blueprint Container.
- `RegistrationListener` - Metadata for a registration listener interested in service registration and unregistration events for a service.
- `ServiceMetadata` - Metadata for a service to be registered by the Blueprint Container when enabled.
- `ServiceReferenceMetadata` - Metadata for a reference to an OSGi service.
- `Target` - A common interface for managed components that can be used as a direct target for method calls.

- `ValueMetadata` - Metadata for a simple `String` value that will be type-converted if necessary before injecting.

121.18.2   public interface BeanArgument

Metadata for a factory method or constructor argument of a bean. The arguments of a bean are obtained from BeanMetadata.getArguments(). This is specified by the `argument` elements of a bean.

Concurrency Thread-safe

121.18.2.1   public int getIndex()

□ Return the zero-based index into the parameter list of the factory method or constructor to be invoked for this argument. This is determined by specifying the `index` attribute for the bean. If not explicitly set, this will return -1 and the initial ordering is defined by its position in the BeanMetadata.getArguments() list. This is specified by the `index` attribute.

Returns The zero-based index of the parameter, or -1 if no index is specified.

121.18.2.2   public Metadata getValue()

□ Return the Metadata for the argument value. This is specified by the `value` attribute.

Returns The Metadata for the argument value.

121.18.2.3   public String getValueType()

□ Return the name of the value type to match the argument and convert the value into when invoking the constructor or factory method. This is specified by the `type` attribute.

Returns The name of the value type to convert the value into, or `null` if no type is specified.

121.18.3   public interface BeanMetadata
extends Target, ComponentMetadata

Metadata for a Bean component.

This is specified by the `bean` element.

Concurrency Thread-safe

121.18.3.1   public static final String SCOPE_PROTOTYPE = "prototype"

The bean has `prototype` scope.

See Also getScope()

121.18.3.2   public static final String SCOPE_SINGLETON = "singleton"

The bean has `singleton` scope.

See Also getScope()

121.18.3.3   public List<BeanArgument> getArguments()

□ Return the arguments for the factory method or constructor of the bean. This is specified by the child `argument` elements.

Returns An immutable List of BeanArgument objects for the factory method or constructor of the bean. The List is empty if no arguments are specified for the bean.

121.18.3.4   public String getClassName()

□ Return the name of the class specified for the bean. This is specified by the `class` attribute of the bean definition.

Returns The name of the class specified for the bean. If no class is specified in the bean definition, because the a factory component is used instead, then this method will return null.

121.18.3.5 **public String getDestroyMethod()**

☐ Return the name of the destroy method specified for the bean. This is specified by the de-stroy-method attribute of the bean definition.

Returns The name of the destroy method specified for the bean, or null if no destroy method is specified.

121.18.3.6 **public Target getFactoryComponent()**

☐ Return the Metadata for the factory component on which to invoke the factory method for the bean. This is specified by the factory-ref attribute of the bean.

When a factory method and factory component have been specified for the bean, this method returns the factory component on which to invoke the factory method for the bean. When no factory component has been specified this method will return null. When a factory method has been specified for the bean but a factory component has not been specified, the factory method must be invoked as a static method on the bean's class.

Returns The Metadata for the factory component on which to invoke the factory method for the bean or null if no factory component is specified.

121.18.3.7 **public String getFactoryMethod()**

☐ Return the name of the factory method for the bean. This is specified by the factory-method attribute of the bean.

Returns The name of the factory method of the bean or null if no factory method is specified for the bean.

121.18.3.8 **public String getInitMethod()**

☐ Return the name of the init method specified for the bean. This is specified by the init-method attribute of the bean definition.

Returns The name of the init method specified for the bean, or null if no init method is specified.

121.18.3.9 **public List<BeanProperty> getProperties()**

☐ Return the properties for the bean. This is specified by the child property elements.

Returns An immutable List of BeanProperty objects, with one entry for each property to be injected in the bean. The List is empty if no property injection is specified for the bean.

121.18.3.10 **public String getScope()**

☐ Return the scope for the bean.

Returns The scope for the bean. Returns null if the scope has not been explicitly specified in the bean definition.

See Also SCOPE_SINGLETON, SCOPE_PROTOTYPE

121.18.4 public interface BeanProperty

Metadata for a property to be injected into a bean. The properties of a bean are obtained from BeanMetadata.getProperties(). This is specified by the property elements of a bean. Properties are defined according to the Java Beans conventions.

Concurrency Thread-safe

121.18.4.1 **public String getName()**

☐ Return the name of the property to be injected. The name follows Java Beans conventions. This is specified by the name attribute.

Returns   The name of the property to be injected.

121.18.4.2   **public Metadata getValue()**

☐ Return the Metadata for the value to be injected into a bean. This is specified by the value attribute or in inlined text.

Returns   The Metadata for the value to be injected into a bean.

121.18.5   public interface CollectionMetadata
extends NonNullMetadata

Metadata for a collection based value. Values of the collection are defined by Metadata objects. This Collection Metadata can constrain the values of the collection to a specific type.

Concurrency   Thread-safe

121.18.5.1   **public Class<?> getCollectionClass()**

☐ Return the type of the collection. The possible types are: array (Object[]), Set, and List. This information is specified in the element name.

Returns   The type of the collection. Object[] is returned to indicate an array.

121.18.5.2   **public List<Metadata> getValues()**

☐ Return Metadata for the values of the collection.

Returns   A List of Metadata for the values of the collection.

121.18.5.3   **public String getValueType()**

☐ Return the type specified for the values of the collection. The value-type attribute specified this information.

Returns   The type specified for the values of the collection.

121.18.6   public interface ComponentMetadata
extends NonNullMetadata

Metadata for managed components. This is the base type for BeanMetadata, ServiceMetadata and ServiceReferenceMetadata.

Concurrency   Thread-safe

121.18.6.1   **public static final int ACTIVATION_EAGER = 1**

The component's manager must eagerly activate the component.

See Also   getActivation()

121.18.6.2   **public static final int ACTIVATION_LAZY = 2**

The component's manager must lazily activate the component.

See Also   getActivation()

121.18.6.3   **public int getActivation()**

☐ Return the activation strategy for the component. This is specified by the activation attribute of a component definition. If this is not set, then the default-activation in the blueprint element is used. If that is also not set, then the activation strategy is ACTIVATION_EAGER.

Returns   The activation strategy for the component.

See Also   ACTIVATION_EAGER, ACTIVATION_LAZY

121.18.6.4   **public List<String> getDependsOn()**

☐ Return the ids of any components listed in a depends-on attribute for the component.

Returns An immutable List of component ids that are explicitly declared as a dependency, or an empty List if none.

121.18.6.5   **public String getId()**

☐ Return the id of the component.

Returns The id of the component. The component id can be null if this is an anonymously defined and/or inlined component.

121.18.7   **public interface IdRefMetadata**
extends NonNullMetadata

Metadata for the verified id of another component managed by the Blueprint Container. The id itself will be injected, not the component to which the id refers. No implicit dependency is created.

Concurrency Thread-safe

121.18.7.1   **public String getComponentId()**

☐ Return the id of the referenced component. This is specified by the component-id attribute of a component.

Returns The id of the referenced component.

121.18.8   **public interface MapEntry**

Metadata for a map entry. This type is used by MapMetadata, PropsMetadata and ServiceMetadata.

Concurrency Thread-safe

121.18.8.1   **public NonNullMetadata getKey()**

☐ Return the Metadata for the key of the map entry. This is specified by the key attribute or element.

Returns The Metadata for the key of the map entry. This must not be null.

121.18.8.2   **public Metadata getValue()**

☐ Return the Metadata for the value of the map entry. This is specified by the value attribute or element.

Returns The Metadata for the value of the map entry. This must not be null.

121.18.9   **public interface MapMetadata**
extends NonNullMetadata

Metadata for a Map based value.

This is specified by the map element.

Concurrency Thread-safe

121.18.9.1   **public List<MapEntry> getEntries()**

☐ Return the entries for the map.

Returns An immutable List of MapEntry objects for each entry in the map. The List is empty if no entries are specified for the map.

121.18.9.2   **public String getKeyType()**

☐ Return the name of the type of the map keys. This is specified by the key-type attribute of the map.

Returns   The name of the type of the map keys, or null if none is specified.

121.18.9.3   **public String getValueType()**

☐   Return the name of the type of the map values. This is specified by the value-type attribute of the map.

Returns   The name of the type of the map values, or null if none is specified.

121.18.10   public interface Metadata

Top level Metadata type. All Metadata types extends this base type.

Concurrency   Thread-safe

121.18.11   public interface NonNullMetadata
extends Metadata

Metadata for a value that cannot null. All Metadata subtypes extend this type except for NullMetadata.

This Metadata type is used for keys in Maps because they cannot be null.

Concurrency   Thread-safe

121.18.12   public interface NullMetadata
extends Metadata

Metadata for a value specified to be null via the ‹null› element.

Concurrency   Thread-safe

121.18.12.1   **public static final NullMetadata NULL**

Singleton instance of NullMetadata.

121.18.13   public interface PropsMetadata
extends NonNullMetadata

Metadata for a java.util.Properties based value.

The MapEntry objects of properties are defined with keys and values of type String.

This is specified by the props element.

Concurrency   Thread-safe

121.18.13.1   **public List‹MapEntry› getEntries()**

☐   Return the entries for the properties.

Returns   An immutable List of MapEntry objects for each entry in the properties. The List is empty if no entries are specified for the properties.

121.18.14   public interface ReferenceListener

Metadata for a reference listener interested in the reference bind and unbind events for a service reference.

Concurrency   Thread-safe

121.18.14.1   **public String getBindMethod()**

☐   Return the name of the bind method. The bind method will be invoked when a matching service is bound to the reference. This is specified by the bind-method attribute of the reference listener.

Returns The name of the bind method.

121.18.14.2 **public Target getListenerComponent()**

☐ Return the Metadata for the component that will receive bind and unbind events. This is specified by the ref attribute or via an inlined component.

Returns The Metadata for the component that will receive bind and unbind events.

121.18.14.3 **public String getUnbindMethod()**

☐ Return the name of the unbind method. The unbind method will be invoked when a matching service is unbound from the reference. This is specified by the unbind-method attribute of the reference listener.

Returns The name of the unbind method.

121.18.15 public interface ReferenceListMetadata extends ServiceReferenceMetadata

Metadata for a list of service references.

This is specified by the reference-list element.

Concurrency Thread-safe

121.18.15.1 **public static final int USE_SERVICE_OBJECT = 1**

Reference list values must be proxies to the actual service objects.

See Also getMemberType()

121.18.15.2 **public static final int USE_SERVICE_REFERENCE = 2**

Reference list values must be ServiceReference objects.

See Also getMemberType()

121.18.15.3 **public int getMemberType()**

☐ Return whether the List will contain service object proxies or ServiceReference objects. This is specified by the member-type attribute of the reference list.

Returns Whether the List will contain service object proxies or ServiceReference objects.

See Also USE_SERVICE_OBJECT, USE_SERVICE_REFERENCE

121.18.16 public interface ReferenceMetadata extends Target, ServiceReferenceMetadata

Metadata for a reference that will bind to a single matching service in the service registry.

This is specified by the reference element.

Concurrency Thread-safe

121.18.16.1 **public long getTimeout()**

☐ Return the timeout for service invocations when a backing service is unavailable. This is specified by the timeout attribute of the reference.

Returns The timeout, in milliseconds, for service invocations when a backing service is unavailable.

121.18.17 public interface RefMetadata extends Target, NonNullMetadata

Metadata for a reference to another component managed by the Blueprint Container.

Concurrency Thread-safe

121.18.17.1   **public String getComponentId()**

☐ Return the id of the referenced component. This is specified by the component-id attribute of a component.

Returns The id of the referenced component.

121.18.18   public interface RegistrationListener

Metadata for a registration listener interested in service registration and unregistration events for a service.

The registration listener is called with the initial state of the service when the registration listener is actuated.

Concurrency Thread-safe

121.18.18.1   **public Target getListenerComponent()**

☐ Return the Metadata for the component that will receive registration and unregistration events. This is specified by the ref attribute or via an inlined component.

Returns The Metadata for the component that will receive registration and unregistration events.

121.18.18.2   **public String getRegistrationMethod()**

☐ Return the name of the registration method. The registration method will be invoked when the associated service is registered with the service registry. This is specified by the registration-method attribute of the registration listener.

Returns The name of the registration method.

121.18.18.3   **public String getUnregistrationMethod()**

☐ Return the name of the unregistration method. The unregistration method will be invoked when the associated service is unregistered from the service registry. This is specified by the unregistration-method attribute of the registration listener.

Returns The name of the unregistration method.

121.18.19   public interface ServiceMetadata
extends ComponentMetadata

Metadata for a service to be registered by the Blueprint Container when enabled.

This is specified by the service element.

Concurrency Thread-safe

121.18.19.1   **public static final int AUTO_EXPORT_ALL_CLASSES = 4**

Advertise all Java classes and interfaces in the component instance type as service interfaces.

See Also getAutoExport()

121.18.19.2   **public static final int AUTO_EXPORT_CLASS_HIERARCHY = 3**

Advertise all Java classes in the hierarchy of the component instance type as service interfaces.

See Also getAutoExport()

121.18.19.3   **public static final int AUTO_EXPORT_DISABLED = 1**

Do not auto-detect types for advertised service interfaces

See Also   getAutoExport()

121.18.19.4   **public static final int AUTO_EXPORT_INTERFACES = 2**

Advertise all Java interfaces implemented by the component instance type as service interfaces.

See Also   getAutoExport()

121.18.19.5   **public int getAutoExport()**

☐   Return the auto-export mode for the service. This is specified by the `auto-export` attribute of the service.

Returns   The auto-export mode for the service.

See Also   AUTO_EXPORT_DISABLED, AUTO_EXPORT_INTERFACES, AUTO_EXPORT_CLASS_HIERARCHY, AUTO_EXPORT_ALL_CLASSES

121.18.19.6   **public List<String> getInterfaces()**

☐   Return the type names of the interfaces that the service should be advertised as supporting. This is specified in the `interface` attribute or child `interfaces` element of the service.

Returns   An immutable List of String for the type names of the interfaces that the service should be advertised as supporting. The List is empty if using `auto-export` or no interface names are specified for the service.

121.18.19.7   **public int getRanking()**

☐   Return the ranking value to use when advertising the service. If the ranking value is zero, the service must be registered without a `service.ranking` service property. This is specified by the `ranking` attribute of the service.

Returns   The ranking value to use when advertising the service.

121.18.19.8   **public Collection<RegistrationListener> getRegistrationListeners()**

☐   Return the registration listeners to be notified when the service is registered and unregistered with the framework. This is specified by the `registration-listener` elements of the service.

Returns   An immutable Collection of RegistrationListener objects to be notified when the service is registered and unregistered with the framework. The Collection is empty if no registration listeners are specified for the service.

121.18.19.9   **public Target getServiceComponent()**

☐   Return the Metadata for the component to be exported as a service. This is specified inline or via the `ref` attribute of the service.

Returns   The Metadata for the component to be exported as a service.

121.18.19.10   **public List<MapEntry> getServiceProperties()**

☐   Return the user declared properties to be advertised with the service. This is specified by the `service-properties` element of the service.

Returns   An immutable List of MapEntry objects for the user declared properties to be advertised with the service. The List is empty if no service properties are specified for the service.

121.18.20   public interface ServiceReferenceMetadata extends ComponentMetadata

Metadata for a reference to an OSGi service. This is the base type for ReferenceListMetadata and ReferenceMetadata.

Concurrency   Thread-safe

121.18.20.1   **public static final int AVAILABILITY_MANDATORY = 1**

A matching service is required at all times.

See Also   getAvailability()

121.18.20.2   **public static final int AVAILABILITY_OPTIONAL = 2**

A matching service is not required to be present.

See Also   getAvailability()

121.18.20.3   **public int getAvailability()**

☐ Return whether or not a matching service is required at all times. This is specified in the availability attribute of the service reference.

Returns   Whether or not a matching service is required at all times.

See Also   AVAILABILITY_MANDATORY, AVAILABILITY_OPTIONAL

121.18.20.4   **public String getComponentName()**

☐ Return the value of the component-name attribute of the service reference. This specifies the id of a component that is registered in the service registry. This will create an automatic filter, appended with the filter if set, to select this component based on its automatic id attribute.

Returns   The value of the component-name attribute of the service reference or null if the attribute is not specified.

121.18.20.5   **public String getFilter()**

☐ Return the filter expression that a matching service must match. This is specified by the filter attribute of the service reference.

Returns   The filter expression that a matching service must match or null if a filter is not specified.

121.18.20.6   **public String getInterface()**

☐ Return the name of the interface type that a matching service must support. This is specified in the interface attribute of the service reference.

Returns   The name of the interface type that a matching service must support or null when no interface name is specified.

121.18.20.7   **public Collection<ReferenceListener> getReferenceListeners()**

☐ Return the reference listeners to receive bind and unbind events. This is specified by the reference-listener elements of the service reference.

Returns   An immutable Collection of ReferenceListener objects to receive bind and unbind events. The Collection is empty if no reference listeners are specified for the service reference.

121.18.21   public interface Target
extends NonNullMetadata

A common interface for managed components that can be used as a direct target for method calls. These are bean, reference, and ref, where the ref must refer to a bean or reference component.

See Also   BeanMetadata, ReferenceMetadata, RefMetadata

Concurrency   Thread-safe

121.18.22   public interface ValueMetadata
extends NonNullMetadata

Metadata for a simple String value that will be type-converted if necessary before injecting.

Concurrency Thread-safe

121.18.22.1   **public String getStringValue()**

□ Return the unconverted string representation of the value. This is specified by the value attribute or text part of the value element.

Returns The unconverted string representation of the value.

121.18.22.2   **public String getType()**

□ Return the name of the type to which the value should be converted. This is specified by the type attribute.

Returns The name of the type to which the value should be converted or null if no type is specified.

121.19   References

[1]   *Spring Framework*
      http://www.springsource.org/

[2]   *Spring Dynamic Modules*
      http://www.springsource.org/osgi

[3]   *Guice*
      http://code.google.com/p/google-guice/

[4]   *Picocontainer*
      http://www.picocontainer.org/

[5]   *Java Beans Specification*
      http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138795.html

[6]   *XML Namespaces*
      http://www.w3.org/TR/REC-xml-names

[7]   *Properties format*
      http://docs.oracle.com/javase/1.4.2/docs/api/java/util/Properties.html#load%28java.io.InputStream%29

[8]   *XML Schema*
      http://www.w3.org/XML/Schema

[9]   *OSGi XML Schemas*
      http://www.osgi.org/Specifications/HomePage

122 Remote Service Admin Service Specification

Version 1.1

122.1 Introduction

The *OSGi Core Release 6* framework specifies a model where bundles can use distributed services. The basic model for OSGi remote services is that a bundle can register services that are *exported* to a communication *Endpoint* and use services that are *imported* from a communication Endpoint. However, chapter *Remote Services* on page 21 does not explain *what* services are exported and/or imported; it leaves such decisions to the distribution provider. The distribution provider therefore performs multiple roles and cannot be leveraged by other bundles in scenarios that the distribution provider had not foreseen.

The primary role of the distribution provider is purely mechanical; it creates Endpoints and registers service proxies and enables their communication. The second role is about the policies around the desired topology. The third role is discovery. To establish a specific topology it is necessary to find out about exported services in other frameworks.

This specification therefore defines an API for the distribution provider and discovery of services in a network. A management agent can use this API to provide an actual distribution policy. This management agent, called the Topology Manager, can control the export and import of services delegating the intrinsic knowledge of the low level details of communication protocols, proxying of services, and discovering services in the network to services defined in this specification.

This specification is an extension of the Remote Service chapter. Though some aspects are repeated in this specification, a full understanding of the Remote Services chapter is required for full understanding of this document.

122.1.1 Essentials
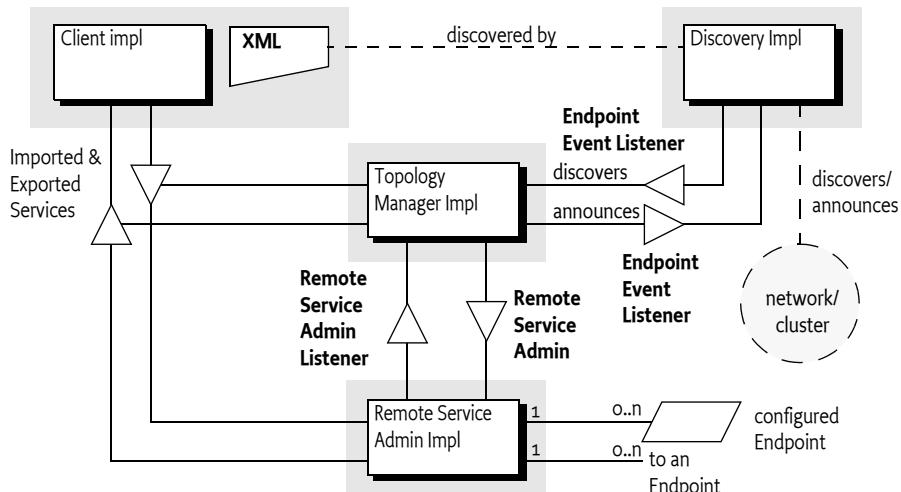
- *Simple* - Make it as simple as possible for a Topology Manager to implement distribution policies.
- *Dynamic* - Discover available Endpoints dynamically, for example through a discovery protocol like [3] *Service Location Protocol (SLP)* or [4] *JGroups*.
- *Inform* - Provide a mechanism to inform other parties about created and removed Endpoints.
- *Configuration* - Allow bundles to describe Endpoints as a bundle resource that are provided to the Distribution Provider.
- *Selective* - Not all parties are interested in all services. Endpoint registries must be able to express the scope of services they are interested in.
- *Multiple* - Allow the collaboration of multiple Topology Managers, Remote Service Admin services, and Discovery Providers.
- *Dynamic* - Allow the dynamic discovery of Endpoints.
- *Federated* - Enable a global view of all available services in a distributed environment.

122.1.2 Entities

- *Remote Service Admin* - An implementation of this specification provides the mechanisms to import and export services through a set of configuration types. The Remote Service Admin service is a passive Distribution Provider, not taking any action to export or import itself.
- *Topology Manager* - The Topology Manager provides the policy for importing and exporting services through the Remote Service Admin service.
- *Endpoint* - An Endpoint is a communications access mechanism to a service in another framework, a (web) service, another process, or a queue or topic destination, etc., requiring some protocol for communications.
- *Endpoint Description* - A properties based description of an Endpoint. Endpoint Descriptions can be exchanged between different frameworks to create connections to each other's services. Endpoint Descriptions can also be created to Endpoints not originating in an OSGi Framework.
- *Endpoint Description Provider* - A party that can inform others about the existence of Endpoints.
- *Endpoint Event Listener* – A listener service that receives events relating to Endpoints that match its scope. This Endpoint Event Listener is used symmetrically to implement a federated registry. The Topology Manager can use it to notify interested parties about created and removed Endpoints, as well as to receive notifications from other parties, potentially remote, about their available Endpoints.
- *Endpoint Listener* – An older version of the Endpoint Event Listener defined by version 1.0 of this specification. The Endpoint Event Listener supersedes the Endpoint Listener, and should be used in preference where possible.
- *Remote Service Admin Listener* - A listener service that is informed of all the primitive actions that the Remote Service Admin performs like importing and exporting as well as errors.
- *Endpoint Configuration Extender* - A bundle that can detect configuration data describing an Endpoint Description in a bundle resource, using the extender pattern.
- *Discovery* – An Endpoint Event Listener that detects the Endpoint Descriptions through some discovery protocol.
- *Cluster* - A group of computing systems that closely work together, usually in a fast network.

Figure 122.1   *Remote Service Admin Entities*

122.1.3 Synopsis

Topology Managers are responsible for the distribution policies of a OSGi framework. To implement a policy, a Topology Manager must be aware of the environment, for this reason, it can register:

- Service listeners to detect services that can be exported according to the Remote Services chapter.
- Listener and Find Hook services to detect bundles that have an interest in specific services that potentially could be imported.
- A Remote Service Admin Listener service to detect the activity of other Topology Managers.
- Endpoint Event Listener and Endpoint Listener services to detect Endpoints that are made available through discovery protocols, configuration data, or other means.

Using this information, the manager implements a topology using the Remote Service Admin service. A Topology Manager that wants to export a service can create an *Export Registration* by providing one or more Remote Service Admin services a Service Reference plus a Map with the required properties. A Remote Service Admin service then creates a number of Endpoints based on the available configuration types and returns a collection of ExportRegistration objects. A collection is returned because a single service can be exported to multiple Endpoints depending on the available configuration type properties.

Each Export Registration is specific for the caller and represents an existing or newly created Endpoint. The Export Registration associates the exported Service Reference with an *Endpoint Description.* If there are problems with the export operation, the Remote Service Admin service reports these on the Export Registration objects. That is, not all the returned Export Registrations have to be valid.

An Endpoint Description is a property based description of an Endpoint. Some of these properties are defined in this specification, other properties are defined by configuration types. These configuration types must follow the same rules as the configuration types defined in the Remote Services chapter. Remote Service Admin services that support the configuration types in the Endpoint Description can import a service from that Endpoint solely based on that Endpoint Description.

In similar vein, the Topology Manager can import a service from a remote system by creating an Import Registration out of an Endpoint Description. The Remote Service Admin service then registers a service that is a proxy for the remote Endpoint and returns an ImportRegistration object. If there are problems with the import, the Remote Service Admin service that cannot be detected early, then the Remote Service Admin service reports these on the returned ImportRegistration object.

For introspection, the Remote Service Admin can list its current set of Import and Export References so that a Topology Manager can get the current state. The Remote Service Admin service also informs all Topology Managers and observers of the creation, deletion, and errors of Import and Export Registrations through the Remote Service Admin Listener service. Interested parties like the Topology Manager can register such a service and will be called back with the initial state as well as any subsequent changes.

An important aspect of the Topology Manager is the distributed nature of the scenarios it plays an orchestrating role in. A Topology Manager needs to be aware of Endpoints in the network, not just the ones provided by Remote Service Admin services in its local framework. The Endpoint Event Listener service is specified for this purpose. This service is provided for both directions, symmetrically. That is, it is used by the Topology Manager to inform any observers about the existence of Endpoints that are locally available, as well as for parties that represent a discovery mechanism. For example Endpoints available on other systems, Endpoint Descriptions embedded in resources in bundles, or Endpoint Descriptions that are available in some other form.

Endpoint Event Listener services are not always interested in the complete set of available Endpoints because this set can potentially be very large. For example, if a remote registry like [5] *UDDI* is used then the number of Endpoints can run into the thousands or more. An Endpoint Event Listener service can therefore scope the set of Endpoints with an OSGi LDAP style filter. Parties that can provide information about Endpoints must only notify Endpoint Event Listener services when the Endpoint Description falls within the scope of the Endpoint Listener service. Parties that use some discovery mechanism can use the scope to trigger directed searches across the network.

122.1.3.1   **Endpoint Listener Services**

The 1.0 version of this specification defined an Endpoint Listener service, which has an identical purpose and similar behaviors to an Endpoint Event Listener service. Unfortunately the design of the Endpoint Listener limited its extensibility, meaning that it had to be replaced in version 1.1 of this specification.

In order to maintain backward compatible interoperability with Remote Service Admin 1.0 actors, Remote Service Admin 1.1 actors must continue to register Endpoint Listener services as well as Endpoint Event Listener services. They must also continue to call Endpoint Listener services as well as EndpointEventListener services.

122.2   Actors
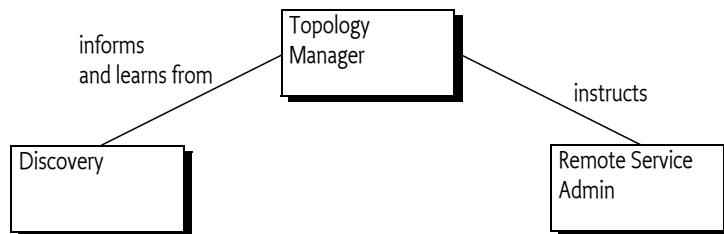
The OSGi Remote Services specification is about the distribution of services. This specification does not outline the details of how the distribution provider knows the desired topology, this policy aspect is left up to implementations. In many situations, this is a desirable architecture because it provides freedom of implementation to the distribution provider. However, such an architecture does not enable a separation of the mechanisms and *policy*. Therefore, this Remote Service Admin specification provides an architecture that enables a separate bundle from the distribution provider to define the topology. It splits the responsibility of the Remote Service specification in a number of *roles*. These roles can all have different implementations but they can collaborate through the services defined in this specification. These roles are:

- *Topology Manager*s - Topology Managers are the (anonymous) players that implement the policies for distributing services; they are closely aligned with the concept of an OSGi *management agent*. It is expected that Topology Managers will be developed for scenarios like import/export all applicable services, configuration based imports- and exports, and scenarios like fail-over, load-balancing, as well as standards like domain managers for the [6] *Service Component Architecture (SCA)*.
- *Remote Service Admin* - The Remote Service Admin service provides the basic mechanism to import and export services. This service is policy free; it will not distribute services without explicitly being told so. A OSGi framework can host multiple Remote Service Admin services that, for example, support different configuration types.
- *Discovery* - To implement a distribution policy, a Topology Manager must be aware of what Endpoints are available. This specification provides an abstraction of a *federated Endpoint registry*. This registry can be used to both publish as well as consume Endpoints from many different sources. The federated registry is defined for local services but is intended to be used with standard and proprietary service discovery protocols. The federated registry is implemented with the Endpoint Event Listener service.

These roles are depicted in Figure 122.2 on page 378.

Figure 122.2   *Roles*

122.3 Topology Managers

Distributed processing has become mainstream because of the massive scale required for Internet applications. Only with distributed architectures is it possible to scale systems to *Internet size* with hundreds of millions of users. To allow a system to scale, servers are grouped in clusters where they can work in unison or geographically dispersed in even larger configurations. The distribution of the work-load is crucial for the amount of scalability provided by an architecture and often has domain specific dispatching techniques. For example, the hash of a user id can be used to select the correct profile database server. In this fast moving world it is very unlikely that a single architecture or distribution policy would be sufficient to satisfy many users. It is therefore that this specification separates the *how* from the *what*. The complex mechanics of importing and exporting services are managed by a Remote Service Admin service (the how) while the different policies are implemented by Topology Managers (the what). This separation of concerns enables the development of Topology Managers that can run on many different systems, providing high user functionality. For example, a Topology Manager could implement a fail-over policy where some strategic services are redirected when their connections fail. Other Topology Managers could use a discovery protocol like SLP to find out about other systems in a cluster and automatically configure the cluster.

The key value of this architecture is demonstrated by the example of an *SCA domain controller*. An SCA domain controller receives a description of a domain (a set of systems and modules) and must ensure that the proper connections are made between the participating SCA modules. By splitting the roles, an SCA domain manager can be developed that can run on any compatible Remote Service Admin service implementation.

122.3.1 Multiple Topology Managers

There is no restriction on the number of Topology Managers, nor is there a restriction on the number of Remote Service Admin service implementations. It is up to the deployer of the OSGi framework to select the appropriate set of these service implementations. It is the responsibility of the Topology Managers to listen to the Remote Service Admin Listener and track Endpoints created and deleted by other Topology Managers and act appropriately.

122.3.2 Example Use Cases

122.3.2.1 Promiscuous Policy

A *cluster* is a set of machines that are connected in a network. The simplest policy for a Topology Manager is to share exported services in such a cluster. Such a policy is very easy to implement with the Remote Services Admin service. In the most basic form, this Topology Manager would use some multicast protocol to communicate with its peers. These peers would exchange EndpointDescription objects of exported services. Each Topology Manager would then import any exported service.

This scenario can be improved by separating the promiscuous policy from the discovery. Instead of embedding the multicast protocol, a Topology manager could use the Endpoint Event Listener service. This service allows the discovery of remote services. At the same time, the Topology Manager could tell all other Endpoint Event Listener services about the services it has created, allowing them to be used by others in the network.

Splitting the Topology Manager and discovery in two bundles allows different implementations of the discovery bundle, for example, to use different protocols. See PROMISCUOUS_POLICY.

122.3.2.2 Fail Over
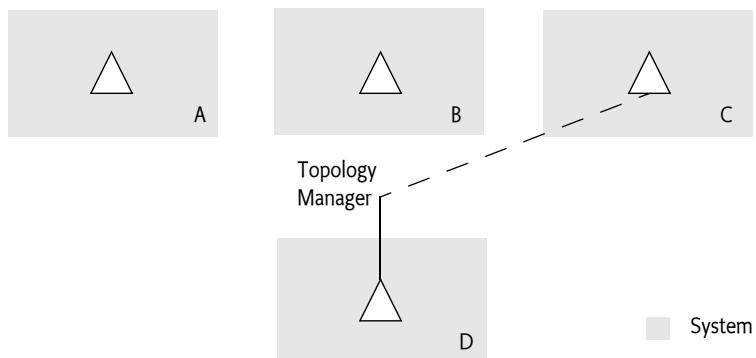
A more elaborate scheme is a *fail-over policy*. In such a policy a service can be replaced by a service from another machine. There are many ways to implement such a policy, an simple example strategy is provided here for illustration.

A Fail-Over Topology Manager is given a list of stateless services that require fail-over, for example through the *Configuration Admin Service Specification* on page 61. The Fail-Over Manager tracks the systems in the its cluster that provide such services. This tracking can use an embedded protocol or it can be based on the Endpoint Event Listener service model.

In the Fail-Over policy, the fail-over manager only imports a single service and then tracks the error status of the imported service through the Remote Service Admin Listener service. If it detects the service is becoming unavailable, it closes the corresponding Import Registration and imports a service from an alternative system instead. In Figure 122.3, there are 4 systems in a cluster. The topology/fail-over manager ensures that there is always one of the services in system A, B, or C available in D.

Figure 122.3   *Fail Over Scenario in a cluster*

There are many possible variations on this scenario. The managers could exchange load information, allowing the service switch to be influenced by the load of the target systems. The important aspect is that the Topology Manager can ignore the complex details of discovery protocols, communication protocols, and service proxying and instead focus on the topology. See FAIL_OVER_POLICY.

122.4 Endpoint Description

An *Endpoint* is a point of rendezvous of distribution providers. It is created by an exporting distribution provider or some other party, and is used by importing distribution providers to create a connection. An *Endpoint Description* describes an Endpoint in such a way that an importing Remote Service Admin service can create this connection if it recognizes the *configuration type* that is used for that Endpoint. The configuration type consists of a name and a set of properties associated with that name.

The core concept of the Endpoint Description is a Map of properties. The structure of this map is the same as service properties, and the defined properties are closely aligned with the properties of an imported service. An EndpointDescription object must only consist of the data types that are supported for service properties. This makes the property map serializable with many different mechanisms. The EndpointDescription class provides a convenient way to access the properties in a type safe way.

An Endpoint Description has case insensitive keys, just like the Service Reference's properties.

The properties map must contain all the prescribed service properties of the exported service after intents have been processed, as if the service was registered as an imported service. That is, the map must not contain any properties that start with service.exported.* but it must contain the service.imported.* variation of these properties. The Endpoint Description must reflect the imported service properties because this simplifies the use of filters from the service hooks. Filters applied to the Endpoint Description can then be the same filters as applied by a bundle to select an imported service from the service registry.

The properties that can be used in an Endpoint Description are listed in Table 122.1. The Remote-Constants class contains the constants for all of these property names.

Table 122.1　　　　*Endpoint Properties*

Endpoint Property Name	Type	Description
service.exported.*		Must not be set
service.imported	*	Must always be set to some value. See SERVICE_IMPORTED.
objectClass	String[]	Must be set to the value of service.exported.interfaces, of the exported service after expanding any wildcards. Though this property will be overridden by the framework for the corresponding service registration, it must be set in the Endpoint Description to simplify the filter matching. These interface names are available with the getInterfaces() method.
service.intents	String+	Intents implemented by the exporting distribution provider and, if applicable, the exported service itself. Any qualified intents must have their expanded form present. These expanded intents are available with the getIntents() method. See SERVICE_INTENTS.
endpoint.service.id	Long	The service id of the exported service. Can be absent or 0 if the corresponding Endpoint is not for an OSGi service. The remote service id is available as getServiceId(). See also ENDPOINT_SERVICE_ID.
endpoint.framework.uuid	String	A universally unique id identifying the instance of the exporting framework. Can be absent if the corresponding Endpoint is not for an OSGi service. See *Framework UUID* on page 383. The remote framework UUID is available with the getFrameworkUUID() method. See also ENDPOINT_FRAMEWORK_UUID.
endpoint.id	String	The Id for this Endpoint, can never be null. This information is available with the getId(). See *Endpoint Id* on page 383 and also ENDPOINT_ID.
endpoint.package. version.<package-name>	String	The Java package version for the embedded <package>. For example, the property endpoint.package.version.com.acme=1.3 describes the version for the com.acme package. The version for a package can be obtained with the getPackageVersion(String). The version does not have to be set, if not set, the value must be assumed to be 0.
service.imported.configs	String+	The configuration types that can be used to implement the corresponding Endpoint. This property maps to the corresponding property in the Remote Services chapter. This property can be obtained with the getConfigurationTypes() method. The Export Registration has all the possible configuration types, where the Import Registration reports the configuration type actually used. SERVICE_IMPORTED_CONFIGS.

Endpoint Property Name	Type	Description
<config>.*	*	Where <config> is one of the configuration type names listed in service.imported.configs. The content of these properties must be valid for creating a connection to the Endpoint in another framework. That is, any locally readable URLs from bundles must be converted in such a form that they can be read by the importing framework. How this is done is configuration type specific.
*	*	All remaining public service properties must be present (that is, not starting with full stop ('.' \u002E)). If the values can not be marshaled by the Distribution Provider then they must be ignored.

The EndpointDescription class has a number of constructors that make it convenient to instantiate it for different purposes:

- EndpointDescription(Map) - Instantiate the Endpoint Description from a Map object.
- EndpointDescription(ServiceReference,Map) - Instantiate an Endpoint Description based on a Service Reference and a Map. The base properties of this Endpoint Description are the Service Reference properties but the properties in the given Map must override any of their case variants in the Service Reference. This allows the construction of an Endpoint Description from an exportable service while still allowing overrides of specific properties by the Topology Manager.

The Endpoint Description must use the allowed properties as given in Table 122.1 on page 381. The Endpoint Description must automatically skip any service.exported.* properties.

The Endpoint Description provides the following methods to access the properties in a more convenient way:

- getInterfaces() - Answers a list of Java interface names. These are the interfaces under which the services must be registered. These interface names can also be found at the objectClass property. A service can only be imported when there is at least one Java interface name available.
- getConfigurationTypes() - Answer the configuration types that are used for exporting this Endpoint. The configuration types are associated with a number of properties.
- getId() - Returns an Id uniquely identifying an Endpoint. The syntax of this Id should be defined in the specification for the associated configuration type. Two Endpoint Descriptions with the same Id describe the same Endpoint.
- getFrameworkUUID() - Get a Universally Unique Identifier (UUID) for the framework instance that has created the Endpoint, *Framework UUID* on page 383.
- getServiceId() - Get the service id for the framework instance that has created the Endpoint. If there is no service on the remote side the value must be 0.
- getPackageVersion(String) - Get the version for the given package.
- getIntents() - Get the list of specified intents.
- getProperties() - Get all the properties.

Two Endpoint Descriptions are deemed equal when their Endpoint Id is equal. The Endpoint Id is a mandatory property of an Endpoint Description, it is further described at *Endpoint Id* on page 383. The hash code is therefore also based on the Endpoint Id.

122.4.1   Validity

A valid Endpoint Description must at least satisfy the following assertions:

- It must have a non-null Id that uniquely identifies the Endpoint

- It must at least have one Java interface name
- It must at least have one configuration type set
- Any version for the packages must have a valid version syntax.

122.4.2   Mutability

An EndpointDescription object is immutable and with all final fields. It can be freely used between different threads.

122.4.3   Endpoint Id

An Endpoint Id is an opaque unique identifier for an Endpoint. This uniqueness must at least hold for the entire network in which the Endpoint is used. There is no syntax defined for this string except that white space at the beginning and ending must be ignored. The actual syntax for this Endpoint Id must be defined by the actual configuration type.

Two Endpoint Descriptions are deemed identical when their Endpoint Id is equal. The Endpoint Ids must be compared as string compares with leading and trailing spaces removed. The Endpoint Description class must use the String class' hash Code from the Endpoint Id as its own hashCode.

The simplest way to ensure that a growth in the number of EndpointDescriptions and/or the size of the connected group does not violate the required uniqueness of Endpoint Ids is for implementations to make their Endpoint Ids globally unique. This protects against clashes regardless of changes to the connected group.

Whilst globally unique identifiers (GUIDs) are a simple solution to the Endpoint Id uniqueness problem, they are not easy to implement in all environments. In some systems they can be prohibitively expensive to create, or of insufficient entropy to be genuinely unique. Some distribution providers may therefore choose not to use random GUIDs.

In the case where no globally unique value is used the following actions are recommended (although not required).

- Distribution Providers protect against intra-framework clashes using some known value unique to the service, for example the service id.
- Distribution Providers protect against inter-provider collisions within a single framework by using some unique value, such as the distribution provider's bundle id. The distribution provider bundle's symbolic name is insufficient, as there may be multiple versions of the same distribution provider installed within a single framework.
- Distribution Providers protect against inter-framework collisions using some value unique to the framework, such as the framework UUID.

122.4.4   Framework UUID

Each framework registers its services with a service id that is only unique for that specific framework. The OSGi framework is not a singleton, making it possible that a single VM process holds multiple OSGi frameworks. Therefore, to identify an OSGi service uniquely it is necessary to identify the framework that has registered it. This identifier is a *Universally Unique IDentifier* (UUID) that is set for each framework. This UUID is contained in the following framework property:

```
org.osgi.framework.uuid
```

If an Endpoint Description has no associated OSGi service then the UUID of that Endpoint Description must not be set and its service id must be 0.

A local Endpoint Description will have its framework UUID set to the local framework. This makes it straightforward to filter for Endpoint Descriptions that are describing local Endpoints or that describe remote Endpoints. For example, a manager can take the filter from a listener and ensure that it is only getting remote Endpoint Descriptions:

```
(&
  (!
    (service.remote.framework.uuid
       =72dc5fd9-5f8f-4f8f-9821-9ebb433a5b72)
  )
  (objectClass=org.osgi.service.log.LogService)
)
```

Where `72dc5fd9-5f8f-4f8f-9821-9ebb433a5b72` is the UUID of the local framework. A discovery bundle can register the following filter in its scope to receive all locally generated Endpoints:

```
(service.remote.framework.uuid
       =72dc5fd9-5f8f-4f8f-9821-9ebb433a5b72)
```

122.4.5 Resource Containment

Configuration types can use URLs to point to local resources describing in detail the Endpoint parameters for specific protocols. However, the purpose of an Endpoint Description is to describe an Endpoint to a remote system. This implies that there is some marshaling process that will transfer the Endpoint Description to another process. This other process is unlikely to be able to access resource URLs. Local bundle resource URLs are only usable in the framework that originates them but even HTTP based URLs can easily run into problems due to firewalls or lack of routing.

Therefore, the properties for a configuration type should be stored in such a way that the receiving process can access them. One way to achieve this is to contain the configuration properties completely in the Endpoint Description and ensure they only use the basic data types that the remote services chapter in the core requires every Distribution Provider to support.

The Endpoint Description XML format provides an `xml` element that is specifically added to make it easy to embed XML based configuration documents. The XML Schema is defined in *Endpoint Description Extender Format* on page 395.

122.5 Remote Service Admin

The Remote Service Admin service abstracts the core functionality of a distribution provider: exporting a service to an Endpoint and importing services from an Endpoint. However, in contrast with the distribution provider of the Remote Services specification, the Remote Service Admin service must be told explicitly what services to import and export.

122.5.1 Exporting

An exportable service can be exported with the exportService(ServiceReference,Map) method. This method creates a number of Endpoints by inspecting the merged properties from the Service Reference and the given Map. Any property in the Map overrides the Service Reference properties, regardless of case. That is, if the map contains a key then it will override any case variant of this key in the Service Reference. However, if the Map contains the `objectClass` or `service.id` property key in any case variant, then these properties must not override the Service Reference's value.

The Remote Service Admin service must interpret the merged properties according to the Remote Services chapter. This means that it must look at the following properties (as defined in chapter *Remote Services* on page 21):

• service.exported.configs - (String+) A list of configuration types that should be used to export this service. Each configuration type represents the configuration parameters for an Endpoint. A Remote Service Admin service should create an Endpoint for each configuration type that it supports and ignore the types it does not recognize. If this property is not set, then the Remote Service Admin implementation must choose a convenient configuration type that then must be

reported on the Endpoint Description with the `service.imported.configs` associated with the returned Export Registration.

- `service.exported.intents` - (`String+`) A list of intents that the Remote Service Admin service must implement to distribute the given service.
- `service.exported.intents.extra` - (`String+`) This property is merged with the `service.exported.intents` property.
- `service.exported.interfaces` - (`String+`) This property must be set; it marks this service for export and defines the interfaces. The list members must all be contained in the types listed in the `objectClass` service property from the Service Reference. The single value of an asterisk ('`*`' \u002A) indicates all interfaces in the registration's `objectClass` property and ignore the classes. Being able to set this property outside the Service Reference implies that the Topology Manager can export any registered service, also services not specifically marked to be exported.
- `service.intents` - (`String+`) A list of intents that this service has implemented.

A Topology Manager cannot remove properties, `null` is invalid as a property value.

The Remote Service Admin returns a collection of `ExportRegistration` objects. This collection must contain an entry for each configuration type the Remote Service Admin has recognized. Unrecognized configuration types must be ignored. However, it is possible that this list contains *invalid registrations*, see *Invalid Registrations* on page 388.

If a Service was already exported then the Remote Service Admin must still return a new `ExportRegistration` object that is linked with the earlier registrations. That is, an Endpoint can be shared between multiple Export Registrations. The Remote Service Admin service must ensure that the corresponding Endpoint remains available as long as there is at least one open Export Registration for that Endpoint.

For each successful creation of an export registration, the Remote Service Admin service must publish an `EXPORT_REGISTRATION` event, see *Events* on page 393. This event must be emitted, even if the Endpoint already existed and is thus shared with another Export Registration. If the creation of an Endpoint runs into an error, an `EXPORT_ERROR` event must be emitted.

Each valid Export Registration corresponds to an Endpoint for the given service. This Endpoint must remain active until all of the Export Registrations are closed that share this Endpoint.

The Endpoint can now be published so that other processes or systems can import this Endpoint. To aid with this import, the Export Registration has a `getExportReference()` method that returns an `ExportReference` object. This reference provides the following information:

- `getExportedEndpoint()` - This is the associated Endpoint Description. This Endpoint Description is a properties based description of an Endpoint. The property keys and their semantics are outlined in *Endpoint Description* on page 380. It can be used to inform other systems of the availability of an Endpoint.
- `getExportedService()` - The Service Reference to the exported service.

Both methods must return `null` when the associated Export Registration is closed.

A Distribution Provider that recognizes the configuration type in an Endpoint can create a connection to an Endpoint on other systems as long as firewalls and networks permit. The Endpoint Description can therefore be communicated to other systems to announce the availability of an Endpoint. The Topology Manager can optionally announce the availability of an Endpoint to the Endpoint Event Listener services, see *Discovery* on page 389. The decision to announce the availability of an Endpoint is one of the policies that is provided by a specific Topology Manager.

The Export Registrations remain open until:

- Explicitly closed by the Topology Manager, or
- The Remote Service Admin service is no longer used by the Topology Manager that created the Export Registration.

If the Remote Service Admin service can no longer maintain the corresponding Endpoint due to failures than these should be reported through the events. However, the registrations should remain open until explicitly closed by the Topology Manager.

See *Registration Life Cycle* on page 388 for more information.

The Export Registrations are not permanent; persistence is in the realm of the Topology Manager.

122.5.2   Importing

To import a service, a Topology Manager must have an Endpoint Description that describes the Endpoint the imported service should connect to. With this Endpoint Description, a Remote Service Admin service can then import the corresponding Endpoint. A Topology Manager can obtain these Endpoint Descriptions through internal configuration; it can use the discovery model enabled by the Endpoint Event Listener service, see *Discovery* on page 389, or some alternate means.

A service can be imported with the Remote Service Admin importService(EndpointDescription) method. This method takes an Endpoint Description and picks one of the embedded configuration types to establish a connection with the corresponding Endpoint to create a local service proxy. This proxy can then be mapped to either a remote OSGi service or an alternative, for example a web service. In certain cases the service proxy can be lazy, only verifying the reachability of the Endpoint when it is actually invoked for the first time. This implies that a service proxy can block when invoked until the proper communication setup has taken place.

If the Remote Service Admin service does not recognize any of the configuration types then it must return null. If there are multiple configuration types recognized then the Remote Service Admin is free to select any one of the recognized types.

The Remote Service Admin service must ensure that service properties are according to the Remote Services chapter for an imported service. This means that it must register the following properties:

- service.imported - (*) Must be set to any value.
- service.imported.configs - (String+) The configuration information used to import this service. Any associated properties for this configuration types must be properly mapped to the importing system. For example, a URL in these properties must point to a valid resource when used in the importing framework, see *Resource Containment* on page 384. Multiple configuration types can be listed if they are synonyms for exactly the same Endpoint that is used to export this service.
- service.intents - (String+) The Remote Service Admin must set this property to convey the combined intents of:
    - The exporting service, and
    - The intents that the exporting distribution provider adds, and
    - The intents that the importing distribution provider adds.
- Any additional properties listed in the Endpoint Description that should not be excluded. See *Endpoint Description* on page 380 for more details about the properties in the Endpoint Description.

A Remote Service Admin service must strictly follow the rules for importing a service as outlined in the Remote Services chapter.

The Remote Service Admin must return an ImportRegistration object or null. Even if an Import Registration is returned, it can still be an *invalid registration*, see *Invalid Registrations* on page 388 if the setup of the connection failed asynchronously. The Import Registration must always be a new object. Each valid Import Registration corresponds to a proxy service, potentially shared, that was created for the given Endpoint. The issues around proxying are described in *Proxying* on page 388.

For each successful creation of an import registration, the Remote Service Admin service must publish an IMPORT_REGISTRATION event, if there is an error it must publish an IMPORT_ERROR, see *Events* on page 393.

For more information see *Registration Life Cycle* on page 388.

The Import Registration provides access to an ImportReference object with the getImportReference(). This object has the following methods:

- getImportedEndpoint() - Provides the Endpoint Description for this imported service.
- getImportedService() - Provides the Service Reference for the service proxy.

The Import Registration will remain open as long as:

- The corresponding remote Endpoint remains available, and
- The Remote Service Admin service is still in use by the Topology Manager that created the Import Registration.

That is, the Import Registrations are not permanent, any persistence is in the realm of the Topology Manager. See *Registration Life Cycle* on page 388 for more details.

122.5.3 Updates

Services Registrations are dynamic and service properties may change during the lifetime of a service. Remote services must mirror these dynamics without making it appear as though the service has become unavailable. This requires that the exporting distribution provider and the importing distribution provider support the changing of service properties.

There are two types of service properties:

- Properties that are intended to be consumed by the distribution provider, such as: the exported interfaces and configuration types, exported intents and configuration type specific properties. These properties are typically prefixed with 'service.' or 'endpoint.' see Table 122.1 on page 381.
- Service properties not intended for the distribution provider. These are typically used to communicate information to the consumer of the service and are often specific to the domain of the service.

The following methods to support the updating of service properties on Export Registrations and the propagation of these updates to the remote proxies via Import Registrations.

- ExportRegistration.update(Map) - Allows the Topology Manager to update an existing export registration it created after receiving a notification of changed properties on the remoted service.
- ImportRegistration.update(EndpointDescription) - Allows the Topology Manager to update the import registration representing a remote service after the remote service properties have been updated. Typically the topology manager is notified of such change via the Discovery mechanism.

The distribution provider must support the updates of service properties *not* intended for the distribution provider, where supported property values are as defined in the *Filter Syntax* of *OSGi Core Release 6*. Distribution providers may support updates to a wider set of properties or data types, but these may fail with other implementations.

122.5.4 Reflection

The Remote Service Admin service provides the following methods to get the list of the current exported and imported services:

- getExportedServices() - List the Export References for services that are exported by this Remote Service Admin service as directed by any of the Topology Managers.
- getImportedEndpoints() - List the Import References for services that have been imported by this Remote Service Admin service as directed by any of the Topology Managers.

122.5.5   Registration Life Cycle

All registrations obtained through a Remote Service Admin service are life cycle bound to the Topology Manager that created it. That is, if a Topology Manager ungets its Remote Service Admin service, all registrations obtained through this service must automatically be closed. This model ensures that all registrations are properly closed if either the Remote Service Admin or the Topology Manager stops because in both cases the framework performs the unget automatically. Such behavior can be achieved by implementing the Remote Service Admin service as a Service Factory.

122.5.6   Invalid Registrations

The Remote Service Admin service is explicitly allowed to return *invalid* Import and Export Registrations. First, in a communications stack it can take time to discover that there are issues, allowing the registration to return before it has completed can potentially save time. Second, it allows the Topology Manager to discover problems with the configuration information. Without the invalid Export Registrations, the Topology Manager would have to scan the log or associate the Remote Service Admin Events with a specific import/export method call, something that can be difficult to do.

If the registration is invalid, the getException() method must return a Throwable object. If the registration has initialized correctly, this method will return null. The getExportReference() and getImportReference() methods must throw an Illegal State Exception when the registration is invalid. A Remote Service Admin service is allowed to block for a reasonable amount of time when any of these methods is called, including the getException method, to finish initialization.

An invalid registration can be considered as never having been opened, it is therefore not necessary to close it; however, closing an invalid or closed registration must be a dummy operation and never throw an Exception. However, a failed registration must generate a corresponding error event.

122.5.7   Proxying

It is the responsibility of the Remote Service Admin service to properly proxy an imported service. This specification does not mandate the technique used to proxy an Endpoint as a service in the OSGi framework. The OSGi Remote Services specification allows a distribution provider to limit what it can proxy.

One of the primary aspects of a proxy is to ensure class space consistency between the exporting bundle and importing bundles. This can require the generation of a proxy-per-bundle to match the proper class spaces. It is the responsibility of the Remote Service Admin to ensure that no Class Cast Exceptions occur.
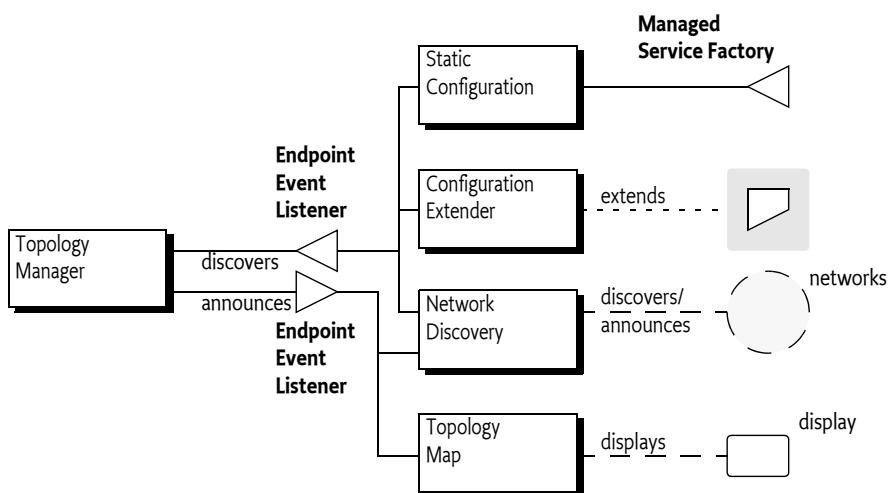
A common technique to achieve maximum class space compatibility is to use a Service Factory. A Service Factory provides the calling bundle when it first gets the service, making it straightforward to verify the package version of the interface that the calling bundle uses. Knowing the bundle that requests the service allows the creation of specialized proxies for each bundle. The interface class(es) for the proxy can then be loaded directly from the bundle, ensuring class compatibility. Interfaces should be loadable by the bundle otherwise that bundle can not use the interface in its code. If an interface cannot be loaded then it can be skipped. A dedicated class loader can then be created that has visibility to all these interfaces and is used to define the proxy class. This design ensures proper visibility and consistency. Implementations can optimize this model by sharing compatible class loaders between bundles.

The proxy will have to call arbitrary methods on arbitrary services. This has a large number of security implications, see *Security* on page 403.

122.6 Discovery

The topology of the distributed system is decided by the Topology Manager. However, in a distributed environment, the Topology Manager needs to *discover* Endpoints in other frameworks. There is a very large number of ways how a Topology Manager could learn about other Endpoints, ranging from static configuration, a centralized administration, all the way to fully dynamic discovery protocols like the Service Location Protocol (SLP) or JGroups. To support the required flexibility, this specification defines an *Endpoint Event Listener* service that allows the dissemination of Endpoint information. This service provides a symmetric solution because the problem is symmetric: it is used by a Topology Manager to announce changes in its local topology as well as find out about other Endpoint Descriptions. Where those other Endpoint Descriptions come from can vary widely. This design is depicted in Figure 122.4 on page 389.

Figure 122.4   *Examples*

The design of the Endpoint Event Listener allows a federated registry of Endpoint Descriptions. Any party that is interested in Endpoint Descriptions should register an Endpoint Event Listener service. This will signal that it is interested in topology information to any *Endpoint Description Providers*. Each Endpoint Event Listener service must be registered with a service property that holds a set of filter strings to indicate the *scope* of its interest. These filters must match an Endpoint Description before the corresponding Endpoint Event Listener service is notified of the availability of an Endpoint Description. Scoping is intended to limit the delivery of unnecessary Endpoint Descriptions as well as signal the need for specific Endpoints.

In addition to providing an Endpoint Event Listener actors must provide an Endpoint Listener. This may, or may not, be the same service object as the Endpoint Event Listener. Registering an Endpoint Listener in addition to an Endpoint Event Listener ensures that Endpoint announcements from version 1.0 actors will continue to be visible. If a service object is advertised as both an Endpoint Listener *and* an Endpoint Event Listener then version 1.1 actors must use the Endpoint Event Listener interface of the service in preference, and not call it as an Endpoint Listener. For this reason the Endpoint Listener interface is marked as `Deprecated`. The reason that the Endpoint Event Listener interface should be preferred is that it supports more advanced notification types, such as modification events.

A Topology Manager has knowledge of its local Endpoints and is likely to be only interested in remote Endpoints. It can therefore set the scope to only match remote Endpoint Descriptions. See *Framework UUID* on page 383 for how to limit the scope to local or remote Endpoints. At the

same time, a Topology manager should inform any locally registered Endpoint Event Listener and Endpoint Listener services about Endpoints that it has created or deleted.
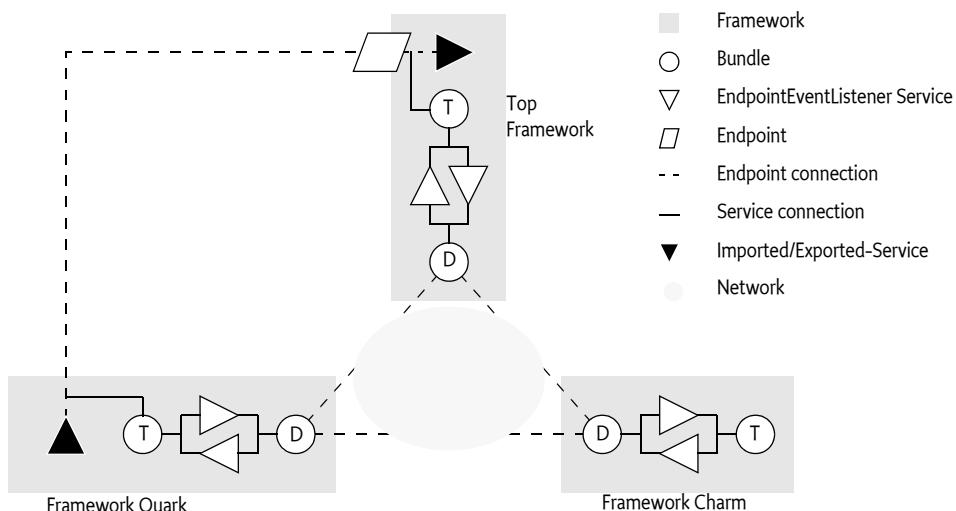
This architecture allows many different use cases. For example, a bundle could display a map of the topology by registering an Endpoint Event Listener with a scope for local Endpoints. Another example is the use of SLP to announce local Endpoints to a network and to discover remote Endpoints from other parties on this network.

An instance of this design is shown in Figure 122.5 on page 390. In this figure, there are 3 frameworks that collaborate through some discovery bundle. The Top framework has created an Endpoint and decides to notify all Endpoint Event Listeners and Endpoint Listeners registered in this framework that are scoped to this new Endpoint. Local bundle D has set its scope to all Endpoint Descriptions that originate from its local framework, it therefore receives the Endpoint Description from T. Bundle D then sends the Endpoint Description to all its peers on the network.

In the Quark framework, the manager bundle T has expressed an interest by setting its scope to a filter that matches the Endpoint Description from the Top framework. When the bundle D on the Quark framework receives the Endpoint Description from bundle D on the Top framework, it matches it against all local Endpoint Event Listener's scope. In this case, the local manager bundle T matches and is given the Endpoint Description. The manager then uses the Remote Service Admin service to import the exported service described by the given Endpoint Description.

Figure 122.5   *Endpoint Discovery Architecture. T=Topology Manager, D=Discovery*

The previous description is just one of the possible usages of the Endpoint Event Listener. For example, the discovery bundles could communicate the scopes to their peers. These peers could then register an Endpoint Event Listener per peer, minimizing the network traffic because Endpoint Descriptions do not have to be broadcast to all peers.

Another alternative usage is described in *Endpoint Description Extender Format* on page 395. In this chapter the extender pattern is used to retrieve Endpoint Descriptions from resources in locally active bundles.

122.6.1   Scope and Filters

An Endpoint Event Listener or Endpoint Listener service is registered with the ENDPOINT_LISTENER_SCOPE service property. This property, which is String+, must be set and must contain at least one filter. If there is not at least one filter, then that Endpoint Event Listener or Endpoint Listener must not receive any Endpoint Descriptions.

Each filter in the scope is applied against the properties of the Endpoint Description until one succeeds. Only if one succeeds is the Endpoint informed about the existence of an Endpoint.

The Endpoint Description is designed to reflect the properties of the imported service, there is therefore a correspondence with the filters that are used by bundles that are listening for service registrations. The purpose of this design is to match the filter available through Listener Hook services, see *On Demand* on page 393.

However, the purpose of the filters is more generic than just this use case. It can also be used to specify the interest in local Endpoints or remote Endpoints. For example, Topology Managers are only interested in remote Endpoints while discoverers are only interested in local Endpoints. It is easy to discriminate between local and remote by filtering on the endpoint.framework.uuid property. Endpoint Descriptions contain the Universally Unique ID (UUID) of the originating framework. This UUID must be available from the local framework as well. See *Framework UUID* on page 383.

122.6.2 Endpoint Event Listener Interface

The EndpointEventListener interface has the following method:

- endpointChanged(EndpointEvent,String) – Notify the Endpoint Event Listener of changes to an Endpoint. The change could entail the addition or removal of an Endpoint or the modification of the properties of an existing Endpoint. Multiple identical events should be counted as a single such event.

These methods must only be called if the Endpoint Event Listener service has a filter in its scope that matches the Endpoint Description properties.

The Endpoint Event Listener interface is *idempotent*. Endpoint Description Providers must inform an Endpoint Event Listener service (and its deprecated predecessor Endpoint Listener service) that is registered of all their matching Endpoints. The only way to find out about all available Endpoints is to register an Endpoint Event Listener (or Endpoint Listener) that is then informed by all available Endpoint Description Providers of their known Endpoint Descriptions that match their scope.

122.6.3 Endpoint Listener Interface

The EndpointListener interface is marked as Deprecated because the EndpointEventListener interface must be used in preference when both are implemented by the same object. The EndpointEvent interface has the following methods:

- endpointAdded(EndpointDescription,String) – Notify the Endpoint Listener of a new Endpoint Description. The second parameter is the filter that matched the Endpoint Description. Registering the same Endpoint multiple times counts as a single registration.
- endpointRemoved(EndpointDescription,String) – Notify the Endpoint Listener that the provided Endpoint Description is no longer available.

These methods must only be called if the Endpoint Listener service has a filter in its scope that matches the Endpoint Description properties. The reason for the filter string in the methods is to simplify and speed up matching an Endpoint Description to the cause of interest. For example, if the Listener Hook is used to do on demand import of services, then the filter can be associated with the Listener Info of the hook, see *On Demand* on page 393. If multiple filters in the scope match the Endpoint Description than the first filter in the scope must be passed.

The Endpoint Listener interface is *idempotent*. Endpoint Description Providers must inform an Endpoint Listener service that is registered of all their matching Endpoints.

122.6.4 Endpoint Event Listener and Endpoint Listener Implementations

An Endpoint Event Listener service tracks the known Endpoints in its given scope. There are potentially a large number of bundles involved in creating this federated registry of Endpoints. To ensure

that no Endpoint Descriptions are orphaned or unnecessarily missed, an Endpoint Event Listener implementation must follow the following rules:

- *Registration* – The Endpoint Event Listener service is called with an event of type ADDED for all known Endpoint Descriptions that the bundles in the local framework are aware of. Similarly, Endpoint Listener services are called with an endpointAdded(EndpointDescription,String) method for all these.
- *Tracking providers* – An Endpoint Event Listener or Endpoint Listener must track the bundles that provide it with Endpoint Descriptions. If a bundle that provided Endpoint Descriptions is stopped, all Endpoint Descriptions that were provided by that bundle must be removed. This can be implemented straightforwardly with a Service Factory.
- *Scope modification* – An Endpoint Event Listener or Endpoint Listener is allowed to modify the set of filters in its scope through a service property modification. This modification must result in new and/or existing Endpoint Descriptions to be added, however, existing Endpoints that are no longer in scope are not required to be explicitly removed by the their sources. It is the responsibility for the Endpoint Listener to remove these orphaned Endpoint Description from its view.
- *Endpoint mutability* – An Endpoint Description can change its Properties. The way this is handled is different for Endpoint Event Listeners and Endpoint Listeners. An Endpoint Event Listener receives a change event of type MODIFIED when the Properties of an existing Endpoint are modified. If the modification means that the Endpoint no longer matches the listener scope an event of type MODIFIED_ENDMATCH is sent instead. Endpoint Listener services receive a sequence of endpointRemoved(EndpointDescription,String) and endpointAdded(EndpointDescription,String) callbacks when the Properties of an Endpoint are modified.

Endpoint Descriptions can be added from different sources and providers of Endpoint Descriptions often use asynchronous and potentially unreliable communications. An implementation must therefore handle the addition of multiple equal Endpoint Descriptions from different sources as well as from the same source. Implementations must not count the number of registrations, a remove operation of an Endpoint Description is final for each source. That is, if source A added Endpoint Description e, then it can only be removed by source A. However, if source A added e multiple times, then it only needs to be removed once. Removals of Endpoint Descriptions that have not been added (or were removed before) should be ignored.

The discovery of Endpoints is a fundamentally indeterministic process and implementations of Endpoint Event Listener services should realize that there are no guarantees that an added Endpoint Description is always describing a valid Endpoint.

122.6.5 Endpoint Description Providers

The Endpoint Event Listener and Endpoint Listener services are based on an asynchronous, unreliable, best effort model because there are few guarantees in a distributed world. It is the task of an Endpoint Description Provider, for example a discovery bundle, to keep the Endpoint Event Listener services up to date of any Endpoint Descriptions the provider is aware of and that match the tracked service's scope.

If an Endpoint Event Listener or Endpoint Listener service is registered, a provider must add all matching Endpoint Descriptions that it is aware of and match the tracked listener's scope. This can be done during registration or asynchronously later. For example, it is possible to use the filters in the scope to request remote systems for any Endpoint Descriptions that match those filters. For expediency reasons, the service registration event should not be delayed until those results return; it is therefore applicable to add these Endpoint Descriptions later when the returns from the remote systems finally arrive.

If a tracked listener service object is advertised as both an Endpoint Event Listener and an Endpoint Listener then the EndpointDescription Provider must ignore the EndpointListener interface, and treat the listener only as an Endpoint Event Listener. Remote Service Admin 1.0 actors will be un-

aware of the EndpointEventListener interface, and will treat the service object purely as an Endpoint Listener. This restriction ensures that all actors will treat the service either as an Endpoint Event Listener, or an Endpoint Listener, but never as both. As a result the listener service will not have to disambiguate duplicate events from a single source. If an Endpoint Description Provider uses both the Endpoint Listener and Endpoint Event Listener interfaces of a single service object then the resulting behavior is undefined. The implementation may detect the misuse and throw an Exception, process or ignore the events from one of the interfaces, or it may simply corrupt the internal registry of Endpoints within the listener.

A tracked Endpoint Event Listener or Endpoint Listener is allowed to modify its scope by setting new properties on its Service Registration. An Endpoint Description provider must process the new scope and add any newly matching Endpoint Descriptions. It is not necessary to remove any Endpoint Descriptions that were added before but no longer match the new scope. Removing those orphaned descriptions is the responsibility of the listener implementation.

It is not necessary to remove any registered Endpoint Descriptions when the Endpoint Event Listener or Endpoint Listener is unregistered; also here it is the responsibility of the listener to do the proper cleanup.

122.6.6   On Demand

A common distribution policy is to import services that are being listened for by local bundles. For example, when a bundle opens a Service Tracker on the Log Service, a Topology Manager could be notified and attempt to find a Log Service in the local cluster and then import this service in the local Service Registry.

The OSGi framework provides service hooks for exactly this purpose. A Topology Manager can register a Listener Hook service and receive the information about bundles that have specified an interests in specific services.

For example, a bundle creates the following Service Tracker:

```
ServiceTracker st = new ServiceTracker(context,
        LogService.class.getName() );
st.open();
```

This Service Tracker will register a Service Listener with the OSGi framework. This will cause the framework to add a ListenerInfo to any Listener Hook services. The getFilter method on a ListenerInfo object provides a filter that is directly applicable for the Endpoint Event Listener's scope. In the previous example, this would be the filter:

```
(objectClass=org.osgi.service.log.LogService)
```

A Topology Manager could verify if this listener is satisfied. That is, if it has at least one service. If no such service could be found, it could then add this filter to its Endpoint Event Listener's scope to detect remote implementations of this service. If such an Endpoint is detected, it could then request the import of this service through the Remote Service Admin service.

122.7   Events

The Remote Service Admin service must synchronously inform any Remote Service Admin Listener services of events as they happen. Client of the events should return quickly and not perform any but trivial processing in the same thread.

The following event types are defined:

- EXPORT_ERROR - An exported service has run into an unrecoverable error, although the Export Registration has not been closed yet. The event carries the Export Registration as well as the Exception that caused the problem, if present.
- EXPORT_REGISTRATION - The Remote Service Admin has registered a new Export Registration.
- EXPORT_UNREGISTRATION - An Export Registration has been closed, the service is no longer exported and the Endpoint is no longer active when this was the last registration for that service/Endpoint combination.
- EXPORT_UPDATE - An exported service is updated. The service properties have changed.
- EXPORT_WARNING - An exported service is experiencing problems but the Endpoint is still available.
- IMPORT_ERROR - An imported service has run into a fatal error and has been shut down. The Import Registration should be closed by the Topology Manager that created them.
- IMPORT_REGISTRATION - A new Import Registration was created for a potentially existing service/Endpoint combination.
- IMPORT_UNREGISTRATION - An Import Registration was closed, removing the proxy if this was the last registration.
- IMPORT_UPDATE - An imported service is updated. The service properties have changed.
- IMPORT_WARNING - An imported service is experiencing problems but can continue to function.

The following properties are available on the event:

- getType() - The type of the event.
- getException() - Any exception, if present.
- getExportReference() - An export reference, if applicable.
- getImportReference() - An import reference, if applicable.
- getSource() - The source of the event, the Remote Service Admin service.

122.7.1   Event Admin Mapping

All Remote Service Admin events must be posted, which is asynchronously, to the Event Admin service, if present, under the following topic:

```
org/osgi/service/remoteserviceadmin/<type>
```

Where <type> represents the type of the event, for example IMPORT_ERROR.

The Event Admin event must have the following properties:

- bundle - (Bundle) The Remote Service Admin bundle
- bundle.id - (Long) The id of the Remote Service Admin bundle.
- bundle.symbolicname - (String) The Bundle Symbolic Name of the Remote Service Admin bundle.version - (Version) The version of the Remote Service Admin bundle.
- bundle.signer - (String[]) Signer of the Remote Service Admin bundle
- exception - (Throwable) The Exception, if present. Also reported on the cause property for backward compatibility.
- exception.class - (String) The fully-qualified class name of the attached Exception.
- exception.message - (String) The message of the attached exception. Only set if the Exception message is not null.
- endpoint.service.id - (Long) Remote service id, if present
- endpoint.framework.uuid - (String) Remote service's Framework UUID, if present
- endpoint.id - (String) The id of the Endpoint, if present
- objectClass - (String[]) The interface names, if present

- `service.imported.configs` - (String+) The configuration types of the imported services, if present
- `timestamp` - (Long) The time when the event occurred
- `event` - (RemoteServiceAdminEvent) The RemoteServiceAdminEvent object that caused this event.

122.8   Endpoint Description Extender Format

The Endpoint Description Extender format is a possibility to deliver Endpoint Descriptions in bundles. This section defines an XML schema and how to locate XML definition resources that use this schema to define Endpoint Descriptions. The definition resource is a simple property based model that can define the same information as the properties on an imported service. If a bundle with the description is *ready* (ACTIVE or lazy activation and in the STARTING state), then this static description can be disseminated through the Endpoint Event Listeners that have specified an interest in this description. If the bundle is stopped, the corresponding Endpoints must be removed.

XML documents containing remote service descriptions must be specified by the Remote-Service header in the manifest. The structure of the Remote Service header is:

```
Remote-Service ::= header // See Common Header Syntax in Core
```

The value of the header is a comma separated list of paths. A path is:

- A directory if it ends with a solidus ('/' \u002F). A directory is scanned for *.xml files.
- A path with wildcards. Such a path can use the wildcards in its last component, as defined in the findEntries method.
- A complete path, not having wildcards not ending in a solidus ('/' \u002F).

The Remote-Service header has no architected directives or attributes, unrecognized attributes and directives must be ignored.

A Remote-Service manifest header specified in a fragment must be ignored. However, XML documents referenced by a bundle's Remote-Service manifest header can be contained in attached fragments. The required behavior for this is implemented in the findEntries method.

The extender must process each XML document specified in this header. If an XML document specified by the header cannot be located in the bundle and its attached fragments, the extender must log an error message with the Log Service, if present, and continue.

For example:

```
Remote-Service: lib/, remote/osgi/*.dsc, cnf/google.xml
```

This matches all resources in the lib directory matching *.xml, all resources in the /remote/osgi directory that end with .dsc, as well as the google.xml resource in the cnf directory.

The namespace of these XML resources must be:

```
 http://www.osgi.org/xmlns/rsa/v1.0.0
```
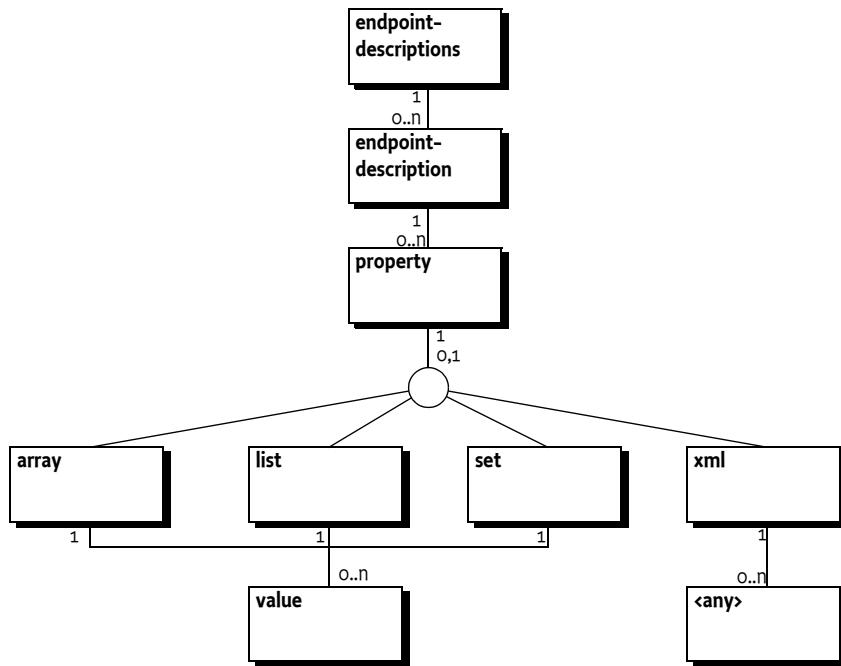
This namespace describes a set of Endpoint Descriptions, where each Endpoint Description can provide a set of properties. The structure of this schema is:

```
endpoint-descriptions  ::= <endpoint-description>*
endpoint-description   ::= <property>*
property               ::= ( <array> | <list> | <set>| <xml> )?
array                  ::= <value> *
```

```
list                    ::= <value> *
set                     ::= <value> *
xml                     ::= <*> *
```

This structure is depicted in Figure 122.6 on page 396.

Figure 122.6   *Endpoint Description XML Structure*

The property element has the attributes listed in table Table 122.2.

Table 122.2   *Property Attributes*

Attribute	Type	Description
name	String	The required name of the property. The type maps to the XML Schema xsd:string type.

Attribute	Type	Description
value-type	String	The optional type name of the property, the default is String. Any value in the value attribute or the value element when collections are used must be converted to the corresponding Java types. If the primitive form, for example byte, is specified for non-array types, then the value must be silently converted to the corresponding wrapper type.
	| long	
	| Long	
	| double	
	| Double	
	| float	
	| Float	
	| int	
	| Integer	
	| byte	
	| Byte	
	| char	
	| Character	
	| boolean	
	| Boolean	
	| short	
	| Short	
value	String	The value. Must be converted to the specified type if this is not the String type. The value attribute must not be used when the property element has a child element.

A property can have an array, list, set, or xml child element. If a child element is present then it is an error if the value attribute is defined. It is also an error of there is no child element and no value attribute.

The array, list, or set are *multi-valued*. That is, they contain 0 or more value elements. A value element contains text (a string) that must be converted to the given value-type or if not specified, left as is. Conversion must *trim* the leading and trailing white space characters as defined in the Character.isWhitespace method. No trimming must be done for strings. An array of primitive integers like int[] {1,42,97} can be encoded as follows:

```
<property name="integers" value-type="int">
    <array>
        <value> 1</value>
        <value>42</value>
        <value>97</value>
    </array>
</property>
```

The xml element is used to convey XML from other namespaces, it is allowed to contain one foreign XML root element, with any number of children, that will act as the root element of an XML document. This root element will be included in the corresponding property as a string. The XML element must be a valid XML document but not contain the XML processing instructions, the part between the <? and ?>. The value-type of the property must be String or not set when an xml element is used, using another type is invalid.

The xml element can be used to embed configuration information, making the Endpoint Description self contained.

The following is an example of an endpoint-descriptions resource.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<endpoint-descriptions xmlns="http://www.osgi.org/xmlns/rsa/v1.0.0">
    <endpoint-description>
        <property name="service.intents">
            <list>
                <value>SOAP</value>
                <value>HTTP</value>
            </list>
        </property>
        <property name="endpoint.id" value="http://ws.acme.com:9000/hello"/>
        <property name="endpoint.package.version.com.acme" value="4.2"/>
        <property name="objectClass">
            <array>
                <value>com.acme.Foo</value>
            </array>
        </property>
        <property name="service.imported.configs" value="com.acme"/>
        <property name="com.acme.ws.xml">
            <xml>
                <config xmlns="http://acme.com/defs">
                    <port>1029</port>
                    <host>www.acme.com</host>
                </config>
            </xml>
        </property>
    </endpoint-description>
</endpoint-descriptions>
```

Besides being in a separate resource, the static configuration as described here could also be part of a larger XML file. In that case the parser must ignore elements not part of the http://www.osgi.org/xmlns/rsa/v1.0.0 namespace schema.

122.8.1   XML Schema

This namespace of the schema is:

http://www.osgi.org/xmlns/rsa/v1.0.0

```xml
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:rsa="http://www.osgi.org/xmlns/rsa/v1.0.0"
  targetNamespace="http://www.osgi.org/xmlns/rsa/v1.0.0"
  elementFormDefault="qualified" version="1.0.1">

  <annotation>
    <documentation xml:lang="en">
      This is the XML Schema for endpoint descriptions used by
      the Remote Service Admin Specification. Endpoint descriptions
      are used to describe remote services to a client in cases
      where a real live Discovery system isn't used. An extender,
      such as a local Discovery Service can look for service
      descriptions in installed bundles and inform the Topology
      Manager of these remote services. The Topology Manager can then
      instruct the Remote Service Admin to create client proxies for
      these services.
    </documentation>
  </annotation>

  <element name="endpoint-descriptions" type="rsa:Tendpoint-descriptions" />

  <complexType name="Tendpoint-descriptions">
    <sequence>
```

```
      <element name="endpoint-description" type="rsa:Tendpoint-description"
              minOccurs="1" maxOccurs="unbounded" />
        <!--
          It is non-deterministic, per W3C XML Schema 1.0:
          http://www.w3.org/TR/xmlschema-1/#cos-nonambig to use
          namespace="##any" below.
        -->
      <any namespace="##other" minOccurs="0" maxOccurs="unbounded"
           processContents="lax" />
    </sequence>
    <anyAttribute processContents="lax" />
  </complexType>

  <complexType name="Tendpoint-description">
    <annotation>
      <documentation xml:lang="en">
        A Distribution Provider can register a proxy with the properties
        provided. Whether or not it is instructed to do so, is up to the
        Topology Manager. If any 'intents' properties are specified then the
        Distribution Provider should only register a proxy if it can support
        those intents.
      </documentation>
    </annotation>
    <sequence>
      <element name="property" type="rsa:Tproperty" minOccurs="1"
              maxOccurs="unbounded" />
      <any namespace="##other" minOccurs="0" maxOccurs="unbounded"
           processContents="lax" />
    </sequence>
    <anyAttribute processContents="lax" />
  </complexType>

  <complexType name="Tproperty" mixed="true">
    <sequence>
      <choice minOccurs="0" maxOccurs="1">
        <element name="array" type="rsa:Tmulti-value"/>
        <element name="list" type="rsa:Tmulti-value"/>
        <element name="set" type="rsa:Tmulti-value"/>
        <element name="xml" type="rsa:Txml"/>
      </choice>
      <any namespace="##other" minOccurs="0" maxOccurs="unbounded"
           processContents="lax" />
    </sequence>
    <attribute name="name" type="string" use="required" />
    <attribute name="value" type="string" use="optional" />
    <attribute name="value-type" type="rsa:Tvalue-types" default="String" use="optional" />
    <anyAttribute processContents="lax" />
  </complexType>

  <complexType name="Tmulti-value">
    <sequence>
      <element name="value" minOccurs="0" maxOccurs="unbounded" type="rsa:Tvalue"/>
      <any namespace="##other" minOccurs="0" maxOccurs="unbounded"
           processContents="lax" />
    </sequence>
    <anyAttribute processContents="lax" />
  </complexType>

  <complexType name="Tvalue" mixed="true">
    <sequence>
      <element name="xml" minOccurs="0" maxOccurs="1" type="rsa:Txml"/>
      <any namespace="##other" minOccurs="0" maxOccurs="unbounded"
           processContents="lax" />
    </sequence>
    <anyAttribute processContents="lax" />
  </complexType>

  <!-- Specifies the data type of a property or of the elements in a multi-value
       property. Numerical and boolean values are trimmed before they are processed.
       Simple types are automatically boxed if needed. Only the array data type
       allows for simple type values. When specifying a simple type on any other
       type of property it will automatically be boxed. -->
  <simpleType name="Tvalue-types">
    <restriction base="string">
      <enumeration value="String" />
```

```
        <enumeration value="long" />
        <enumeration value="Long" />
        <enumeration value="double" />
        <enumeration value="Double" />
        <enumeration value="float" />
        <enumeration value="Float" />
        <enumeration value="int" />
        <enumeration value="Integer" />
        <enumeration value="byte" />
        <enumeration value="Byte" />
        <enumeration value="char" />
        <enumeration value="Character" />
        <enumeration value="boolean" />
        <enumeration value="Boolean" />
        <enumeration value="short" />
        <enumeration value="Short" />
      </restriction>
</simpleType>

<!-- This complex type allows literal XML to be specified in an <xml/> tag (which
     is more convenient than putting it in a CDATA section).
     The embedded XML must be well-formed and not be in the rsa namespace. It will
     be put in a String value of a property or in an element of a multi-value
     property of base type String. The XML will be prefixed with the standard
     <?XML ?> header which is copied from the enclosing document. Hence it will
     carry the same version and encoding as the document in the rsa namespace. -->
<complexType name="Txml">
   <sequence>
     <any namespace="##other" minOccurs="1" maxOccurs="1"
          processContents="lax" />
   </sequence>
   <anyAttribute processContents="lax" />
</complexType>

<attribute name="must-understand" type="boolean" default="false">
   <annotation>
     <documentation xml:lang="en">
       This attribute should be used by extensions to documents
       to require that the document consumer understand the
       extension.
     </documentation>
   </annotation>
</attribute>
</schema>
```

122.9   Capability Namespaces

122.9.1   Local Discovery Extender

A bundle containing Endpoint Description Extender resources can indicate its dependency on the Remote Service Admin extender by declaring a requirement on the osgi.extender namespace.

```
Require-Capability: osgi.extender;
     filter:="(&(osgi.extender=osgi.remoteserviceadmin.localdiscovery)
              (version>=1.0)(!(version>=2.0)))"
```

With this constraint declared a bundle that depends on the extender will fail to resolve if no extender is present in the framework.

Implementations of this specification must provide this extender capability at version 1.0 as follows:

```
Provide-Capability: osgi.extender;
     osgi.extender="osgi.remoteserviceadmin.localdiscovery";
     version:Version="1.0";
     uses:="org.osgi.service.remoteserviceadmin"
```

The reason that the extender capability is declared at version 1.0 is because the extender is unchanged from version 1.0 of this specification.

122.9.2 Discovery Provider Capability

Discovery Providers use the `osgi.remoteserviceadmin.discovery` namespace to declare themselves as such. The version defined for this namespace indicates the version of this specification that the discovery provider supports.

This namespace has a defined attribute, `protocols` of type `List<String>`, which contains a list of the discovery protocols supported by the discovery provider. Local discovery providers (using the *Endpoint Description Extender Format* on page 395), should use the value `local` to indicate that they support this. Additionally, it defines a `version` attribute. Other values for the protocols attribute are implementation specific.

Table 122.3   *osgi.remoteserviceadmin.discovery Namespace*

Name	Kind	M/O	Type	Syntax	Description
protocols	CA	M	List<String>	symbolic-name	The discovery protocols supported. A value of local indicates support for the *Endpoint Description Extender Format* on page 395.
version	CA	M	Version	version	This version must correspond to the version of the Remote Service Admin specification.

Example: A discovery provider that provides local and SLP discovery:

```
Provide-Capability: osgi.remoteserviceadmin.discovery;
    protocols:List<String>="SLP,local"; version:Version=1.1
```

122.9.3 Distribution Provider Capability

Distribution providers advertise their supported distribution mechanisms using configuration types. These are selected at runtime using the `service.exported.configs` service property. Distribution providers can use the `osgi.remoteserviceadmin.distribution` namespace with attribute `configs`, of type `List<String>`, to advertise the supported config types.

Table 122.4   *osgi.remoteserviceadmin.distribution Namespace*

Name	Kind	M/O	Type	Syntax	Description
configs	CA	M	List<String>	symbolic-name	Supported configuration types. See *Endpoint Description* on page 380 .
version	CA	M	Version	version	This version must correspond to the version of the Remote Service Admin specification.

Example: A Distribution provider that supports the `org.acme.jaxws` and `org.acme.jaxrs` configuration types:

```
Provide-Capability: osgi.remoteserviceadmin.distribution;
    configs:List<String>="org.acme.jaxws,org.acme.jaxrs"; version:Version=1.1
```

122.9.4 Topology Manager Capability

Remote Service Admin topology managers may use different policies when determining which services to export and/or import. Topology managers use the namespace `osgi.remoteserviceadmin.topology` to declare this behavior. This namespace defines the policy at-

tribute of type List<String>. Values are implementation specific, but example definitions can be found at *Example Use Cases* on page 379.

Table 122.5   *osgi.remoteserviceadmin.topology Namespace*

Name	Kind	M/O	Type	Syntax	Description
policy	CA	M	List<String>	symbolic-name	The policy used for importing and exporting services. In general the policy is implementation specific.
version	CA	M	Version	version	This version must correspond to the version of the Remote Service Admin specification.

Example: A Topology manager that supports a *promiscuous* policy:

```
Provide-Capability: osgi.remoteserviceadmin.topology;
    policy:List<String>=promiscuous; version:Version=1.1
```

122.9.5   Service Capability

The Distribution Provider provides the *Remote Service Admin* service. To inform tools about this service it must provide the osgi.service namespace representing the RemoteServiceAdmin service. This capability must also declare a uses constraint for the org.osgi.service.remoteserviceadmin package:

```
Provide-Capability: osgi.service;
  objectClass:List<String>=
    "org.osgi.service.remoteserviceadmin.RemoteServiceAdmin";
  uses:="org.osgi.service.remoteserviceadmin"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 685.

122.10   Advice to implementations

This section is not intended to be normative, but offers advice to implementations as to how the complexity of supporting both the new Endpoint Event Listener and Endpoint Listener services can be managed and minimized. This advice applies to both Discovery Providers and Topology Managers implementing Remote Service Admin 1.1.

122.10.1   Notifying listeners

Endpoint Event Listeners and Endpoint Listeners have a very similar behavior and lifecycle. They also use the same property names to define their scope filter. It is therefore relatively simple for an Endpoint Description Provider to notify both Endpoint Listener and Endpoint Event Listeners using a single code path.

One possible mechanism is to track both the listener types using the same Service Tracker. If the tracked Service Reference advertises the EndpointEventListener interface then it must be treated as an Endpoint Event Listener. If not then the Endpoint Listener service can be wrapped in an adapter that converts Endpoint Event Listener events into the appropriate Endpoint Listener calls. The main notification code path can then treat every listener as an Endpoint Event Listener.

122.10.2   Receiving Endpoint lifecycle notifications

The Remote Service Admin 1.1 specification is backward compatible with version 1.0, meaning that version 1.1 actors must register an Endpoint Listener service. There is no restriction requiring this listener to be the same service as the Endpoint Event Listener, however there is a significant advantage to combining the listeners into a single service registration.

By making the two listeners a single service object a bundle can guarantee that it will not receive multiple notifications for the same event. If the service registrations are separate then Endpoint Description Providers will see two separate listeners, and notify them both. As a single service registration only one event will occur, and using the highest mutually supported version of the Remote Service Admin Specification.

122.11 Security

From a security point of view distribution is a significant threat. A Distribution Provider requires very significant capabilities to be able to proxy services. In many situations it will be required to grant the distribution provider All Permission. It is therefore highly recommended that Distribution Providers use trusted links and ensure that it is not possible to attack a system through the Remote Services Admin service and used discovery protocols.

122.11.1 Import and Export Registrations

Import and Export Registrations are *capabilities*. That is, they can only be obtained when the caller has the proper permissions but once obtained they are no longer checked. The caller should therefore be careful to share those objects with other bundles. Export and Import References are free to share.

122.11.2 Endpoint Permission

The Remote Service Admin implementation requires a large set of permissions because it must be able to distribute potentially any service. Giving these extensive capabilities to all Topology Managers would make it harder to developer general Topology Managers that implements specific scenarios. For this reason, this specification provides an Endpoint Permission.

When an Endpoint Permission must be verified, it must be created with an Endpoint Description as argument, like:

```
sm.checkPermission( new EndpointPermission(anEndpoint,localUUID,READ));
```

The standard name and action constructor is used to define a permission. The name argument is a filter expression. The filter for an Endpoint Permission is applied to the properties of an Endpoint Description. The localUUID must map to the UUID of the framework of the caller of this constructor, see *Framework UUID* on page 383. This localUUID is used to allow a the permissions to use the <<LOCAL>> magic name in the permission filter to refer to the local framework.

The filter expression can use the following magic value:

- <<LOCAL>> - This value represents the framework UUID of the framework that this bundle belongs to. The following example restricts the visibility to descriptions of local Endpoints:

```
ALLOW {
  ...EndpointPermission
      "(endpoint.framework.uuid=<<LOCAL>>)"
      "READ" }
```

An Endpoint Permission that has the actions listed in the following table.

Table 122.6   *Endpoint Permission Actions*

Action	Methods	Description
IMPORT	importService(EndpointDescription)	Import an Endpoint
EXPORT	exportService(ServiceReference,Map)	Export a service

Action	Methods	Description
READ	getExportedServices()	See the presence of distributed ser-
	getImportedEndpoints()	vices. The IMPORT and EXPORT action
	remoteAdminEvent(RemoteServiceAdminEvent)	imply READ. Distribution of events to the Remote Service Admin Listener. The Remote Service Admin must verify that the listener's bundle has the proper permission. No events should be delivered that are not implied.

122.12   org.osgi.service.remoteserviceadmin

Remote Service Admin Package Version 1.1.

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

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

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

122.12.1   Summary

- EndpointDescription - A description of an endpoint that provides sufficient information for a compatible distribution provider to create a connection to this endpoint An Endpoint Description is easy to transfer between different systems because it is property based where the property keys are strings and the values are simple types.
- EndpointEvent - An Endpoint Event.
- EndpointEventListener - A white board service that represents a listener for endpoints.
- EndpointListener - Deprecated white board service that represents a listener for endpoints.
- EndpointPermission - A bundle's authority to export, import or read an Endpoint.
- ExportReference - An Export Reference associates a service with a local endpoint.
- ExportRegistration - An Export Registration associates a service to a local endpoint.
- ImportReference - An Import Reference associates an active proxy service to a remote endpoint.
- ImportRegistration - An Import Registration associates an active proxy service to a remote endpoint.
- RemoteConstants - Provide the definition of the constants used in the Remote Service Admin specification.
- RemoteServiceAdmin - A Remote Service Admin manages the import and export of services.
- RemoteServiceAdminEvent - Provides the event information for a Remote Service Admin event.
- RemoteServiceAdminListener - A RemoteServiceAdminEvent listener is notified synchronously of any export or import registrations and unregistrations.

122.12.2   public class EndpointDescription

A description of an endpoint that provides sufficient information for a compatible distribution provider to create a connection to this endpoint An Endpoint Description is easy to transfer between different systems because it is property based where the property keys are strings and the values are simple types. This allows it to be used as a communications device to convey available end-

point information to nodes in a network. An Endpoint Description reflects the perspective of an *importer*. That is, the property keys have been chosen to match filters that are created by client bundles that need a service. Therefore the map must not contain any service.exported.* property and must contain the corresponding service.imported.* ones. The service.intents property must contain the intents provided by the service itself combined with the intents added by the exporting distribution provider. Qualified intents appear fully expanded on this property.

Concurrency　Immutable

122.12.2.1　　　**public EndpointDescription(Map<String,?> properties)**

properties　The map from which to create the Endpoint Description. The keys in the map must be type String and, since the keys are case insensitive, there must be no duplicates with case variation.

□　Create an Endpoint Description from a Map.

The endpoint.id, service.imported.configs and objectClass properties must be set.

Throws　IllegalArgumentException– When the properties are not proper for an Endpoint Description.

122.12.2.2　　　**public EndpointDescription(ServiceReference<?> reference,Map<String,?> properties)**

reference　A service reference that can be exported.

properties　Map of properties. This argument can be null. The keys in the map must be type String and, since the keys are case insensitive, there must be no duplicates with case variation.

□　Create an Endpoint Description based on a Service Reference and a Map of properties. The properties in the map take precedence over the properties in the Service Reference.

This method will automatically set the endpoint.framework.uuid and endpoint.service.id properties based on the specified Service Reference as well as the service.imported property if they are not specified as properties.

The endpoint.id, service.imported.configs and objectClass properties must be set.

Throws　IllegalArgumentException– When the properties are not proper for an Endpoint Description

122.12.2.3　　　**public boolean equals(Object other)**

other　The EndpointDescription object to be compared.

□　Compares this EndpointDescription object to another object.

An Endpoint Description is considered to be **equal to** another Endpoint Description if their ids are equal.

Returns　true if object is a EndpointDescription and is equal to this object; false otherwise.

122.12.2.4　　　**public List<String> getConfigurationTypes()**

□　Returns the configuration types. A distribution provider exports a service with an endpoint. This endpoint uses some kind of communications protocol with a set of configuration parameters. There are many different types but each endpoint is configured by only one configuration type. However, a distribution provider can be aware of different configuration types and provide synonyms to increase the change a receiving distribution provider can create a connection to this endpoint. This value of the configuration types is stored in the RemoteConstants.SERVICE_IMPORTED_CONFIGS service property.

Returns　An unmodifiable list of the configuration types used for the associated endpoint and optionally synonyms.

122.12.2.5　　　**public String getFrameworkUUID()**

□　Return the framework UUID for the remote service, if present. The value of the remote framework UUID is stored in the RemoteConstants.ENDPOINT_FRAMEWORK_UUID endpoint property.

Returns   Remote Framework UUID, or null if this endpoint is not associated with an OSGi framework having a framework UUID.

122.12.2.6   **public String getId()**

☐   Returns the endpoint's id. The id is an opaque id for an endpoint. No two different endpoints must have the same id. Two Endpoint Descriptions with the same id must represent the same endpoint. The value of the id is stored in the RemoteConstants.ENDPOINT_ID property.

Returns   The id of the endpoint, never null. The returned value has leading and trailing whitespace removed.

122.12.2.7   **public List<String> getIntents()**

☐   Return the list of intents implemented by this endpoint. The intents are based on the service.intents on an imported service, except for any intents that are additionally provided by the importing distribution provider. All qualified intents must have been expanded. This value of the intents is stored in the RemoteConstants.SERVICE_INTENTS service property.

Returns   An unmodifiable list of expanded intents that are provided by this endpoint.

122.12.2.8   **public List<String> getInterfaces()**

☐   Provide the list of interfaces implemented by the exported service. The value of the interfaces is derived from the objectClass property.

Returns   An unmodifiable list of Java interface names implemented by this endpoint.

122.12.2.9   **public Version getPackageVersion(String packageName)**

packageName   The name of the package for which a version is requested.

☐   Provide the version of the given package name. The version is encoded by prefixing the given package name with endpoint.package.version., and then using this as an endpoint property key. For example:

    endpoint.package.version.com.acme

The value of this property is in String format and will be converted to a Version object by this method.

Returns   The version of the specified package or Version.emptyVersion if the package has no version in this Endpoint Description.

Throws   IllegalArgumentException– If the version property value is not String.

122.12.2.10   **public Map<String,Object> getProperties()**

☐   Returns all endpoint properties.

Returns   An unmodifiable map referring to the properties of this Endpoint Description.

122.12.2.11   **public long getServiceId()**

☐   Returns the service id for the service exported through this endpoint. This is the service id under which the framework has registered the service. This field together with the Framework UUID is a globally unique id for a service. The value of the remote service id is stored in the RemoteConstants.ENDPOINT_SERVICE_ID endpoint property.

Returns   Service id of a service or 0 if this Endpoint Description does not relate to an OSGi service.

122.12.2.12   **public int hashCode()**

☐   Returns a hash code value for the object.

Returns   An integer which is a hash code value for this object.

122.12.2.13   **public boolean isSameService(EndpointDescription other)**

other   The Endpoint Description to look at

☐   Answers if this Endpoint Description refers to the same service instance as the given Endpoint De-
scription. Two Endpoint Descriptions point to the same service if they have the same id or their
framework UUIDs and remote service ids are equal.

Returns   True if this endpoint description points to the same service as the other

122.12.2.14   **public boolean matches(String filter)**

filter   The filter to test.

☐   Tests the properties of this EndpointDescription against the given filter using a case insensitive
match.

Returns   true If the properties of this EndpointDescription match the filter, false otherwise.

Throws   IllegalArgumentException– If filter contains an invalid filter string that cannot be parsed.

122.12.2.15   **public String toString()**

☐   Returns the string representation of this EndpointDescription.

Returns   String form of this EndpointDescription.

122.12.3   public class EndpointEvent

An Endpoint Event.

EndpointEvent objects are delivered to all registered EndpointEventListener services
where the EndpointDescription properties match one of the filters specified in the
EndpointEventListener.ENDPOINT_LISTENER_SCOPE registration properties of the Endpoint
Event Listener.

A type code is used to identify the type of event. The following event types are defined:

- ADDED
- REMOVED
- MODIFIED
- MODIFIED_ENDMATCH

Additional event types may be defined in the future.

See Also   EndpointEventListener

Since   1.1

Concurrency   Immutable

122.12.3.1   **public static final int ADDED = 1**

An endpoint has been added.

This EndpointEvent type indicates that a new endpoint has been added. The endpoint is represented
by the associated EndpointDescription object.

122.12.3.2   **public static final int MODIFIED = 4**

The properties of an endpoint have been modified.

This EndpointEvent type indicates that the properties of an existing endpoint have been modified.
The endpoint is represented by the associated EndpointDescription object and its properties can be

obtained via EndpointDescription.getProperties(). The endpoint properties still match the filters as specified in the EndpointEventListener.ENDPOINT_LISTENER_SCOPE filter.

122.12.3.3   public static final int MODIFIED_ENDMATCH = 8

The properties of an endpoint have been modified and the new properties no longer match the listener's filter.

This EndpointEvent type indicates that the properties of an existing endpoint have been modified and no longer match the filter. The endpoint is represented by the associated EndpointDescription object and its properties can be obtained via EndpointDescription.getProperties(). As a consequence of the modification the filters as specified in the EndpointEventListener.ENDPOINT_LISTENER_SCOPE do not match any more.

122.12.3.4   public static final int REMOVED = 2

An endpoint has been removed.

This EndpointEvent type indicates that an endpoint has been removed. The endpoint is represented by the associated EndpointDescription object.

122.12.3.5   public EndpointEvent(int type,EndpointDescription endpoint)

type   The event type. See getType().

endpoint   The endpoint associated with the event.

☐ Constructs a EndpointEvent object from the given arguments.

122.12.3.6   public EndpointDescription getEndpoint()

☐ Return the endpoint associated with this event.

Returns   The endpoint associated with the event.

122.12.3.7   public int getType()

☐ Return the type of this event.

The type values are:

- ADDED
- REMOVED
- MODIFIED
- MODIFIED_ENDMATCH

Returns   The type of this event.

122.12.4   public interface EndpointEventListener

A white board service that represents a listener for endpoints. An Endpoint Event Listener represents a participant in the distributed model that is interested in Endpoint Descriptions. This white board service can be used in many different scenarios. However, the primary use case is to allow a remote manager to be informed of Endpoint Descriptions available in the network and inform the network about available Endpoint Descriptions. Both the network bundle and the manager bundle register an Endpoint Event Listener service. The manager informs the network bundle about Endpoints that it creates. The network bundles then uses a protocol like SLP to announce these local end-points to the network. If the network bundle discovers a new Endpoint through its discovery protocol, then it sends an Endpoint Description to all the Endpoint Listener services that are registered (except its own) that have specified an interest in that endpoint. Endpoint Event Listener services can express their *scope* with the service property ENDPOINT_LISTENER_SCOPE. This service property is a list of filters. An Endpoint Description should only be given to a Endpoint Event Listen-

er when there is at least one filter that matches the Endpoint Description properties. This filter model is quite flexible. For example, a discovery bundle is only interested in locally originating Endpoint Descriptions. The following filter ensures that it only sees local endpoints.

> (org.osgi.framework.uuid=72dc5fd9-5f8f-4f8f-9821-9ebb433a5b72)

In the same vein, a manager that is only interested in remote Endpoint Descriptions can use a filter like:

> (!(org.osgi.framework.uuid=72dc5fd9-5f8f-4f8f-9821-9ebb433a5b72))

Where in both cases, the given UUID is the UUID of the local framework that can be found in the Framework properties. The Endpoint Event Listener's scope maps very well to the service hooks. A manager can just register all filters found from the Listener Hook as its scope. This will automatically provide it with all known endpoints that match the given scope, without having to inspect the filter string. In general, when an Endpoint Description is discovered, it should be dispatched to all registered Endpoint Event Listener services. If a new Endpoint Event Listener is registered, it should be informed about all currently known Endpoints that match its scope. If a getter of the Endpoint Listener service is unregistered, then all its registered Endpoint Description objects must be removed. The Endpoint Event Listener models a *best effort* approach. Participating bundles should do their utmost to keep the listeners up to date, but implementers should realize that many endpoints come through unreliable discovery processes. The Endpoint Event Listener supersedes the EndpointListener interface as it also supports notifications around modifications of endpoints.

Since 1.1

Concurrency Thread-safe

122.12.4.1   **public static final String ENDPOINT_LISTENER_SCOPE = "endpoint.listener.scope"**

Specifies the interest of this listener with filters. This listener is only interested in Endpoint Descriptions where its properties match the given filter. The type of this property must be String+.

122.12.4.2   **public void endpointChanged(EndpointEvent event,String filter)**

event The event containing the details about the change.

filter The filter from the ENDPOINT_LISTENER_SCOPE that matches (or for EndpointEvent.MODIFIED_ENDMATCH and EndpointEvent.REMOVED used to match) the endpoint, must not be null.

☐ Notification that an endpoint has changed. Details of the change is captured in the Endpoint Event provided. This could be that an endpoint was added, removed or modified.

122.12.5   public interface EndpointListener

Deprecated white board service that represents a listener for endpoints. An Endpoint Listener represents a participant in the distributed model that is interested in Endpoint Descriptions. The Endpoint Listener is called back when matching endpoints are added or removed. Consumers interested in the modification of endpoints, when associated service properties are changed, should use an EndpointEventListener instead. This white board service can be used in many different scenarios. However, the primary use case is to allow a remote manager to be informed of Endpoint Descriptions available in the network and inform the network about available Endpoint Descriptions. Both the network bundle and the manager bundle register an Endpoint Listener service. The manager informs the network bundle about Endpoints that it creates. The network bundles then uses a protocol like SLP to announce these local end-points to the network. If the network bundle discovers a new Endpoint through its discovery protocol, then it sends an Endpoint Description to all the Endpoint Listener services that are registered (except its own) that have specified an interest in that endpoint. Endpoint Listener services can express their *scope* with the service property ENDPOINT_LISTENER_SCOPE. This service property is a list of filters. An Endpoint Description should only be given to a Endpoint Listener when there is at least one filter that matches the End-

point Description properties. This filter model is quite flexible. For example, a discovery bundle is only interested in locally originating Endpoint Descriptions. The following filter ensure that it only sees local endpoints.

    (org.osgi.framework.uuid=72dc5fd9-5f8f-4f8f-9821-9ebb433a5b72)

In the same vein, a manager that is only interested in remote Endpoint Descriptions can use a filter like:

    (!(org.osgi.framework.uuid=72dc5fd9-5f8f-4f8f-9821-9ebb433a5b72))

Where in both cases, the given UUID is the UUID of the local framework that can be found in the Framework properties. The Endpoint Listener's scope maps very well to the service hooks. A manager can just register all filters found from the Listener Hook as its scope. This will automatically provide it with all known endpoints that match the given scope, without having to inspect the filter string. In general, when an Endpoint Description is discovered, it should be dispatched to all registered Endpoint Listener services. If a new Endpoint Listener is registered, it should be informed about all currently known Endpoints that match its scope. If a getter of the Endpoint Listener service is unregistered, then all its registered Endpoint Description objects must be removed. The Endpoint Listener models a *best effort* approach. Participating bundles should do their utmost to keep the listeners up to date, but implementers should realize that many endpoints come through unreliable discovery processes.

Deprecated   As of 1.1. Replaced by EndpointEventListener.

Concurrency   Thread-safe

122.12.5.1   **public static final String ENDPOINT_LISTENER_SCOPE = "endpoint.listener.scope"**

Specifies the interest of this listener with filters. This listener is only interested in Endpoint Descriptions where its properties match the given filter. The type of this property must be String+.

122.12.5.2   **public void endpointAdded(EndpointDescription endpoint,String matchedFilter)**

endpoint   The Endpoint Description to be published

matchedFilter   The filter from the ENDPOINT_LISTENER_SCOPE that matched the endpoint, must not be null.

☐   Register an endpoint with this listener. If the endpoint matches one of the filters registered with the ENDPOINT_LISTENER_SCOPE service property then this filter should be given as the matchedFilter parameter. When this service is first registered or it is modified, it should receive all known endpoints matching the filter.

122.12.5.3   **public void endpointRemoved(EndpointDescription endpoint,String matchedFilter)**

endpoint   The Endpoint Description that is no longer valid.

matchedFilter   The filter from the ENDPOINT_LISTENER_SCOPE that matched the endpoint, must not be null.

☐   Remove the registration of an endpoint. If an endpoint that was registered with the endpointAdded(EndpointDescription, String) method is no longer available then this method should be called. This will remove the endpoint from the listener. It is not necessary to remove endpoints when the service is unregistered or modified in such a way that not all endpoints match the interest filter anymore.

122.12.6   public final class EndpointPermission
extends Permission

A bundle's authority to export, import or read an Endpoint.

- The export action allows a bundle to export a service as an Endpoint.
- The import action allows a bundle to import a service from an Endpoint.

- The read action allows a bundle to read references to an Endpoint.

Permission to read an Endpoint is required in order to detect events regarding an Endpoint. Untrusted bundles should not be able to detect the presence of certain Endpoints unless they have the appropriate EndpointPermission to read the specific service.

Concurrency   Thread-safe

122.12.6.1   **public static final String EXPORT = "export"**

The action string export. The export action implies the read action.

122.12.6.2   **public static final String IMPORT = "import"**

The action string import. The import action implies the read action.

122.12.6.3   **public static final String READ = "read"**

The action string read.

122.12.6.4   **public EndpointPermission(String filterString,String actions)**

filterString   The filter string or "∗" to match all endpoints.

actions   The actions read, import, or export.

☐   Create a new EndpointPermission with the specified filter.

The filter will be evaluated against the endpoint properties of a requested EndpointPermission.

There are three possible actions: read, import and export. The read action allows the owner of this permission to see the presence of distributed services. The import action allows the owner of this permission to import an endpoint. The export action allows the owner of this permission to export a service.

Throws   IllegalArgumentException– If the filter has an invalid syntax or the actions are not valid.

122.12.6.5   **public EndpointPermission(EndpointDescription endpoint,String localFrameworkUUID,String actions)**

endpoint   The requested endpoint.

localFrameworkU-   The UUID of the local framework. This is used to support matching the endpoint.framework.uuid
UID   endpoint property to the <<LOCAL>> value in the filter expression.

actions   The actions read, import, or export.

☐   Creates a new requested EndpointPermission object to be used by code that must perform checkPermission. EndpointPermission objects created with this constructor cannot be added to an EndpointPermission permission collection.

Throws   IllegalArgumentException– If the endpoint is null or the actions are not valid.

122.12.6.6   **public boolean equals(Object obj)**

obj   The object to test for equality.

☐   Determines the equality of two EndpointPermission objects. Checks that specified object has the same name, actions and endpoint as this EndpointPermission.

Returns   true If obj is a EndpointPermission, and has the same name, actions and endpoint as this EndpointPermission object; false otherwise.

122.12.6.7   **public String getActions()**

☐   Returns the canonical string representation of the actions. Always returns present actions in the following canonical order: read, import, export.

Returns   The canonical string representation of the actions.

122.12.6.8   **public int hashCode()**

  ☐ Returns the hash code value for this object.

Returns Hash code value for this object.

122.12.6.9   **public boolean implies(Permission p)**

  p The target permission to check.

  ☐ Determines if a EndpointPermission object "implies" the specified permission.

Returns true if the specified permission is implied by this object; false otherwise.

122.12.6.10   **public PermissionCollection newPermissionCollection()**

  ☐ Returns a new PermissionCollection object for storing EndpointPermission objects.

Returns A new PermissionCollection object suitable for storing EndpointPermission objects.

122.12.7   public interface ExportReference

An Export Reference associates a service with a local endpoint. The Export Reference can be used to reference an exported service. When the service is no longer exported, all methods must return null.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

122.12.7.1   **public EndpointDescription getExportedEndpoint()**

  ☐ Return the Endpoint Description for the local endpoint.

Returns The Endpoint Description for the local endpoint. Must be null when the service is no longer exported.

122.12.7.2   **public ServiceReference<?> getExportedService()**

  ☐ Return the service being exported.

Returns The service being exported. Must be null when the service is no longer exported.

122.12.8   public interface ExportRegistration

An Export Registration associates a service to a local endpoint. The Export Registration can be used to delete the endpoint associated with an this registration. It is created with the RemoteServiceAdmin.exportService(ServiceReference,Map) method. When this Export Registration has been closed, all methods must return null.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

122.12.8.1   **public void close()**

  ☐ Delete the local endpoint and disconnect any remote distribution providers. After this method returns, all methods must return null. This method has no effect when this registration has already been closed or is being closed.

122.12.8.2   **public Throwable getException()**

  ☐ Return the exception for any error during the export process. If the Remote Service Admin for some reasons is unable to properly initialize this registration, then it must return an exception from this method. If no error occurred, this method must return null. The error must be set before this Export Registration is returned. Asynchronously occurring errors must be reported to the log.

Returns The exception that occurred during the initialization of this registration or null if no exception occurred.

122.12.8.3   public ExportReference getExportReference()

☐ Return the Export Reference for the exported service.

Returns The Export Reference for this registration, or null if this Import Registration is closed.

Throws IllegalStateException– When this registration was not properly initialized. See getException().

122.12.8.4   public EndpointDescription update(Map<String,?> properties)

properties properties to be merged with the current service properties for the ServiceReference represented by this ExportRegistration. If null is passed then the original properties passed to RemoteServiceAdmin.exportService(ServiceReference, Map) will be used.

☐ Update the endpoint represented by this ExportRegistration and return an updated EndpointDescription. If this method returns an updated EndpointDescription, then the object returned via getExportReference() must also have been updated to return this new object. If this method does not return an updated EndpointDescription then the object returned via getExportReference() should remain unchanged. When creating the updated EndpointDescription the ServiceReference originally passed to RemoteServiceAdmin.exportService(ServiceReference, Map) must be queried to pick up any changes to its service properties. If this argument is null then the original properties passed when creating this ExportRegistration should be used when constructing the updated EndpointDescription. Otherwise the new properties should be used, and replace the original properties for subsequent calls to the update method.

Returns The updated EndpointDescription for this registration or null if there was a failure updating the endpoint. If a failure occurs then it can be accessed using getException().

Throws IllegalStateException– If this registration is closed, or when this registration was not properly initialized. See getException().

Since 1.1

122.12.9   public interface ImportReference

An Import Reference associates an active proxy service to a remote endpoint. The Import Reference can be used to reference an imported service. When the service is no longer imported, all methods must return null.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

122.12.9.1   public EndpointDescription getImportedEndpoint()

☐ Return the Endpoint Description for the remote endpoint.

Returns The Endpoint Description for the remote endpoint. Must be null when the service is no longer imported.

122.12.9.2   public ServiceReference<?> getImportedService()

☐ Return the Service Reference for the proxy for the endpoint.

Returns The Service Reference to the proxy for the endpoint. Must be null when the service is no longer imported.

122.12.10   public interface ImportRegistration

An Import Registration associates an active proxy service to a remote endpoint. The Import Registration can be used to delete the proxy associated with an endpoint. It is created with the

RemoteServiceAdmin.importService(EndpointDescription) method. When this Import Registration has been closed, all methods must return null.

Concurrency   Thread-safe

Provider Type   Consumers of this API must not implement this type

122.12.10.1   **public void close()**

□   Close this Import Registration. This must close the connection to the endpoint and unregister the proxy. After this method returns, all other methods must return null. This method has no effect when this registration has already been closed or is being closed.

122.12.10.2   **public Throwable getException()**

□   Return the exception for any error during the import process. If the Remote Service Admin for some reasons is unable to properly initialize this registration, then it must return an exception from this method. If no error occurred, this method must return null. The error must be set before this Import Registration is returned. Asynchronously occurring errors must be reported to the log.

Returns   The exception that occurred during the initialization of this registration or null if no exception occurred.

122.12.10.3   **public ImportReference getImportReference()**

□   Return the Import Reference for the imported service.

Returns   The Import Reference for this registration, or null if this Import Registration is closed.

Throws   IllegalStateException– When this registration was not properly initialized. See getException().

122.12.10.4   **public boolean update(EndpointDescription endpoint)**

endpoint   The updated endpoint

□   Update the local service represented by this ImportRegistration. After this method returns the EndpointDescription returned via getImportReference() must have been updated.

Returns   true if the endpoint was successfully updated, false otherwise. If the update fails then the failure can be retrieved from getException().

Throws   IllegalStateException– When this registration is closed, or if it was not properly initialized. See getException().

IllegalArgumentException– When the supplied EndpointDescription does not represent the same endpoint as this ImportRegistration.

Since   1.1

122.12.11   public class RemoteConstants

Provide the definition of the constants used in the Remote Service Admin specification.

Concurrency   Immutable

122.12.11.1   **public static final String ENDPOINT_FRAMEWORK_UUID = "endpoint.framework.uuid"**

Endpoint property identifying the universally unique id of the exporting framework. Can be absent if the corresponding endpoint is not for an OSGi service.

The value of this property must be of type String.

122.12.11.2   **public static final String ENDPOINT_ID = "endpoint.id"**

Endpoint property identifying the id for this endpoint. This service property must always be set.

The value of this property must be of type String.

122.12.11.3 **public static final String ENDPOINT_PACKAGE_VERSION_ = "endpoint.package.version."**

Prefix for an endpoint property identifying the interface Java package version for an interface. For example, the property endpoint.package.version.com.acme=1.3 describes the version of the package for the com.acme.Foo interface. This endpoint property for an interface package does not have to be set. If not set, the value must be assumed to be 0.

Since endpoint properties are stored in a case insensitive map, case variants of a package name are folded together.

The value of properties having this prefix must be of type String.

122.12.11.4 **public static final String ENDPOINT_SERVICE_ID = "endpoint.service.id"**

Endpoint property identifying the service id of the exported service. Can be absent or 0 if the corresponding endpoint is not for an OSGi service.

The value of this property must be of type Long.

122.12.11.5 **public static final String REMOTE_CONFIGS_SUPPORTED = "remote.configs.supported"**

Service property identifying the configuration types supported by a distribution provider. Registered by the distribution provider on one of its services to indicate the supported configuration types.

The value of this property must be of type String, String[], or Collection of String.

See Also Remote Services Specification

122.12.11.6 **public static final String REMOTE_INTENTS_SUPPORTED = "remote.intents.supported"**

Service property identifying the intents supported by a distribution provider. Registered by the distribution provider on one of its services to indicate the vocabulary of implemented intents.

The value of this property must be of type String, String[], or Collection of String.

See Also Remote Services Specification

122.12.11.7 **public static final String SERVICE_EXPORTED_CONFIGS = "service.exported.configs"**

Service property identifying the configuration types that should be used to export the service. Each configuration type represents the configuration parameters for an endpoint. A distribution provider should create an endpoint for each configuration type that it supports.

This property may be supplied in the properties Dictionary object passed to the BundleContext.registerService method. The value of this property must be of type String, String[], or Collection of String.

See Also Remote Services Specification

122.12.11.8 **public static final String SERVICE_EXPORTED_INTENTS = "service.exported.intents"**

Service property identifying the intents that the distribution provider must implement to distribute the service. Intents listed in this property are reserved for intents that are critical for the code to function correctly, for example, ordering of messages. These intents should not be configurable.

This property may be supplied in the properties Dictionary object passed to the BundleContext.registerService method. The value of this property must be of type String, String[], or Collection of String.

See Also Remote Services Specification

122.12.11.9 **public static final String SERVICE_EXPORTED_INTENTS_EXTRA = "service.exported.intents.extra"**

Service property identifying the extra intents that the distribution provider must implement to distribute the service. This property is merged with the service.exported.intents property before the

distribution provider interprets the listed intents; it has therefore the same semantics but the property should be configurable so the administrator can choose the intents based on the topology. Bundles should therefore make this property configurable, for example through the Configuration Admin service.

This property may be supplied in the properties Dictionary object passed to the BundleContext.registerService method. The value of this property must be of type String, String[], or Collection of String.

See Also   Remote Services Specification

122.12.11.10   **public static final String SERVICE_EXPORTED_INTERFACES = "service.exported.interfaces"**

Service property marking the service for export. It defines the interfaces under which this service can be exported. This list must be a subset of the types under which the service was registered. The single value of an asterisk ('∗' \u002A) indicates all the interface types under which the service was registered excluding the non-interface types. It is strongly recommended to only export interface types and not concrete classes due to the complexity of creating proxies for some type of concrete classes.

This property may be supplied in the properties Dictionary object passed to the BundleContext.registerService method. The value of this property must be of type String, String[], or Collection of String.

See Also   Remote Services Specification

122.12.11.11   **public static final String SERVICE_IMPORTED = "service.imported"**

Service property identifying the service as imported. This service property must be set by a distribution provider to any value when it registers the endpoint proxy as an imported service. A bundle can use this property to filter out imported services.

The value of this property may be of any type.

See Also   Remote Services Specification

122.12.11.12   **public static final String SERVICE_IMPORTED_CONFIGS = "service.imported.configs"**

Service property identifying the configuration types used to import the service. Any associated properties for this configuration types must be properly mapped to the importing system. For example, a URL in these properties must point to a valid resource when used in the importing framework. If multiple configuration types are listed in this property, then they must be synonyms for exactly the same remote endpoint that is used to export this service.

The value of this property must be of type String, String[], or Collection of String.

See Also   Remote Services Specification, SERVICE_EXPORTED_CONFIGS

122.12.11.13   **public static final String SERVICE_INTENTS = "service.intents"**

Service property identifying the intents that this service implement. This property has a dual purpose:

- A bundle can use this service property to notify the distribution provider that these intents are already implemented by the exported service object.
- A distribution provider must use this property to convey the combined intents of: The exporting service, and the intents that the exporting distribution provider adds, and the intents that the importing distribution provider adds.

To export a service, a distribution provider must expand any qualified intents. Both the exporting and importing distribution providers must recognize all intents before a service can be distributed. The value of this property must be of type String, String[], or Collection of String.

See Also   Remote Services Specification

122.12.12   public interface RemoteServiceAdmin

A Remote Service Admin manages the import and export of services. A Distribution Provider can expose a control interface. This interface allows a Topology Manager to control the export and import of services. The API allows a Topology Manager to export a service, to import a service, and find out about the current imports and exports.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

122.12.12.1   public Collection<ExportRegistration> exportService(ServiceReference<?> reference,Map<String,?> properties)

reference The Service Reference to export.

properties The properties to create a local Endpoint that can be implemented by this Remote Service Admin. If this is null, the Endpoint will be determined by the properties on the service. The properties are the same as given for an exported service. They override any properties in the specified Service Reference (case insensitive). The properties objectClass and service.id, in any case variant, are ignored. Those properties in the Service Reference cannot be overridden. This parameter can be null, this should be treated as an empty map.

☐ Export a service to a given Endpoint. The Remote Service Admin must create an Endpoint from the given description that can be used by other Distribution Providers to connect to this Remote Service Admin and use the exported service. The property keys of a Service Reference are case insensitive while the property keys of the specified properties map are case sensitive. A property key in the specified properties map must therefore override any case variant property key in the properties of the specified Service Reference.

If the caller does not have the appropriate EndpointPermission[endpoint,EXPORT] for an Endpoint, and the Java Runtime Environment supports permissions, then the getException method on the corresponding returned ExportRegistration will return a SecurityException.

Returns A Collection of ExportRegistrations for the specified Service Reference and properties. Multiple Export Registrations may be returned because a single service can be exported to multiple Endpoints depending on the available configuration type properties and the intents that they support. The result is never null but may be empty if this Remove Service Admin does not recognize any of the configuration types, or if they Remote Service Admin cannot support the relevant intents.

Throws IllegalArgumentException– If any of the properties for this configuration type has a value that is not syntactically correct, or if the service properties and the overlaid properties do not contain a RemoteConstants.SERVICE_EXPORTED_INTERFACES entry. This means that implementations must not ignore invalid values for property names that they recognize.

122.12.12.2   public Collection<ExportReference> getExportedServices()

☐ Return the currently active Export References.

If the caller does not have the appropriate EndpointPermission[endpoint,READ] for an Endpoint, and the Java Runtime Environment supports permissions, then returned collection will not contain a reference to the exported Endpoint.

Returns A Collection of ExportReferences that are currently active.

122.12.12.3   public Collection<ImportReference> getImportedEndpoints()

☐ Return the currently active Import References.

If the caller does not have the appropriate EndpointPermission[endpoint,READ] for an Endpoint, and the Java Runtime Environment supports permissions, then returned collection will not contain a reference to the imported Endpoint.

Returns A Collection of ImportReferences that are currently active.

122.12.12.4   **public ImportRegistration importService(EndpointDescription endpoint)**

endpoint   The Endpoint Description to be used for import.

☐   Import a service from an Endpoint. The Remote Service Admin must use the given Endpoint to create a proxy. This method can return null if the service could not be imported.

Returns   An Import Registration that combines the Endpoint Description and the Service Reference or null if the Endpoint could not be imported.

Throws   SecurityException– If the caller does not have the appropriate
EndpointPermission[endpoint,IMPORT] for the Endpoint, and the Java Runtime Environment supports permissions.

122.12.13   public class RemoteServiceAdminEvent

Provides the event information for a Remote Service Admin event.

Concurrency   Immutable

122.12.13.1   **public static final int EXPORT_ERROR = 6**

A fatal exporting error occurred. The Export Registration has been closed.

122.12.13.2   **public static final int EXPORT_REGISTRATION = 2**

Add an export registration. The Remote Service Admin will send this event when it exports a service. When the RemoteServiceAdminListener service is registered, the Remote Service Admin must notify the listener of all existing Export Registrations.

122.12.13.3   **public static final int EXPORT_UNREGISTRATION = 3**

Remove an export registration. The Remote Service Admin will send this event when it removes the export of a service.

122.12.13.4   **public static final int EXPORT_UPDATE = 10**

Update an export registration. The Remote Service Admin will send this event when it exports a service.

Since   1.1

122.12.13.5   **public static final int EXPORT_WARNING = 7**

A problematic situation occurred, the export is still active.

122.12.13.6   **public static final int IMPORT_ERROR = 5**

A fatal importing error occurred. The Import Registration has been closed.

122.12.13.7   **public static final int IMPORT_REGISTRATION = 1**

Add an import registration. The Remote Service Admin will send this event when it imports a service. When the RemoteServiceAdminListener service is registered, the Remote Service Admin must notify the listener of all existing Import Registrations.

122.12.13.8   **public static final int IMPORT_UNREGISTRATION = 4**

Remove an import registration. The Remote Service Admin will send this event when it removes the import of a service.

122.12.13.9   **public static final int IMPORT_UPDATE = 9**

Update an import registration. The Remote Service Admin will send this event when it updates a service.

Since 1.1

122.12.13.10   **public static final int IMPORT_WARNING = 8**

A problematic situation occurred, the import is still active.

122.12.13.11   **public RemoteServiceAdminEvent(int type,Bundle source,ExportReference exportReference,Throwable exception)**

type   The event type.

source   The source bundle, must not be null.

exportReference   The exportReference, can not be null.

exception   Any exceptions encountered, can be null.

☐ Create a Remote Service Admin Event for an export notification.

122.12.13.12   **public RemoteServiceAdminEvent(int type,Bundle source,ImportReference importReference,Throwable exception)**

type   The event type.

source   The source bundle, must not be null.

importReference   The importReference, can not be null.

exception   Any exceptions encountered, can be null.

☐ Create a Remote Service Admin Event for an import notification.

122.12.13.13   **public Throwable getException()**

☐ Return the exception for this event.

Returns   The exception or null.

122.12.13.14   **public ExportReference getExportReference()**

☐ Return the Export Reference for this event.

Returns   The Export Reference or null.

122.12.13.15   **public ImportReference getImportReference()**

☐ Return the Import Reference for this event.

Returns   The Import Reference or null.

122.12.13.16   **public Bundle getSource()**

☐ Return the bundle source of this event.

Returns   The bundle source of this event.

122.12.13.17   **public int getType()**

☐ Return the type of this event.

Returns   The type of this event.

122.12.14   **public interface RemoteServiceAdminListener**

A RemoteServiceAdminEvent listener is notified synchronously of any export or import registrations and unregistrations.

If the Java Runtime Environment supports permissions, then filtering is done. RemoteServiceAdminEvent objects are only delivered to the listener if the bundle which defines the listener object's

class has the appropriate EndpointPermission[endpoint,READ] for the endpoint referenced by the event.

See Also   RemoteServiceAdminEvent

Concurrency   Thread-safe

122.12.14.1   **public void remoteAdminEvent(RemoteServiceAdminEvent event)**

event   The RemoteServiceAdminEvent object.

☐   Receive notification of any export or import registrations and unregistrations as well as errors and warnings.

122.13   org.osgi.service.remoteserviceadmin.namespace

Remote Service Admin Namespaces Version 1.0.

Bundles should not need to import this package at runtime since all the types in this package just contain constants for capability and requirement namespaces specified by the OSGi Alliance.

122.13.1   Summary

- DiscoveryNamespace - Remote Services Discovery Provider Capability and Requirement Namespace.
- DistributionNamespace - Remote Services Distribution Provider Capability and Requirement Namespace.
- TopologyNamespace - Remote Services Topology Manager Capability and Requirement Namespace.

122.13.2   public final class DiscoveryNamespace
extends Namespace

Remote Services Discovery Provider Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace.

Concurrency   Immutable

122.13.2.1   **public static final String CAPABILITY_PROTOCOLS_ATTRIBUTE = "protocols"**

The capability attribute used to specify the discovery protocols supported by this discovery provider. The value of this attribute must be of type String or List<String>.

122.13.2.2   **public static final String DISCOVERY_NAMESPACE = "osgi.remoteserviceadmin.discovery"**

Namespace name for Remote Services discovery provider capabilities and requirements.

122.13.3   public final class DistributionNamespace
extends Namespace

Remote Services Distribution Provider Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace.

Concurrency   Immutable

122.13.3.1   **public static final String CAPABILITY_CONFIGS_ATTRIBUTE = "configs"**

The capability attribute used to specify the config types supported by this distribution provider. The value of this attribute must be of type String or List<String>.

122.13.3.2 **public static final String DISTRIBUTION_NAMESPACE = "osgi.remoteserviceadmin.distribution"**

Namespace name for Remote Services distribution provider capabilities and requirements.

122.13.4 public final class TopologyNamespace extends Namespace

Remote Services Topology Manager Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace.

Concurrency Immutable

122.13.4.1 **public static final String CAPABILITY_POLICY_ATTRIBUTE = "policy"**

The capability attribute used to specify the policy or policies supported by this topology manager. The value of this attribute must be of type String or List<String>. Policy names are typically implementation specific, however the Remote Services Specification defines the *promiscuous* and *fail-over* policies for common use cases.

122.13.4.2 **public static final String FAIL_OVER_POLICY = "fail-over"**

The attribute value for Topology managers with a fail-over policy

See Also TopologyNamespace.CAPABILITY_POLICY_ATTRIBUTE

122.13.4.3 **public static final String PROMISCUOUS_POLICY = "promiscuous"**

The attribute value for Topology managers with a promiscuous policy

See Also TopologyNamespace.CAPABILITY_POLICY_ATTRIBUTE

122.13.4.4 **public static final String TOPOLOGY_NAMESPACE = "osgi.remoteserviceadmin.topology"**

Namespace name for Remote Services topology manager capabilities and requirements.

122.14 References

[1] *OSGi Service Property Namespace*
http://www.osgi.org/Specifications/ServicePropertyNamespace

[2] *UUIDs*
http://en.wikipedia.org/wiki/Universally_Unique_Identifier

[3] *Service Location Protocol (SLP)*
http://en.wikipedia.org/wiki/Service_Location_Protocol

[4] *JGroups*
http://www.jgroups.org/

[5] *UDDI*
http://en.wikipedia.org/wiki/Universal_Description_Discovery_and_Integration

[6] *Service Component Architecture (SCA)*
http://www.osoa.org/display/Main/Home

122.15 Changes

- Introduced Endpoint Event Listener.
- Deprecated Endpoint Listener.

- Added update to Import Registration and Export Registration.
- Added capability namespace definitions.
- Removed endpoint sharing.
- Removed text around setting of Framework UUID property (not needed for Core Frameworks version 4.3 or newer).
- Clarified the rules concerning Remote Service Endpoint identifier uniqueness.

123   JTA Transaction Services Specification

Version 1.0

123.1   Introduction

Transactions are the key abstraction to provide reliability with large scale distributed systems and are a primary component of enterprise systems. This specification provides an OSGi service based design for the Java Transaction Architecture (JTA) Specification, which describes the standard transaction model for Java applications. Providing the JTA specification as a service based model enables the use of independent implementations. This JTA Transaction Services Specification provides a managed model, where an Application Container (such as the Java EE EJB container) manages the transaction and the enlistment of resources, and an unmanaged model, where each application is responsible for these tasks itself.

This specification provides a brief overview of JTA and then the use of it through 3 transaction services: User Transaction, Transaction Manager, and Transaction Synchronization.

This specification is based on [1] *Java Transaction API Specification 1.1*.

123.1.1   Essentials

- *Portability* - It is important that applications are easy to port from other environments that support JTA.
- *Pluggability* - Allow different vendors to provide implementations of this specification.
- *JTA Compatible* - Support full JTA 1.1 Specification
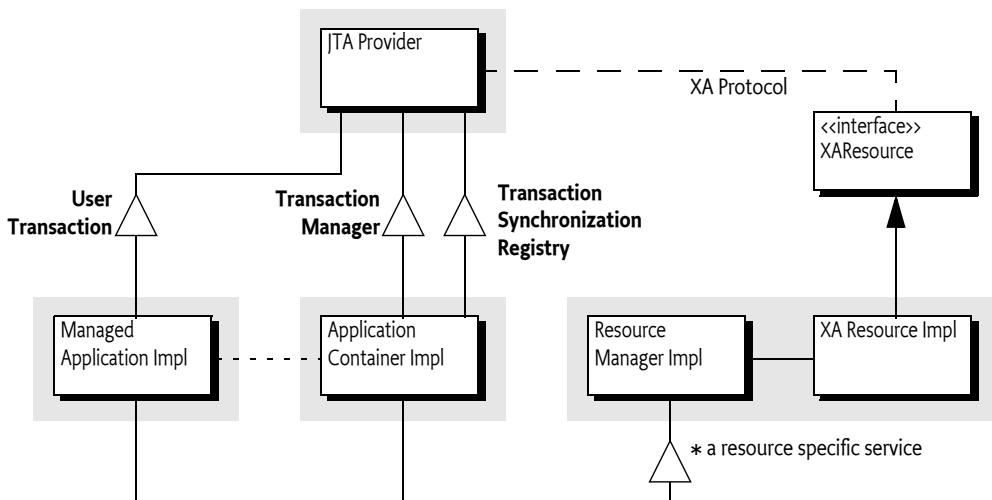
123.1.2   Entities

- *JTA Provider* - Implementation of this specification. It is responsible, on request from a Transaction Originator, for starting and ending transactions and coordinating the work of Resource Managers that become involved in each Transaction. This entity provides the User Transaction service, Transaction Manager service, and the Transaction Synchronization Registry service.
- *Transaction* - An atomic unit of work that is associated with a thread of execution.
- *Transaction Originator* - An Application or its Container, that directs the JTA Provider to begin and end Transactions.
- *User Transaction* - A service used by a Transaction Originator for beginning and ending transactions.
- *Transaction Manager* - A service used by a Transaction Originator for managing both transaction demarcation and enlistment of Durable Resources or Volatile Resources.
- *Transaction Synchronization Registry* - A service for enlistment of Volatile Resources for getting notifications before and after ending Transactions.
- *Application Bundle* - An entity that initiates work that executes under a Transaction.
- *Container* - An entity that is distinct from the Application and which provides a managed environment for Applications. Unmanaged environments do not distinguish between the Application and Container entities.

- *Resource Manager* - Provides the transactional resources whose work is externally coordinated by a JTA Provider. Examples of Resource Managers include databases, Java Message Service providers and enterprise information systems.
- *Durable Resource* - A resource whose work is made durable when the Transaction is successfully committed. Durable Resources can be enlisted with a Transaction to ensure that work is performed within the scope of the Transaction and to participate in the outcome of a Transaction. Durable Resource enlistment is the responsibility of the Application Bundle or its Container. Durable Resources implement the `javax.transaction.xa.XAResource` interface
- *Volatile Resource* - Resources that are associated with a Transaction but are no longer needed after the Transaction, for example transaction-scoped caches. Volatile Resources are registered with the JTA Provider to receive notifications before and after the outcome of the Transaction. Volatile Resources implement the `javax.transaction.Synchronization` interface
- *Transaction Services* - The triplet of the User Transaction, Transaction Manager, and Transaction Synchronization Registry services registered by the JTA Provider.

Figure 123.1   Transaction Service Specification Entities

123.1.3   Dependencies

This specification is based on the following packages:

```
javax.transaction
javax.transaction.xa
```

These packages must be exported as version 1.1.

123.1.4   Synopsis

The JTA Provider register the Transaction Services:

- *User Transaction* - Offers transaction demarcation capabilities to an Application bundle.
- *Transaction Manager* - Offers transaction demarcation and further transaction management capabilities to an Application Bundle or an Application Container.
- *Transaction Synchronization Registry* - Offers a callback registration service for volatile transactional participants wishing to be notified of the completion of the transaction.

A JTA Provider must register these services when it is started. A JTA Provider may put restrictions on which bundles can use these services. For example, in a Java EE environment, the JTA Provider does not expose the `TransactionManager` interface to applications. An OSGi environment which

supports the Java EE specifications will typically provide access to the Transaction Manager service only to Java EE Containers.

A typical example of the use of a transaction is for transferring money from one bank account to another. Two Durable Resources are involved, one provided by the database from which the money is to be withdrawn and another provided by the database to which the money will be deposited. An Application Bundle acting as the Transaction Originator gets the User Transaction service and uses it to begin a transaction. This transaction is associated with the current thread (implicitly) by the JTA Provider. On the same thread of execution, the Application Bundle connects to the database from which the money is to be withdrawn and updates the balance in the source account by the amount to be debited.

The database is a resource manager whose connections have associated XA Resources; the first time a connection is used within the scope of a new transaction the Application Bundle, or a Container, obtains the XA Resource associated with the connection and enlists it with the JTA Provider through the Transaction Manager service. On the same thread of execution, the Application Bundle connects to the second database and updates the balance in the target account by the amount to be credited. An XA Resource for the second connection is enlisted with the Transaction Manager service as well by the Application Bundle or a Container.

Now that the money has been transferred the Transaction Originator requests a commit of the Transaction (on the same thread of execution) via the User Transaction Service, causing the JTA Provider to initiate the two-phase commit process with the two Resource Managers through the enlisted XA Resources. The transaction is then atomically committed or rolled back.

123.2 JTA Overview

A transaction is a unit of work in which interactions with multiple participants can be coordinated by a third party such that the final outcome of these interactions has well-defined transactional semantics. A variety of well-known transaction models exist with specific characteristics; the transactions described in this specification provide *Atomic Consistent Isolated and Durable* (ACID) semantics as defined in [2] *XA+ Specification* whereby all the participants in a transaction are coordinated to an *atomic* outcome in which the work of all the participants is either completely committed or completely rolled back.

The [2] *XA+ Specification* defines a *Distributed Transaction Processing* (DTP) software architecture for transactional work that is distributed across multiple Resource Managers and coordinated externally by a Transaction Manager using the two-phase commit XA protocol. The DTP architecture defines the roles of the *Transaction Manager* and *Resource Manager*; this specification uses the term *JTA Provider* rather than *Transaction Manager* to distinguish it from the *Transaction Manager service*. Note that Distributed Transaction Processing does not imply distribution of transactions across multiple frameworks or JVMs.

The [1] *Java Transaction API Specification 1.1* defines the Java interfaces required for the management of transactions on the enterprise Java platform.

123.2.1 Global and Local Transactions

A transaction may be a *local transaction* or a *global transaction*. A local transaction is a unit of work that is local to a single Resource Manager and may succeed or fail independently of the work of other Resource Managers. A global transaction, sometimes referred to as a distributed transaction, is a unit of work that may encompass multiple Resource Managers and is coordinated by a JTA Provider external to the Resource Manager(s) as described in the DTP architecture. The term *transaction* in this specification always refers to a global transaction.

The JTA Provider is responsible for servicing requests from a Transaction Originator to create and complete transactions, it manages the state of each transaction it creates, the association of each

transaction with the thread of execution, and the coordination of any Resource Managers that become involved in the global transaction. The JTA Provider ensures that each transaction is associated with, at most, one application thread at a time and provides the means to move that association from one thread to another as needed.

The model for resource commit coordination is the *two phase commit* XA protocol, with Resource Managers being directed by the JTA Provider. The first time an Application accesses a Resource Manager within the scope of a new global transaction, the Application, or its Container, obtains an XA Resource from the Resource Manager and *enlists* this XA Resource with the JTA Provider.

At the end of a transaction, the Transaction Originator must decide whether to initiate a *commit* or *rollback* request for all the changes made within the scope of the Transaction. The Transaction Originator requests that the JTA Provider completes the transaction. The JTA Provider then negotiates with each enlisted Resource Manager to reach a coordinated outcome. A failure in the transaction at any point before the second phase of two-phase commit results in the transaction being rolled back.

XA is a *presumed abort* protocol and implementations of XA-compliant JTA Providers and Resource Managers can be highly optimized to perform no logging of transactional state until a commit decision is required. A Resource Manager durably records its prepare decision, and a JTA Provider durably records any commit decision it makes. Failures between a decision on the outcome of a transaction and the enactment of that outcome are handled during *transaction recovery* to ensure the atomic outcome of the transaction.

123.2.2 Durable Resource

Durable Resources are provided by Resource Managers and must implement the XAResource interface described in the [1] *Java Transaction API Specification 1.1*. An XAResource object is enlisted with a transaction to ensure that the work of the Resource Manager is associated with the correct transaction and to participate in the two-phase commit process. The XAResource interface is driven by the JTA Provider during the completion of the transaction and is used to direct the Resource Manager to commit or rollback any changes made under the corresponding transaction.

123.2.3 Volatile Resource

Volatile resources are components that do not participate in the two phase commit but are called immediately prior to and after the two phase commit. They implement the [1] *Java Transaction API Specification 1.1* Synchronization interface. If a request is made to commit a transaction then the volatile participants have the opportunity to perform some *before completion* processing such as flushing cached updates to persistent storage. Failures during the *before completion* processing must cause the transaction to rollback. In both the commit and rollback cases the volatile resources are called after two phase commit to perform *after completion* processing. *After completion* procession cannot affect the outcome of the transaction.

123.2.4 Threading

As noted above in *Global and Local Transactions* on page 425, a global transaction must not be associated with more than one application thread at a time but can be moved over time from one application thread to another. In some environments Applications run in containers which restrict the ability of the Application component to explicitly manage the transaction-thread association by restricting access to the Transaction Manager. For example, Java EE application servers provide web and EJB Containers for application components and, while the Containers themselves can explicitly manage transaction-thread associations, these containers do not allow the Applications to do so. Applications running in these containers are required to complete any transactions they start on that same application thread. In general, Applications that run inside a Container must follow the rules defined by that Container. For further details of the considerations specific to Java EE containers, see the section *Transactions and Threads* in [4] *Java Platform, Enterprise Edition (Java EE) Specification, v5.*

123.3   Application

An *Application* is a bundle that may use transactions, either as a Transaction Originator or as a bundle that is called as part of an existing transaction. A Transaction Originator Application bundle starts a transaction and end it with a commit or rollback using the User Transaction or Transaction Manager service.

A Transaction Originator Application bundle may not make use of Resource Managers itself but may simply provide transaction demarcation and then call other bundles which do use Resource Managers. In such a case the Transaction Originator Application bundle requires only the use of the User Transaction service for transaction demarcation. The called bundles may use the Transaction Manager service if they use Resource Managers.

Application Bundles that use Resource Managers have to know the enlistment strategy for the Resource Managers they use. There are two possibilities:

- *Application Bundle Enlistment* - The Application Bundle must enlist the Resource Managers itself. For each Resource Manager it uses it must enlist that Resource Manager with the Transaction Manager.
- *Container-Managed Enlistment* - An Application runs in a container, such as a Java EE Container, which manages the Resource Manager enlistment on behalf of the Application.

These scenarios are explained in the following sections.

123.3.1   No Enlistment

A Transaction Originator Application bundle that uses no Resource Managers itself but starts a Transaction before calling another bundle may use the *User Transaction* service to control the Transaction demarcation.

For example, an Application can use the User Transaction service to begin a global transaction:

```
UserTransaction ut = getUserTransaction();
ut.begin();
```

The User Transaction service associates a transaction with the current thread until that transaction is completed via:

```
UserTransaction ut = getUserTransaction();
ut.commit();
```

Or the equivalent rollback method. The getUserTransaction method implementation (not shown) can get the User Transaction service directly from the service registry or from an injected field.

123.3.2   Application Bundle Enlistment

An Application Bundle is responsible for enlisting Resource Managers itself. That is, it must enlist Resource Manager it uses with the *Transaction Manager* service. The Transaction Manager service is an implementation of the JTA TransactionManager interface, registered by the JTA Provider.

For example, an Application Bundle can get an XADataSource object from a Data Source Factory service. Such a Data Source object can provide an XAConnection object that then can provide an XAResource object. XAResource objects can then be enlisted with the Transaction Manager service.

For example:

```
TransactionManager tm;
XADataSource        left;
```

```
XADataSource        right;

void acid() throws Exception {
    tm.begin();
     Transaction transaction = tm.getTransaction();
    try {
        XAConnection left = this.left.getXAConnection();
        XAConnection right = this.right.getXAConnection();
        transaction.enlistResource( left.getXAResource());
        transaction.enlistResource( right.getXAResource());
        doWork(left.getConnection(), right.getConnection());
        tm.commit();
    } catch( Throwable t ) {
        tm.rollback();
        throw t; } }
// ...
void setTransactionManager( TransactionManager tm ) { this.tm= tm; }
void setDataSourceFactory( DataSourceFactory dsf ) {
    left = dsf.createXADataSource( getLeftProperties() );
    right = dsf.createXADataSource( getRightProperties() );
}
```

In the previous example, the Transaction Manager service could have been injected with a component model like Declarative Services:
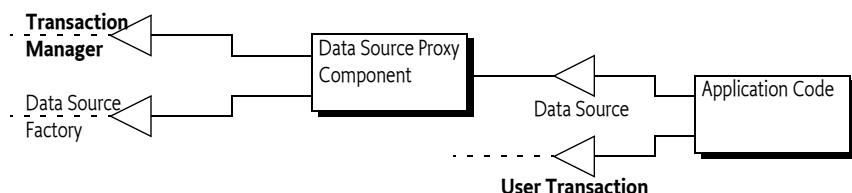
```
<reference interface="javax.transaction.TransactionManager"
    bind="setTransactionManager"/>
<reference name="dsf" interface="org.osgi.service.jdbc.DataSourceFactory"
    bind="setDataSourceFactory"/>
```

For example, it is possible to provide a Data Source service that provides automatic enlistment of the Connection as an XA Resource when one of its getConnection methods is called inside a transaction. The following code contains a Declarative Service component that implement this design. The component references a Transaction Manager service and a Data Source Factory service and provides a Data Source service that proxies an XA Data Source. Applications depend on the Data Source service, assuming that the Data Source service automatically enlists the connections it uses inside a transaction. See for an overview Figure 123.2 on page 428.

Figure 123.2   *Data Source Proxy*

This general purpose Data Source Proxy component can be fully configured by the Configuration Admin service to instantiate this component for each needed database connection. The Declarative Services service properties can be used to select a Data Source Factory for the required database driver (using the target), as well as provide the configuration properties for the creation of an XA Data Source. That is, such a component could be part of a support library.

The code for such an Application component could start like:

```
public class DataSourceProxy implements DataSource{
    Properties         properties  = new Properties();
    TransactionManager tm;
    XADataSource       xads;
```

The activate method is called when the component's dependencies are met, that is, there is a Transaction Manager service as well as a matching Data Source Factory service. In this method, the properties of the component are copied to a Properties object to be compatible with the Data Source Factory factory methods.

```
void activate(ComponentContext c) {
    // copy the properties set by the Config Admin into properties
    ...
}
```

The relevant methods in the Data Source Proxy component are the getConnection methods. The contract for this proxy component is that it enlists the XA Data Connection's XA Resource when it is called inside a transaction. This enlistment is done in the private enlist method.

```
public Connection getConnection() throws SQLException{
    XAConnection connection = xads.getXAConnection();
    return enlist(connection); }

public Connection getConnection(String username, String password)
        throws SQLException {
    XAConnection connection = xads.getXAConnection(username,password);
    return enlist(connection); }
```

The enlist method checks if there currently is a transaction active. If not, it ignores the enlistment, the connection will then not be connection to the transaction. If there is a current transaction, it enlists the corresponding XA Resource.

```
private Connection enlist(XAConnection connection)throws SQLException {
    try {
        Transaction transaction = tm.getTransaction();
        if (transaction != null)
            transaction.enlistResource( connection.getXAResource());
    } catch (Exception e) {
        SQLException sqle=
            new SQLException("Failed to enlist");
        sqle.initCause(e);
        throw sqle;
    }
    return connection.getConnection();
}
```

What remains are a number of boilerplate methods that forward to the XA Data Source or set the dependencies.

```
void setTransactionManager(TransactionManagertm) { this.tm = tm;}
void setDataSourceFactory(DataSourceFactory dsf) throws Exception{
    xads = dsf.createXADataSource(properties);}
public PrintWriter getLogWriter()
    throws SQLException { return xads.getLogWriter(); }

public int getLoginTimeout()
```

```
        throws SQLException { return xads.getLoginTimeout();}

public void setLogWriter(PrintWriter out)
    throws SQLException { xads.setLogWriter(out); }

public void setLoginTimeout(int seconds)
    throws SQLException { xads.setLoginTimeout(seconds);}
```

This is a fully coded example, it only lacks the configuration definitions for the Configuration Admin service.

This example Data Source proxy component makes it possible for an Application to depend on a Data Source service. The connections the Application uses from this Data Source are automatically transactional as long as there is a current transaction when the service is called. However, this approach only works when all bundles in the OSGi framework follow the same enlistment strategy because this specification does not provide a common enlistment strategy.

123.3.3   Container Managed Enlistment

The Application Container is responsible for enlisting Resource Managers used by the Application. For example, the Java EE Web and EJB Containers have a well defined model for managing resources within a transaction. If an Application runs inside a Java EE Container then it is the responsibility of the Java EE Container to handle the resource enlistment, the actual rules are beyond this specification.

A Transaction Originator Application bundle running inside a Container which manages any Resource Managers enlistment may use the User Transaction service for transaction demarcation, assuming this service is made available by the Container.

When a Java EE Container runs inside an OSGi framework then it must ensure that any services seen by its contained Applications are the same Transaction services as other bundles on that OSGi framework.

123.4   Resource Managers

Resource Managers perform work that needs to be committed or rolled back in a transaction. To participate in a transaction, a Resource Manager must have an XA Resource enlisted with the current transaction. This specification does not define how OSGi service implementations should be enlisted. This can be done by a Java EE Container, the Applications themselves, or through some other unspecified means.

123.5   The JTA Provider

The JTA Provider is the entity that provides the transaction services:

*   *User Transaction - A service that implements the JTA* UserTransaction *interface.*
*   *Transaction Manager - A service that implements the JTA* TransactionManager *interface.*
*   *Transaction Synchronization Registry - A service that implements the JTA* TransactionSynchronizationRegistry *interface.*

There can be at most one JTA Provider in an OSGi framework and this JTA Provider must ensure that at most one transaction is associated with an application thread at any moment in time. All JTA Provider's transaction services must map to the same underlying JTA implementation. All JTA services should only be registered once.

123.5.1 User Transaction

The User Transaction service may be used by an Application bundle, acting as the Transaction Originator, to demarcate transaction boundaries when the bundle has no need to perform resource enlistement.

123.5.2 Transaction Manager

The Transaction Manager service offers transaction demarcation and further transaction management capabilities, such as Durable and Volatile resource enlistment, to an Application bundle or Application Container.

123.5.3 Transaction Synchronization Service

The Transaction Synchronization Registry service may be used by an Application bundle or a Container. The service provides for the registration of Volatile Resources that implement the JTA Synchronization interface.

For example:

```
private class MyVolatile implements Synchronization{...}
TransactionSynchronizationRegistry tsr = ...; // may be injected
tsr.registerInterposedSynchronization(new MyVolatile());
```

123.6 Life Cycle

123.6.1 JTA Provider

The life cycle of the transaction services and bundles that make up the JTA Provider must be dealt with appropriately such that implementations always ensure the atomic nature of transactions. When the JTA Provider is stopped and its services are unregistered, the JTA Provider must make sure that all active transactions are dealt with appropriately. A JTA Provider can decide to rollback all active transactions or it can decide to keep track of existing active transactions and allow them to continue to their normal conclusion but not allow any new transactions to be created. Any failures caused by executing code outside their life cycle can be dealt with as general failures. From a transactional consistency point of view, stopping the bundle(s) that implement the JTA Provider while transactional work is in-flight, is no different from a failure of the framework hosting the JTA Provider. In either case transaction recovery is initiated by the JTA Provider after it has re-started.

There are well-defined XA semantics between a JTA Provider and Resource Managers in the event of a failure of either at any point in a transaction. If a Resource Manager bundle is stopped while it is involved in-flight transactions then the JTA Provider should exhibit the same external behavior it does in the event of a communication failure with the Resource Manager. For example a JTA Provider will respond to an XAER_RMFAIL response resulting from calling the XAResource commit method by retrying the commit. The mechanism used by the JTA Provider to determine when to retry the commit is a detail of the implementation.

123.6.2 Application Bundles

Applications can act in the role of the Transaction Originator. There is no guarantee that an Application that starts a transaction will always be available to complete the transaction since the client can fail independently of the JTA Provider. A failure of the Application Bundle to complete, in a timely fashion, a transaction it originated must finally result in the JTA Provider rolling back the transaction.

123.6.3   Error Handling

This specification does not define a specific error handling strategy. Exceptions and errors that occur during transaction processing can result in the transaction being marked *rollback-only* by the container or framework in which an Application runs or may be left for the Application to handle. An Application which receives an error or an exception while running under a transaction can choose to mark the transaction rollback-only.

123.7   Security

This specification relies on the security model of JTA.

123.8   References

[1]   *Java Transaction API Specification 1.1*
       http://www.oracle.com/technetwork/java/javaee/jta/index.html

[2]   *XA+ Specification*
       Version 2, The Open Group, ISBN: 1-85912-046-6

[3]   *Transaction Processing*
       J. Gray and A. Reuter. Morgan Kaufmann Publishers, ISBN 1.55860-190-2

[4]   *Java Platform, Enterprise Edition (Java EE) Specification, v5*
       http://jcp.org/en/jsr/detail?id=244

124 JMX™ Management Model Specification

Version 1.1

124.1 Introduction

The Java Management Extensions (JMX) is the standard API specification for providing a management interface to Java SE and Java EE applications. The JMX specification defines the design patterns, APIs, services and architecture for application management, network management and monitoring in the Java programming language. The need to administer, monitor and manage a container is today recognized as a prerequisite in the enterprise software domain.

While OSGi defines a rich API for controlling all aspects of the framework, this API is not suitable for direct usage in the JMX framework because it was not designed to be remoted. This specification provides an interface adaptation of the existing OSGi framework, which can be used to expose an OSGi Framework manipulation API to any JMX compliant implementation. Interfaces and system semantics for a monitoring system are specified for exposing the underlying artifacts of the OSGi framework such as services and bundles. Additionally, the management of a number of core and compendium services have been standardized in this document.

Finally, a standardized JMX object naming standard is proposed so that management objects are uniformly named across implementations such that any JMX compliant system can find, manipulate and interact with the framework and artifacts that it manages.

This specification requires version 1.2 or later of JMX, which implies the use of Java 5.

124.1.1 Essentials

- *Life Cycle* - Must allow support of full life cycle management of bundles.
- *Batch* - Support batch oriented operations to minimize the influence of network capacity and latency.
- *Compatible* - This specification must work naturally with JMX.
- *Efficient* - Minimize the number of registered objects to not overload the MBean Server and communication channels.
- *Open MBean* - Support the Open MBean layer of JMX instead of using domain specific objects.
- *Core* - Supports all the Framework's operations.
- *Core Services* - Support the framework services if registered, except for Conditional Permission Admin.
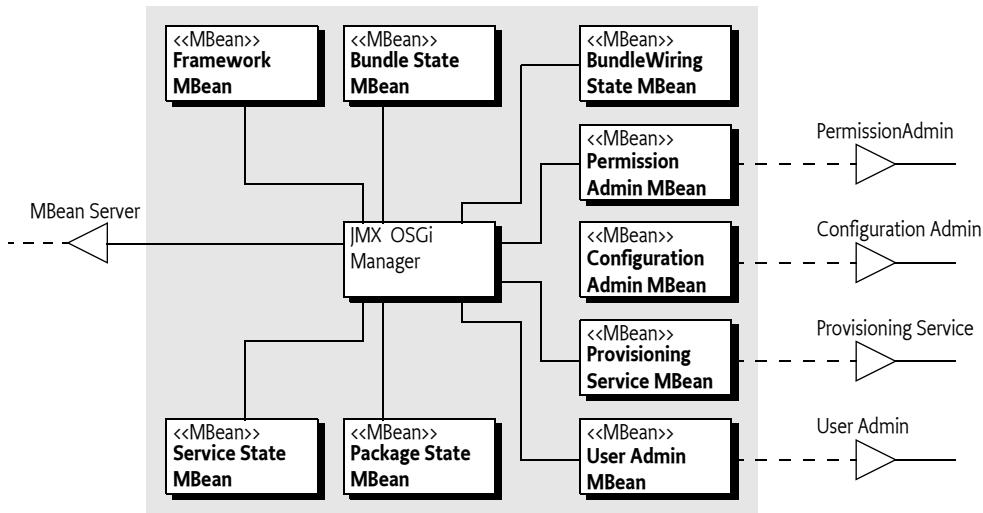
124.1.2 Entities

- *MBean* - A Managed Bean. The core concept of JMX to manage an entity.
- *MBean Server* - The MBean Server is the access point for registering MBeans.
- *Manager* - The entity that implements the MBeans and registers them with the registered MBean servers.
- *Object Name* - A name for an MBean registered with an MBean Server.

- *Bundle State MBean* - Provides central access to the state of a bundle in a framework.
- *Framework MBean* - Represents the general framework's state and can be used to manage the life cycle of bundles.
- *Bundle Wiring State MBean* - Provides access to the wiring state of the framework.
- *Service State MBean* - Provides access to the service information in the service registry. It provides both a general MBean interface as well as an Open Type description.
- *Configuration Admin MBean* - Can be used to manipulate a Configuration Admin service.
- *Permission Admin MBean* - Provides access to the Permission Admin service.
- *Provisioning Service MBean* - Provides access to the Provisioning Service.
- *User Admin MBean* - Provides access to the User Admin service.
- *Item* - A helper class to create Open Types. This class is intended to make the Javadoc easier to navigate and keep definitions close together. This is otherwise hard to do with Open Type. This class has no utility for management applications.
- *Open Type* - A JMX metadata standard to describe MBeans.
- *Remote Manager* - The entity accessing a MBean Server remotely.
- *JConsole* - The default Java Remote Manager.

Figure 124.1   MBeans

124.1.3   Synopsis

This specification plays a part in both the OSGi framework as well as in a remote manager.

A JMX OSGi manager bundle obtains one or more MBean servers that are registered as services. The JMX OSGi manager then registers all its managed beans: Framework MBean, Bundle State MBean, Package State MBean, and the Service State MBean under their JMX object names. If a number of optional services are registered, then the JMX OSGi bundle must also register a corresponding MBean with the MBean server for each of the services that it can obtain.

A remote manager can access an MBean Server running in a (remote) VM. The remote manager can then discover any MBeans. These MBeans can be manipulated as dynamic types or as specific types as outlined in this specification.

124.2   JMX Overview
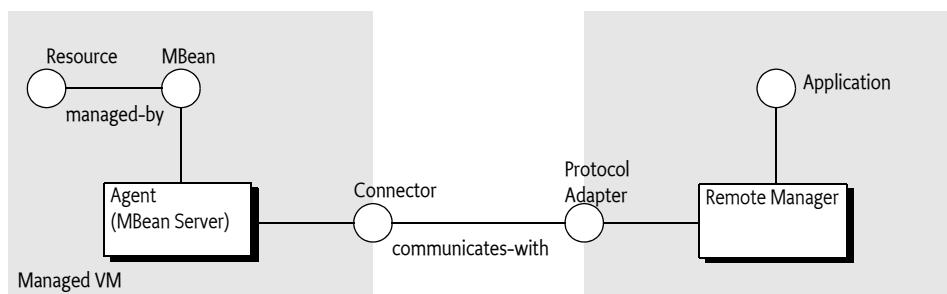
JMX is a specification which defines how arbitrary remote communication protocols and mechanisms can be adapted to interact with the underlying management APIs exposed by JMX compliant implementations. JMX is not a remote communication standard, the actual protocols can vary. The JMX architecture is composed of three levels:

- *Instrumentation* - The managed resources of the system are instrumented using *managed beans* (a.k.a. MBeans) which expose their management interfaces through a JMX agent for remote management and monitoring.
- *Agent* - The JMX agent layer is mainly represented by the MBean server. This is the *managed object* server where the MBeans are registered. The JMX agent includes a set of functions for manipulating the registered MBeans, which directly expose and control the underlying resources, and then make them available to remote managers.
- *Remote Manager* - The remote management layer provides the specification for the actual remote communication protocol adapters and defines standard *connectors* which make the JMX agent accessible to remote managers outside of the Java process that hosts the agent.

The JMX Architecture is depicted in Figure 124.2.

Figure 124.2   *JMX Architecture*

124.2.1   Connectors and Adapters

Connectors are used to connect an agent with a remote JMX-enabled managers. This form of communication involves a connector in the JMX agent and a connector client in the management application. Protocol adapters provide a management view of the JMX agent through a given protocol.

Remote managers that connect to a protocol adapter are usually specific to the given protocol. Remote Managers can be generic consoles (such as JConsole; see [8] *Using JConsole to Monitor Applications*), or domain-specific monitoring applications. External applications can interact with the MBeans through the use of JMX connectors and protocol adapters.

124.2.2   Object Name

All managed objects in JMX are referenced via JMX *Object Names*. Object Names are strings which can be resolved within the context of a JMX MBean Server in order. An Object Name consists of two parts:

```
ObjectName  ::= domain ':' properties
properties  ::= property ( ',' property )*
```

To avoid collisions between MBeans supplied by different vendors, a recommended convention is to begin the domain name with the reverse DNS name of the organization that specifies the MBeans, followed by a full stop ('.' \u002E) and a string whose interpretation is determined by that organization.

MBeans specified by the OSGi Alliance have domains that start with osgi.

124.2.3   MBeans

Any object can be registered with an MBean Server and manipulated remotely over an *MBean Server Connection*. An MBean Server Connection can represent the a local MBean Server or a remote MBean Server. An MBean is always identified by an *Object Name*. The Object Name identifies a remote MBean uniquely within a specific MBean Server Connection.

Standard manipulations of a remote MBean are done through *attributes* and *operations*, which are similar to properties and methods for Java beans. Not all methods on the implementation class can be used, the registering party must specifically provide access to the methods that can be called remotely. The registrar can define the exposed operations with the following mechanisms:

*   *Design Pattern* - Let the registered object implement an *MBean interface* that has the fully qualified name of the implementation class suffixed with MBean. The MBean server will then limit access to attributes and properties defined in the MBean interface. For example, the com.acme.Resource class should implement the com.acme.ResourceMBean interface. The com.acme.ResourceMBean interface would define the properties and operations.
*   *Dynamic MBean* - Register a Dynamic MBean, which handles the access to the operations and attributes programmatically. The JMX specification provides the DynamicMBean interface for this purpose. If the MBean registered with an MBean Server implements this interface, then the MBean Server must get the MBean's metadata through the DynamicMBean interface instead of using reflection. Therefore, Dynamic MBeans can provide more rich metadata that describes their operations and attributes.
*   *Standard MBean* - Register a Standard MBean. A standard MBean works the same as the previous bullet but does not require the implementation class name to map to the MBean interface name.

Attributes map to properties on the registered MBean interface and operations allow the invocation of an arbitrary method on the remote MBean with arbitrary parameters. The following code example shows how to get a the size property of a remote MBean in this way:

```
void drop( MBeanServerConnection mbs, ObjectNameobjectName) {
  Integer sizeI = (Integer)
        mbs.getAttribute(objectName, "Size");
  int size = sizeI.intValue();
  if (size > desiredSize) {
    mbs.invoke(objectName,"dropOldest",
        new Integer[] {new Integer(size - desiredSize)},
        new String[] {"int"});
  }
}
```

In release 1.2 the JMX specification introduced the *MBean Server Invocation Handler* to simplify the manipulation of the remote MBeans by creating a *proxy* for an *MBean interface* that implements all the relevant methods. An MBean interface defines the methods and properties for an MBean. The proxy has a reference to an *MBean Server Connection*, it can therefore automate the invocation of the appropriate methods from the MBean interface. Therefore, by using an MBean interface, it is possible to simplify the remote manager:

```
MBeanServer mbs = ...;
CacheControlMBean cacheControl = (CacheControlMBean)
 MbeanServerInvocationHandler.newProxyInstance(
    mbs, objectName, CacheControlMBean.class, false);

int size = cacheControl.getSize();
if (size > desiredSize)
```

```
cacheControl.dropOldest(size - desiredSize);
```

The creation of the proxy is somewhat verbose, but once it is available, the MBean can be accessed like a local object. The proxy is much easier to use and read, and much less error-prone, than accessing the MBean Server method through invoking operations and getting attributes.

The MBean interface can also ensure a certain amount of type safety. The MBean implementation can implement the MBean interface and the remote manager uses the proxy implementing this interface. However, neither is required. The MBean can directly implement the methods without implementing the interface and the remote manager can directly manipulate the attributes and invocations.

The key advantage is therefore the documentation of the management interface. Using an MBean interface, this can be done very concisely and it allows the usage of standard tools for Java source code and Javadoc.

124.2.4   Open Types

The distributed nature of remote management poses a number of problems for exchanging general objects.

- *Versioning* - All participating parties require access to the same version of the object's class.
- *Serialization* - Not all objects are easy to serialize.
- *Size* - Arbitrary objects can transitively link to large amounts of data.
- *Descriptive* - Classes provide little or no support for editing.
- *Limited* - Classes are Java specific, making it harder to interact with non-Java environments.

An alternative is to limit the management types to be exchanged to small, well defined set. Open MBeans limit the used data types to small number of types called the *basic types*. These types are supported by all JMX 1.2 and later implementations. This basic set of types contains:

- *Primitives* - boolean, byte, char, short, int, long, float, double.
- *Primitive Arrays* - boolean[], byte[], char[], short[], int[], long[], float[], double[].
- *Wrappers* - Boolean, Byte, Character, Short, Integer, Long, Float, Double.
- *Scalars* - String, BigDecimal, BigInteger, Date, ObjectName.
- *Complex* - CompositeData, TabularData, and complex arrays.
- *Return* - Void, operation return only.

The Complex types are unique to JMX, they are used to provide access to complex data (like objects) without using classes. The complex types are *self describing*. The metadata associated with these complex types allow a remote manager to discover the structure and automatically construct a (graphic) user interface for these complex objects.

Open MBeans must be Dynamic MBeans when registered. Furthermore, they must provide Open MBean variations of the Info objects that describe the operations and attributes.

124.3   OSGi JMX Management

The OSGi JMX Management model is based on Open MBeans, see *Open Types* on page 437. This specification declares a number of MBeans for the core Framework, some of the core services, and a number of compendium services. Though Open MBeans are based on Dynamic MBeans, this specification uses the traditional MBean interface to define the management interaction patterns. The implementer of this specification must register an implementation of these interfaces as a Dynamic MBean. An implementation should provide the additional Open MBeans Info objects for the operations and attributes.

This specification defines the following Open MBeans:

- *Core Framework* - `FrameworkMBean`, `BundleStateMBean`, `ServiceStateMBean`, `BundleWiringStateMBean`, and `PackageStateMBean`.
- *Core Services* - `PermissionAdminMBean`. The Conditional Permission Admin is not included in this specification.
- *Compendium Services* - `ConfigurationAdminMBean`, `UserAdminMBean`, `ProvisioningServiceMBean`

124.3.1 Naming

The MBean interfaces have been named after the service they manage. That is the `ConfigurationAdminMBean` interface manages the Configuration Admin service, which is modelled with the `ConfigurationAdmin` interface.

Package names are constructed from taking the corresponding resource package and inserting `jmx.` after `org.osgi`. For example

```
org.osgi.framework          org.osgi.jmx.framework
org.osgi.service.cm         org.osgi.jmx.service.cm
```

It is not possible to use the MBean interface design pattern because the MBean interfaces are in OSGi packages. The design pattern requires the fully qualified name of the implementation suffixed with MBean to match the MBean interface name. This would require that the implementation class resides in an OSGi package, which would extend these packages.

However, the `StandardMBean` class allows the association of one of the OSGi MBean interfaces with an arbitrary class.

124.3.2 Object Naming

Object Names for OSGi managed MBeans must follow the following structure:

```
object-name    ::= ( core | compendium )
                   ',version=' version
                   ',framework=' framework
                   ',uuid=' uuid
                   (',' key '=' value )*
core           ::= 'osgi.core:' framework-type
compendium     ::= 'osgi.compendium:' service-type
framework-type ::= ( 'type=' token ) | service-type
service-type   ::= 'service=' token
framework      ::= ‹Bundle symbolic name of the system bundle›
uuid           ::= ‹org.osgi.framework.uuid Framework property'svalue›
key            ::= ‹any jmx supported key›
value          ::= ‹any jmx supported value›
```

There are the following additional constraints:

- *Spaces* - Spaces between any of the terminals are not permitted.
- *Version* - The `version` must be limited to a major and minor version part. The given version must identify the package of the corresponding resource. For example, if the Configuration Admin service is on version `1.3.2.200910101250`, then the version in the Object Name must be `1.3`.
- *Service* - The `service-type` should use the package name of the corresponding service. For example, for Configuration Admin this would be `service=cm`.

The Object Name must contain the framework bundle symbolic name and its UUID so that multiple instances on the same VM can be discriminated. An example of an Object Name is:

```
osgi.core:type=framework,version=1.7,framework=org.apache.felix.framework, «
```

```
uuid=f81d4fae-7dec-11d0-a765-00a0c91e6bf6
```

The advantage of the framework property is that it can be used to simplify the querying for the MBeans using Object Name *patterns*. Patterns are names have an asterisk ('∗' \u002A). For instance, the following query allows a client to find all Framework MBeans for an Apache Felix implementation without having to rely on knowing the UUID:

```
ObjectName on = new ObjectName(
    "osgi.core:type=framework,"
    + "version=1.7,framework=org.apache.felix.framework,*");
Set<ObjectInstance> instances = mserver.queryMBeans(on,null);
```

Furthermore, in many cases, a JMX client may appropriately assume that only a single instance of the OSGi framework exists in the managed system, as in the following example:

```
ObjectName on = new ObjectName("osgi.core:type=framework,version=1.7,*");
Set<ObjectInstance> instances = mserver.queryMBeans(on,null);
```

The uuid and framework property keys are only applicable to OSGi JMX Management Model Specification Version 1.1 and above.

To maintain backward compatibility, a OSGi JMX Framework package Version 1.7 may register the first instantiation of an OSGi framework using both the Version 1.0 Object Names as well as the Object Names outlined in this specification. In other words, a JMX client may not specify the uuid and/ or framework properties, and still retrieve the MBeans for a OSGi framework instance.

The actual object name prefixes are defined in the MBean interfaces. For example, the Object Name for the Configuration Admin MBean is:

```
osgi.compendium:service=cm,version=1.3
```

It is the responsibility of the party registering the MBean to suffix this with the framework and UUID.

In this specification, all management interfaces are specified to return opaque Strings or longs rather than Object Names so that the MBean interfaces contain no JMX specific artifacts and can be used with a variety of remote access protocols such as SNMP, etc. Non JMX use of these APIs can use these Strings as their own opaque identifiers without any change to the interfaces themselves.

124.3.3 The MBean Server

An implementation of this specification must find all MBean Servers services that it has access to. It should then register all MBeans with each server found in the service registry.

A compliant implementation must register all the framework's MBeans: FrameworkMBean, BundleStateMBean, ServiceStateMBean, BundleWiringStateMBean and PackageStateMBean. The registration of the compendium services is optional. However, if they are registered they must implement the behavior as defined in this specification.

124.3.4 Registrations

The OSGi MBeans are designed to minimize the notifications. That is, the objects model a command interface to access the required information. Their registration is not intended to signify anything else than the start of the manager bundle and the availability of the underlying resource.

Implementations must always register only one of each of the Framework MBean types (Framework MBean, Service State MBean, Bundle State MBean, Wiring State MBean, and Package State MBean). All other MBean types depend on the registered services they manage. Each service requires its unique MBean. If no corresponding service is present, then no MBean should be registered. Modified events must be ignored. If a manager supports a specific OSGi MBean for a compendium service then it must register an MBean for each instance of that service.

124.4   MBeans

This specification defines MBean interfaces listed in the following table. The Object Name specified in this table is broken into a number of lines for readability, however, newlines and whitespace is not allowed in the Object Name.

Table 124.1   *MBeans*

MBean	Object Name	Description
FrameworkMBean	osgi.core: type=framework, version=1.7	Provides access to bundle life cycle methods of the framework including batch install and update operations.
BundleStateMBean	osgi.core: type=bundleState, version=1.7	Provides detailed access to the state of one bundle and aggregated state of a group of bundles.
ServiceStateMBean	osgi.core: type=serviceState, version=1.7	Provides detailed access to the state of one service and aggregated state of a group of services.
PackageStateMBean	osgi.core: type=packageState, version=1.5	Provides detailed access to the state of one package and aggregated state of a group of packages.
PermissionAdminMBean	osgi.core: service=permissionadmin, version=1.2	Based on the Permission Admin service.
ConfigurationAdminMBean	osgi.compendium: service=cm, version=1.3	Manages a Configuration Admin service.
ProvisioningServiceMBean	osgi.compendium: service=provisioning, version=1.2	Manages a Provisioning Service.
UserAdminMBean	osgi.compendium: service=useradmin, version=1.1	Manages a User Admin service.
BundleWiringStateMBean	osgi.core: service=wiringState, version=1.1	Reflects the Framework's wiring state.

124.5   Item

The MBean interfaces do not only define the Java interface, they also define the Open Types. These types are defined with the Item class in this specification to simplify the definitions; the Item class has no role in a management application. The Item class is used to allow the items used in Compos-

ite Types to be encoded in the interface. This is not possible with the standard Open Types because they use exceptions and use parallel arrays. For example, the following code defines a static Open Type without the Item class:

```
static CompositeType HEADER;
static {
   try {
     HEADER = new CompositeType( "HEADER" "This is a header",
         new String[] {"KEY", "VALUE"},
         new String[] {"A key for a header", "A value for a header"},
         new OpenType[] { SimpleType.STRING, SimpleType.STRING });
   catch(OpenDataException e) {
     ...
   }
}
```

This code can be replaced with the Item class:

```
static Item KEY = new Item("KEY", "A key forheader", SimpleType.STRING );
static Item VALUE = new Item("VALUE", "A value for header",SimpleType.STRING );
static CompositeType HEADER = Item.composite( "HEADER", "Thisis a header",
    KEY, VALUE );
```

The Item class also provides a number of convenience methods to construct the different Open Types. However, the intention is to simplify the specification definitions, not as an aid in management operations.

124.6   Security

Exposing any system remotely opens up a, potentially, devastating security hole in a system. Remote entities should establish their identity and the management system should be able to control the access these entities have over the management system. JMX seamlessly inter operates with the Java Authentication and Authorization Service (JAAS) and Java 2 platform Standard Edition (Java SE) Security Architecture.

The JMX OSGi manager must have access to the services it manages and the operations it invokes. It is likely that this bundle requires All Permission because it needs to invoke operations on the Conditional Permission Admin. It is strongly advised that implementations limit the set of available permissions based on authenticating the remote manager.

124.7   org.osgi.jmx

OSGi JMX Package Version 1.1.

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

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

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

124.7.1   Summary

- Item - The item class enables the definition of open types in the appropriate interfaces.
- JmxConstants - Constants for OSGi JMX Specification.

124.7.2   public class Item

The item class enables the definition of open types in the appropriate interfaces. This class contains a number of methods that make it possible to create open types for CompositeType, TabularType, and ArrayType. The normal creation throws a checked exception, making it impossible to use them in a static initializer. The constructors are also not very suitable for static construction. An Item instance describes an item in a Composite Type. It groups the triplet of name, description, and Open Type. These Item instances allows the definitions of an item to stay together.

Concurrency Immutable

124.7.2.1   public Item(String name,String description,OpenType type,String ... restrictions)

name The name of the item.

description The description of the item.

type The Open Type of this item.

restrictions Ignored, contains list of restrictions

☐ Create a triple of name, description, and type. This triplet is used in the creation of a Composite Type.

124.7.2.2   public static ArrayType arrayType(int dim,OpenType elementType)

dim The dimension

elementType The element type

☐ Return a new Array Type.

Returns A new Array Type

124.7.2.3   public static CompositeType compositeType(String name,String description,Item ... items)

name The name of the Tabular Type.

description The description of the Tabular Type.

items The items that describe the composite type.

☐ Create a Composite Type

Returns a new Composite Type

Throws RuntimeException– when the Tabular Type throws an OpenDataException

124.7.2.4   public static CompositeType extend(CompositeType parent,String name,String description,Item ... items)

parent The parent type, can be null

name The name of the type

description The description of the type

items The items that should be added/override to the parent type

☐ Extend a Composite Type by adding new items. Items can override items in the parent type.

Returns A new Composite Type that extends the parent type

Throws RuntimeException– when an OpenDataException is thrown

124.7.2.5   **public static TabularType tabularType(String name,String description,CompositeType rowType,String ...**
                     index)

name   The name of the Tabular Type.

description   The description of the Tabular Type.

rowType   The Open Type for a row

index   The names of the items that form the index .

☐   Create a Tabular Type.

Returns   A new Tabular Type composed from the parameters.

Throws   RuntimeException– when the Tabular Type throws an OpenDataException

124.7.3   public class JmxConstants

Constants for OSGi JMX Specification. Additionally, this class contains a number of utility types that are used in different places in the specification. These are LONG_ARRAY_TYPE, STRING_ARRAY_TYPE, and PROPERTIES_TYPE.

Concurrency   Immutable

124.7.3.1   **public static final String ARRAY_OF = "Array of "**

For an encoded array we need to start with ARRAY_OF. This must be followed by one of the names in SCALAR.

124.7.3.2   **public static final String BIGDECIMAL = "BigDecimal"**

Value for PROPERTY_TYPE value in the case of java.math.BigDecimal

124.7.3.3   **public static final String BIGINTEGER = "BigInteger"**

Value for PROPERTY_TYPE value in the case of java.math.BigInteger

124.7.3.4   **public static final String BOOLEAN = "Boolean"**

Value for PROPERTY_TYPE value in the case of java.lang.Boolean

124.7.3.5   **public static final String BYTE = "Byte"**

Value for PROPERTY_TYPE value in the case of java.lang.Byte

124.7.3.6   **public static final String CHARACTER = "Character"**

Value for PROPERTY_TYPE value in the case of java.lang.Character

124.7.3.7   **public static final String DOUBLE = "Double"**

Value for PROPERTY_TYPE value in the case of java.lang.Double

124.7.3.8   **public static final String FLOAT = "Float"**

Value for PROPERTY_TYPE value in the case of java.lang.Float

124.7.3.9   **public static final String INTEGER = "Integer"**

Value for PROPERTY_TYPE value in the case of java.lang.Integer

124.7.3.10   **public static final String KEY = "Key"**

The key KEY.

124.7.3.11   **public static final Item KEY_ITEM**

The key of a property. The key is KEY and the type is SimpleType.STRING.

124.7.3.12	**public static final String LONG = "Long"**
	Value for PROPERTY_TYPE value in the case of java.lang.Long
124.7.3.13	**public static final ArrayType LONG_ARRAY_TYPE**
	The MBean Open type for an array of longs
124.7.3.14	**public static final String OSGI_COMPENDIUM = "osgi.compendium"**
	The domain name of the selected OSGi compendium MBeans
124.7.3.15	**public static final String OSGI_CORE = "osgi.core"**
	The domain name of the core OSGi MBeans
124.7.3.16	**public static final String P_BOOLEAN = "boolean"**
	Value for PROPERTY_TYPE value in the case of the boolean primitive type.
124.7.3.17	**public static final String P_BYTE = "byte"**
	Value for PROPERTY_TYPE value in the case of the byte primitive type.
124.7.3.18	**public static final String P_CHAR = "char"**
	Value for PROPERTY_TYPE value in the case of the char primitive type.
124.7.3.19	**public static final String P_DOUBLE = "double"**
	Value for PROPERTY_TYPE value in the case of the double primitive type.
124.7.3.20	**public static final String P_FLOAT = "float"**
	Value for PROPERTY_TYPE value in the case of the float primitive type.
124.7.3.21	**public static final String P_INT = "int"**
	Value for PROPERTY_TYPE value in the case of the int primitive type.
124.7.3.22	**public static final String P_LONG = "long"**
	Value for PROPERTY_TYPE value in the case of the long primitive type.
124.7.3.23	**public static final String P_SHORT = "short"**
	Value for PROPERTY_TYPE value in the case of the short primitive type.
124.7.3.24	**public static final TabularType PROPERTIES_TYPE**
	Describes a map with properties. The row type is PROPERTY_TYPE. The index is defined to the KEY of the property.
124.7.3.25	**public static final CompositeType PROPERTY_TYPE**
	A Composite Type describing a a single property. A property consists of the following items KEY_ITEM, VALUE_ITEM, and TYPE_ITEM.
124.7.3.26	**public static final List<String> SCALAR**
	A set of all scalars that can be used in the TYPE property of a PROPERTIES_TYPE. This contains the following names:

- BIGDECIMAL

- BIGINTEGER
- BOOLEAN
- BYTE
- CHARACTER
- DOUBLE
- FLOAT
- INTEGER
- LONG
- SHORT
- STRING
- VERSION
- P_BYTE
- P_CHAR
- P_DOUBLE
- P_FLOAT
- P_INT
- P_LONG
- P_SHORT

124.7.3.27 **public static final String SHORT = "Short"**

Value for PROPERTY_TYPE value in the case of java.lang.Short

124.7.3.28 **public static final String STRING = "String"**

Value for PROPERTY_TYPE value in the case of java.lang.String

124.7.3.29 **public static final ArrayType STRING_ARRAY_TYPE**

The MBean Open type for an array of strings

124.7.3.30 **public static final String TYPE = "Type"**

The key TYPE.

124.7.3.31 **public static final Item TYPE_ITEM**

The type of the property. The key is TYPE and the type is SimpleType.STRING. This string must follow the following syntax:

```
type      ::= scalar | vector | array
vector    ::= 'Vector of' scalar
array     ::= 'Array of' (scalar | primitive)
scalar    ::= 'String' | 'BigInteger' | 'BigDecimal'
              | 'Byte' | 'Character' | 'Short'
              | 'Integer' | 'Long' | 'Float'
              | 'Double' | 'Version'
primitive ::= 'byte' | 'char' | 'short'
              | 'int' | 'long' | 'float'
              | 'double'
```

This encoding does not support arrays in vectors or arrays. Arrays and vectors can only contain scalars. List properties are encoded as arrays. Empty lists, arrays or vectors are not represented. Null is not an allowed value.

For example, the encoding of a byte array byte[] {1,2,3,5,7} would look like:

```
type:  'Array of byte'
value: 1,2,3,5,7
```

Quoting can be used as follows:

```
type:  'Array of String'
value: 'abc', 'def', '\'quoted\'', "'quoted'", "\\"
```

124.7.3.32   **public static final String VALUE = "Value"**

The key VALUE.

124.7.3.33   **public static final Item VALUE_ITEM**

The value of a property. The key is VALUE and the type is SimpleType.STRING. A value will be encoded by the string given in TYPE. The syntax for this type is given in TYPE_ITEM.

124.7.3.34   **public static final String VECTOR_OF = "Vector of "**

For an encoded vector we need to start with ARRAY_OF. This must be followed by one of the names in SCALAR.

124.7.3.35   **public static final String VERSION = "Version"**

Value for PROPERTY_TYPE value in the case of Version

Since 1.1

124.8 org.osgi.jmx.framework

OSGi JMX Framework Package Version 1.7.

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

Example import for consumers using the API in this package:

Import-Package: org.osgi.jmx.framework; version="[1.7,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.jmx.framework; version="[1.7,1.8)"

124.8.1 Summary

- `BundleStateMBean` - This MBean represents the Bundle state of the framework.
- `FrameworkMBean` - The FrameworkMBean provides mechanisms to exert control over the framework.
- `PackageStateMBean` - This MBean provides information about the package state of the framework.
- `ServiceStateMBean` - This MBean represents the Service state of the framework.

124.8.2 public interface BundleStateMBean

This MBean represents the Bundle state of the framework. This MBean also emits events that clients can use to get notified of the changes in the bundle state of the framework.

Concurrency Thread-safe

124.8.2.1   **public static final String ACTIVATION_POLICY_USED = "ActivationPolicyUsed"**

The key ACTIVATION_POLICY_USED, used in ACTIVATION_POLICY_USED_ITEM.

124.8.2.2   **public static final Item ACTIVATION_POLICY_USED_ITEM**

The item containing the indication whether the bundle activation policy must be used in
BUNDLE_TYPE. The key is ACTIVATION_POLICY_USED and the type is SimpleType.BOOLEAN.

124.8.2.3   **public static final String ACTIVE = "ACTIVE"**

Constant ACTIVE for the STATE

124.8.2.4   **public static final CompositeType BUNDLE_EVENT_TYPE**

The Composite Type that represents a bundle event. This composite consists of:

- IDENTIFIER
- LOCATION
- SYMBOLIC_NAME
- EVENT

124.8.2.5   **public static final CompositeType BUNDLE_TYPE**

The Composite Type that represents a bundle. This composite consist of:

- EXPORTED_PACKAGES
- FRAGMENT
- FRAGMENTS
- HEADERS
- HOSTS
- IDENTIFIER
- IMPORTED_PACKAGES
- LAST_MODIFIED
- LOCATION
- ACTIVATION_POLICY_USED
- PERSISTENTLY_STARTED
- REGISTERED_SERVICES
- REMOVAL_PENDING
- REQUIRED
- REQUIRED_BUNDLES
- REQUIRING_BUNDLES
- START_LEVEL
- STATE
- SERVICES_IN_USE
- SYMBOLIC_NAME
- VERSION

It is used by BUNDLES_TYPE.

124.8.2.6   **public static final TabularType BUNDLES_TYPE**

The Tabular Type for a list of bundles. The row type is BUNDLE_TYPE and the index is IDENTIFIER.

124.8.2.7   **public static final String EVENT = "BundleEvent"**

The key EVENT, used in EVENT_ITEM.

124.8.2.8   **public static final Item EVENT_ITEM**

The item containing the event type. The key is EVENT and the type is SimpleType.INTEGER

124.8.2.9   **public static final String EXPORTED_PACKAGES = "ExportedPackages"**

The key EXPORTED_PACKAGES, used in EXPORTED_PACKAGES_ITEM.

124.8.2.10   **public static final Item EXPORTED_PACKAGES_ITEM**

The item containing the exported package names in BUNDLE_TYPE .The key is
EXPORTED_PACKAGES and the the type is JmxConstants.STRING_ARRAY_TYPE.

124.8.2.11   **public static final String FRAGMENT = "Fragment"**

The key FRAGMENT, used in FRAGMENT_ITEM.

124.8.2.12   **public static final Item FRAGMENT_ITEM**

The item containing the fragment status in BUNDLE_TYPE. The key is FRAGMENT and the the type
is SimpleType.BOOLEAN.

124.8.2.13   **public static final String FRAGMENTS = "Fragments"**

The key FRAGMENTS, used in FRAGMENTS_ITEM.

124.8.2.14   **public static final Item FRAGMENTS_ITEM**

The item containing the list of fragments the bundle is host to in BUNDLE_TYPE. The key is FRAG-
MENTS and the type is JmxConstants.LONG_ARRAY_TYPE.

124.8.2.15   **public static final CompositeType HEADER_TYPE**

The Composite Type describing an entry in bundle headers. It consists of KEY_ITEM and
VALUE_ITEM.

124.8.2.16   **public static final String HEADERS = "Headers"**

The key HEADERS, used in HEADERS_ITEM.

124.8.2.17   **public static final Item HEADERS_ITEM**

The item containing the bundle headers in BUNDLE_TYPE. The key is HEADERS and the the type is
HEADERS_TYPE.

124.8.2.18   **public static final TabularType HEADERS_TYPE**

The Tabular Type describing the type of the Tabular Data value that is returned from
getHeaders(long) method. The primary item is KEY_ITEM.

124.8.2.19   **public static final String HOSTS = "Hosts"**

The key HOSTS, used in HOSTS_ITEM.

124.8.2.20   **public static final Item HOSTS_ITEM**

The item containing the bundle identifiers representing the hosts in BUNDLE_TYPE. The key is
HOSTS and the type is JmxConstants.LONG_ARRAY_TYPE

124.8.2.21   **public static final String IDENTIFIER = "Identifier"**

The key IDENTIFIER, used in IDENTIFIER_ITEM.

124.8.2.22   **public static final Item IDENTIFIER_ITEM**

The item containing the bundle identifier in BUNDLE_TYPE. The key is IDENTIFIER and the the type is SimpleType.LONG.

124.8.2.23   **public static final String IMPORTED_PACKAGES = "ImportedPackages"**

The key IMPORTED_PACKAGES, used in EXPORTED_PACKAGES_ITEM.

124.8.2.24   **public static final Item IMPORTED_PACKAGES_ITEM**

The item containing the imported package names in BUNDLE_TYPE .The key is IMPORTED_PACKAGES and the the type is JmxConstants.STRING_ARRAY_TYPE.

124.8.2.25   **public static final String INSTALLED = "INSTALLED"**

Constant INSTALLED for the STATE

124.8.2.26   **public static final String KEY = "Key"**

The key KEY, used in KEY_ITEM.

124.8.2.27   **public static final Item KEY_ITEM**

The item describing the key of a bundle header entry. The key is KEY and the type is SimpleType.STRING.

124.8.2.28   **public static final String LAST_MODIFIED = "LastModified"**

The key LAST_MODIFIED, used in LAST_MODIFIED_ITEM.

124.8.2.29   **public static final Item LAST_MODIFIED_ITEM**

The item containing the last modified time in the BUNDLE_TYPE. The key is LAST_MODIFIED and the the type is SimpleType.LONG.

124.8.2.30   **public static final String LOCATION = "Location"**

The key LOCATION, used in LOCATION_ITEM.

124.8.2.31   **public static final Item LOCATION_ITEM**

The item containing the bundle location in BUNDLE_TYPE. The key is LOCATION and the the type is SimpleType.STRING.

124.8.2.32   **public static final String OBJECTNAME = "osgi.core:type=bundleState,version=1.7"**

The Object Name prefix for this mbean. The full object name also contains the framework name and uuid as properties.

124.8.2.33   **public static final String PERSISTENTLY_STARTED = "PersistentlyStarted"**

The key PERSISTENTLY_STARTED, used in PERSISTENTLY_STARTED_ITEM.

124.8.2.34   **public static final Item PERSISTENTLY_STARTED_ITEM**

The item containing the indication of persistently started in BUNDLE_TYPE. The key is PERSISTENTLY_STARTED and the the type is SimpleType.BOOLEAN.

124.8.2.35   **public static final String REGISTERED_SERVICES = "RegisteredServices"**

The key REGISTERED_SERVICES, used in REGISTERED_SERVICES_ITEM.

124.8.2.36   **public static final Item REGISTERED_SERVICES_ITEM**

The item containing the registered services of the bundle in BUNDLE_TYPE. The key is REGISTERED_SERVICES and the the type is JmxConstants.LONG_ARRAY_TYPE.

124.8.2.37   **public static final String REMOVAL_PENDING = "RemovalPending"**

The key REMOVAL_PENDING, used in REMOVAL_PENDING_ITEM.

124.8.2.38   **public static final Item REMOVAL_PENDING_ITEM**

The item containing the indication of removal pending in BUNDLE_TYPE. The key is
REMOVAL_PENDING and the type is SimpleType.BOOLEAN.

124.8.2.39   **public static final String REQUIRED = "Required"**

The key REQUIRED, used in REQUIRED_ITEM.

124.8.2.40   **public static final String REQUIRED_BUNDLES = "RequiredBundles"**

The key REQUIRED_BUNDLES, used in REQUIRED_BUNDLES_ITEM.

124.8.2.41   **public static final Item REQUIRED_BUNDLES_ITEM**

The item containing the required bundles in BUNDLE_TYPE. The key is REQUIRED_BUNDLES and
the type is JmxConstants.LONG_ARRAY_TYPE

124.8.2.42   **public static final Item REQUIRED_ITEM**

The item containing the required status in BUNDLE_TYPE. The key is REQUIRED and the the type
is SimpleType.BOOLEAN.

124.8.2.43   **public static final String REQUIRING_BUNDLES = "RequiringBundles"**

The key REQUIRING_BUNDLES, used in REQUIRING_BUNDLES_ITEM.

124.8.2.44   **public static final Item REQUIRING_BUNDLES_ITEM**

The item containing the bundles requiring this bundle in BUNDLE_TYPE. The key is
REQUIRING_BUNDLES and the type is JmxConstants.LONG_ARRAY_TYPE

124.8.2.45   **public static final String RESOLVED = "RESOLVED"**

Constant RESOLVED for the STATE

124.8.2.46   **public static final String SERVICES_IN_USE = "ServicesInUse"**

The key SERVICES_IN_USE, used in SERVICES_IN_USE_ITEM.

124.8.2.47   **public static final Item SERVICES_IN_USE_ITEM**

The item containing the services in use by this bundle in BUNDLE_TYPE. The key is
SERVICES_IN_USE and the the type is JmxConstants.LONG_ARRAY_TYPE.

124.8.2.48   **public static final String START_LEVEL = "StartLevel"**

The key START_LEVEL, used in START_LEVEL_ITEM.

124.8.2.49   **public static final Item START_LEVEL_ITEM**

The item containing the start level in BUNDLE_TYPE. The key is START_LEVEL and the the type is
SimpleType.INTEGER.

124.8.2.50   **public static final String STARTING = "STARTING"**

Constant STARTING for the STATE

124.8.2.51   **public static final String STATE = "State"**

The key STATE, used in STATE_ITEM.

124.8.2.52   **public static final Item STATE_ITEM**

The item containing the bundle state in BUNDLE_TYPE. The key is STATE and the the type is SimpleType.STRING. The returned values must be one of the following strings:

- INSTALLED
- RESOLVED
- STARTING
- ACTIVE
- STOPPING
- UNINSTALLED
- UNKNOWN

124.8.2.53   **public static final String STOPPING = "STOPPING"**

Constant STOPPING for the STATE

124.8.2.54   **public static final String SYMBOLIC_NAME = "SymbolicName"**

The key SYMBOLIC_NAME, used in SYMBOLIC_NAME_ITEM.

124.8.2.55   **public static final Item SYMBOLIC_NAME_ITEM**

The item containing the symbolic name in BUNDLE_TYPE. The key is SYMBOLIC_NAME and the the type is SimpleType.STRING.

124.8.2.56   **public static final String UNINSTALLED = "UNINSTALLED"**

Constant UNINSTALLED for the STATE

124.8.2.57   **public static final String UNKNOWN = "UNKNOWN"**

Constant UNKNOWN for the STATE

124.8.2.58   **public static final String VALUE = "Value"**

The key VALUE, used in VALUE_ITEM.

124.8.2.59   **public static final Item VALUE_ITEM**

The item describing the value of a bundle header entry. The key is VALUE and the type is SimpleType.STRING.

124.8.2.60   **public static final String VERSION = "Version"**

The key VERSION, used in VERSION_ITEM.

124.8.2.61   **public static final Item VERSION_ITEM**

The item containing the symbolic name in BUNDLE_TYPE. The key is SYMBOLIC_NAME and the the type is SimpleType.STRING.

124.8.2.62   **public CompositeData getBundle(long bundleIdentifier) throws IOException**

bundleIdentifier   the bundle identifier of the requested bundle

☐   Obtain the information regarding a single bundle. The result is defined by the BUNDLE_TYPE CompositeType.

Returns   A CompositeData object with the bundle information

Throws   IOException– if the operation fails

IllegalArgumentException– if the bundle indicated does not exist

124.8.2.63   **public long[] getBundleIds() throws IOException**

☐ List all bundle IDs in the framework.

Returns all the bundle ids in the framework.

Throws IOException– if the operation fails

124.8.2.64   **public String[] getExportedPackages(long bundleId) throws IOException**

bundleId

☐ Answer the list of exported packages for this bundle.

Returns the array of package names, combined with their version in the format ‹packageName;version›

Throws IOException– if the operation fails

IllegalArgumentException– if the bundle indicated does not exist

124.8.2.65   **public long[] getFragments(long bundleId) throws IOException**

bundleId

☐ Answer the list of the bundle ids of the fragments associated with this bundle

Returns the array of bundle identifiers

Throws IOException– if the operation fails

IllegalArgumentException– if the bundle indicated does not exist

124.8.2.66   **public String getHeader(long bundleId,String key) throws IOException**

bundleId the unique identifier of the bundle

key the key of the header to look up

☐ Retrieve a single header from the bundle headers.

Returns the value of associated header

Throws IOException– if the operation fails

IllegalArgumentException– if the bundle indicated does not exist

124.8.2.67   **public String getHeader(long bundleId,String key,String locale) throws IOException**

bundleId the unique identifier of the bundle

key the key of the header to look up

locale the locale name into which the header value is to be localized. The value of this parameter follows the same rules as the locale parameter in Bundle.getHeaders(String locale)

☐ Retrieve a single header from the bundle headers.

This method performs the same function as getHeaders(long bundleId) except the manifest header values are localized to the specified locale.

Returns the value of associated header

Throws IOException– if the operation fails

IllegalArgumentException– if the bundle indicated does not exist

124.8.2.68   **public TabularData getHeaders(long bundleId) throws IOException**

bundleId the unique identifier of the bundle

□ Answer the headers for the bundle uniquely identified by the bundle id. The Tabular Data is typed by the HEADERS_TYPE.

Returns the table of associated header key and values

Throws IOException– if the operation fails

IllegalArgumentException– if the bundle indicated does not exist

124.8.2.69 **public TabularData getHeaders(long bundleId,String locale) throws IOException**

bundleId the unique identifier of the bundle

locale the locale name into which the header values are to be localized. The value of this parameter follows the same rules as the locale parameter in Bundle.getHeaders(String locale)

□ Answer the headers for the bundle uniquely identified by the bundle id. The Tabular Data is typed by the HEADERS_TYPE.

This method performs the same function as getHeaders(long bundleId) except the manifest header values are localized to the specified locale.

Returns the table of associated header key and values

Throws IOException– if the operation fails

IllegalArgumentException– if the bundle indicated does not exist

124.8.2.70 **public long[] getHosts(long fragment) throws IOException**

fragment the bundle id of the fragment

□ Answer the list of bundle ids of the bundles which host a fragment

Returns the array of bundle identifiers

Throws IOException– if the operation fails

IllegalArgumentException– if the bundle indicated does not exist

124.8.2.71 **public String[] getImportedPackages(long bundleId) throws IOException**

bundleId the bundle identifier

□ Answer the array of the packages imported by this bundle

Returns the array of package names, combined with their version in the format ‹packageName;version›

Throws IOException– if the operation fails

IllegalArgumentException– if the bundle indicated does not exist

124.8.2.72 **public long getLastModified(long bundleId) throws IOException**

bundleId the unique identifier of a bundle

□ Answer the last modified time of a bundle

Returns the last modified time

Throws IOException– if the operation fails

IllegalArgumentException– if the bundle indicated does not exist

124.8.2.73 **public String getLocation(long bundleId) throws IOException**

bundleId the identifier of the bundle

□ Answer the location of the bundle.

Returns The location string of this bundle

Throws	IOException– if the operation fails
	IllegalArgumentException– if the bundle indicated does not exist

124.8.2.74   **public long[] getRegisteredServices(long bundleId) throws IOException**

bundleId	the bundle identifier
□	Answer the list of service identifiers representing the services this bundle exports
Returns	the list of service identifiers
Throws	IOException– if the operation fails
	IllegalArgumentException– if the bundle indicated does not exist

124.8.2.75   **public long[] getRequiredBundles(long bundleIdentifier) throws IOException**

bundleIdentifier	the bundle identifier to find the dependencies for
□	Answer an array of ids of bundles the given bundle depends on.
Returns	the bundle identifiers of the dependencies
Throws	IOException– if the operation fails
	IllegalArgumentException– if the bundle indicated does not exist

124.8.2.76   **public long[] getRequiringBundles(long bundleIdentifier) throws IOException**

bundleIdentifier	the bundle identifier
□	Answer the list of identifiers of the bundles which require this bundle
Returns	the list of bundle identifiers
Throws	IOException– if the operation fails
	IllegalArgumentException– if the bundle indicated does not exist

124.8.2.77   **public long[] getServicesInUse(long bundleIdentifier) throws IOException**

bundleIdentifier	the bundle identifier
□	Answer the list of service identifiers which refer to the the services this bundle is using
Returns	the list of service identifiers
Throws	IOException– if the operation fails
	IllegalArgumentException– if the bundle indicated does not exist

124.8.2.78   **public int getStartLevel(long bundleId) throws IOException**

bundleId	the identifier of the bundle
□	Answer the start level of the bundle
Returns	the start level
Throws	IOException– if the operation fails
	IllegalArgumentException– if the bundle indicated does not exist

124.8.2.79   **public String getState(long bundleId) throws IOException**

bundleId	the identifier of the bundle
□	Answer the symbolic name of the state of the bundle
Returns	the string name of the bundle state
Throws	IOException– if the operation fails

IllegalArgumentException– if the bundle indicated does not exist

124.8.2.80   **public String getSymbolicName(long bundleId) throws IOException**

bundleId   the identifier of the bundle

☐   Answer the symbolic name of the bundle

Returns   the symbolic name

Throws   IOException– if the operation fails

IllegalArgumentException– if the bundle indicated does not exist

124.8.2.81   **public String getVersion(long bundleId) throws IOException**

bundleId   the identifier of the bundle

☐   Answer the location of the bundle.

Returns   The location string of this bundle

Throws   IOException– if the operation fails

IllegalArgumentException– if the bundle indicated does not exist

124.8.2.82   **public boolean isActivationPolicyUsed(long bundleId) throws IOException**

bundleId   the identifier of the bundle

☐   Answer whether the specified bundle's autostart setting indicates that the activation policy declared in the bundle's manifest must be used.

Returns   true if the bundle's autostart setting indicates the activation policy declared in the manifest must be used. false if the bundle must be eagerly activated.

Throws   IOException– if the operation fails

IllegalArgumentException– if the bundle indicated does not exist

124.8.2.83   **public boolean isFragment(long bundleId) throws IOException**

bundleId   the identifier of the bundle

☐   Answer whether the bundle is a fragment or not

Returns   true if the bundle is a fragment

Throws   IOException– if the operation fails

IllegalArgumentException– if the bundle indicated does not exist

124.8.2.84   **public boolean isPersistentlyStarted(long bundleId) throws IOException**

bundleId   the identifier of the bundle

☐   Answer if the bundle is persistently started when its start level is reached

Returns   true if the bundle is persistently started

Throws   IOException– if the operation fails

IllegalArgumentException– if the bundle indicated does not exist

124.8.2.85   **public boolean isRemovalPending(long bundleId) throws IOException**

bundleId   the identifier of the bundle

☐   Answer true if the bundle is pending removal

Returns   true if the bundle is pending removal

Throws IOException– if the operation fails

IllegalArgumentException– if the bundle indicated does not exist

124.8.2.86   **public boolean isRequired(long bundleId) throws IOException**

bundleId the identifier of the bundle

☐ Answer true if the bundle is required by another bundle

Returns true if the bundle is required by another bundle

Throws IOException– if the operation fails

IllegalArgumentException– if the bundle indicated does not exist

124.8.2.87   **public TabularData listBundles() throws IOException**

☐ Answer the bundle state of the system in tabular form. Each row of the returned table represents a single bundle. The Tabular Data consists of Composite Data that is type by BUNDLES_TYPE.

Returns the tabular representation of the bundle state

Throws IOException– if the operation fails

124.8.2.88   **public TabularData listBundles(String ... items) throws IOException**

items The names of the items to include in the result.

☐ Answer the bundle state of the system in tabular form. Each row of the returned table represents a single bundle. The Tabular Data consists of Composite Data that is type by BUNDLES_TYPE. This method supports specifying the items that are included in the result. Note that the IDENTIFIER item is always returns as this the key in the TabularData structure.

Returns the tabular representation of the bundle state

Throws IOException– if the operation fails

124.8.3   public interface FrameworkMBean

The FrameworkMbean provides mechanisms to exert control over the framework. For many operations, it provides a batch mechanism to avoid excessive message passing when interacting remotely.

Concurrency Thread-safe

124.8.3.1   **public static final CompositeType BATCH_ACTION_RESULT_TYPE**

The Composite Type for a batch action result. refreshBundle(long) and refreshBundles(long[]). Notice that a batch action result returns uses an id for the BUNDLE_IN_ERROR while the BATCH_INSTALL_RESULT_TYPE uses a location. This Composite Type consists of the following items:

•   BUNDLE_IN_ERROR_ID_ITEM
•   COMPLETED_ITEM
•   ERROR_ITEM
•   REMAINING_ID_ITEM
•   SUCCESS_ITEM

124.8.3.2   **public static final CompositeType BATCH_INSTALL_RESULT_TYPE**

The Composite Type which represents the result of a batch install operation. It is used in installBundles(String[]) and installBundlesFromURL(String[], String[]). This Composite Type consists of the following items:

•   BUNDLE_IN_ERROR_LOCATION_ITEM

- COMPLETED_ITEM
- ERROR_ITEM
- REMAINING_LOCATION_ITEM
- SUCCESS_ITEM

124.8.3.3   **public static final CompositeType BATCH_RESOLVE_RESULT_TYPE**

The Composite Type which represents the result of a batch resolve operation. It is used in refreshBundlesAndWait(long[]) and resolve(long[]). This Composite Type consists of the following items:

- COMPLETED_ITEM
- SUCCESS_ITEM

124.8.3.4   **public static final String BUNDLE_IN_ERROR = "BundleInError"**

The key for BUNDLE_IN_ERROR. This key is used with two different items: BUNDLE_IN_ERROR_ID_ITEM and BUNDLE_IN_ERROR_LOCATION_ITEM that each have a different type for this key. It is used in BATCH_ACTION_RESULT_TYPE and BATCH_INSTALL_RESULT_TYPE.

124.8.3.5   **public static final Item BUNDLE_IN_ERROR_ID_ITEM**

The item containing the bundle which caused the error during the batch operation. This item describes the bundle in error as an id. The key is BUNDLE_IN_ERROR and the type is SimpleType.LONG. It is used in BATCH_ACTION_RESULT_TYPE.

See Also BUNDLE_IN_ERROR_LOCATION_ITEM for the item that has a location for the bundle in error.

124.8.3.6   **public static final Item BUNDLE_IN_ERROR_LOCATION_ITEM**

The item containing the bundle which caused the error during the batch operation. This item describes the bundle in error as a location. The key is BUNDLE_IN_ERROR and the type is SimpleType.STRING. It is used in BATCH_INSTALL_RESULT_TYPE.

See Also BUNDLE_IN_ERROR_ID_ITEM for the item that has the id for the bundle in error.

124.8.3.7   **public static final String COMPLETED = "Completed"**

The key COMPLETED, used in COMPLETED_ITEM.

124.8.3.8   **public static final Item COMPLETED_ITEM**

The item containing the list of bundles completing the batch operation. The key is COMPLETED and the type is JmxConstants.LONG_ARRAY_TYPE. It is used in BATCH_ACTION_RESULT_TYPE and BATCH_INSTALL_RESULT_TYPE.

124.8.3.9   **public static final String ERROR = "Error"**

The key ERROR, used in ERROR_ITEM.

124.8.3.10   **public static final Item ERROR_ITEM**

The item containing the error message of the batch operation. The key is ERROR and the type is SimpleType.STRING. It is used in BATCH_ACTION_RESULT_TYPE and BATCH_INSTALL_RESULT_TYPE.

124.8.3.11   **public static final String OBJECTNAME = "osgi.core:type=framework,version=1.7"**

The Object Name prefix for this mbean. The full object name also contains the framework name and uuid as properties.

124.8.3.12   **public static final String REMAINING = "Remaining"**

The key REMAINING, used in REMAINING_ID_ITEM and REMAINING_LOCATION_ITEM.

124.8.3.13   **public static final Item REMAINING_ID_ITEM**

The item containing the list of remaining bundles unprocessed by the failing batch operation. The key is REMAINING and the type is JmxConstants.LONG_ARRAY_TYPE. It is used in BATCH_ACTION_RESULT_TYPE and BATCH_INSTALL_RESULT_TYPE.

124.8.3.14   **public static final Item REMAINING_LOCATION_ITEM**

The item containing the list of remaining bundles unprocessed by the failing batch operation. The key is REMAINING and the type is JmxConstants.STRING_ARRAY_TYPE. It is used in BATCH_ACTION_RESULT_TYPE and BATCH_INSTALL_RESULT_TYPE.

124.8.3.15   **public static final String SUCCESS = "Success"**

The SUCCESS, used in SUCCESS_ITEM.

124.8.3.16   **public static final Item SUCCESS_ITEM**

The item that indicates if this operation was successful. The key is SUCCESS and the type is SimpleType.BOOLEAN. It is used in BATCH_ACTION_RESULT_TYPE and BATCH_INSTALL_RESULT_TYPE.

124.8.3.17   **public long[] getDependencyClosure(long[] bundles) throws IOException**

bundles The initial bundles IDs for which to generate the dependency closure.

☐ Returns the dependency closure for the specified bundles.

A graph of bundles is computed starting with the specified bundles. The graph is expanded by adding any bundle that is either wired to a package that is currently exported by a bundle in the graph or requires a bundle in the graph. The graph is fully constructed when there is no bundle outside the graph that is wired to a bundle in the graph. The graph may contain UNINSTALLED bundles that are removal pending.

Returns A bundle ID array containing a snapshot of the dependency closure of the specified bundles, or an empty array if there were no specified bundles.

Throws IOException– if the operation failed

IllegalArgumentException– if a bundle indicated does not exist

124.8.3.18   **public int getFrameworkStartLevel() throws IOException**

☐ Retrieve the framework start level

Returns the framework start level

Throws IOException– if the operation failed

124.8.3.19   **public int getInitialBundleStartLevel() throws IOException**

☐ Answer the initial start level assigned to a bundle when it is first started

Returns the start level

Throws IOException– if the operation failed

124.8.3.20   **public String getProperty(String key) throws IOException**

key The name of the requested property.

☐ Returns the value of the specified property. If the key is not found in the Framework properties, the system properties are then searched. The method returns null if the property is not found.

Returns The value of the requested property, or null if the property is undefined.

Throws IOException– if the operation failed

124.8.3.21   **public long[] getRemovalPendingBundles() throws IOException**

□ Returns the bundles IDs that have non-current, in use bundle wirings. This is typically the bundles which have been updated or uninstalled since the last call to refreshBundles(long[]).

Returns A bundle ID array containing a snapshot of the bundles which have non-current, in use bundle wirings, or an empty array if there are no such bundles.

Throws IOException– if the operation failed

124.8.3.22   **public long installBundle(String location) throws IOException**

location the location of the bundle to install

□ Install the bundle indicated by the bundleLocations

Returns the bundle id the installed bundle

Throws IOException– if the operation does not succeed

124.8.3.23   **public long installBundleFromURL(String location,String url) throws IOException**

location the location to assign to the bundle

url the URL which will supply the bytes for the bundle

□ Install the bundle indicated by the bundleLocations

Returns the bundle id the installed bundle

Throws IOException– if the operation does not succeed

124.8.3.24   **public CompositeData installBundles(String[] locations) throws IOException**

locations the array of locations of the bundles to install

□ Batch install the bundles indicated by the list of bundleLocationUrls

Returns the resulting state from executing the operation

Throws IOException– if the operation does not succeed

See Also BATCH_INSTALL_RESULT_TYPE for the precise specification of the CompositeData type representing the returned result.

124.8.3.25   **public CompositeData installBundlesFromURL(String[] locations,String[] urls) throws IOException**

locations the array of locations to assign to the installed bundles

urls the array of urls which supply the bundle bytes

□ Batch install the bundles indicated by the list of bundleLocationUrls

Returns the resulting state from executing the operation

Throws IOException– if the operation does not succeed

See Also for the precise specification of the CompositeData type representing the returned result.

124.8.3.26   **public void refreshBundle(long bundleIdentifier) throws IOException**

bundleIdentifier the bundle identifier

□ Force the update, replacement or removal of the packages identified by the specified bundle.

Throws IOException– if the operation failed

IllegalArgumentException– if the bundle indicated does not exist

124.8.3.27   **public boolean refreshBundleAndWait(long bundleIdentifier) throws IOException**

bundleIdentifier the bundle identifier

□ Force the update, replacement or removal of the packages identified by the specified bundle and wait until completed.

Returns whether the bundle was successfully resolved after being refreshed.

Throws IOException– if the operation failed

IllegalArgumentException– if the bundle indicated does not exist

124.8.3.28   **public void refreshBundles(long[] bundleIdentifiers) throws IOException**

bundleIdentifiers The identifiers of the bundles to refresh, or null for all bundles with packages pending removal.

□ Force the update, replacement or removal of the packages identified by the list of bundles.

Throws IOException– if the operation failed

IllegalArgumentException– if a bundle indicated does not exist

124.8.3.29   **public CompositeData refreshBundlesAndWait(long[] bundleIdentifiers) throws IOException**

bundleIdentifiers The identifiers of the bundles to refresh, or null for all bundles with packages pending removal.

□ Force the update, replacement or removal of the packages identified by the list of bundles and wait until completed.

Returns the result of the refresh operation

Throws IOException– if the operation failed

IllegalArgumentException– if a bundle indicated does not exist

See Also for the precise specification of the CompositeData type representing the returned result.

124.8.3.30   **public CompositeData resolve(long[] bundleIdentifiers) throws IOException**

bundleIdentifiers The identifiers of the bundles to resolve, or null to resolve all unresolved bundles.

□ Same as resolveBundles(long[]) but with a more detailed return type.

Returns the resulting state from executing the operation

Throws IOException– if the operation failed

IllegalArgumentException– if a bundle indicated does not exist

See Also for the precise specification of the CompositeData type representing the returned result.

124.8.3.31   **public boolean resolveBundle(long bundleIdentifier) throws IOException**

bundleIdentifier the bundle identifier

□ Resolve the bundle indicated by the unique symbolic name and version

Returns true if the bundle was resolved, false otherwise

Throws IOException– if the operation does not succeed

IllegalArgumentException– if the bundle indicated does not exist

124.8.3.32   **public boolean resolveBundles(long[] bundleIdentifiers) throws IOException**

bundleIdentifiers The identifiers of the bundles to resolve, or null to resolve all unresolved bundles.

□ Batch resolve the bundles indicated by the list of bundle identifiers

Returns true if the bundles were resolved, false otherwise

Throws IOException– if the operation does not succeed

IllegalArgumentException– if a bundle indicated does not exist

124.8.3.33   **public void restartFramework() throws IOException**

☐ Restart the framework by updating the system bundle

Throws IOException– if the operation failed

124.8.3.34   **public void setBundleStartLevel(long bundleIdentifier,int newlevel) throws IOException**

bundleIdentifier the bundle identifier

newlevel the new start level for the bundle

☐ Set the start level for the bundle identifier

Throws IOException– if the operation failed

124.8.3.35   **public CompositeData setBundleStartLevels(long[] bundleIdentifiers,int[] newlevels) throws IOException**

bundleIdentifiers the array of bundle identifiers

newlevels the array of new start level for the bundles

☐ Set the start levels for the list of bundles.

Returns the resulting state from executing the operation

Throws IOException– if the operation failed

See Also for the precise specification of the CompositeData type representing the returned result.

124.8.3.36   **public void setFrameworkStartLevel(int newlevel) throws IOException**

newlevel the new start level

☐ Set the start level for the framework

Throws IOException– if the operation failed

124.8.3.37   **public void setInitialBundleStartLevel(int newlevel) throws IOException**

newlevel the new start level

☐ Set the initial start level assigned to a bundle when it is first started

Throws IOException– if the operation failed

124.8.3.38   **public void shutdownFramework() throws IOException**

☐ Shutdown the framework by stopping the system bundle

Throws IOException– if the operation failed

124.8.3.39   **public void startBundle(long bundleIdentifier) throws IOException**

bundleIdentifier the bundle identifier

☐ Start the bundle indicated by the bundle identifier

Throws IOException– if the operation does not succeed

IllegalArgumentException– if the bundle indicated does not exist

124.8.3.40   **public CompositeData startBundles(long[] bundleIdentifiers) throws IOException**

bundleIdentifiers the array of bundle identifiers

☐ Batch start the bundles indicated by the list of bundle identifier

Returns the resulting state from executing the operation

Throws IOException– if the operation does not succeed

See Also for the precise specification of the CompositeData type representing the returned result.

124.8.3.41 **public void stopBundle(long bundleIdentifier) throws IOException**

bundleIdentifier the bundle identifier

  □ Stop the bundle indicated by the bundle identifier

Throws IOException– if the operation does not succeed

  IllegalArgumentException– if the bundle indicated does not exist

124.8.3.42 **public CompositeData stopBundles(long[] bundleIdentifiers) throws IOException**

bundleIdentifiers the array of bundle identifiers

  □ Batch stop the bundles indicated by the list of bundle identifier

Returns the resulting state from executing the operation

Throws IOException– if the operation does not succeed

See Also BATCH_ACTION_RESULT_TYPE for the precise specification of the CompositeData type representing the returned result.

124.8.3.43 **public void uninstallBundle(long bundleIdentifier) throws IOException**

bundleIdentifier the bundle identifier

  □ Uninstall the bundle indicated by the bundle identifier

Throws IOException– if the operation does not succeed

  IllegalArgumentException– if the bundle indicated does not exist

124.8.3.44 **public CompositeData uninstallBundles(long[] bundleIdentifiers) throws IOException**

bundleIdentifiers the array of bundle identifiers

  □ Batch uninstall the bundles indicated by the list of bundle identifiers

Returns the resulting state from executing the operation

Throws IOException– if the operation does not succeed

See Also BATCH_ACTION_RESULT_TYPE for the precise specification of the CompositeData type representing the returned result.

124.8.3.45 **public void updateBundle(long bundleIdentifier) throws IOException**

bundleIdentifier the bundle identifier

  □ Update the bundle indicated by the bundle identifier

Throws IOException– if the operation does not succeed

  IllegalArgumentException– if the bundle indicated does not exist

124.8.3.46 **public void updateBundleFromURL(long bundleIdentifier,String url) throws IOException**

bundleIdentifier the bundle identifier

url the URL to use to update the bundle

  □ Update the bundle identified by the bundle identifier

Throws IOException– if the operation does not succeed

  IllegalArgumentException– if the bundle indicated does not exist

124.8.3.47   **public CompositeData updateBundles(long[] bundleIdentifiers) throws IOException**

bundleIdentifiers the array of bundle identifiers

□ Batch update the bundles indicated by the list of bundle identifier.

Returns the resulting state from executing the operation

Throws IOException– if the operation does not succeed

See Also BATCH_ACTION_RESULT_TYPE for the precise specification of the CompositeData type representing the returned result.

124.8.3.48   **public CompositeData updateBundlesFromURL(long[] bundleIdentifiers,String[] urls) throws IOException**

bundleIdentifiers the array of bundle identifiers

urls the array of URLs to use to update the bundles

□ Update the bundle uniquely identified by the bundle symbolic name and version using the contents of the supplied urls.

Returns the resulting state from executing the operation

Throws IOException– if the operation does not succeed

IllegalArgumentException– if the bundle indicated does not exist

See Also BATCH_ACTION_RESULT_TYPE for the precise specification of the CompositeData type representing the returned result.

124.8.3.49   **public void updateFramework() throws IOException**

□ Update the framework by updating the system bundle.

Throws IOException– if the operation failed

124.8.4   public interface PackageStateMBean

This MBean provides information about the package state of the framework.

Concurrency Thread-safe

124.8.4.1   **public static final String EXPORTING_BUNDLES = "ExportingBundles"**

The key EXPORTING_BUNDLE, used in EXPORTING_BUNDLES_ITEM.

124.8.4.2   **public static final Item EXPORTING_BUNDLES_ITEM**

The item containing the bundle identifier in PACKAGE_TYPE. The key is EXPORTING_BUNDLES and the type is JmxConstants.LONG_ARRAY_TYPE.

124.8.4.3   **public static final String IMPORTING_BUNDLES = "ImportingBundles"**

The key IMPORTING_BUNDLES, used in IMPORTING_BUNDLES_ITEM.

124.8.4.4   **public static final Item IMPORTING_BUNDLES_ITEM**

The item containing the bundle identifier in PACKAGE_TYPE. The key is IMPORTING_BUNDLES and the type is JmxConstants.LONG_ARRAY_TYPE.

124.8.4.5   **public static final String NAME = "Name"**

The key NAME, used in NAME_ITEM.

124.8.4.6   **public static final Item NAME_ITEM**

The item containing the name of the package in PACKAGE_TYPE. The key is NAME and the type is SimpleType.LONG.

124.8.4.7   **public static final String OBJECTNAME = "osgi.core:type=packageState,version=1.5"**

The fully qualified object name of this MBean.

124.8.4.8   **public static final CompositeType PACKAGE_TYPE**

The Composite Type for a CompositeData representing a package. This type consists of:

- EXPORTING_BUNDLES_ITEM
- IMPORTING_BUNDLES_ITEM
- NAME_ITEM
- REMOVAL_PENDING_ITEM
- VERSION_ITEM

The key is defined as NAME and EXPORTING_BUNDLES

124.8.4.9   **public static final TabularType PACKAGES_TYPE**

The Tabular Type used in listPackages(). They key is NAME, VERSION, and EXPORTING_BUNDLES.

124.8.4.10   **public static final String REMOVAL_PENDING = "RemovalPending"**

The name of the item containing the pending removal status of the package in the CompositeData. Used

124.8.4.11   **public static final Item REMOVAL_PENDING_ITEM**

The item representing the removal pending status of a package. The key is REMOVAL_PENDING and the type is SimpleType.BOOLEAN.

124.8.4.12   **public static final String VERSION = "Version"**

The name of the item containing the package version in the CompositeData. Used in VERSION_ITEM.

124.8.4.13   **public static final Item VERSION_ITEM**

The item containing the version of the package in PACKAGE_TYPE. The key is VERSION and the type is SimpleType.STRING.

124.8.4.14   **public long[] getExportingBundles(String packageName,String version) throws IOException**

packageName - the package name

version - the version of the package

☐ Answer the identifier of the bundle exporting the package

Returns the bundle identifiers exporting such a package

Throws IOException– if the operation fails

IllegalArgumentException– if the package indicated does not exist

124.8.4.15   **public long[] getImportingBundles(String packageName,String version,long exportingBundle) throws IOException**

packageName The package name

version The version of the package

exportingBundle The exporting bundle for the given package

☐ Answer the list of identifiers of the bundles importing the package

Returns the list of bundle identifiers

Throws	IOException– if the operation fails
	IllegalArgumentException– if the package indicated does not exist

124.8.4.16   **public boolean isRemovalPending(String packageName,String version,long exportingBundle) throws IOException**

packageName	The package name
version	The version of the package
exportingBundle	The bundle exporting the package
□	Answer if this package is exported by a bundle which has been updated or uninstalled
Returns	true if this package is being exported by a bundle that has been updated or uninstalled.
Throws	IOException– if the operation fails
	IllegalArgumentException– if the package indicated does not exist

124.8.4.17   **public TabularData listPackages() throws IOException**

□	Answer the package state of the system in tabular form The Tabular Data is typed by PACKAGES_TYPE, which has PACKAGE_TYPE as its Composite Type.
Returns	the tabular representation of the package state
Throws	IOException– When fails

124.8.5   public interface ServiceStateMBean

This MBean represents the Service state of the framework. This MBean also emits events that clients can use to get notified of the changes in the service state of the framework.

Concurrency   Thread-safe

124.8.5.1   **public static final String BUNDLE_IDENTIFIER = "BundleIdentifier"**

The key BUNDLE_IDENTIFIER, used in BUNDLE_IDENTIFIER_ITEM.

124.8.5.2   **public static final Item BUNDLE_IDENTIFIER_ITEM**

The item containing the bundle identifier in SERVICE_TYPE. The key is BUNDLE_IDENTIFIER and the type is SimpleType.LONG .

124.8.5.3   **public static final String BUNDLE_LOCATION = "BundleLocation"**

The key BUNDLE_LOCATION, used in SERVICE_EVENT_TYPE.

124.8.5.4   **public static final Item BUNDLE_LOCATION_ITEM**

The item containing the bundle location in EVENT_ITEM. The key is BUNDLE_LOCATION and the the type is SimpleType.STRING .

124.8.5.5   **public static final String BUNDLE_SYMBOLIC_NAME = "BundleSymbolicName"**

The key BUNDLE_SYMBOLIC_NAME, used in SERVICE_EVENT_TYPE.

124.8.5.6   **public static final Item BUNDLE_SYMBOLIC_NAME_ITEM**

The item containing the symbolic name in EVENT. The key is BUNDLE_SYMBOLIC_NAME and the the type is SimpleType.STRING.

124.8.5.7   **public static final String EVENT = "ServiceEvent"**

The key EVENT, used in EVENT_ITEM.

124.8.5.8   **public static final Item EVENT_ITEM**

The item containing the event type. The key is EVENT and the type is SimpleType.INTEGER

124.8.5.9   **public static final String IDENTIFIER = "Identifier"**

The key IDENTIFIER, used IDENTIFIER_ITEM.

124.8.5.10   **public static final Item IDENTIFIER_ITEM**

The item containing the service identifier in SERVICE_TYPE. The key is IDENTIFIER and the type is SimpleType.LONG.

124.8.5.11   **public static final String OBJECT_CLASS = "objectClass"**

The key OBJECT_CLASS, used OBJECT_CLASS_ITEM.

124.8.5.12   **public static final Item OBJECT_CLASS_ITEM**

The item containing the interfaces of the service in SERVICE_TYPE. The key is OBJECT_CLASS and the type is JmxConstants.STRING_ARRAY_TYPE.

124.8.5.13   **public static final String OBJECTNAME = "osgi.core:type=serviceState,version=1.7"**

The Object Name prefix for this mbean. The full object name also contains the framework name and uuid as properties.

124.8.5.14   **public static final String PROPERTIES = "Properties"**

The key PROPERTIES, used in PROPERTIES_ITEM.

124.8.5.15   **public static final Item PROPERTIES_ITEM**

The item containing service properties in SERVICE_TYPE. The key is PROPERTIES and the type is JmxConstants.PROPERTIES_TYPE.

124.8.5.16   **public static final CompositeType SERVICE_EVENT_TYPE**

The Composite Type that represents a service event. This composite consists of:

- IDENTIFIER
- OBJECT_CLASS
- BUNDLE_LOCATION
- BUNDLE_SYMBOLIC_NAME
- EVENT

124.8.5.17   **public static final CompositeType SERVICE_TYPE**

The Composite Type for a CompositeData representing a service. This type consists of:

- BUNDLE_IDENTIFIER
- IDENTIFIER
- OBJECT_CLASS
- PROPERTIES
- USING_BUNDLES

124.8.5.18   **public static final TabularType SERVICES_TYPE**

The Tabular Type for a Service table. The rows consists of SERVICE_TYPE Composite Data and the index is IDENTIFIER .

124.8.5.19   **public static final String USING_BUNDLES = "UsingBundles"**

The key USING_BUNDLES, used in USING_BUNDLES_ITEM.

124.8.5.20   **public static final Item USING_BUNDLES_ITEM**

The item containing the bundles using the service in SERVICE_TYPE. The key is USING_BUNDLES and the type is JmxConstants.LONG_ARRAY_TYPE.

124.8.5.21   **public long getBundleIdentifier(long serviceId) throws IOException**

serviceId   the identifier of the service

☐ Answer the bundle identifier of the bundle which registered the service

Returns   the identifier for the bundle

Throws   IOException– if the operation fails

IllegalArgumentException– if the service indicated does not exist

124.8.5.22   **public String[] getObjectClass(long serviceId) throws IOException**

serviceId   the identifier of the service

☐ Answer the list of interfaces that this service implements

Returns   the list of interfaces

Throws   IOException– if the operation fails

IllegalArgumentException– if the service indicated does not exist

124.8.5.23   **public TabularData getProperties(long serviceId) throws IOException**

serviceId   the identifier of the service

☐ Answer the map of properties associated with this service.

Returns   the table of properties. These include the standard mandatory service.id and objectClass properties as defined in the org.osgi.framework.Constants interface

Throws   IOException– if the operation fails

IllegalArgumentException– if the service indicated does not exist

See Also   for the details of the TabularType

124.8.5.24   **public CompositeData getProperty(long serviceId,String key) throws IOException**

serviceId   the identifier of the service

key   the property key

☐ Return a single property from the specified service.

Returns   a CompositeData object holding the value and data type of the property.

Throws   IOException– if the operation fails

See Also   for the details of the CompositeType.

124.8.5.25   **public CompositeData getService(long serviceId) throws IOException**

serviceId   the ID of the service to look up

☐ Obtain information about a given service. The result is defined by the CompositeType SERVICE_TYPE.

Returns   A CompositeData object with the service information

Throws   IOException– if the operation fails

IllegalArgumentException– if the service indicated does not exist

124.8.5.26 **public long[] getServiceIds() throws IOException**

□ List all service IDs in the framework.

Returns all the service ids in the framework.

Throws IOException– if the operation fails

124.8.5.27 **public long[] getUsingBundles(long serviceId) throws IOException**

serviceId the identifier of the service

□ Answer the list of identifiers of the bundles that use the service

Returns the list of bundle identifiers

Throws IOException– if the operation fails

IllegalArgumentException– if the service indicated does not exist

124.8.5.28 **public TabularData listServices() throws IOException**

□ Answer the service state of the system in tabular form.

Returns the tabular representation of the service state

Throws IOException– If the operation fails

IllegalArgumentException– if the service indicated does not exist

See Also for the details of the TabularType

124.8.5.29 **public TabularData listServices(String clazz,String filter) throws IOException**

clazz The class name with which the services were registered or null for any class name.

filter A filter expression to match the services or null for no additional filter.

□ Answer the service state of the system in tabular form. This method allows the specification of a class name and a filter to select services to be provided.

Returns the tabular representation of the service state

Throws IOException– If the operation fails

IllegalArgumentException– if the service indicated does not exist

See Also for the details of the TabularType

124.8.5.30 **public TabularData listServices(String clazz,String filter,String ... serviceTypeItems) throws IOException**

clazz The class name with which the services were registered or null for any class name.

filter A filter expression to match the services or null for no additional filter.

serviceTypeItems The names of the SERVICE_TYPE items to include in the result. For example "objectClass" or "Properties". Note that the result always returns the "Identifier" item since this serves as the key in the resulting table.

□ Answer the service state of the system in tabular form. Apart from class name and filter, this method allows the specification of a subset of the SERVICE_TYPE items to be included in the result. Selecting only the relevant Service Type items may save bandwidth and improve performance over a remote connection.

Returns the tabular representation of the service state

Throws IOException– If the operation fails

IllegalArgumentException– if the service indicated does not exist

See Also　for the details of the TabularType

124.9　org.osgi.jmx.service.cm

OSGi JMX Configuration Admin Package Version 1.3.

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

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

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

124.9.1　Summary

- ConfigurationAdminMBean - This MBean provides the management interface to the OSGi Configuration Administration Service.

124.9.2　public interface ConfigurationAdminMBean

This MBean provides the management interface to the OSGi Configuration Administration Service.

Concurrency　Thread-safe

124.9.2.1　public static final String OBJECTNAME = "osgi.compendium:service=cm,version=1.3"

The object name for this mbean.

124.9.2.2　public String createFactoryConfiguration(String factoryPid) throws IOException

factoryPid　the persistent id of the factory

□　Create a new configuration instance for the supplied persistent id of the factory, answering the PID of the created configuration

Returns　the PID of the created configuration

Throws　IOException– if the operation failed

124.9.2.3　public String createFactoryConfigurationForLocation(String factoryPid,String location) throws IOException

factoryPid　the persistent id of the factory

location　the bundle location

□　Create a factory configuration for the supplied persistent id of the factory and the bundle location bound to bind the created configuration to, answering the PID of the created configuration

Returns　the pid of the created configuation

Throws　IOException– if the operation failed

124.9.2.4　public void delete(String pid) throws IOException

pid　the persistent identifier of the configuration

□　Delete the configuration

Throws　IOException– if the operation fails

124.9.2.5   **public void deleteConfigurations(String filter) throws IOException**

filter   the string representation of the org.osgi.framework.Filter

☐   Delete the configurations matching the filter specification.

Throws   IOException– if the operation failed

IllegalArgumentException– if the filter is invalid

124.9.2.6   **public void deleteForLocation(String pid,String location) throws IOException**

pid   the persistent identifier of the configuration

location   the bundle location

☐   Delete the configuration

Throws   IOException– if the operation fails

124.9.2.7   **public String getBundleLocation(String pid) throws IOException**

pid   the persistent identifier of the configuration

☐   Answer the bundle location the configuration is bound to

Returns   the bundle location

Throws   IOException– if the operation fails

124.9.2.8   **public String[][] getConfigurations(String filter) throws IOException**

filter   the string representation of the org.osgi.framework.Filter

☐   Answer the list of PID/Location pairs of the configurations managed by this service

Returns   the list of configuration PID/Location pairs

Throws   IOException– if the operation failed

IllegalArgumentException– if the filter is invalid

124.9.2.9   **public String getFactoryPid(String pid) throws IOException**

pid   the persistent identifier of the configuration

☐   Answer the factory PID if the configuration is a factory configuration, null otherwise.

Returns   the factory PID

Throws   IOException– if the operation fails

124.9.2.10   **public String getFactoryPidForLocation(String pid,String location) throws IOException**

pid   the persistent identifier of the configuration

location   the bundle location

☐   Answer the factory PID if the configuration is a factory configuration, null otherwise.

Returns   the factory PID

Throws   IOException– if the operation fails

124.9.2.11   **public TabularData getProperties(String pid) throws IOException**

pid   the persistent identifier of the configuration

☐   Answer the contents of the configuration.

Returns   the table of contents

Throws   IOException– if the operation fails

See Also JmxConstants.PROPERTIES_TYPE for the details of the TabularType

124.9.2.12 **public TabularData getPropertiesForLocation(String pid,String location) throws IOException**

pid the persistent identifier of the configuration

location the bundle location

□ Answer the contents of the configuration.

Returns the table of contents

Throws IOException– if the operation fails

See Also JmxConstants.PROPERTIES_TYPE for the details of the TabularType

124.9.2.13 **public void setBundleLocation(String pid,String location) throws IOException**

pid the persistent identifier of the configuration

location the bundle location

□ Set the bundle location the configuration is bound to

Throws IOException– if the operation fails

124.9.2.14 **public void update(String pid,TabularData properties) throws IOException**

pid the persistent identifier of the configuration

properties the table of properties

□ Update the configuration with the supplied properties For each property entry, the following row is supplied.

Throws IOException– if the operation fails

See Also JmxConstants.PROPERTIES_TYPE for the details of the TabularType

124.9.2.15 **public void updateForLocation(String pid,String location,TabularData properties) throws IOException**

pid the persistent identifier of the configuration

location the bundle location

properties the table of properties

□ Update the configuration with the supplied properties For each property entry, the following row is supplied.

Throws IOException– if the operation fails

See Also JmxConstants.PROPERTIES_TYPE for the details of the TabularType

124.10 org.osgi.jmx.service.permissionadmin

OSGi JMX Permission Admin Package Version 1.2.

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

Example import for consumers using the API in this package:

Import-Package: org.osgi.jmx.service.permissionadmin; version="[1.2,2.0)"

Example import for providers implementing the API in this package:

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

124.10.1   Summary

- `PermissionAdminMBean` - This MBean represents the OSGi Permission Manager Service

124.10.2   public interface PermissionAdminMBean

This MBean represents the OSGi Permission Manager Service

Concurrency   Thread-safe

124.10.2.1   public static final String OBJECTNAME = "osgi.core:service=permissionadmin,version=1.2"

Permission Admin MBean object name.

124.10.2.2   public String[] getPermissions(String location) throws IOException

location   location identifying the bundle

☐   Answer the list of encoded permissions of the bundle specified by the bundle location

Returns   the array of String encoded permissions

Throws   IOException– if the operation fails

124.10.2.3   public String[] listDefaultPermissions() throws IOException

☐   Answer the list of encoded permissions representing the default permissions assigned to bundle locations that have no assigned permissions

Returns   the array of String encoded permissions

Throws   IOException– if the operation fails

124.10.2.4   public String[] listLocations() throws IOException

☐   Answer the bundle locations that have permissions assigned to them

Returns   the bundle locations

Throws   IOException– if the operation fails

124.10.2.5   public void setDefaultPermissions(String[] encodedPermissions) throws IOException

encodedPermis-   the string encoded permissions
sions

☐   Set the default permissions assigned to bundle locations that have no assigned permissions

Throws   IOException– if the operation fails

124.10.2.6   public void setPermissions(String location,String[] encodedPermissions) throws IOException

location   the location of the bundle

encodedPermis-   the string encoded permissions to set
sions

☐   Set the permissions on the bundle specified by the bundle location

Throws   IOException– if the operation fails

124.11   org.osgi.jmx.service.provisioning

OSGi JMX Initial Provisioning Package Version 1.2.

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

Example import for consumers using the API in this package:

Import-Package: org.osgi.jmx.service.provisioning; version="[1.2,2.0)"

Example import for providers implementing the API in this package:

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

124.11.1 Summary

- ProvisioningServiceMBean - This MBean represents the management interface to the OSGi Initial Provisioning Service

124.11.2 public interface ProvisioningServiceMBean

This MBean represents the management interface to the OSGi Initial Provisioning Service

Concurrency Thread-safe

124.11.2.1 public static final String OBJECTNAME = "osgi.compendium:service=provisioning,version=1.2"

Provisioning MBean object name.

124.11.2.2 public void addInformation(TabularData info) throws IOException

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.

Throws IOException– if the operation fails

See Also JmxConstants.PROPERTIES_TYPE for details of the Tabular Data

124.11.2.3 public void addInformationFromZip(String zipURL) throws IOException

zipURL the String form of the URL that will be resolved into a ZipInputStream which 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 contents of the provided zipURL 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 of the URL. No additions will be made to the Provisioning Information dictionary and no bundles must be started or installed.

124.11.2.4 public TabularData listInformation() throws IOException

□ Returns a table representing the Provisioning Information Dictionary.

Returns The table representing the manager dictionary.

Throws IOException– if the operation fails

See Also JmxConstants.PROPERTIES_TYPE for details of the Tabular Data

124.11.2.5 **public void setInformation(TabularData info) throws IOException**

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 entries of the supplied table. This method causes the PROVISIONING_UPDATE_COUNT to be incremented.

Throws IOException– if the operation fails

See Also JmxConstants.PROPERTIES_TYPE for details of the Tabular Data

124.12 org.osgi.jmx.service.useradmin

OSGi JMX User Admin Package Version 1.1.

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

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

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

124.12.1 Summary

• UserAdminMBean - This MBean provides the management interface to the OSGi User Manager Service

124.12.2 public interface UserAdminMBean

This MBean provides the management interface to the OSGi User Manager Service

Concurrency Thread-safe

124.12.2.1 **public static final CompositeType AUTORIZATION_TYPE**

The Composite Type for an Authorization object. It consists of the NAME_ITEM and ROLES_ITEM items.

124.12.2.2 **public static final String CREDENTIALS = "Credentials"**

The CREDENTIALS key, used in CREDENTIALS_ITEM.

124.12.2.3 **public static final Item CREDENTIALS_ITEM**

The item containing the credentials of a user. The key is CREDENTIALS and the type is JmxConstants.PROPERTIES_TYPE .

124.12.2.4 **public static final CompositeType GROUP_TYPE**

The Composite Type for a Group. It extends USER_TYPE and adds MEMBERS_ITEM, and REQUIRED_MEMBERS_ITEM. This type extends the USER_TYPE. It adds:

• MEMBERS
• REQUIRED_MEMBERS

If there are no members or required members an empty array is returned in the respective items.

124.12.2.5 **public static final String MEMBERS = "Members"**

The MEMBERS key, used in MEMBERS_ITEM.

124.12.2.6 **public static final Item MEMBERS_ITEM**

The item containing the members of a group. The key is MEMBERS and the type is JmxConstants.STRING_ARRAY_TYPE. It is used in GROUP_TYPE.

124.12.2.7 **public static final String NAME = "Name"**

The key NAME, used in NAME_ITEM.

124.12.2.8 **public static final Item NAME_ITEM**

The item for the user name for an authorization object. The key is NAME and the type is SimpleType.STRING.

124.12.2.9 **public static final String OBJECTNAME = "osgi.compendium:service=useradmin,version=1.1"**

User Admin MBean object name.

124.12.2.10 **public static final String PROPERTIES = "Properties"**

The PROPERTIES key, used in PROPERTIES_ITEM.

124.12.2.11 **public static final Item PROPERTIES_ITEM**

The item containing the properties of a Role. The key is PROPERTIES and the type is JmxConstants.PROPERTIES_TYPE.

124.12.2.12 **public static final String REQUIRED_MEMBERS = "RequiredMembers"**

The REQUIRED_MEMBERS key, used in REQUIRED_MEMBERS_ITEM.

124.12.2.13 **public static final Item REQUIRED_MEMBERS_ITEM**

The item containing the required members of a group. The key is REQUIRED_MEMBERS and the type is JmxConstants.STRING_ARRAY_TYPE. It is used in GROUP_TYPE .

124.12.2.14 **public static final CompositeType ROLE_TYPE**

The Composite Type for a Role. It contains the following items:

- NAME
- TYPE
- PROPERTIES

124.12.2.15 **public static final String ROLES = "Roles"**

The key ROLES, used in ROLES_ITEM.

124.12.2.16 **public static final Item ROLES_ITEM**

The item containing the roles for this authorization object. The key is ROLES. and the type is JmxConstants.STRING_ARRAY_TYPE.

124.12.2.17 **public static final String TYPE = "Type"**

The Role TYPE key, used in TYPE_ITEM.

124.12.2.18 **public static final Item TYPE_ITEM**

The item containing the type of the roles encapsulated by this authorization object. The key is TYPE and the type is SimpleType.INTEGER.

124.12.2.19   **public static final CompositeType USER_TYPE**

A Composite Type for a User. A User contains its Role description and adds the credentials. It extends ROLE_TYPE and adds CREDENTIALS_ITEM. This type extends the ROLE_TYPE. It adds:

• CREDENTIALS

124.12.2.20   **public void addCredential(String key,byte[] value,String username) throws IOException**

key   The key of the credential to add

value   The value of the credential to add

username   The name of the user that gets the credential.

☐ Add credentials to a user, associated with the supplied key

Throws   IOException– if the operation fails

IllegalArgumentException– if the username is not a User

124.12.2.21   **public void addCredentialString(String key,String value,String username) throws IOException**

key   The key of the credential to add

value   The value of the credential to add

username   The name of the user that gets the credential.

☐ Add credentials to a user, associated with the supplied key

Throws   IOException– if the operation fails

IllegalArgumentException– if the username is not a User

124.12.2.22   **public boolean addMember(String groupname,String rolename) throws IOException**

groupname   The group name that receives the rolename as member.

rolename   The rolename (User or Group) that must be added.

☐ Add a member to the group.

Returns   true if the role was added to the group

Throws   IOException– if the operation fails

IllegalArgumentException– if an invalid group name or role name is specified

124.12.2.23   **public void addProperty(String key,byte[] value,String rolename) throws IOException**

key   The added property key

value   The added byte[] property value

rolename   The role name that receives the property

☐ Add or update a property on a role.

Throws   IOException– if the operation fails

IllegalArgumentException– if an invalid role name is specified

124.12.2.24   **public void addPropertyString(String key,String value,String rolename) throws IOException**

key   The key of the property to add

value   The value of the property to add (String)

rolename   The role name

☐ Add or update a property on a role

Throws IOException– if the operation fails

IllegalArgumentException– if an invalid role name is specified

124.12.2.25 **public boolean addRequiredMember(String groupname,String rolename) throws IOException**

groupname The group name that is addded

rolename The role that

☐ Add a required member to the group

Returns true if the role was added to the group

Throws IOException– if the operation fails

IllegalArgumentException– if an invalid group name or role name is specified

124.12.2.26 **public void createGroup(String name) throws IOException**

name Name of the group to create

☐ Create a Group

Throws IOException– if the operation fails

124.12.2.27 **public void createRole(String name) throws IOException**

name Ignored.

☐ This method was specified in error and must not be used.

Throws IOException– This method will throw an exception if called.

Deprecated This method was specified in error. It does not function and must not be used. Use either createUser(String) or createGroup(String).

124.12.2.28 **public void createUser(String name) throws IOException**

name Name of the user to create

☐ Create a User

Throws IOException– if the operation fails

124.12.2.29 **public CompositeData getAuthorization(String user) throws IOException**

user The user name

☐ Answer the authorization for the user name. The Composite Data is typed by AUTORIZATION_TYPE.

Returns the Authorization typed by AUTORIZATION_TYPE.

Throws IOException– if the operation fails

IllegalArgumentException– if the user name is not a User

124.12.2.30 **public TabularData getCredentials(String username) throws IOException**

username The user name

☐ Answer the credentials associated with a user. The returned Tabular Data is typed by JmxConstants.PROPERTIES_TYPE.

Returns the credentials associated with the user, see JmxConstants.PROPERTIES_TYPE

Throws IOException– if the operation fails

IllegalArgumentException– if the user name is not a User

124.12.2.31   **public CompositeData getGroup(String groupname) throws IOException**

groupname The group name

☐ Answer the Group associated with the group name. The returned Composite Data is typed by GROUP_TYPE

Returns the Group, see GROUP_TYPE

Throws IOException– if the operation fails

IllegalArgumentException– if the group name is not a Group

124.12.2.32   **public String[] getGroups(String filter) throws IOException**

filter The filter to apply

☐ Answer the list of group names

Returns The list of group names

Throws IOException– if the operation fails

124.12.2.33   **public String[] getImpliedRoles(String username) throws IOException**

username The name of the user that has the implied roles

☐ Answer the list of implied roles for a user

Returns The list of role names

Throws IOException– if the operation fails

IllegalArgumentException– if the username is not a User

124.12.2.34   **public String[] getMembers(String groupname) throws IOException**

groupname The name of the group to get the members from

☐ Answer the the user names which are members of the group

Returns The list of user names

Throws IOException– if the operation fails

IllegalArgumentException– if the groupname is not a Group

124.12.2.35   **public TabularData getProperties(String rolename) throws IOException**

rolename The name of the role to get properties from

☐ Answer the properties associated with a role. The returned Tabular Data is typed by JmxConstants.PROPERTIES_TYPE.

Returns the properties associated with the role, see JmxConstants.PROPERTIES_TYPE

Throws IOException– if the operation fails

IllegalArgumentException– if the rolename is not a role

124.12.2.36   **public String[] getRequiredMembers(String groupname) throws IOException**

groupname The name of the group to get the required members from

☐ Answer the list of user names which are required members of this group

Returns The list of user names

Throws IOException– if the operation fails

IllegalArgumentException– if the group name is not a Group

124.12.2.37   **public CompositeData getRole(String name) throws IOException**

name The name of the role to get the data from

    ☐ Answer the role associated with a name. The returned Composite Data is typed by ROLE_TYPE.

Returns the Role, see ROLE_TYPE

Throws IOException– if the operation fails

        IllegalArgumentException– if the name is not a role

124.12.2.38   **public String[] getRoles(String filter) throws IOException**

filter The string representation of the org.osgi.framework.Filter that is used to filter the roles by applying to the properties, if null all roles are returned.

    ☐ Answer the list of role names which match the supplied filter

Returns The list the role names

Throws IOException– if the operation fails

124.12.2.39   **public CompositeData getUser(String username) throws IOException**

username The name of the requested user

    ☐ Answer the User associated with the user name. The returned Composite Data is typed by USER_TYPE.

Returns The User, see USER_TYPE

Throws IOException– if the operation fails

        IllegalArgumentException– if the username is not a User

124.12.2.40   **public String[] getUsers(String filter) throws IOException**

filter The filter to apply

    ☐ Answer the list of user names in the User Admin database

Returns The list of user names

Throws IOException– if the operation fails

124.12.2.41   **public String getUserWithProperty(String key,String value) throws IOException**

key The key to compare

value The value to compare

    ☐ Answer the user name with the given property key-value pair from the User Admin service database.

Returns The User

Throws IOException– if the operation fails

124.12.2.42   **public String[] listGroups() throws IOException**

    ☐ Answer the list of group names

Returns The list of group names

Throws IOException– if the operation fails

124.12.2.43   **public String[] listRoles() throws IOException**

    ☐ Answer the list of role names in the User Admin database

Returns The list of role names

Throws IOException– if the operation fails

124.12.2.44   **public String[] listUsers() throws IOException**

  □ Answer the list of user names in the User Admin database

Returns The list of user names

Throws IOException– if the operation fails

124.12.2.45   **public void removeCredential(String key,String username) throws IOException**

key The key of the credential to remove

username The name of the user for which the credential must be removed

  □ Remove the credential associated with the given user

Throws IOException– if the operation fails

          IllegalArgumentException– if the username is not a User

124.12.2.46   **public boolean removeGroup(String name) throws IOException**

name

  □ Remove the Group associated with the name

Returns true if the remove succeeded

Throws IOException– if the operation fails

          IllegalArgumentException– if the name is not a Group

124.12.2.47   **public boolean removeMember(String groupname,String rolename) throws IOException**

groupname The group name

rolename

  □ Remove a role from the group

Returns true if the role was removed from the group

Throws IOException– if the operation fails

          IllegalArgumentException– if the groupname is not a Group

124.12.2.48   **public void removeProperty(String key,String rolename) throws IOException**

key

rolename

  □ Remove a property from a role

Throws IOException– if the operation fails

          IllegalArgumentException– if the rolename is not a role

124.12.2.49   **public boolean removeRole(String name) throws IOException**

name

  □ Remove the Role associated with the name

Returns true if the remove succeeded

Throws IOException– if the operation fails

          IllegalArgumentException– if the name is not a role

| 124.12.2.50 | **public boolean removeUser(String name) throws IOException** |

name

☐ Remove the User associated with the name

Returns true if the remove succeeded

Throws IOException– if the operation fails

IllegalArgumentException– if the name is not a User

124.13 org.osgi.jmx.framework.wiring

OSGi JMX Framework Wiring Package Version 1.1.

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

Example import for consumers using the API in this package:

Import-Package: org.osgi.jmx.framework.wiring; version="[1.1,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.jmx.framework.wiring; version="[1.1,1.2)"

124.13.1 Summary

- BundleWiringStateMBean - This MBean represents the bundle wiring state.

124.13.2 public interface BundleWiringStateMBean

This MBean represents the bundle wiring state.

It can be used to retrieve the declared capabilities, declared requirements, and wiring for the current and past revisions of bundles.

Concurrency Thread-safe

124.13.2.1 public static final String ATTRIBUTES = "Attributes"

The key of ATTRIBUTES_ITEM.

124.13.2.2 public static final Item ATTRIBUTES_ITEM

The item containing the attributes of a capability or requirement. Used in BUNDLE_REQUIREMENT_TYPE and BUNDLE_CAPABILITY_TYPE. The key is ATTRIBUTES and the type is ATTRIBUTES_TYPE.

124.13.2.3 public static final TabularType ATTRIBUTES_TYPE

The Tabular Type that holds the attributes for a capability or requirements. The row type is JmxConstants.PROPERTY_TYPE and the index is JmxConstants.KEY.

124.13.2.4 public static final String BUNDLE_CAPABILITY = "BundleCapability"

The key of BUNDLE_CAPABILITY_ITEM.

124.13.2.5 public static final Item BUNDLE_CAPABILITY_ITEM

The item containing a capability for a bundle in BUNDLE_WIRE_TYPE. The key is BUNDLE_CAPABILITY and the type is BUNDLE_CAPABILITY_TYPE.

124.13.2.6   **public static final CompositeType BUNDLE_CAPABILITY_TYPE**

The Composite Type that represents the capability of a bundle. The composite consists of:

- NAMESPACE
- ATTRIBUTES
- DIRECTIVES

124.13.2.7   **public static final String BUNDLE_ID = "BundleId"**

The key of BUNDLE_ID_ITEM.

124.13.2.8   **public static final Item BUNDLE_ID_ITEM**

The item containing a bundle ID. They key is BUNDLE_ID and the type is a long.

124.13.2.9   **public static final String BUNDLE_REQUIREMENT = "BundleRequirement"**

The key of BUNDLE_REQUIREMENT_ITEM.

124.13.2.10   **public static final Item BUNDLE_REQUIREMENT_ITEM**

The item containing a requirement for a bundle in BUNDLE_WIRE_TYPE. The key is BUNDLE_REQUIREMENT and the type is BUNDLE_REQUIREMENT_TYPE.

124.13.2.11   **public static final CompositeType BUNDLE_REQUIREMENT_TYPE**

The Composite Type that represents the requirement of a bundle. The composite consists of:

- NAMESPACE
- ATTRIBUTES
- DIRECTIVES

124.13.2.12   **public static final String BUNDLE_REVISION_ID = "BundleRevisionId"**

The key of BUNDLE_REVISION_ID_ITEM.

124.13.2.13   **public static final Item BUNDLE_REVISION_ID_ITEM**

The item containing a bundle revision ID. A bundle revision ID is always local to the result of a JMX invocation and do not have a defined meaning across invocation calls. They are used where a result can potentially contain multiple revisions of the same bundle. The key is BUNDLE_REVISION_ID and the type is an integer.

124.13.2.14   **public static final CompositeType BUNDLE_WIRE_TYPE**

The Composite type that represents a bundle wire describing the live association between a provider of a capability and a requirer of the corresponding requirement.

The composite consists of:

- BUNDLE_REQUIREMENT
- BUNDLE_CAPABILITY
- PROVIDER_BUNDLE_ID
- PROVIDER_BUNDLE_REVISION_ID
- REQUIRER_BUNDLE_ID
- REQUIRER_BUNDLE_REVISION_ID

124.13.2.15   **public static final ArrayType BUNDLE_WIRES_TYPE_ARRAY**

An array of BUNDLE_WIRE_TYPEs.

124.13.2.16   **public static final CompositeType BUNDLE_WIRING_TYPE**

The Composite Type that represents a bundle wiring. The composite consists of:

- BUNDLE_ID
- BUNDLE_REVISION_ID
- REQUIREMENTS
- CAPABILITIES
- REQUIRED_WIRES
- PROVIDED_WIRES

124.13.2.17   **public static final TabularType BUNDLES_WIRING_TYPE**

The Tabular Type to hold the wiring of a number of bundles. The row type is BUNDLE_WIRING_TYPE and the index is the combination of the BUNDLE_ID and the BUNDLE_REVISION_ID.

124.13.2.18   **public static final String CAPABILITIES = "Capabilities"**

The key of CAPABILITIES_ITEM.

124.13.2.19   **public static final Item CAPABILITIES_ITEM**

The item containing the capabilities in REVISION_CAPABILITIES_TYPE and BUNDLE_WIRING_TYPE. The key is CAPABILITIES and the type is CAPABILITY_TYPE_ARRAY.

124.13.2.20   **public static final ArrayType CAPABILITY_TYPE_ARRAY**

An array of BUNDLE_CAPABILITY_TYPEs.

124.13.2.21   **public static final CompositeType DIRECTIVE_TYPE**

The Composite Type that represents a directive for a capability or requirement. The composite consists of:

- KEY
- VALUE

124.13.2.22   **public static final String DIRECTIVES = "Directives"**

The key of DIRECTIVES_ITEM.

124.13.2.23   **public static final Item DIRECTIVES_ITEM**

The item containing the directives of a capability or requirement. Used in BUNDLE_REQUIREMENT_TYPE and BUNDLE_CAPABILITY_TYPE. The key is DIRECTIVES and the type is DIRECTIVES_TYPE.

124.13.2.24   **public static final TabularType DIRECTIVES_TYPE**

The Tabular Type that holds the directives for a capability or requirement. The row type is DIRECTIVE_TYPE and the index is KEY.

124.13.2.25   **public static final String KEY = "Key"**

The key of KEY_ITEM.

124.13.2.26   **public static final Item KEY_ITEM**

The item containing the key of a capability or requirement directive. Used in DIRECTIVE_TYPE. The key is KEY and the type is a String.

124.13.2.27	**public static final String NAMESPACE = "Namespace"**

The key of NAMESPACE_ITEM.

124.13.2.28	**public static final Item NAMESPACE_ITEM**

The item containing the namespace for a capability or requirement. Used in BUNDLE_REQUIREMENT_TYPE and BUNDLE_CAPABILITY_TYPE. The key is NAMESPACE and the type is a String.

124.13.2.29	**public static final String OBJECTNAME = "osgi.core:type=wiringState,version=1.1"**

The Object Name prefix for this mbean. The full object name also contains the framework name and uuid as properties.

124.13.2.30	**public static final String PROVIDED_WIRES = "ProvidedWires"**

The key of PROVIDED_WIRES_ITEM.

124.13.2.31	**public static final Item PROVIDED_WIRES_ITEM**

The item containing the provided wires in BUNDLE_WIRING_TYPE. The key is PROVIDED_WIRES and the type is BUNDLE_WIRES_TYPE_ARRAY.

124.13.2.32	**public static final String PROVIDER_BUNDLE_ID = "ProviderBundleId"**

The key of PROVIDER_BUNDLE_ID_ITEM.

124.13.2.33	**public static final Item PROVIDER_BUNDLE_ID_ITEM**

The item containing the provider bundle ID in BUNDLE_WIRE_TYPE. The key is PROVIDER_BUNDLE_ID and the type is a long.

124.13.2.34	**public static final String PROVIDER_BUNDLE_REVISION_ID = "ProviderBundleRevisionId"**

The key of PROVIDER_BUNDLE_REVISION_ID_ITEM.

124.13.2.35	**public static final Item PROVIDER_BUNDLE_REVISION_ID_ITEM**

The local ID of a provider revision in BUNDLE_WIRE_TYPE. This ID is local to the result where it resides and has no defined meaning across multiple invocations. The key is PROVIDER_BUNDLE_REVISION_ID and the type is an int.

124.13.2.36	**public static final String REQUIRED_WIRES = "RequiredWires"**

The key of REQUIRED_WIRES_ITEM.

124.13.2.37	**public static final Item REQUIRED_WIRES_ITEM**

The item containing the required wires in BUNDLE_WIRING_TYPE. The key is REQUIRED_WIRES and the type is BUNDLE_WIRES_TYPE_ARRAY.

124.13.2.38	**public static final ArrayType REQUIREMENT_TYPE_ARRAY**

An array of BUNDLE_REQUIREMENT_TYPEs.

124.13.2.39	**public static final String REQUIREMENTS = "Requirements"**

The key of REQUIREMENTS_ITEM.

124.13.2.40	**public static final Item REQUIREMENTS_ITEM**

The item containing the requirements in REVISION_REQUIREMENTS_TYPE and BUNDLE_WIRING_TYPE. The key is REQUIREMENTS and the type is REQUIREMENT_TYPE_ARRAY.

124.13.2.41　　　**public static final String REQUIRER_BUNDLE_ID = "RequirerBundleId"**

The key of REQUIRER_BUNDLE_ID_ITEM.

124.13.2.42　　　**public static final Item REQUIRER_BUNDLE_ID_ITEM**

The item containing the requirer bundle ID in BUNDLE_WIRE_TYPE. The key is
REQUIRER_BUNDLE_ID and the type is long.

124.13.2.43　　　**public static final String REQUIRER_BUNDLE_REVISION_ID = "RequirerBundleRevisionId"**

The key of REQUIRER_BUNDLE_REVISION_ID_ITEM.

124.13.2.44　　　**public static final Item REQUIRER_BUNDLE_REVISION_ID_ITEM**

The local ID of a requirer revision in BUNDLE_WIRE_TYPE. This ID is local to the re-
sult where it resides and has no defined meaning across multiple invocations. The key is
REQUIRER_BUNDLE_REVISION_ID and the type is an int.

124.13.2.45　　　**public static final CompositeType REVISION_CAPABILITIES_TYPE**

The Composite Type that represents the capabilities for a revision. The composite consists of:

- BUNDLE_REVISION_ID
- CAPABILITIES

124.13.2.46　　　**public static final CompositeType REVISION_REQUIREMENTS_TYPE**

The Composite Type that represents the requirements of a revision. The composite consists of:

- BUNDLE_REVISION_ID
- REQUIREMENTS

124.13.2.47　　　**public static final TabularType REVISIONS_CAPABILITIES_TYPE**

The Tabular Type that holds the capabilities of a revision. The row type is
REVISION_CAPABILITIES_TYPE and the index is BUNDLE_REVISION_ID.

124.13.2.48　　　**public static final TabularType REVISIONS_REQUIREMENTS_TYPE**

The Tabular Type that hold the requirements of a revision. The row type is
REVISION_REQUIREMENTS_TYPE and the index is BUNDLE_REVISION_ID.

124.13.2.49　　　**public static final String VALUE = "Value"**

The key of VALUE.

124.13.2.50　　　**public static final Item VALUE_ITEM**

The item containing the value of a capability or requirement directive. Used in DIRECTIVE_TYPE.
They key is VALUE and the type is a String.

124.13.2.51　　　**public CompositeData[] getCurrentRevisionDeclaredCapabilities(long bundleId,String namespace) throws
IOException, JMException**

bundleId　The bundle ID.

namespace　The namespace of the capabilities to be returned by this operation.

　□　Returns the capabilities for the current bundle revision.

Returns　the declared capabilities for the current revision of bundleId and namespace.

Throws　JMException– if there is a JMX problem.

IOException– if the connection could not be made because of a communication problem.

See Also for the details of the CompositeData.

124.13.2.52 **public CompositeData[] getCurrentRevisionDeclaredRequirements(long bundleId,String namespace) throws IOException, JMException**

bundleId The bundle ID.

namespace The namespace of the requirements to be returned by this operation.

□ Returns the requirements for the current bundle revision.

Returns the declared requirements for the current revision of `bundleId` and `namespace`.

Throws JMException– if there is a JMX problem.

IOException– if the connection could not be made because of a communication problem.

See Also for the details of the CompositeData.

124.13.2.53 **public CompositeData getCurrentWiring(long bundleId,String namespace) throws IOException, JMException**

bundleId The bundle ID.

namespace The namespace of the requirements and capabilities for which to return information.

□ Returns the bundle wiring for the current bundle revision.

Returns the wiring information for the current revision of `bundleId` and `namespace`.

Throws JMException– if there is a JMX problem.

IOException– if the connection could not be made because of a communication problem.

See Also for the details of the CompositeData.

124.13.2.54 **public TabularData getCurrentWiringClosure(long rootBundleId,String namespace) throws IOException, JMException**

rootBundleId the root bundle of the closure.

namespace The namespace of the requirements and capabilities for which to return information.

□ Returns the bundle wiring closure for the current revision of the specified bundle. The wiring closure contains all the wirings from the root bundle revision to all bundle revisions it is wired to and all their transitive wirings.

Returns a tabular representation of all the wirings in the closure. The bundle revision IDs only have meaning in the context of the current result. The revision of the rootBundle is set to 0. Therefore the root bundle of the closure can be looked up in the table by its bundle ID and revision 0.

Throws JMException– if there is a JMX problem.

IOException– if the connection could not be made because of a communication problem.

See Also for the details of the TabularData.

124.13.2.55 **public TabularData getRevisionsDeclaredCapabilities(long bundleId,String namespace) throws IOException, JMException**

bundleId The bundle ID.

namespace The namespace of the capabilities to be returned by this operation.

□ Returns the capabilities for all revisions of the bundle.

Returns the declared capabilities for all revisions of `bundleId`

Throws JMException– if there is a JMX problem.

IOException– if the connection could not be made because of a communication problem.

See Also	for the details of TabularData. The capabilities are in no particular order, and may change in subsequent calls to this operation.

124.13.2.56 **public TabularData getRevisionsDeclaredRequirements(long bundleId,String namespace) throws IOException, JMException**

bundleId	The bundle ID.
namespace	The namespace of the requirements to be returned by this operation.
□	Returns the requirements for all revisions of the bundle.
Returns	the declared requirements for all revisions of bundleId.
Throws	JMException– if there is a JMX problem.
	IOException– if the connection could not be made because of a communication problem.
See Also	for the details of TabularData. The requirements are in no particular order, and may change in subsequent calls to this operation.

124.13.2.57 **public TabularData getRevisionsWiring(long bundleId,String namespace) throws IOException, JMException**

bundleId	The bundle ID.
namespace	The namespace of the requirements and capabilities for which to return information.
□	Returns the bundle wirings for all revisions of the bundle.
Returns	the wiring information for all revisions of bundleId and namespace.
Throws	JMException– if there is a JMX problem.
	IOException– if the connection could not be made because of a communication problem.
See Also	for the details of TabularData. The bundle wirings are in no particular order, and may change in subsequent calls to this operations.

124.13.2.58 **public TabularData getRevisionsWiringClosure(long rootBundleId,String namespace) throws IOException, JMException**

rootBundleId	The root bundle ID.
namespace	The namespace of the requirements and capabilities for which to return information.
□	Returns the bundle wiring closure for all revisions of the specified bundle. The wiring closure contains all the wirings from the root bundle revision to all bundle revisions it is wired to and all their transitive wirings.
Returns	a tabular representation of all the wirings in the closure. The bundle revision IDs only have meaning in the context of the current result.
Throws	JMException– if there is a JMX problem.
	IOException– if the connection could not be made because of a communication problem.
See Also	for the details of TabularData. The bundle wirings are in no particular order, and may change in subsequent calls to this operation. Furthermore, the bundle revision IDs are local and cannot be reused across invocations.

124.14 References

[1] *JMX*
http://en.wikipedia.org/wiki/JMX

[2] *Java Management Extensions (JMX) Technology Overview*

http://docs.oracle.com/javase/1.5.0/docs/guide/jmx/overview/JMXoverviewTOC.html

[3]   *JSR 3: Java Management Extensions (JMX) Specification*
        http://www.jcp.org/en/jsr/detailid=3

[4]   *JSR 255: Java Management Extensions (JMX) Specification, version 2.0*
        http://www.jcp.org/en/jsr/detailid=255

[5]   *JSR 160: JavaTM Management Extensions (JMX) Remote API*
        http://www.jcp.org/en/jsr/detailid=160

[6]   *JSR 262: Web Services Connector for Java Management Extensions (JMX) Agents*
        http://www.jcp.org/en/jsr/detailid=262

[7]   *JavaTM Management Extensions (JMXTM)API Specification*
        http://docs.oracle.com/javase/1.5.0/docs/guide/jmx/spec.html

[8]   *Using JConsole to Monitor Applications*
        http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html

125   JDBC™ Service Specification

Version 1.0

125.1   Introduction

The Java Database Connectivity (JDBC) standard provides an API for applications to interact with relational database systems from different vendors. To abstract over concrete database systems and vendor specific characteristics, the JDBC specification provides various classes and Service Provider Interfaces (SPI) that can be used for database interaction. Implementations are database specific and provided by the corresponding driver. This specification defines how OSGi-aware JDBC drivers can provide access to their implementations. Applications can rely on this mechanism to transparently access drivers and to stay independent from driver specific classes. Additionally, this mechanism helps to use common OSGi practices and to avoid class loading problems.

This specification uses a number of packages that are defined in Java SE 1.4 or later.
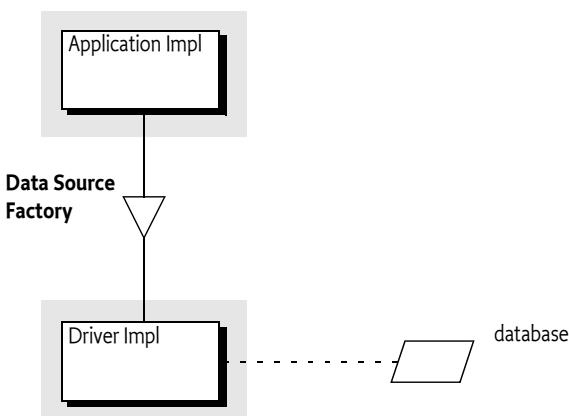
125.1.1   Essentials

- *Registration* - Provide a mechanism for JDBC driver announcements.
- *Lookup* - Inspect available database drivers and provide means for driver access.
- *Services* - Uses a service model for getting the driver objects.
- *Compatible* - Minimize the amount of work needed to support this specification for existing drivers.

125.1.2   Entities

- *Relational Database Management Systems* - (RDBMS) An external database system.
- *Database Driver* - JDBC-compliant database driver that is delivered in a bundle.
- *Data Source Factory* - Provides one of the different Data Sources that gives access to a database driver.
- *Application* - The application that wants to access a relational database system.

Figure 125.1   *JDBC Class/Service Overview*

125.1.3   Dependencies

The classes and interfaces used in this specification come from the following packages:

```
javax.sql
java.sql
```

These packages have no associated version. It is assumed they come from the runtime environment. This specification is based on Java SE 1.4 or later.

125.1.4   Synopsis

A JDBC *Database Driver* is the software that maps the JDBC specification to a specific implementation of a relational database. For OSGi, JDBC drivers are delivered as driver bundles. A driver bundle registers a Data Source Factory service when it is ACTIVE. Service properties are used to specify the database driver name, version, etc. The Data Source Factory service provides methods to create DataSource, ConnectionPoolDataSource, XADataSource, or Driver objects. These objects are then used by an application to interact with the relational database system in the standard way.

The application can query the service registry for available Data Source Factory services. It can select particular drivers by filtering on the service properties. This service based model is easy to use with dependency injection frameworks like Blueprint or Declarative Services.

125.2   Database Driver

A Database Driver provides the connection between an *Application* and a particular database. A single OSGi Framework can contain several Database Drivers simultaneously. To make itself available to Applications, a Database Driver must register a Data Source Factory service. Applications must be able to find the appropriate Database Driver. The Database Driver must therefore register the Data Source Factory service with the following service properties:

- OSGI_JDBC_DRIVER_CLASS - (String) The required name of the driver implementation class. This property is the primary key to find a driver's Data Source Factory. It is not required that there is an actual class with this name.
- OSGI_JDBC_DRIVER_NAME - (String) The optional driver name. This property is informational.
- OSGI_JDBC_DRIVER_VERSION - (String) The driver version. The version is not required to be an OSGi version, it should be treated as an opaque string. This version is likely not related to the package of the implementation class or its bundle.

The previous properties are vendor-specific and are meant to further describe the Database Driver to the Application.

Each Data Source Factory service must relate to a single Database Driver. The Database Driver implementation bundle does not necessarily need to be the registrar of the Data Source Factory service. Any bundle can provide the Data Source Factory service and delegate to the appropriate driver specific implementation classes. However, as JDBC driver implementations evolve to include built-in support for OSGi they can provide the Data Source Factory service themselves. This implies that the same driver can be registered multiple times.

125.2.1   Life Cycle

A Data Source Factory service should be registered while its Driver Bundle is in the ACTIVE state or when it has a lazy activation policy and is in the STARTING state.

What happens to the objects created by the Data Source Factory service, and the objects they created, is undefined in this specification. Database Drivers are not mandated to track the proper life cycle of these objects.

125.2.2   Package Dependencies

A Database Driver must import the javax.sql package. The java.sql package that contains the Driver and SQLException interface is automatically imported because it starts with java.. Both packages are contained in the JRE since Java SE 1.4. These packages are not normally versioned with OSGi version numbers. Bundles using the Data Source Factory must therefore ensure they get the proper imports, which is usually from the JRE. Due to the lack of specified metadata, the deployer is responsible for ensuring this.

125.3   Applications

125.3.1   Selecting the Data Source Factory Service

Applications can query the OSGi service registry for available Database Drivers by getting a list of Data Source Factory services. Normally, the application needs access to specific drivers that match their needed relational database type. The service properties can be used to find the desired Database Driver. This model is well supported by dependency injection frameworks like Blueprint or Declarative Services. However, it can of course also be used with the basic service methods. The following code shows how a Service Tracker can be used to get a Database Driver called ACME DB.

```
Filter filter = context.createFilter(
    "(&(objectClass="  +
        DataSourceFactory.class.getName() +
    ")(" +
        DataSourceFactory.OSGI_JDBC_DRIVER_CLASS + "=com.acme.db.Driver))");

ServiceTracker tracker = new ServiceTracker(context, filter, null);
tracker.open();

DataSourceFactory dsf = (DataSourceFactory) tracker.getService();
```

125.3.2   Using Database Drivers

The Data Source Factory service can be used to obtain instances for the following JDBC related types:

- javax.sql.DataSource
- javax.sql.ConnectionPoolDataSource
- javax.sql.XADataSource
- java.sql.Driver

Which type of Connection provider that is actually required depends on the Application and the use case. For each type, the Data Source Factory service provides a method that returns the corresponding instance. Each method takes a Properties object as a parameter to pass a configuration to the Database Driver implementation. The configuration is driver-specific and can be used to specify the URL for the database and user credentials. Common property names for these configuration properties are also defined in the DataSourceFactory interface.

A Data Source Factory is not required to implement all of the factory methods. If an implementation does not support a particular type then it must throw a SQL Exception. This specification does not provide a mechanism to depend on a Data Source Factory service that implements a particular factory method.

The following code shows how a DataSource object could be created.

```
Properties props = new Properties();
```

```
props.put(DataSourceFactory.JDBC_URL,      "jdbc:acme:ACMEDB");
props.put(DataSourceFactory.JDBC_USER,     "foo");
props.put(DataSourceFactory.JDBC_PASSWORD, "secret");
DataSource dataSource = dsf.createDataSource(props);
```

The DataSourceFactory interface has several static fields that represent common property keys for the Properties instance. General properties are:

- JDBC_DATABASE_NAME
- JDBC_DATASOURCE_NAME
- JDBC_DESCRIPTION
- JDBC_NETWORK_PROTOCOL
- JDBC_PASSWORD
- JDBC_PORT_NUMBER
- JDBC_ROLE_NAME
- JDBC_SERVER_NAME
- JDBC_USER
- JDBC_URL

The following additional property keys are provided for applications that want to create a ConnectionPoolDataSource object or a XAPoolDataSource object:

- JDBC_INITIAL_POOL_SIZE
- JDBC_MAX_IDLE_TIME
- JDBC_MAX_POOL_SIZE
- JDBC_MAX_STATEMENTS
- JDBC_MIN_POOL_SIZE
- JDBC_PROPERTY_CYCLE

Which property keys and values are supported depends on the driver implementation. Drivers can support additional custom configuration properties.

125.3.3 Using JDBC in OSGi and Containers

The JDBC service provides JDBC driver services, not *container* services. A typical client would only use the DataSourceFactory.createDataSource() method to procure a regular Data Source from which they can obtain (usually non-pooled) connections.

Containers generally offer connection pools and support XA transactions. The container manages the pools and does this by using Pooled Connection or XA Connection objects from a driver-implemented respective Connection Pool Data Source or XA Data Source. To support containers, frameworks, or any client that wants to manage a pool, these Data Source types are included in the Data Source Factory service. Drivers are permitted to implement their own Data Source using an underlying connection pooling scheme. This is driver-dependent and not related to the OSGi specifications.

The usual set of JDBC properties are defined in the services for use with the Data Source types. They are the same as what is defined for JDBC and the caller should know which properties make sense when passed to a given Data Source type. The same result should occur in OSGi as occurs outside of OSGi. If the driver does not support a given property with a given Data Source type then it can ignore it or it can throw an Exception.

125.4 Security

This specification depends on the JDBC specification for security.

125.5   org.osgi.service.jdbc

JDBC Service Package Version 1.0.

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

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.jdbc; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.jdbc; version="[1.0,1.1)"

125.5.1   Summary

- DataSourceFactory - A factory for JDBC connection factories.

125.5.2   public interface DataSourceFactory

A factory for JDBC connection factories. There are 3 preferred connection factories for getting JDBC connections: javax.sql.DataSource, javax.sql.ConnectionPoolDataSource, and javax.sql.XADataSource. DataSource providers should implement this interface and register it as an OSGi service with the JDBC driver class name in the OSGI_JDBC_DRIVER_CLASS property.

Concurrency Thread-safe

125.5.2.1   public static final String JDBC_DATABASE_NAME = "databaseName"

The "databaseName" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.2   public static final String JDBC_DATASOURCE_NAME = "dataSourceName"

The "dataSourceName" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.3   public static final String JDBC_DESCRIPTION = "description"

The "description" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.4   public static final String JDBC_INITIAL_POOL_SIZE = "initialPoolSize"

The "initialPoolSize" property that ConnectionPoolDataSource and XADataSource clients may supply a value for when calling createConnectionPoolDataSource(Properties) or createXADataSource(Properties) on drivers that support this property.

125.5.2.5   public static final String JDBC_MAX_IDLE_TIME = "maxIdleTime"

The "maxIdleTime" property that ConnectionPoolDataSource and XADataSource clients may supply a value for when calling createConnectionPoolDataSource(Properties) or createXADataSource(Properties) on drivers that support this property.

125.5.2.6   public static final String JDBC_MAX_POOL_SIZE = "maxPoolSize"

The "maxPoolSize" property that ConnectionPoolDataSource and XADataSource clients may supply a value for when calling createConnectionPoolDataSource(Properties) or createXADataSource(Properties) on drivers that support this property.

125.5.2.7   **public static final String JDBC_MAX_STATEMENTS = "maxStatements"**

The "maxStatements" property that ConnectionPoolDataSource and XADataSource clients may supply a value for when calling createConnectionPoolDataSource(Properties) or createXADataSource(Properties) on drivers that support this property.

125.5.2.8   **public static final String JDBC_MIN_POOL_SIZE = "minPoolSize"**

The "minPoolSize" property that ConnectionPoolDataSource and XADataSource clients may supply a value for when calling createConnectionPoolDataSource(Properties) or createXADataSource(Properties) on drivers that support this property.

125.5.2.9   **public static final String JDBC_NETWORK_PROTOCOL = "networkProtocol"**

The "networkProtocol" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.10   **public static final String JDBC_PASSWORD = "password"**

The "password" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.11   **public static final String JDBC_PORT_NUMBER = "portNumber"**

The "portNumber" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.12   **public static final String JDBC_PROPERTY_CYCLE = "propertyCycle"**

The "propertyCycle" property that ConnectionPoolDataSource and XADataSource clients may supply a value for when calling createConnectionPoolDataSource(Properties) or createXADataSource(Properties) on drivers that support this property.

125.5.2.13   **public static final String JDBC_ROLE_NAME = "roleName"**

The "roleName" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.14   **public static final String JDBC_SERVER_NAME = "serverName"**

The "serverName" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.15   **public static final String JDBC_URL = "url"**

The "url" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.16   **public static final String JDBC_USER = "user"**

The "user" property that DataSource clients should supply a value for when calling createDataSource(Properties).

125.5.2.17   **public static final String OSGI_JDBC_DRIVER_CLASS = "osgi.jdbc.driver.class"**

Service property used by a JDBC driver to declare the driver class when registering a JDBC DataSourceFactory service. Clients may filter or test this property to determine if the driver is suitable, or the desired one.

125.5.2.18   **public static final String OSGI_JDBC_DRIVER_NAME = "osgi.jdbc.driver.name"**

Service property used by a JDBC driver to declare the driver name when registering a JDBC DataSourceFactory service. Clients may filter or test this property to determine if the driver is suitable, or the desired one.

125.5.2.19   **public static final String OSGI_JDBC_DRIVER_VERSION = "osgi.jdbc.driver.version"**

Service property used by a JDBC driver to declare the driver version when registering a JDBC DataSourceFactory service. Clients may filter or test this property to determine if the driver is suitable, or the desired one.

125.5.2.20   **public ConnectionPoolDataSource createConnectionPoolDataSource(Properties props) throws SQLException**

props The properties used to configure the ConnectionPoolDataSource. null indicates no properties. If the property cannot be set on the ConnectionPoolDataSource being created then a SQLException must be thrown.

☐ Create a new ConnectionPoolDataSource using the given properties.

Returns A configured ConnectionPoolDataSource.

Throws SQLException– If the ConnectionPoolDataSource cannot be created.

125.5.2.21   **public DataSource createDataSource(Properties props) throws SQLException**

props The properties used to configure the DataSource . null indicates no properties. If the property cannot be set on the DataSource being created then a SQLException must be thrown.

☐ Create a new DataSource using the given properties.

Returns A configured DataSource.

Throws SQLException– If the DataSource cannot be created.

125.5.2.22   **public Driver createDriver(Properties props) throws SQLException**

props The properties used to configure the Driver. null indicates no properties. If the property cannot be set on the Driver being created then a SQLException must be thrown.

☐ Create a new Driver using the given properties.

Returns A configured Driver.

Throws SQLException– If the Driver cannot be created.

125.5.2.23   **public XADataSource createXADataSource(Properties props) throws SQLException**

props The properties used to configure the XADataSource. null indicates no properties. If the property cannot be set on the XADataSource being created then a SQLException must be thrown.

☐ Create a new XADataSource using the given properties.

Returns A configured XADataSource.

Throws SQLException– If the XADataSource cannot be created.

125.6   References

[1]   *Java SE 1.4*
http://www.oracle.com/technetwork/java/archive-139210.html

126   JNDI Services Specification

Version 1.0

126.1   Introduction

Naming and directory services have long been useful tools in the building of software systems. The ability to use a programming interface to publish and consume objects can provide many benefits to any system. The Java Naming and Directory Interface (JNDI) is a registry technology in Java applications, both in the Java SE and Java EE space. JNDI provides a vendor-neutral set of APIs that allow clients to interact with a naming service from different vendors.

The JNDI as used in the Java SE environment relies on the class loading model provided by the JDK to find providers. By default, it attempts to load the JNDI provider class using the Thread Context Class Loader. In an OSGi environment, this type of Context creation is not desirable since it relies on the JNDI provider classes being visible to the JNDI client, or require it to set the Context Class Loader; in both cases breaking modularity. For modularity reasons, it is important that clients are not required to express a dependency on the implementation of services they use.

This specification will define how JNDI can be utilized from within an OSGi framework. The specification consists of three key parts:

- *OSGi Service Model* - How clients interact with JNDI when running inside an OSGi Framework.
- *JNDI Provider Model* - How JNDI providers can advertise their existence so they are available to OSGi and traditional clients.
- *Traditional Model* - How traditional JNDI applications and providers can continue to work in an OSGi Framework without needing to be rewritten when certain precautions are taken.

126.1.1   Essentials
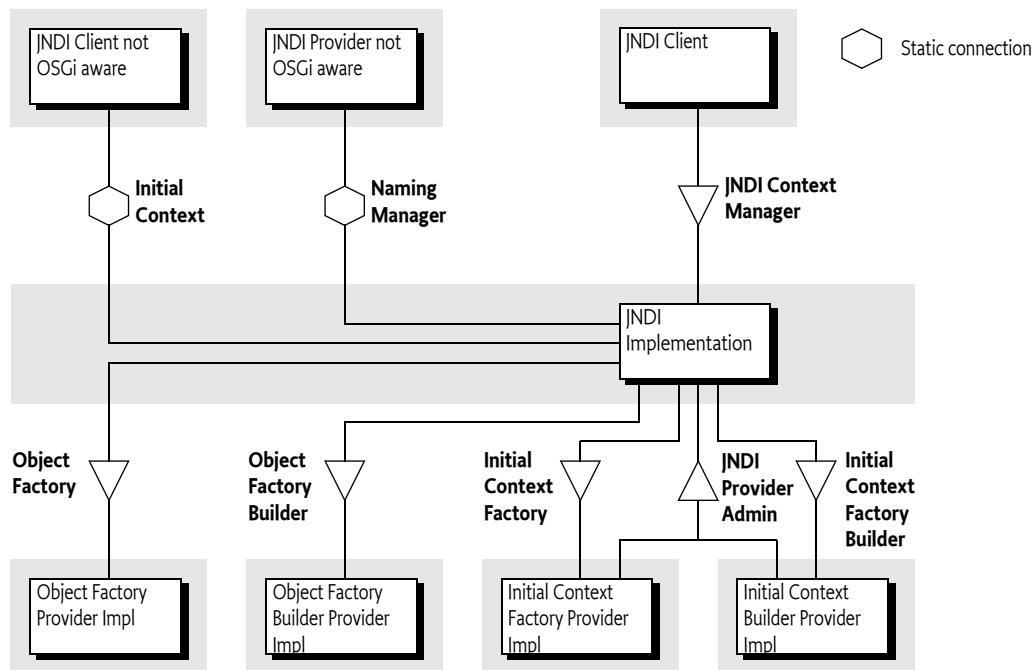
- *Naming Service* - Provide an integration model for JNDI API clients and providers.
- *Flexible* - Provide a standard mechanism for publishing and locating JNDI providers.
- *Compatibility* - Support the traditional JNDI programming model used by Java SE and Java EE clients.
- *Service Based* - Provide a service model that clients and providers can use to leverage JNDI facilities.
- *Migration* - Provide a mechanism to access OSGi services from a JNDI context.

126.1.2   Entities

- *JNDI Implementation* - The Implementer of the JNDI Context Manager, JNDI Provider Admin, and setter of the JNDI static singletons.
- *JNDI Client* - Any code running within an OSGi bundle that needs to use JNDI.
- *JNDI Context Manager* - A service that allows clients to obtain Contexts via a service.
- *JNDI Provider Admin* - A service that allows the conversion of objects for providers.
- *JNDI Provider* - Provides a Context implementation.
- *Context* - A Context abstracts a namespace. Implementations are provided by JNDI providers and the Contexts are used by JNDI clients. The corresponding interface is javax.naming.Context.

- *Dir Context* - A sub-type of Context that provides mechanisms for examining and updating the attributes of an object in a directory structure, and for performing searches in an hierarchical naming systems like LDAP. The corresponding interface is javax.naming.directory.DirContext.
- *Initial Context Factory* - A factory for creating instances of Context objects. This factory is used to integrate new JNDI Providers. In general, a single Initial Context Factory constructs Context objects for a single provider implementation. The corresponding interface is javax.naming.spi.InitialContextFactory.
- *Initial Context Factory Builder* - A factory for InitialContextFactory objects. A single Initial Context Factory Builder can construct InitialContextFactory objects for different types of Contexts. The interface is javax.naming.spi.InitialContextFactoryBuilder.
- *Object Factory* - Used in conversion of objects. The corresponding interface is javax.naming.spi.ObjectFactory.
- *Dir Object Factory* - An Object Factory that takes attribute information for object conversion. The corresponding interface is javax.naming.spi.DirObjectFactory.
- *Object Factory Builder* - A factory for ObjectFactory objects. A single Object Factory Builder can construct ObjectFactory instances for different types of conversions. The corresponding interface is javax.naming.spi.ObjectFactoryBuilder.
- *Reference* - A description of an object that can be turned into an object through an Object Factory. The associated Referenceable interface implemented on an object indicates that it can provide a Reference object.

Figure 126.1   *JNDI Service Specification Service Entities*

126.1.3   Dependencies

The classes and interfaces used in this specification come from the following packages:

```
javax.naming
javax.naming.spi
javax.naming.directory
```

These packages have no associated version. It is assumed they come from the runtime environment. This specification is based on Java SE 1.4 or later.

126.1.4 Synopsis

A client bundle wishing to make use of JNDI in order to access JNDI Providers such as LDAP or DNS in OSGi should not use the Naming Manager but instead use the JNDI Context Manager service. This service can be asked for a Context based on environment properties. The environment properties are based on an optional argument in the `newInitialContext` method, the Java System properties, and an optional resource in the caller's bundle.

These environment properties can specify an implementation class name for a factory that can create a `Context` object. If such a class name is specified, then it is searched for in the service registry. If such a service is found, then that service is used to create a new Context, which is subsequently returned. If no class name is specified, the service registry is searched for Initial Context Factory services. These services are tried in ranking order to see if they can create an appropriate Context, the first one that can create a Context is then used.

If no class name is specified, all Initial Context Factory Builder services are tried to see if they can create a Context, the first non-`null` result is used. If no Context can be found, a No Initial Context Exception is thrown. Otherwise, the JNDI Context Manager service returns an initial Context that uses the just created Context from a provider as the backing service. This initial Context delegates all operations to this backing Context, except operations that use a name that can be interpreted as a URL, that is, the name contains a colon. URL operations are delegated a URL Context that is associated with the used scheme. URL Contexts are found through the general object conversion facility provided by the JNDI Provider Admin service.

The JNDI Provider Admin service provides a general object conversion facility that can be extended with Object Factory and Object Factory Builder services that are traditionally provided through the Naming Manager `getObjectInstance` method. A specific case for this conversion is the use of `Reference` objects. `Reference` objects can be used to store objects persistently in a Context implementation. Reference objects must be converted to their corresponding object when retrieved from a Context.

During the client's use of a Context it is possible that its provider's service is unregistered. In this case the JNDI Context Manager must release the backing Context. If the initial Context is used and no backing Context is available, the JNDI Context Manager must re-create a new Context, if possible. Otherwise a Naming Exception is thrown. If subsequently a proper new backing Context can be created, the initial Context must start operating again.

The JNDI Context Manager service must track the life cycle of a calling bundle and ensure that any returned `Context` objects are closed and returned objects are properly cleaned up when the bundle is closed or the JNDI Context Manager service is unget.

When the client bundle is stopped, any returned initial Context objects are closed and discarded. If the Initial Context Factory, or Initial Context Factory Builder, service that created the initial Context goes away then the JNDI Context Manager service releases the Context backing the initial Context and attempts to create a replacement Context.

Clients and JNDI Context providers that are unaware of OSGi use static methods to connect to the JRE JNDI implementation. The `InitialContext` class provides access to a Context from a provider and providers use the static `NamingManager` methods to do object conversion and find URL Contexts. This traditional model is not aware of OSGi and can therefore only be used reliably if the consequences of this lack of OSGi awareness are managed.

126.2   JNDI Overview

The Java Naming and Directory Interface (JNDI) provides an abstraction for namespaces that is included in Java SE. This section describes the basic concepts of JNDI as provided in Java SE. These concepts are later used in the service model provided by this specification.

126.2.1   Context and Dir Context

The *[1] Java Naming and Directory Interface* (JNDI) defines an API for *namespaces*. These namespaces are abstracted with the Context interface. Namespaces that support *attributes*, such as a namespace as the Lightweight Directory Access Protocol (LDAP), are represented by the DirContext class, which extends the Context class. If applicable, a Context object can be cast to a DirContext object. The distinction is not relevant for this specification, except in places where it is especially mentioned.

The Context interface models a set of name-to-object *bindings* within a namespace. These bindings can be looked-up, created, and updated through the Context interface. The Context interface can be used for federated, flat, or hierarchical namespaces.

126.2.2   Initial Context
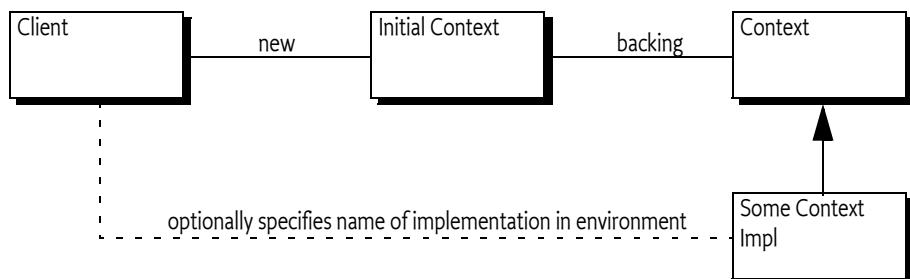
Obtaining a Context for a specific namespace, for example DNS, is handled through the InitialContext class. Creating an instance of this class will cause the JRE to find a *backing* Context. The Initial Context is only a facade for the backing Context. The facade context provides URL based lookups.

The backing Context is created by a *JNDI Provider*. How this backing Context is created is an elaborate process using class loading techniques or a provisioning mechanism involving *builders*, see *Naming Manager Singletons* on page 501 for more information about the builder provisioning mechanism.

If there is no Initial Context Factory Builder set, the class name of a class implementing the InitialContextFactory interface is specified as a property in the *environment*. The environment is a Hashtable object that is constructed from different sources and then merged with System properties and a resource in the calling bundle, see *Environment* on page 501. In a standard Java SE JNDI, the given class name is then used to construct an InitialContextFactory object and this object is then used to create the backing Context. This process is depicted in Figure 126.2 on page 500.

Figure 126.2   *Backing Context*

126.2.3   URL Context Factory

The InitialContext class implements the Context interface. It can therefore delegate all the Context interface methods to the backing Context object. However, it provides a special URL lookup behavior for names that are formed like URLs, that is, names that contain a colon (':' \u003A) character. This behavior is called a *URL lookup*.

URL lookups are not delegated to the backing Context but are instead first tried via a *URL Context* based lookup on the given scheme, like:

```
myscheme: foo
```

For example a lookup using `acme:foo/javax.sql.DataSource` results in a URL Context being used, rather than the backing Context.

JNDI uses class loading techniques to search for an `ObjectFactory` class that can be used to create this URL Context. The Naming Manager provides a static method `getURLContext` for this purpose. If such a URL Context is found, it is used with the requested operation and uses the full URL. If no such URL Context can be found, the backing Context is asked to perform the operation with the given name.

The URL lookup behavior is only done when the backing Context was created by the JNDI implementation in the JRE. If the backing Context had been created through the singleton provisioning mechanism, then no URL lookup is done for names that have a colon. The URL lookup responsibility is then left to the backing Context implementation.

126.2.4 Object and Reference Conversion

The `NamingManager` class provides a way to create objects from a *description* with the `getObjectInstance` method. In general, it will iterate over a number of `ObjectFactory` objects and ask each one of them to provide the requested object. The first non-null result indicates success. These `ObjectFactory` objects are created from an environment property.

A special case for the description argument in the `getObjectInstance` method is the *Reference*. A Reference is a description of an object that can be stored persistently. It can be re-created into an actual object through the static `getObjectInstance` method of the `NamingManager` class. The `Reference` object describes the actual `ObjectFactory` implementing class that must be used to create the object.

This default behavior is completely replaced with the Object Factory Builder singleton by getting the to be used `ObjectFactory` object directly from the set singleton Object Factory Builder.

126.2.5 Environment

JNDI clients need a way to set the configuration properties to select the proper JNDI Provider. For example, a JNDI Provider might require an identity and a password in order to access the service. This type of configuration is referred to as the *environment* of a Context. The environment is a set of properties. Common property names can be found in [3] *JNDI Standard Property Names*. The set of properties is build from the following sources (in priority order, that is later entries are shadowed by earlier entries):

1. Properties set in the environment `Hashtable` object given in the constructor argument (if any) of the `InitialContext` class.
2. Properties from the Java System Properties
3. Properties found in $JAVA_HOME/lib/jndi.properties

There are some special rules around the handling of specific properties.

126.2.6 Naming Manager Singletons

The default behavior of the JRE implementation of JNDI can be extended in a standardized way. The `NamingManager` class has two static singletons that allow JNDI Providers outside the JRE to provide `InitialContextFactory` and `ObjectFactory` objects. These singletons are set with the following static methods on the `NamingManager` class:

- `setObjectFactoryBuilder(ObjectFactoryBuilder)` - A hook to provide `ObjectFactory` objects.
- `setInitialContextFactoryBuilder(InitialContextFactoryBuilder)` - A hook to provide `InitialContextFactory` objects. This hook is consulted to create a `Context` object that will be associated with an `InitialContext` object the client creates.

These JNDI Provider hooks are *singletons* and must be set *before* any application code creates an InitialContext object or any objects are converted. If these singletons are not set, the JNDI implementation in the JRE will provide a default behavior that is based on searching through classes defined in an environment property.

Both singletons can only be set once. A second attempt to set these singletons results in an Illegal State Exception being thrown.

126.2.7 Built-In JNDI Providers

The Java Runtime Environment (JRE) defines the following default providers:

- *LDAP* - Lightweight Directory Access Protocol (LDAP) service provider
- *COS* - Corba Object Service (COS) naming service provider
- *RMI* - Remote Method Invocation (RMI) Registry service provider
- *DNS* - Domain Name System (DNS) service provider

Although these are the default JNDI Service Providers, the JNDI architecture provides a number of mechanisms to plug-in new types of providers.

126.3 JNDI Context Manager Service

The JNDI Context Manager service allows clients to obtain a Context using the OSGi service model. By obtaining a JNDI Context Manager service, a client can get a Context object so that it can interact with the available JNDI Providers. This service replaces the approach where the creation of a new InitialContext object provided the client with access to an InitialContext object that was backed by a JNDI Provider's Context.

The JNDIContextManager interface defines the following methods for obtaining Context objects:

- newInitialContext() - Obtain a Context object using the default environment properties.
- newInitialContext(Map) - Get a Context object using the default environment properties merged with the given properties.
- newInitialDirContext() - Get a DirContext object using a default environment properties.
- newInitialDirContext(Map) -Get a DirContext object using the default environment properties merged with the given properties.

The JNDI Context Manager service returns Context objects that implement the same behavior as the InitialContext class; the returned Context object does not actually extend the InitialContext class, its only guarantee is that it implements the Context interface.

This Context object is a facade for the context that is created by the JNDI Provider. This JNDI Provider's Context is called the *backing Context.* This is similar to the behavior of the InitialContext class. However, in this specification, the facade can change or loose the backing Context due to the dynamics of the OSGi framework.

The returned facade must also provides URL lookups, just like an Initial Context. However, the URL Context lookup must be based on Object Factory services with a service property that defines the scheme.

The environment properties used to create the backing Context are constructed in a similar way as the environment properties of the Java SE JNDI, see *Environment and Bundles* on page 503.

The following sections define in detail how a JNDI Provider Context must be created and managed.

126.3.1 Environment and Bundles

The Java SE JNDI looks for a file in $JAVAHOME/lib/jndi.properties, see *Environment* on page 501. A JNDI Implementation must not use this information but it must use a resource in the bundle that uses the JNDI Context Manager service. The order is therefore:

1. Properties set in the environment Hashtable object given in the constructor argument (if any) of the InitialContext class.
2. Properties from the Java System Properties
3. A properties resource from the bundle that uses the service called /jndi.properties.

The following four properties do not overwrite other properties but are merged:

- java.naming.factory.object
- java.naming.factory.state
- java.naming.factory.control
- java.naming.factory.url.pkgs

These property values are considered lists and the ultimate value used by the JNDI Providers is taken by merging the values found in each stage into a single colon separated list. For more information see [3] *JNDI Standard Property Names*.

The environment consists of the merged properties. This environment is then passed to the Initial Context Factory Builder for the creation of an Initial Context Factory.

126.3.2 Context Creation

When a client calls one of the newInitialContext (or newInitialDirContext) methods, the JNDI Context Manager service must construct an object that implements the Context interface based on the environment properties. All factory methods in the InitialContextFactory and InitialContextFactoryBuilder classes take a Hashtable object with the environment as an argument, see *Environment and Bundles* on page 503.

The caller normally provides a specific property in the environment that specifies the class name of a provider class. This property is named:

```
java.naming.factory.initial
```

The algorithm to find the provider of the requested Context can differ depending on the presence or absence of the java.naming.factory.initial property in the environment.

In the following sections the cases for presence or absence of the java.naming.factory.initial property are described. Several steps in these algorithm iterate over a set of available services. This iteration must always take place in service *ranking order*. Service ranking order follows the ordering of the service.ranking service property, which is the highest service.ranking value, or when equal, the lowest service.id value.

Exception handling in the following steps is as follows:

- If an Exception is thrown by an Initial Context Factory Builder service, then this Exception must be logged but further ignored.
- Exceptions thrown by the InitialContextFactory objects when creating a Context must be thrown to the caller.

126.3.2.1 Implementation Class Present in Environment

If the implementation class is specified, a JNDI Provider is searched in the service registry with the following steps, which stop when a backing Context can be created:

1. Find a service in ranking order that has a name matching the given implementation class name as well as the InitialContextFactory class name. The searching must take place through the Bundle Context of the requesting bundle but must not require that the requesting bundle imports the package of the implementation class. If such a matching Initial Context Factory service is found, it must be used to construct the Context object that will act as the backing Context.

2. Get all the Initial Context Factory Builder services. For each such service, in ranking order:
   - Ask the Initial Context Factory Builder service to create a new InitialContextFactory object. If this is null then continue with the next service.
   - Create the Context with the found Initial Context Factory and return it.

3. If no backing Context could be found using these steps, then the JNDI Context Manager service must throw a No Initial Context Exception.

126.3.2.2 No Implementation Class Specified

If the environment does not contain a value for the java.naming.factory.initial property then the following steps must be used to find a backing Context object.

1. Get all the Initial Context Factory Builder services. For each such service, in ranking order, do:
   - Ask the Initial Context Factory Builder service to create a new InitialContextFactory object. If this is null, then continue with the next service.
   - Create the backing Context object with the found Initial Context Factory service and return it.

2. Get all the Initial Context Factory services. For each such service, in ranking order, do:
   - Ask the Initial Context Factory service to create a new Context object. If this is null then continue with the next service otherwise create a new Context with the created Context as the backing Context.

3. If no Context has been found, an initial Context is returned without any backing. This returned initial Context can then only be used to perform URL based lookups.

126.3.3 Rebinding

A JNDI Provider can be added or removed to the service registry at any time because it is an OSGi service; OSGi services are by their nature dynamic. When a JNDI Provider unregisters an Initial Context Factory that was used to create a backing service then the JNDI Context Manager service must remove the association between any returned Contexts and their now invalid backing Contexts.

The JNDI Context Manager service must try to find a replacement whenever it is accessed and no backing Context is available. However, if no such replacement can be found the called function must result in throwing a No Initial Context Exception.

126.3.4 Life Cycle and Dynamism

When a client has finished with a Context object, then the client must close this Context object by calling the close method. When a Context object is closed, the resources held by the JNDI Implementation on the client's behalf for that Context must all be released. Releasing these resources must not affect other, independent, Context objects returned to the same client.

If a client ungets the JNDI Context Manager service, all the Context objects returned through that service instance must automatically be closed by the JNDI Context Manager. When the JNDI Context Manager service is unregistered, the JNDI Context Manager must automatically close all Contexts held.

For more information about life cycle issues, see also *Life Cycle Mismatch* on page 511.

126.4   JNDI Provider Admin service

JNDI provides a general object conversion service, see *Object and Reference Conversion* on page 501. For this specification, the responsibility of the static method on the NamingManager getObjectIn-stance is replaced with the JNDI Provider Admin service. The JNDIProviderAdmin interface provides the following methods that can be used to convert a description object to an object:

- getObjectInstance(Object,Name,Context,Map) - Used by Context implementations to convert a description object to another object.
- getObjectInstance(Object,Name,Context,Map,Attributes) - Used by a Dir Context implementations to convert a description object to another object.

In either case, the first argument is an object, called the *description*. JNDI allows a number of different Java types here. When either method is called, the following algorithm is followed to find a matching Object Factory to find/create the requested object. This algorithm is identical for both methods, except that the call that takes the Attributes argument consults Dir Object Factory services first and then Object Factory services while the method without the Attributes parameter only consults Object Factory services.

1. If the description object is an instance of Referenceable, then get the corresponding Reference object and use this as the description object.
2. If the description object is not a Reference object then goto step 5.
3. If a factory class name is specified, the JNDI Provider Admin service uses its own Bundle Context to search for a service registered under the Reference's factory class name. If a matching Object Factory is found then it is used to create the object from the Reference object and the algorithm stops here.
4. If no factory class name is specified, iterate over all the Reference object's StringRefAddrs objects with the address type of URL. For each matching address type, use the value to find a matching URL Context, see *URL Context Provider* on page 507, and use it to recreate the object. See the Naming Manager for details. If an object is created then it is returned and the algorithm stops here.
5. Iterate over the Object Factory Builder services in ranking order. Attempt to use each such service to create an ObjectFactory or DirObjectFactory instance. If this succeeds (non null) then use this ObjectFactory or DirObjectFactory instance to recreate the object. If successful, the algorithm stops here.
6. If the description was a Reference and without a factory class name specified, or if the description was not of type Reference, then attempt to convert the object with each Object Factory service (or Dir Object Factory service for directories) service in ranking order until a non-null value is returned.
7. If no ObjectFactory implementations can be located to resolve the given description object, the description object is returned.

If an Exception occurs during the use of an Object Factory Builder service then this exception should be logged but must be ignored. If, however, an Exception occurs during the calling of a found ObjectFactory or DirObjecFactory object then this Exception must be re-thrown to the caller of the JNDI Provider Admin service.

126.5   JNDI Providers

JNDI Providers can be registered by registering an appropriate service. These services are consulted by the JNDI Implementation for creating a Context as well as creating/finding/converting general objects.

126.5.1　Initial Context Factory Builder Provider

An Initial Context Factory Builder provider is asked to provide an Initial Context Factory when no implementation class is specified or no such implementation can be found. An Initial Context Factory Builder service can be used by containers for other bundles to control the initial Context their applications receive.

An Initial Context Factory Builder provider must register an Initial Context Factory Builder service. The service.ranking property defines the iteration ordering of multiple Initial Context Factory Builder services. Implementations must be careful to correctly provide defaults.

For example, a container could use a thread local variable to mark the stack for a specific application. The implementation of the Initial Context Factory Builder can then detect specific calls from this application. To make the next code example work, an instance must be registered as an Initial Context Factory Builder service.

```java
public class Container implements InitialContextFactoryBuilder {
    ThreadLocal<Application> apps;

    void startApp(final Application app) {
        Thread appThread = new Thread(app.getName()) {
            public void run() {
                apps.set(app);
                    app.run();
    }}}

    public InitialContextFactory
        createInitialContextFactory( Hashtable<?,?> ht ) {
        final Application app = apps.get();
        if ( app == null )
          return null;

        return new InitialContextFactory() {
            public Context getInitialContext( Hashtable<?,?>env) {
                return app.getContext(env);
            }
        };
    } }
```

126.5.2　Initial Context Factory Provider

An Initial Context Factory provides Contexts of a specific type. For example, those contexts allow communications with an LDAP server. An Initial Context Factory Provider must register the its Initial Context Factory service under the following names:

- *Implementation Class* - An Initial Context Factory provider must register a service under the name of the implementation class. This allows the JNDI Context Manager to find implementations specified in the environment properties.
- *Initial Context Factory* - As a general Initial Context Factory. If registered as such, it can be consulted for a default Initial Context. Implementations must be careful to only return a Context when the environment properties are appropriate. See *No Implementation Class Specified* on page 504

An Initial Context Factory service can create both DirContext as well as Context objects.

For example, SUN JREs for Java SE provide an implementation of a Context that can answer DNS questions. The name of the implementation class is a well known constant. The following class can be used with Declarative Services to provide a lazy implementation of a DNS Context:

```
public class DNSProvider implements InitialContextFactory{
    public Context createInitialContextFactory( Hashtable<?,?>env ) throws
        NamingException {
        try {
            Class<InitialContextFactory> cf = (Class<InitialContextFactory>)
                l.loadClass("com.sun.jndi.dns.DnsContextFactory" );
            InitialContextFactory icf = cf.newInstance();
            return icf.createInitialContextFactory(env);
        } catch( Throwable t ) {
            return null;
        }
    }
}
```

126.5.3 Object Factory Builder Provider

An Object Factory Builder provider must register an Object Factory Builder service. Such a service can be used to provide ObjectFactory and/or DirObjectFactory objects. An Object Factory Builder service is requested for such an object when no specific converter can be found. This service can be leveraged by bundles that act as a container for other bundles to control the object conversion for their subjects.

126.5.4 Object Factory Provider

An Object Factory provider can participate in the conversion of objects. It must register a service under the following names:

- *Implementation Class* - A service registered under its implementation class can be leveraged by a description that is a Reference object. Such an object can contain the name of the factory class. The implementation class can implement the DirObjectFactory interface or the ObjectFactory interface.
- *Object Factory* - The ObjectFactory interface is necessary to ensure class space consistency.
- *Dir Object Factory* - If the Object Factory provider can accept the additional Attributes argument in the getObjectInstance method of the JNDI Provider Admin service than it must also register as a Dir Object Factory service.

126.5.5 URL Context Provider

A *URL Context Factory* is a special type of an Object Factory service. A URL Context Factory must be registered as an Object Factory service with the following service property:

- osgi.jndi.url.scheme - The URL scheme associated with this URL Context, for example acme. The scheme must not contain the colon (':' \u003A).

A URL Context is used for URL based operations on an initial Context. For example, a lookup to acme:foo/javax.sql.DataSource must not use the provider based lookup mechanism of the backing Context but instead causes a lookup for the requested URL Context. A URL Context also provides a secondary mechanism for restoring Reference objects.

When an initial Context returned by the JNDI Context Manager service is given a URL based operation, it searches in the service registry for an Object Factory service that is published with the URL scheme property that matches the scheme used from the lookup request.

It then calls the getInstance method on the Object Factory service with the following parameters:

- *Object* - Should be either a String, String[], or null.
- *Name* - must be null
- *Context* - must be null

- *Hashtable* - The environment properties.

Calling the getInstance method must return a Context object. This context is then used to perform the lookup.

The life cycle of the Object Factory used to create the URL Context is tied to the JNDI context that was used to perform the URL based JNDI operation. By the time JNDI context is closed any Object-Factory objects held to process the URL lookups must be released (unget).

126.5.6 JRE Context Providers

The Java Runtime Environment (JRE) defines a number of default naming providers., see *Built-In JNDI Providers* on page 502. These naming providers are not OSGi aware, but are commonly used and are provided by the JRE. These naming providers rely on the NamingManager class for object conversion and finding URL Contexts.

The JRE default providers are made available by the JNDI Implementation. This JNDI Implementation must register a *built-in* Initial Context Factory Builder service that is capable of loading any InitialContextFactory classes of the JRE providers.

When this built-in Initial Context Factory Builder is called to create an InitialContextFactory object it must look in the environment properties that were given as an argument and extract the java.naming.factory.initial property; this property contains the name of the class of a provider. The built-in Initial Context Factory Builder then must use the bootstrap class loader to load the given InitialContextFactory class and creates a new instance with the no arguments constructor and return it. If this fails, it must return null. This mechanism will allow loading of any built-in providers.

This built-in Initial Context Factory Builder service must be registered with no service.ranking property. This will give it the default ranking and allows other providers to override the default.

126.6 OSGi URL Scheme

A URL scheme is available that allows JNDI based applications to access services in the service registry, see *Services and State* on page 510 about restrictions on these services. The URL scheme is specified as follows:

```
service   ::= 'osgi:service/' query
query     ::= jndi-name |  qname ( '/' filter )?
jndi-name ::= <any string>
```

No spaces are allowed between the terms.

This OSGi URL scheme can be used to perform a lookup of a single matching service using the interface name and filter. The URL Context must use the *owning bundle* to perform the service queries. The owning bundle is the bundle that requested the initial Context from the JNDI Context Manager service or received its Context through the InitialContext class. The returned objects must not be incompatible with the class space of the owning bundle.

The lookup for a URL with the osgi: scheme and service path returns the service with highest service.ranking and the lowest service.id. This scheme only allows a single service to be found. Multiple services can be obtained with the osgi: scheme and servicelist path:

```
servicelist ::= 'osgi:servicelist/' query?
```

If this osgi:servicelist scheme is used from a lookup method then a Context object is returned instead of a service object. Calling the listBindings method will produce a NamingEnumeration object that provides Binding objects. A Binding object contains the name, class of the service, and the service object. The bound object is the service object contained in the given Context.

When the `Context` class list method is called, the Naming Enumeration object provides a `NameClassPair` object. This `NameClassPair` object will include the name and class of each service in the Context. The `list` method can be useful in cases where a client wishes to iterate over the available services without actually getting them. If the service itself is required, then `listBindings` method should be used.

If multiple services matched the criteria listed in the URL, there would be more than one service available in the Context, and the corresponding Naming Enumeration would contain the same number of services.

If multiple services match, a call to `listBindings` on this Context would return a list of bindings whose name are a string with the `service.id` number, for example:

`1283`

Thus the following lookup is valid:

`osgi:servicelist/javax.sql.DataSource/(&(db=mydb)(version=3.1))`

A service can provide a *JNDI service name* if it provides the following service property:

- `osgi.jndi.service.name` - An alternative name that the service can be looked up by when the `osgi:` URL scheme is used.

If a service is published with a JNDI service name then the service matches any URL that has this service name in the place of `interface`. For example, if the JNDI service name is foo, then the following URL selects this service:

`osgi:service/foo`

Using a JNDI service name that can be interpreted as an interface name must be avoided, if this happens the result is undefined.

A JNDI client can also obtain the Bundle Context of the owning bundle by using the `osgi:` scheme namespace with the `framework/bundleContext` name. The following URL must return the Bundle Context of the owning bundle:

`osgi:framework/bundleContext`

After the `NamingEnumeration` object has been used it must be closed by the client. Implementations must then unget any gotten services or perform other cleanup.

126.6.1   Service Proxies

The OSGi URL Context handles the complexities by hiding the dynamic nature of OSGi. The OSGi URL Context must handle the dynamics by *proxying* the service objects. This proxy must implement the interface given in the URL. If the JNDI service name instead of a class name is used, then all interfaces under which the service is registered must be implemented. If an interface is not compatible with the owning bundle's class space then it must not be implemented on the proxy, it must then be ignored. If this results in no implemented interfaces then an Illegal Argument Exception must be thrown.

Interfaces can always be proxied but classes are much harder. For this reason, an implementation is free to throw an Illegal Argument Exception when a class is used in the URL or in one of the registration names.

Getting the actual service object can be delayed until the proxy is actually used to call a method. If a method is called and the actual service has been unregistered, then the OSGi URL Context must attempt to rebind it to another service that matches the criteria given in the URL the next time it is

called. When no alternative service is available, a Service Exception with the UNREGISTERED type code must be thrown. Services obtained with the osgi: URL scheme must therefore be stateless because the rebinding to alternative services is not visible to the caller; there are no listeners defined for this rebinding, see *Services and State* on page 510.

If the reference was looked up using osgi:servicelist then proxies must still be used, however, these proxies must not rebind when their underlying service is unregistered. Instead, they must throw a Service Exception with the UNREGISTERED type whenever the proxy is used and the proxied service is no longer available.

126.6.2   Services and State

A service obtained through a URL Context lookup is proxied. During the usage of this service, the JNDI Implementation can be forced to transparently rebind this service to another instance. The JNDI specification is largely intended for portability. For this reason, it has no mechanism architected to receive notifications about this rebinding. The client code is therefore unable to handle the dynamics.

The consequence of this model is that stateful services require extra care because applications cannot rely on the fact that they always communicate with the same service. Virtually all OSGi specified services have state.

126.7   Traditional Client Model

A JNDI Implementation must at startup register the InitialContextFactoryBuilder object and the ObjectFactoryBuilder object with the NamingManager class. As described in *JNDI Overview* on page 500, the JNDI code in the JRE will then delegate all Context related requests to the JNDI Implementation. Setting these singletons allows code that is not aware of the OSGi framework to use Context implementations from JNDI Providers registered with the OSGi service registry and that are managed as bundles. The JNDI Implementation therefore acts as a broker to the service registry for OSGi unaware code.

This brokering role can only be played when the JNDI Implementation can set the singletons as specified in *Naming Manager Singletons* on page 501. If the JNDI Implementation cannot set these singletons then it should log an error with the Log Service, if available. It can then not perform the following sections.

126.7.1   New Initial Context

The client typically requests a Context using the following code:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
InitialContext ctx = new InitialContext(env);
```

The created InitialContext object is a facade for the real Context that is requested by the caller. It provides the bootstrapping mechanism for JNDI Provider pluggability. In order to obtain the provider's Context, the InitialContext class makes a call to the static getContext method on the NamingManager class. The JNDI code in the JRE then delegates any request for an initial Context object to the JNDI Implementation through the registered InitialContextFactoryBuilder singleton. The JNDI Implementation then determines the Bundle Context of the caller as described in *Caller's Bundle Context* on page 511. If no such Bundle Context can be found, a No Initial Context Exception is thrown to the caller. This Bundle Context must be from an ACTIVE bundle.

This Bundle Context is then used to get the JNDI Context Manager service. This service is then used as described in *Context Creation* on page 503 to get an initial Context. This initial Context is then used in the InitialContext object as the *default initial context*. In this specification this is normally

called the backing context. An InitialContext object constructed through an Initial Context Factory Builder will not use the URL lookup mechanism, it must delegate all operations to the its backing context. A Context obtained through the JNDI Context Manager provides the URL lookup behavior instead.

126.7.2   Static Conversion

JNDI provides a general object conversion facility that is used by the URL Context and the process of restoring an object from a Reference object, see *Object and Reference Conversion* on page 501. A JNDI Implementation must take over this conversion by setting the static Object Factory Builder singleton, see *Naming Manager Singletons* on page 501. Non-OSGi aware Context implementations will use the NamingManager static getObjectInstance method for object conversion. This method then delegates to the set singleton Object Factory Builder to obtain an ObjectFactory object that understands how to convert the given description to an object. The JNDI Implementation must return an Object Factory that understands the OSGi service registry. If the getObjectInstance method is called on this object it must use the same rules as defined for the JNDI Provider Admin service getObjectInstance(Object,javax.naming.Name,javax.naming.Context,Map) method, see *JNDI Provider Admin service* on page 505. The Bundle Context that must be used with respect to this service is the caller's Bundle Context, see *Caller's Bundle Context* on page 511. If the Bundle Context is not found, the description object must be returned. The calling bundle must not be required to import the org.osgi.service.jndi package.

126.7.3   Caller's Bundle Context

The following mechanisms are used to determine the callers Bundle Context:

1. Look in the JNDI environment properties for a property called

    osgi.service.jndi.bundleContext

    If a value for this property exists then use it as the Bundle Context. If the Bundle Context has been found stop.
2. Obtain the Thread Context Class Loader; if it, or an ancestor class loader, implements the BundleReference interface, call its getBundle method to get the client's Bundle; then call getBundleContext on the Bundle object to get the client's Bundle Context. If the Bundle Context has been found stop.
3. Walk the call stack until the invoker is found. The invoker can be the caller of the InitialContext class constructor or the NamingManager or DirectoryManager getObjectInstance methods.
    *   Get the class loader of the caller and see if it, or an ancestor, implements the BundleReference interface.
    *   If a Class Loader implementing the BundleReference interface is found call the getBundle method to get the clients Bundle; then call the getBundleContext method on the Bundle to get the clients Bundle Context.
    *   If the Bundle Context has been found stop, else continue with the next stack frame.

126.7.4   Life Cycle Mismatch

The use of static access to the JNDI mechanisms, NamingManager and InitialContext class methods, in the traditional client programming model produces several problems with regard to the OSGi life cycle. The primary problem being that there is no dependency management in place when static methods are used. These problems do not exist for the JNDI Context Manager service. Therefore, OSGi applications are strongly encouraged to use the JNDI Context Manager service.

The traditional programming model approach relies on two JVM singletons in the Naming Manager, see *Naming Manager Singletons* on page 501. The JNDI Implementation bundle must set both singletons before it registers its JNDI Context Manager service and JNDI Provider Admin service. However, in OSGi there is no defined start ordering, primarily because bundles can be updated at

any moment in time and will at such time not be available to provide their function anyway. For this reason, OSGi bundles express their dependencies with services.

The lack of start ordering means that a bundle could create an InitialContext object before the JNDI Implementation has had the chance to set the static Initial Context Factory Builder singleton. This means that the JNDI implementation inside the JRE will provide its default behavior and likely have to throw an exception. A similar exception is thrown for the Object Factory Builder singleton.

There is a also a (small) possibility that a client will call new InitialContext() after the singletons have been set, but before the JNDI Context Manager and JNDI Provider Admin services have been registered. This specification requires that these services are set after the singletons are set. In this race condition the JNDI Implementation should throw a No Initial Context Exception, explaining that the JNDI services are not available yet.

126.8 Security

126.8.1 JNDI Implementation

A JNDI Implementation may wish to assert that the user of the provider has some relevant Java 2 security permission. Since the JNDI implementation is an intermediary between the JNDI client and provider this means that the JNDI implementation needs to have any permissions required to access any JNDI Provider. As a result the JNDI implementation needs All Permission. This will result in the JNDI clients permissions being checked to see if it has the relevant permission to access the JNDI Provider.

The JNDI Implementation must make any invocation to access these services in a doPriviledged check. A JNDI client must therefore not be required to have the following permissions, which are needed by a JNDI Implementation:

```
ServicePermission ..ObjectFactory                 REGISTER,GET
ServicePermission ..DirObjectFactory              REGISTER,GET
ServicePermission ..ObjectFactoryBuilder          REGISTER,GET
ServicePermission ..InitialContextFactory         REGISTER,GET
ServicePermission ..InitialContextFactoryBuilder  REGISTER,GET
ServicePermission ..JNDIProviderAdmin             REGISTER,GET
```

The JNDI Implementation bundle must have the appropriate permissions to install the InitialContextFactoryBuilder and ObjectFactoryBuilder instances using the appropriate methods on the NamingManager class. This requires the following permission:

```
RuntimePermission "setFactory"
```

126.8.2 JNDI Clients

A JNDI client using the JNDI Context Manager service must have the following permissions:

```
ServicePermission ..JNDIContextManager            GET
```

Obtaining a reference to a JNDI Context Manager service should be considered a privileged operation and should be guarded by permissions.

126.8.3 OSGi URL namespace

A JNDI client must not be able to obtain services or a Bundle Context that the client bundle would not be able to get via the core OSGi API. To allow a client to use the osgi namespace to get a service the bundle must have the corresponding Service Permission. When using the osgi namespace to obtain the Bundle Context the client bundle must have Admin Permission for the Bundle Context.

These permissions must be enforced by the osgi URL namespace handler. If there is no proper permission, the implementation must throw a Name Not Found Exception to prevent exposing the existence of such services.

126.9 org.osgi.service.jndi

JNDI Package Version 1.0.

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

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.jndi; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.jndi; version="[1.0,1.1)"

126.9.1 Summary

- JNDIConstants - Constants for the JNDI implementation.
- JNDIContextManager - This interface defines the OSGi service interface for the JNDIContextManager.
- JNDIProviderAdmin - This interface defines the OSGi service interface for the JNDIProviderAdmin service.

126.9.2 public class JNDIConstants

Constants for the JNDI implementation.

Concurrency Immutable

126.9.2.1 public static final String BUNDLE_CONTEXT = "osgi.service.jndi.bundleContext"

This JNDI environment property can be used by a JNDI client to indicate the caller's BundleContext. This property can be set and passed to an InitialContext constructor. This property is only useful in the "traditional" mode of JNDI.

126.9.2.2 public static final String JNDI_SERVICENAME = "osgi.jndi.service.name"

This service property is set on an OSGi service to provide a name that can be used to locate the service other than the service interface name.

126.9.2.3 public static final String JNDI_URLSCHEME = "osgi.jndi.url.scheme"

This service property is set by JNDI Providers that publish URL Context Factories as OSGi Services. The value of this property should be the URL scheme that is supported by the published service.

126.9.3 public interface JNDIContextManager

This interface defines the OSGi service interface for the JNDIContextManager. This service provides the ability to create new JNDI Context instances without relying on the InitialContext constructor.

Concurrency Thread-safe

126.9.3.1 public Context newInitialContext() throws NamingException

☐ Creates a new JNDI initial context with the default JNDI environment properties.

Returns   an instance of javax.naming.Context

Throws   NamingException– upon any error that occurs during context creation

126.9.3.2   **public Context newInitialContext(Map environment) throws NamingException**

environment   JNDI environment properties specified by caller

☐   Creates a new JNDI initial context with the specified JNDI environment properties.

Returns   an instance of javax.naming.Context

Throws   NamingException– upon any error that occurs during context creation

126.9.3.3   **public DirContext newInitialDirContext() throws NamingException**

☐   Creates a new initial DirContext with the default JNDI environment properties.

Returns   an instance of javax.naming.directory.DirContext

Throws   NamingException– upon any error that occurs during context creation

126.9.3.4   **public DirContext newInitialDirContext(Map environment) throws NamingException**

environment   JNDI environment properties specified by the caller

☐   Creates a new initial DirContext with the specified JNDI environment properties.

Returns   an instance of javax.naming.directory.DirContext

Throws   NamingException– upon any error that occurs during context creation

126.9.4   public interface JNDIProviderAdmin

This interface defines the OSGi service interface for the JNDIProviderAdmin service. This service provides the ability to resolve JNDI References in a dynamic fashion that does not require calls to NamingManager.getObjectInstance(). The methods of this service provide similar reference resolution, but rely on the OSGi Service Registry in order to find ObjectFactory instances that can convert a Reference to an Object. This service will typically be used by OSGi-aware JNDI Service Providers.

Concurrency   Thread-safe

126.9.4.1   **public Object getObjectInstance(Object refInfo,Name name,Context context,Map environment) throws Exception**

refInfo   Reference info

name   the JNDI name associated with this reference

context   the JNDI context associated with this reference

environment   the JNDI environment associated with this JNDI context

☐   Resolve the object from the given reference.

Returns   an Object based on the reference passed in, or the original reference object if the reference could not be resolved.

Throws   Exception– in the event that an error occurs while attempting to resolve the JNDI reference.

126.9.4.2   **public Object getObjectInstance(Object refInfo,Name name,Context context,Map environment,Attributes attributes) throws Exception**

refInfo   Reference info

name   the JNDI name associated with this reference

context   the JNDI context associated with this reference

environment   the JNDI environment associated with this JNDI context

attributes   the naming attributes to use when resolving this object

▫ Resolve the object from the given reference.

Returns   an Object based on the reference passed in, or the original reference object if the reference could not be resolved.

Throws   Exception– in the event that an error occurs while attempting to resolve the JNDI reference.

126.10   References

[1]   *Java Naming and Directory Interface*
http://docs.oracle.com/javase/6/docs/technotes/guides/jndi/index.html

[2]   *Java Naming and Directory Interface Tutorial from Sun Microsystems*
http://download.oracle.com/javase/6/docs/technotes/guides/jndi/index.html

[3]   *JNDI Standard Property Names*
http://download.oracle.com/javase/1.5.0/docs/api/javax/naming/Context.html

127 JPA Service Specification

Version 1.0

127.1 Introduction

The Java Persistence API (JPA) is a specification that sets a standard for persistently storing objects in enterprise and non-enterprise Java based environments. JPA provides an Object Relational Mapping (ORM) model that is configured through persistence descriptors. This Java Persistence Service specification defines how persistence units can be published in an OSGi framework, how client bundles can find these persistence units, how database drivers are found with the OSGi JDBC Specification, as well as how JPA providers can be made available within an OSGi framework.

Applications can be managed or they can be unmanaged. Managed applications run inside a Java EE Container and unmanaged applications run in a Java SE environment. The managed case requires a provider interface that can be used by the container, while in the unmanaged case the JPA provider is responsible for supporting the client directly. This specification is about the unmanaged model of JPA except in the areas where the managed model is explicitly mentioned. Additionally, multiple concurrent providers for the unmanaged case are not supported.

This JPA Specification supports both [1] *JPA 1.0* and [2] *JPA 2.0*.

127.1.1 Essentials

- *Dependencies* - There must be a way for persistence clients, if they so require, to manage their dependencies on a compatible persistence unit.
- *Compatibility* - The Persistence Unit service must be able to function in non-managed mode according to existing standards and interfaces outlined in the JPA specification.
- *Modularity* - Persistent classes and their accompanying configuration can exist in a separate bundle from the client that is operating on them using the Persistence Unit service.
- *JDBC* - Leverage the OSGi JDBC Specification for access to the database.
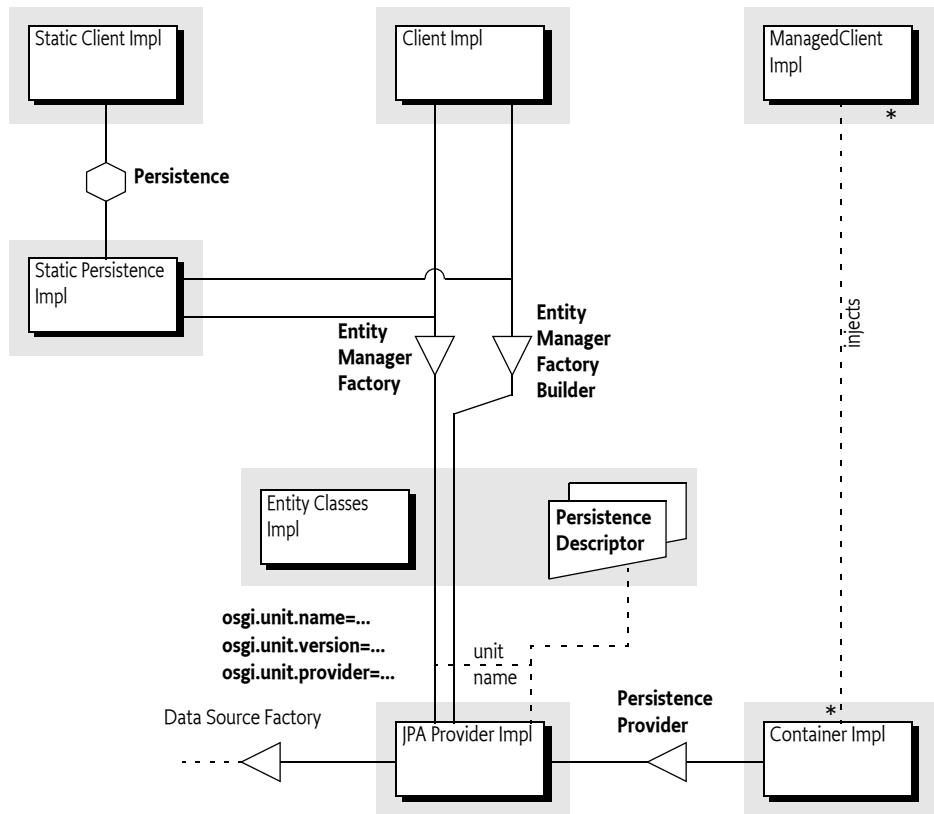
127.1.2 Entities

- *JPA* - The Java Persistence API, [1] *JPA 1.0* and [2] *JPA 2.0*.
- *JPA Provider* - An implementation of JPA, providing the Persistence Provider and JPA Services to Java EE Containers and Client Bundles.
- *Interface Bundle* - A bundle containing the interfaces and classes in the `javax.persistence` namespace (and its sub-namespaces) that are defined by the JPA specification.
- *Persistence Bundle* - A bundle that includes, a Meta-Persistence header, one or more Persistence Descriptor resources, and the entity classes specified by the Persistence Units in those resources.
- *Client Bundle* - The bundle that uses the Persistence Bundle to retrieve and store objects.
- *Persistence Descriptor* - A resource describing one or more Persistence Units.
- *Persistence Unit* - A named configuration for the object-relational mappings and database access as defined in a Persistence Descriptor.
- *Entity Manager* - The interface that provides the control point of retrieving and persisting objects in a relational database based on a single Persistence Unit for a single session.

- *Entity Manager Factory* - A service that can create Entity Managers based on a Persistence Unit for different sessions.
- *Entity Manager Factory Builder* - A service that can build an Entity Manager Factory for a specific Persistence Unit with extra configuration parameters.
- *Managed Client* - A Client Bundle that is managed by a Container
- *Static Client* - A Client that uses the static factory methods in the Persistence class instead of services.
- *Static Persistence* - The actor that enables the use of the Persistence class static factory methods to obtain an Entity Manager Factory.
- *JDBC Provider* - The bundle providing a Data Source Factory service.

Figure 127.1   *JPA Service overview*

127.1.3   Dependencies

This specification is based on JPA 1.0 and JPA 2.0. JPA 2.0 is backward compatible with JPA 1.0. For this reason, the versions of the packages follow the OSGi recommended version policy with the addition of a special JPA marker that annotates the specification version for JPA. All JPA Packages must also have an attribute called jpa that specifies the JPA version. The purpose of this attribute is to make it clear what JPA version belongs to this package.

Table 127.1   *Dependency versions*

JPA	Packages	Export Version	Client Import Range	Provider Imp. Range
JPA 1.0	javax.persistence	1.0	[1.0,2.0)	[1.0,1.1)
	javax.persistence.spi	1.0	[1.0,2.0)	[1.0,1.1)

JPA	Packages	Export Version	Client Import Range	Provider Imp. Range
JPA 2.0	javax.persistence	1.1	[1.1,2.0)	[1.1,1.2)
	javax.persistence.spi	1.1	[1.1,2.0)	[1.1,1.2)

For example, JPA should have an export declaration like:

```
Export-Package: javax.persistence; version=1.1; jpa=2.0, ...
```

127.1.4   Synopsis

A JPA Provider tracks Persistence Bundles; a Persistence Bundle contains a Meta-Persistence manifest header. This manifest header enumerates the Persistence Descriptor resources in the Persistence Bundle. Each resource's XML schema is defined by the JPA 1.0 or JPA 2.0 specification. The JPA Provider reads the resource accordingly and extracts the information for one or more Persistence Units. For each found Persistence Unit, the JPA Provider registers an Entity Manager Factory Builder service. If the database is defined in the Persistence Unit, then the JPA Provider registers an Entity Manager Factory service during the availability of the corresponding Data Source Factory.

The identification of these services is handled through a number of service properties. The Entity Manager Factory service is named by the standard JPA interface, the Builder version is OSGi specific; it is used when the Client Bundle needs to create an Entity Manager Factory based on configuration properties.

A Client Bundle that wants to persist or retrieve its entity classes depends on an Entity Manager Factory (Builder) service that corresponds to a Persistence Unit that lists the entity classes. If such a service is available, the client can use this service to get an Entity Manager, allowing the client to retrieve and persist objects as long as the originating Entity Manager Factory (Builder) service is registered.

In a non-OSGi environment, it is customary to get an Entity Manager Factory through the Persistence class. This Persistence class provides a number of static methods that give access to any locally available JPA providers. This approach is not recommended in an OSGi environment due to class loading and start ordering issues. However, OSGi environments can support access through this static factory with a Static Persistence bundle.
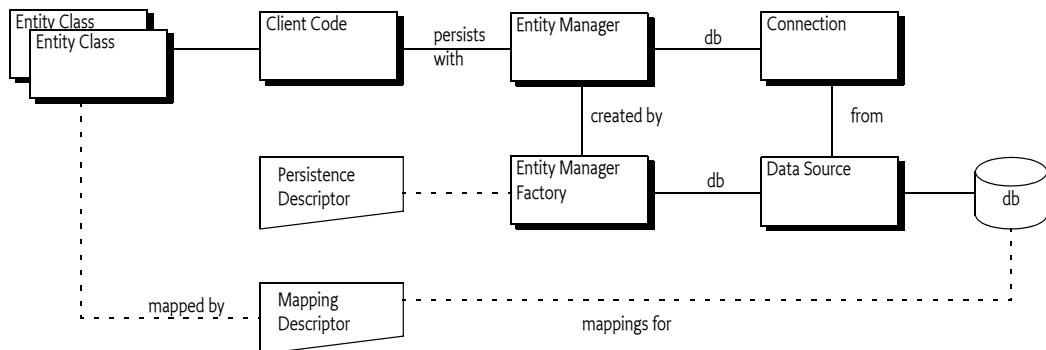
127.2   JPA Overview

Java Persistence API (JPA) is a specification that is part of [3] *Java EE 5*. This OSGi Specification is based on [1] *JPA 1.0* and [2] *JPA 2.0*. This section provides an overview of JPA as specified in the JCP. The purpose of this section is to introduce the concepts behind JPA and define the terminology that will be used in the remainder of the chapter.

The purpose of JPA is to simplify access to relational databases for applications on the object-oriented Java platform. JPA provides support for storing and retrieving objects in a relational database. The JPA specification defines in detail how objects are mapped to tables and columns under the full control of the application. The core classes involved are depicted in Figure 127.2.

Figure 127.2   *JPA Client View*

The JPA specifications define a number of concepts that are defined in this section for the purpose of this OSGi specification. However, the full syntax and semantics are defined in the JPA specifications.

127.2.1   Persistence

Classes that are stored and retrieved through JPA are called the *entity classes*. In this specification, the concept of entity classes includes the *embeddable* classes, which are classes that do not have any persistent identity, and mapped super classes that allow mappings, but are not themselves persistent. Entity classes are not required to implement any interface or extend a specific superclass, they are Plain Old Java Objects (POJOs). It is the responsibility of the *JPA Provider* to connect to a database and map the store and retrieve operations of the entity classes to their tables and columns. For performance reasons, the entity classes are sometimes *enhanced*. This enhancement can take place during build time, deploy time, or during class loading time. Some enhancements use byte code weaving, some enhancements are based on sub-classing.

The JPA Provider cannot automatically perform its persistence tasks; it requires configuration information. This configuration information is stored in the *Persistence Descriptor*. A Persistence Descriptor is an XML file according of one of the two following namespaces:

```
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd
```

The JPA standard Persistence Descriptor must be stored in META-INF/persistence.xml. It is usually in the same class path entry (like a JAR or directory) as the entity classes.

The JPA Provider parses the Persistence Descriptor and extracts one or more *Persistence Units*. A Persistence Unit includes the following aspects:

*   *Name* - Every Persistence Unit must have a name to identify it to clients. For example: Accounting.
*   *Provider Selection* - Restriction to a specific JPA Provider, usually because there are dependencies in the application code on provider specific functionality.
*   *JDBC Driver Selection* - Selects the JDBC driver, the principal and the credentials for selecting and accessing a relational database. See *JDBC Access in JPA* on page 522.
*   *Properties* - Standard and JPA Provider specific properties.
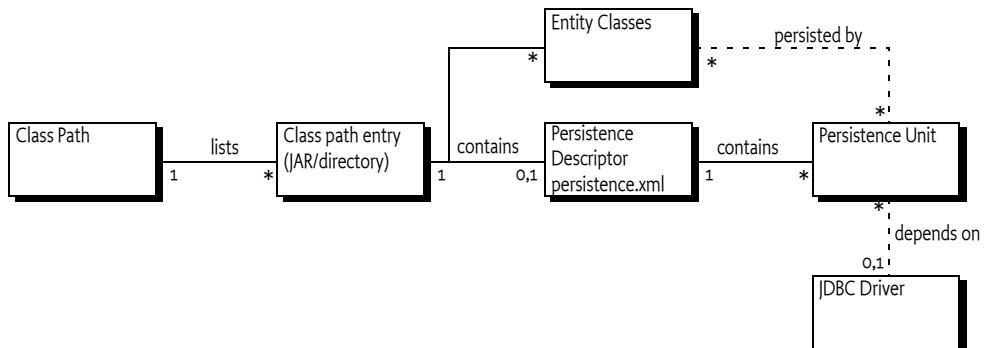
The object-relational mappings are stored in special mapping resources or are specified in annotations.

A Persistence Unit can be *complete* or *incomplete*. A complete Persistence Unit identifies the database driver that is needed for the Persistence Unit, though it does not have to contain the credentials. An incomplete Persistence Unit lacks this information.

The relations between the class path, its entries, the entity classes, the Persistence Descriptor and the Persistence Unit is depicted in Figure 127.3 on page 521.

Figure 127.3   *JPA Configuration*

JPA recognizes the concept of a *persistence root*. The persistence root is the root of the JAR (or directory) on the class path that contains the META-INF/persistence.xml resource.

127.2.2   JPA Provider

The JPA specifications provide support for multiple JPA Providers in the same application. An Application selects a JPA Provider through the Persistence class, using static factory methods. One of these methods accepts a map with *configuration properties*. Configuration properties can override information specified in a Persistence Unit or these properties add new information to the Persistence Unit.

The default implementation of the Persistence class discovers providers through the Java EE services model, this model requires a text resource in the class path entry called:

    META-INF/services/javax.persistence.PersistenceProvider

This text resource contains the name of the JPA Provider implementation class.

The Persistence class createEntityManagerFactory method provides the JPA Provider with the name of a Persistence Unit. The JPA Provider must then scan the class path for any META-INF/persistence.xml entries, these are the available Persistence Descriptors. It then extracts the Persistence Units to find the requested Persistence Unit. If no such Persistence Unit can be found, or the JPA Provider is restricted from servicing this Persistence Unit, then null is returned. The Persistence class will then continue to try the next found or registered JPA Provider.

A Persistence Unit can restrict JPA Providers by specifying a *JPA Provider class*, this introduces a *provider dependency*. The specified JPA Provider class must implement the PersistenceProvider interface. This *implementation class name* must be available from the JPA Provider's documentation. JPA Providers that do not own the specified JPA Provider class must ignore such a Persistence Unit.

Otherwise, if the Persistence Unit is not restricted, the JPA Provider is *assigned* to this Persistence Unit; it must be ready to provide an EntityManagerFactory object when the application requests one.

The JPA Provider uses the Persistence Unit, together with any additional configuration properties, to construct an *Entity Manager Factory*. The application then uses this Entity Manager Factory to construct an *Entity Manager*, optionally providing additional configuration properties. The Entity Manager then provides the operations for the application to store and retrieve entity classes from the database.
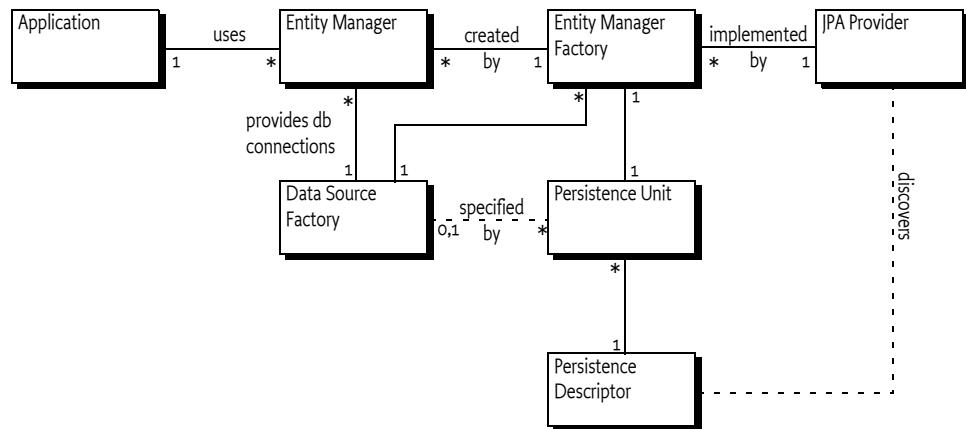
The additional configuration properties provided with the creation of the Entity Manager Factory or the Entity Manager are often used to specify the database driver and the credentials. This allows the

Persistence Unit to be specified without committing to a specific database, leaving the choice to the application at runtime.

The relations between the application, Entity Manager, Entity Manager Factory and the JPA Provider are depicted in Figure 127.4 on page 522.

Figure 127.4   *JPA Dynamic Model*

127.2.3   Managed and Unmanaged

The JPA specifications make a distinction between a *managed* and an *unmanaged* mode. In the managed mode the presence of a Java EE Container is assumed. Such a container provides many services for its contained applications like transaction handling, dependency injection, etc. One of these aspects can be the interface to the relational database. The JPA specifications therefore have defined a special method for Java EE Containers to manage the persistence aspects of their Managed Clients. This method is the createContainerEntityManagerFactory method on the PersistenceProvider interface. This method is purely intended for Java EE Containers and should not be used in other environments.

The other method on the PersistenceProvider interface is intended to be used by the Persistence class static factory methods. The Persistence class searches for an appropriate JPA Provider by asking all available JPA Providers to create an Entity Manager Factory based on configuration properties. The first JPA Provider that is capable of providing an Entity Manager Factory wins. The use of these static factory methods is called the *unmanaged mode*. It requires a JPA Provider to scan the class path to find the assigned Persistence Units.

127.2.4   JDBC Access in JPA

A Persistence Unit is configured to work with a relational database. JPA Providers communicate with a relational database through compliant JDBC database drivers. The database and driver parameters are specified in the Persistence Unit or configured during Entity Manager Factory or Entity Manager creation with the configuration properties. The configuration properties for selecting a database in non-managed mode were proprietary in JPA 1.0 but have been standardized in version 2.0 of JPA:

- javax.persistence.jdbc.driver - Fully-qualified name of the driver class
- javax.persistence.jdbc.url - Driver-specific URL to indicate database information
- javax.persistence.jdbc.user - User name to use when obtaining connections
- javax.persistence.jdbc.password - Password to use when obtaining connections

127.3   Bundles with Persistence

The primary goal of this specification is to simplify the programming model for bundles that need persistence. In this specification there are two application roles:

- *Persistence Bundle* - A Persistence Bundle contains the entity classes and one or more Persistence Descriptors, each providing one or more Persistence Units.
- *Client Bundle* - A Client Bundle contains the code that manipulates the entity classes and uses an Entity Manager to store and retrieve these entity classes with a relational database. The Client Bundle obtains the required Entity Manager(s) via a service based model.

These roles can be combined in a single bundle.

127.3.1   Services

A JPA Provider uses Persistence Units to provide Client Bundles with a configured *Entity Manager Factory* service and/or an *Entity Manager Factory Builder* service for each assigned Persistence Unit:

- *Entity Manager Factory service* - Provides an `EntityManagerFactory` object that depends on a complete Persistence Unit. That is, it is associated with a registered Data Source Factory service.
- *Entity Manager Factory Builder service* - The Entity Manager Factory Builder service provides the capability of creating an `EntityManagerFactory` object with additional configuration properties.

These services are collectively called the *JPA Services*. Entity Managers obtained from such JPA Services can only be used to operate on entity classes associated with their corresponding Persistence Unit.

127.3.2   Persistence Bundle

A *Persistence Bundle* is a bundle that specifies the Meta-Persistence header, see *Meta Persistence Header* on page 526. This header refers to one or more Persistence Descriptors in the Persistence Bundle. Commonly, this is the `META-INF/persistence.xml` resource. This location is the standard for non-OSGi environments, however an OSGi bundle can also use other locations as well as multiple resources.

For example, the contents of a simple Persistence Bundle with a single `Person` entity class could look like:

```
META-INF/
META-INF/MANIFEST.MF
OSGI-INF/address.xml
com/acme/Person.class
```

The corresponding manifest would then look like:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Meta-Persistence: OSGI-INF/address.xml
Bundle-SymbolicName: com.acme.simple.persistence
Bundle-Version: 3.2.4.200912231004
```

A Persistence Bundle is a normal bundle; it must follow all the rules of OSGi and can use all OSGi constructs like Bundle-Classpath, fragment bundles, import packages, export packages, etc. However, there is one limitation: any entity classes must originate in the bundle's JAR, it cannot come from a fragment. This requirement is necessary to simplify enhancing entity classes.

127.3.3   Client Bundles

A Client Bundle uses the entity classes from a Persistence Bundle to provide its required functionality. To store and retrieve these entity classes a Client Bundle requires an Entity Manager that is configured for the corresponding Persistence Unit.

An Entity Manager is intended to be used by a single session, it is not thread safe. Therefore, a client needs an Entity Manager Factory to create an Entity Manager. In an OSGi environment, there are multiple routes to obtain an Entity Manager Factory.

A JPA Provider must register an Entity Manager Factory service for each assigned Persistence Unit that is *complete*. Complete means that it is a configured Persistence Unit, including the reference to the relational database. The Entity Manager Factory service is therefore bound to a Data Source Factory service and Client Bundles should not attempt to rebind the Data Source Factory with the configuration properties of the createEntityManager(Map) method. See *Rebinding* on page 530 for the consequences. If the Data Source Factory must be bound by the Client Bundle then the Client Bundle should use the *Custom Configured Entity Manager* on page 524.

The Entity Manager Factory service must be registered with the service properties as defined in *Service Registrations* on page 527. These are:

- osgi.unit.name - (String) The name of the Persistence Unit
- osgi.unit.version - (String) The version of the associated Persistence Bundle
- osgi.unit.provider - (String) The implementation class name of the JPA Provider

The life cycle of the Entity Manager Factory service is bound to the Persistence Bundle, the JPA Provider, and the selected Data Source Factory service.

A Client Bundle that wants to use an Entity Manager Factory service should therefore use an appropriate filter to select the Entity Manager Factory service that corresponds to its required Persistence Unit. For example, the following snippet uses Declarative Services, see *Declarative Services Specification* on page 175, to statically depend on such a service:

```
<reference name="accounting"
   target="(&amp;(osgi.unit.name=Accounting)(osgi.unit.version=3.2.*))"
    interface="javax.persistence.EntityManagerFactory"/>
```

127.3.4   Custom Configured Entity Manager

If a Client Bundle needs to provide configuration properties for the creation of an Entity Manager Factory it should use the *Entity Manager Factory Builder* service. This can for example be used to provide the database selection properties when the Persistence Unit is incomplete or if the database selection needs to be overridden.

The Entity Manager Factory Builder service's life cycle must not depend on the availability of any Data Source Factory, even if a JDBC driver class name is specified in the Persistence Descriptor. The Entity Manager Factory Builder service is registered with the same service properties as the corresponding Entity Factory service, see *Service Registrations* on page 527.

The following method is defined on the EntityManagerFactoryBuilder interface:

- createEntityManagerFactory(Map) - Returns a custom configured EntityManagerFactory instance for the Persistence Unit associated with the service. Accepts a map with the configuration properties to be applied during Entity Manager Factory creation. The method must return a proper Entity Manager Factory or throw an Exception.

The createEntityManagerFactory method allows standard and vendor-specific properties to be passed in and applied to the Entity Manager Factory being created. However, some properties cannot be honored by the aforementioned method. For example, the javax.persistence.provider JPA prop-

erty, as a means to specify a specific JPA Provider at runtime, cannot be supported because the JPA Provider has already been decided; it is the JPA Provider that registered the Entity Manager Factory Builder service. A JPA Provider should throw an Exception if it recognizes the property but it cannot use the property when specified through the builder. Unrecognized properties must be ignored.

Once an Entity Manager Factory is created the specified Data Source becomes associated with the Entity Manager Factory. It is therefore not possible to re-associate an Entity Manager Factory with another Data Source by providing different properties. A JPA Provider must throw an Exception when an attempt is made to re-specify the database properties. See *Rebinding* on page 530 for further information.

As an example, a sample snippet of a client that wants to operate on a persistence unit named Accounting and pass in the JDBC user name and password properties is:

```
ServiceReference[] refs = context.getServiceReferences(
    EntityManagerFactoryBuilder.class.getName(),
    "(osgi.unit.name=Accounting)");
if ( refs != null ) {
    EntityManagerFactoryBuilder emfBuilder =
        (EntityManagerFactoryBuilder) context.getService(refs[0]);
    if ( emfBuilder != null ) {
      Map<String,Object> props = new HashMap<String,Object>();
      props.put("javax.persistence.jdbc.user", userString);
      props.put("javax.persistence.jdbc.password",passwordString);
      EntityManagerFactory emf = emfBuilder.createEntityManagerFactory(props);
      EntityManager em = emf.createEntityManager();
     ...
}
```

The example does not handle the dynamic dependencies on the associated Data Source Factory service.

127.4 Extending a Persistence Bundle

A Persistence Bundle is identified by its Meta-Persistence manifest header that references a number of Persistence Descriptor resources. Persistence bundles must be detected by a JPA Provider. The JPA Provider must parse any Persistence Descriptors in these bundles and detect the assigned Persistence Units. For each assigned Persistence Unit, the JPA Provider must register an Entity Manager Factory Builder service when the Persistence Bundle is ready, see *Ready Phase* on page 527.

For complete and assigned Persistence Units, the JPA Provider must find the required Data Source Factory service based on the driver name. When the Persistence Bundle is ready and the selected Data Source Factory is available, the JPA Provider must have an Entity Manager Factory service registered that is linked to that Data Source Factory.

When the Persistence Bundle is stopped (or the JPA Provider stops), the JPA Provider must close all connections and cleanup any resources associated with the Persistence Bundle.

This process is outlined in detail in the following sections.

127.4.1 Class Space Consistency

A JPA Provider must ignore Persistence Bundles that are in another class space for the javax.persistence.* packages. Such a JPA Provider cannot create JPA Services that would be visible and usable by the Client Bundles.

127.4.2 Meta Persistence Header

A *Persistence Bundle* is a bundle that contains the Meta-Persistence header. If this header is not present, then this specification does not apply and a JPA Provider should ignore the corresponding bundle.

The persistence root of a Persistence Unit is the root of the Persistence Bundle's JAR

The Meta-Persistence header has a syntax of:

```
Meta-Persistence ::= ( jar-path ( ',' jar-path)* )?
jar-path         ::= path ( '!/' spath )?
spath            ::= path   // must not start with solidus ('/' \u002F)
```

The header may include zero or more comma-separated jar-paths, each a path to a Persistence Descriptor resource in the bundle. Paths may optionally be prefixed with the solidus ('/' \u002F) character. The JPA Provider must always include the META-INF/persistence.xml first if it is not one of the listed paths. Wildcards in directories are not supported. The META-INF/persistence.xml is therefore the default location for an empty header.

For example:

```
Meta-Persistence: META-INF/jpa.xml, persistence/jpa.xml
```

The previous example will instruct the JPA Provider to process the META-INF/persistence.xml resource first, even though it is not explicitly listed. The JPA Provider must then subsequently process META-INF/jpa.xml and the persistence/jpa.xml resources.

The paths in the Meta-Persistence header must be used with the Bundle.getEntry() method, or a mechanism with similar semantics, to obtain the corresponding resource. The getEntry method does not force the bundle to resolve when still unresolved; resolving might interfere with the efficiency of any required entity class enhancements. However, the use of the getEntry method implies that fragment bundles cannot be used to contain Persistence Descriptors nor entity classes.

Paths in the Meta-Persistence header can reference JAR files that are nested in the bundle by using the !/ jar: URL syntax to separate the JAR file from the path within the JAR, for example:

```
Meta-Persistence: embedded.jar!/META-INF/persistence.xml
```

This example refers to a resource in the embedded.jar resource, located in the META-INF directory of embedded.jar.

The !/ splits the jar-path in a prefix and a suffix:

- *Prefix* - The prefix is a path to a JAR resource in the bundle.
- *Suffix* - The suffix is a path to a resource in the JAR identified by the prefix.

For example:

```
embedded.jar!/META-INF/persistence.xml
prefix:     embedded.jar
suffix:     META-INF/persistence.xml
```

It is not required that all listed or implied resources are present in the bundle's JAR. For example, it is valid that the default META-INF/persistence.xml resource is absent. However, if no Persistence Units are found at all then the absence of any Persistence Unit is regarded as an error that should be logged. In this case, the Persistence Bundle is further ignored.

127.4.3 Processing

A JPA Provider can detect a Persistence Bundle as early as its installation time. This early detection allows the JPA Provider to validate the Persistence Bundle as well as prepare any mechanisms to en-

hance the classes for better performance. However, this process can also be delayed until the bundle is started.

The JPA Provider must validate the Persistence Bundle. A valid Persistence Bundle must:

- Have no parsing errors of the Persistence Descriptors
- Validate all Persistence Descriptors against their schemas
- Have at least one assigned Persistence Unit
- Have all entity classes mentioned in the assigned Persistence Units on the Persistence Bundle's JAR.

A Persistence Bundle that uses multiple providers for its Persistence Units could become incompatible with future versions of this specification.

If any validation fails, then this is an error and should be logged. Such a bundle is ignored completely even if it also contains valid assigned Persistence Units. Only a bundle update can recover from this state.

Persistence Units can restrict JPA Providers by specifying a provider dependency. JPA Providers that do not own this JPA Provider implementation class must ignore such a Persistence Unit completely. Otherwise, if the JPA Provider can service a Persistence Unit, it assigns itself to this Persistence Unit.

If after the processing of all Persistence Descriptors, the JPA Provider has no assigned Persistence Units, then the JPA Provider must further ignore the Persistence Bundle.

127.4.4   Ready Phase

A Persistence Bundle is *ready* when its state is ACTIVE or, when a lazy activation policy is used, STARTING. A JPA Provider must track the ready state of Persistence Bundles that contain assigned Persistence Units.

While a Persistence Bundle is ready, the JPA Provider must have, for each assigned Persistence Unit, an Entity Manager Factory Builder service registered to allow Client Bundles to create new Entity-ManagerFactory objects. The JPA Provider must also register an Entity Manager Factory for each assigned and complete Persistence Unit that has its corresponding Data Source available in the service registry.

The service registration process is asynchronous with the Persistence Bundle start because a JPA Provider could start after a Persistence Bundle became ready.

127.4.5   Service Registrations

The JPA Services must be registered through the Bundle Context of the corresponding Persistence Bundle to ensure proper class space consistency checks by the OSGi Framework.

JPA Services are always related to an assigned Persistence Unit. To identify this Persistence Unit and the assigned JPA Provider, each JPA Service must have the following service properties:

- osgi.unit.name - (String) The name of the Persistence Unit. This property corresponds to the name attribute of the persistence-unit element in the Persistence Descriptor. It is used by Client Bundles as the primary filter criterion to obtain a JPA Service for a required Persistence Unit. There can be multiple JPA Services registered under the same osgi.unit.name, each representing a different version of the Persistence Unit.
- osgi.unit.version - (String) The version of the Persistence Bundle, as specified in Bundle-Version header, that provides the corresponding Persistence Unit. Client Bundles can filter their required JPA Services based on a particular Persistence Unit version.
- osgi.unit.provider - (String) The JPA Provider implementation class name that registered the service. The osgi.unit.provider property allows Client Bundles to know the JPA Provider that is servicing the Persistence Unit. Client Bundles should be careful when filtering on this proper-

ty, however, since the JPA Provider that is assigned a Persistence Unit may not be known by the Client Bundle ahead of time. If there is a JPA Provider dependency, it is better to specify this dependency in the Persistence Unit because other JPA Providers are then not allowed to assign such a Persistence Unit and will therefore not register a service.

127.4.6 Registering the Entity Manager Factory Builder Service

Once the Persistence Bundle is ready, a JPA Provider must register an Entity Manager Factory Builder service for each assigned Persistence Unit from that Persistence Bundle.

The Entity Manager Factory Builder service must be registered with the service properties listed in *Service Registrations* on page 527. The Entity Manager Factory Builder service is registered under the org.osgi.service.jpa.EntityManagerFactoryBuilder name. This interface is using the JPA packages and is therefore bound to one of the two supported versions, see *Dependencies* on page 518.

The Entity Manager Factory Builder service enables the creation of a parameterized version of an Entity Factory Manager by allowing the caller to specify configuration properties. This approach is necessary if, for example, the Persistence Unit is not complete.

127.4.7 Registering the Entity Manager Factory

A complete Persistence Unit is configured with a specific relational database driver, see *JDBC Access in JPA* on page 522. A JPA Provider must have an Entity Manager Factory service registered for each assigned and complete Persistence Unit when:

- The originating Persistence Bundle is ready, and
- A *matching* Data Source Factory service is available. Matching a Data Source Factory service to a Persistence Unit is discussed in *Database Access* on page 529.

A JPA Provider must track the life cycle of the matching Data Source Factory service; while this service is unavailable the Entity Manager Factory service must also be unavailable. Any active Entity Managers created by the Entity Manager Factory service become invalid to use at that time.

The Entity Manager Factory service must be registered with the same service properties as described for the Entity Manager Factory Builder service, see *Service Registrations* on page 527. It should be registered under the following name:

```
javax.persistence.EntityManagerFactory
```

The EntityManagerFactory interface is from the JPA packages and is therefore bound to one of the two supported versions, see *Dependencies* on page 518.

An Entity Manager Factory is bound to a Data Source Factory service because its assigned Persistence Unit was complete. However, a Client Bundle could still provide JDBC configuration properties for the createEntityManager(Map) method. This not always possible, see *Rebinding* on page 530.

127.4.8 Stopping

If a Persistence Bundle is being stopped, then the JPA Provider must ensure that any resources allocated on behalf of the Persistence Bundle are cleaned up and all open connections are closed. This cleanup must happen synchronously with the STOPPING event. Any Exceptions being thrown while cleaning up should be logged but must not stop any further clean up.

If the JPA Provider is being stopped, the JPA Provider must unregister all JPA Services that it registered through the Persistence Bundles and clean up as if those bundles were stopped.

127.5   JPA Provider

JPA Providers supply the implementation of the JPA Services and the Persistence Provider service. It is the responsibility of a JPA Provider to store and retrieve the entity classes from a relational database. It is the responsibility of the JPA Provider to register a Persistence Provider and start tracking Persistence Bundles, see *Extending a Persistence Bundle* on page 525.

127.5.1   Managed Model

A JPA Provider that supports running in managed mode should register a specific service for the Java EE Containers: the Persistence Provider service. The interface is the standard JPA Persistence-Provider interface. See *Dependencies* on page 518 for the issues around the multiple versions that this specification supports.

The service must be registered with the following service property:

* javax.persistence.provider - The JPA Provider implementation class name, a documented name for all JPA Providers.

The Persistence Provider service enables a Java EE Container to find a particular JPA Provider. This service is intended for containers only, not for Client Bundles because there are implicit assumptions in the JPA Providers about the Java EE environment. A Java EE Container must obey the life cycle of the Persistence Provider service. If this service is unregistered then it must close all connections and clean up the corresponding resources.

127.5.2   Database Access

A Persistence Unit is configured to work with a relational database. JPA Providers must communicate with a relational database through a compliant JDBC database driver. The database and driver parameters are specified with properties in the Persistence Unit or the configuration properties when a Entity Manager Factory Builder is used to build an Entity Manager Factory. All JPA Providers, regardless of version, in an OSGi environment must support the following properties for database access:

* javax.persistence.jdbc.driver - Fully-qualified name of the driver class.
* javax.persistence.jdbc.url - Driver-specific URL to indicate database information
* javax.persistence.jdbc.user - User name to use when obtaining connections
* javax.persistence.jdbc.password - Password to use when obtaining connections

There are severe limitations in specifying these properties after the Entity Manager Factory is created for the first time, see *Rebinding* on page 530.

127.5.3   Data Source Factory Service Matching

Providers must use the javax.persistence.jdbc.driver property, as defined in *JDBC Access in JPA* on page 522, to obtain a Data Source Factory service. The Data Source Factory is specified in *JDBC™ Service Specification* on page 489. The javax.persistence.jdbc.driver property must be matched with the value of the Data Source Factory service property named osgi.jdbc.driver.class.

The Data Source Factory service is registered with the osgi.jdbc.driver.class service property that holds the class name of the driver. This property must match the javax.persistence.jdbc.driver service property of the Persistence Unit.

For example, if the Persistence Unit specifies the com.acme.db.Driver database driver in the javax.persistence.jdbc.driver property (or in the Persistence Descriptor property element), then the following filter would select an appropriate Data Source Factory:

```
(&(objectClass=org.osgi.service.jdbc.DataSourceFactory)
    (osgi.jdbc.driver.class=com.acme.db.Driver))
```

Once the Data Source Factory is obtained, the JPA Provider must obtain a `DataSource` object. This Data Source object must then be used for all relational database access.

In [1] *JPA 1.0* the JPA JDBC properties were not standardized. JPA Providers typically defined a set of JDBC properties, similar to those defined in JPA 2.0, to configure JDBC driver access. JPA 1.0 JPA Providers must look up the Data Source Factory service first using the JPA 2.0 JDBC properties. If these properties are not defined then they should fall back to their proprietary driver properties.

127.5.4   Rebinding

In this specification, the Entity Manager Factory service is only registered when the Persistence Unit is complete and a matching Data Source Factory service is available. However, the API of the Entity Manager Factory Builder allows the creation of an Entity Manager Factory with configuration properties. Those configuration properties could contain the JDBC properties to bind to another Data Source Factory service than it had already selected.

This case must not be supported by a JPA Provider, an Illegal Argument Exception must be thrown. If such a case would be supported then the life cycle of the Entity Manager Factory service would still be bound to the first Data Source Factory. There would be no way for the JPA Provider to signal to the Client Bundle that the returned Entity Manager Factory is no longer valid because the rebound Data Source Factory was unregistered.

Therefore, when an Entity Manager Factory is being created using the Entity Manager Factory Builder, a JPA Provider must verify that the new properties are compatible with the properties of the already created Entity Manager Factory. If no, then an Exception must be thrown. If they are compatible, then an instance of the previous Entity Manager Factory should be returned.

127.5.5   Enhancing Entity Classes

JPA Providers may choose to implement the JPA specifications using various implementation approaches and techniques. This promotes innovation in the area, but also opens the door to limitations and constraints arising due to implementation choices. For example, there are JPA Providers that perform byte code weaving during the entity class loading. Dynamic byte code weaving requires that the entity classes are not loaded until the JPA Provider is first able to intercept the loading of the entity class and be given an opportunity to do its weaving. It also implies that the Persistence Bundle and any other bundles that import packages from that bundle must be refreshed if the JPA Provider needs to be changed.

This is necessary because the JPA Services are registered against the Bundle Contexts of the Persistence Bundles and not the Bundle Context of the JPA Providers. Client Bundles must then unget the service to unbind themselves from the uninstalled JPA Provider. However, since most JPA Providers perform some kind of weaving or class transformation on the entity classes, the Persistence Bundle will likely need to be refreshed. This will cause the Client Bundles to be refreshed also because they depend on the packages of the entity classes.

127.5.6   Class Loading

JPA Providers cannot have package dependencies on entity classes in Persistence Bundles because they cannot know at install time what Persistence Bundles they will be servicing. However, when a JPA Provider is servicing a Persistence Bundle, it must be able to load classes and resources from that Persistence Bundle according to the OSGi bundle rules. To do this class loading it must obtain a class loader that has the same visibility as the Persistence Bundle's bundle class loader. This will also allow it to load and manage metadata for the entity classes and resources for that Persistence Bundle's assigned Persistence Units. These resources and entity classes must reside directly in the Persistence Bundle, they must be accessed using the `getEntry` method. Entity classes and resources must not reside in fragments.

127.5.7 Validation

There is not yet an OSGi service specification defined for validation providers. If validation is required, the validation implementation will need to be included with the JPA Provider bundle.

127.6 Static Access

Non-managed client usage of JPA has traditionally been achieved through the `Persistence` class. Invoking a static method on the `Persistence` class is a dependency on the returned JPA Provider that cannot be managed by the OSGi framework.

However, such an unmanaged dependency is supported in this specification by the Static Persistence bundle. This bundle provides backwards compatibility for programs that use existing JPA access patterns. However, usage of this static model requires that the deployer ensures that the actors needed are in place at the appropriate times by controlling the life cycles of all participating bundles. The normal OSGi safe-guards and dependency handling do not work in the case of static access.

A Static Persistence Bundle must provide static access from the `Persistence` class to the JPA Services.

127.6.1 Access

There are two methods on the `Persistence` class:

- `createEntityManagerFactory(String)`
- `createEntityManagerFactory(String,Map)`

Both methods take the name of a Persistence Unit. The last method also takes a map that contains extra configuration properties. To support the usage of the static methods on the `Persistence` class, the implementation of the `Persistence.createEntityManagerFactory` method family must do a lookup of one of the JPA Services associated with the selected Persistence Unit.

If no configuration properties are specified, the Static Persistence Bundle must look for an Entity Manager Factory service with the `osgi.unit.name` property set to the given name. The default service should be used because no selector for a version is provided. If no such service is available, `null` must be returned. Provisioning of multiple versioned Persistence Units is not supported. Deployers should ensure only a single version of a Persistence Unit with the same name is present in an OSGi framework at any moment in time.

Otherwise, if configuration properties are provided, the Static Access implementation must look for an Entity Manager Factory Builder service with the `osgi.unit.name` property set to the given Persistence Unit name. If no such service exists, `null` must be returned. Otherwise, the default service must be used to create an Entity Manager Factory with the given configuration properties. The result must be returned to the caller.

For service lookups, the Static Persistence Bundle must use its own Bundle Context, it must not attempt to use the Bundle Context of the caller. All exceptions should be passed to the caller.

The class space of the Entity Manager Factory and the class space of the client cannot be enforced to be consistent by the framework because it is the `Persistence` class that is doing the lookup of the service, and not the actual calling Client Bundle that will be using the Entity Manager Factory. The framework cannot make the connection and therefore cannot enforce that the class spaces correspond. Deployers should therefore ensure that the involved class spaces are correctly wired.

127.7 Security

The security for this specification is based on the JPA specification.

127.8 org.osgi.service.jpa

JPA Package Version 1.0.

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

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.jpa; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.jpa; version="[1.0,1.1)"

127.8.1 Summary

- `EntityManagerFactoryBuilder` - This service interface offers JPA clients the ability to create instances of EntityManagerFactory for a given named persistence unit.

127.8.2 public interface EntityManagerFactoryBuilder

This service interface offers JPA clients the ability to create instances of EntityManagerFactory for a given named persistence unit. A service instance will be created for each named persistence unit and can be filtered by comparing the value of the osgi.unit.name property containing the persistence unit name. This service is used specifically when the caller wants to pass in factory-scoped properties as arguments. If no properties are being used in the creation of the EntityManagerFactory then the basic EntityManagerFactory service should be used.

127.8.2.1 public static final String JPA_UNIT_NAME = "osgi.unit.name"

The name of the persistence unit.

127.8.2.2 public static final String JPA_UNIT_PROVIDER = "osgi.unit.provider"

The class name of the provider that registered the service and implements the JPA javax.persistence.PersistenceProvider interface.

127.8.2.3 public static final String JPA_UNIT_VERSION = "osgi.unit.version"

The version of the persistence unit bundle.

127.8.2.4 public EntityManagerFactory createEntityManagerFactory(Map<String,Object> props)

props   Properties to be used, in addition to those in the persistence descriptor, for configuring the EntityManagerFactory for the persistence unit.

□   Return an EntityManagerFactory instance configured according to the properties defined in the corresponding persistence descriptor, as well as the properties passed into the method.

Returns   An EntityManagerFactory for the persistence unit associated with this service. Must not be null.

127.9 References

[1]   *JPA 1.0*
      http://jcp.org/en/jsr/summary?id=220

[2]   *JPA 2.0*
      http://jcp.org/en/jsr/summary?id=317

[3]   *Java EE 5*
        http://www.oracle.com/technetwork/java/javaee/tech/index.html

128   Web Applications Specification

Version 1.0

128.1   Introduction

The Java EE Servlet model has provided the backbone of web based applications written in Java. Given the popularity of the Servlet model, it is desirable to provide a seamless experience for deploying existing and new web applications to Servlet containers operating on the OSGi framework. Previously, the Http Service in the catalogue of OSGi compendium services was the only model specified in OSGi to support the Servlet programming model. However, the Http Service, as defined in that specification, is focused on the run time, as well as manual construction of the servlet context, and thus does not actually support the standard Servlet packaging and deployment model based on the Web Application Archive, or WAR format.

This specification defines the Web Application Bundle, which is a bundle that performs the same role as the WAR in Java EE. A WAB uses the OSGi life cycle and class/resource loading rules instead of the standard Java EE environment. WABs are normal bundles and can leverage the full set of features of the OSGi framework.

Web applications can also be installed as traditional WARs through a manifest rewriting process. During the install, a WAR is transformed into a WAB. This specification was based on ideas developed in [5] *PAX Web Extender*.

This Web Application Specification provides support for web applications written to the Servlet 2.5 specification, or later. Given that Java Server Pages, or JSPs, are an integral part of the Java EE web application framework, this specification also supports the JSP 2.1 specification or greater if present. This specification details how a web application packaged as a WAR may be installed into an OSGi framework, as well as how this application may interact with, and obtain, OSGi services.

128.1.1   Essentials

- *Extender* - Enable the configuration of components inside a bundle based on configuration data provided by the bundle developer.
- *Services* - Enable the use of OSGi services within a Web Application.
- *Deployment* - Define a mechanism to deploy Web Applications, both OSGi aware and non OSGi aware, in the OSGi environment.
- *WAR File Support* - Transparently enhance the contents of a WAR's manifest during installation to add any headers necessary to deploy a WAR as an OSGi bundle.
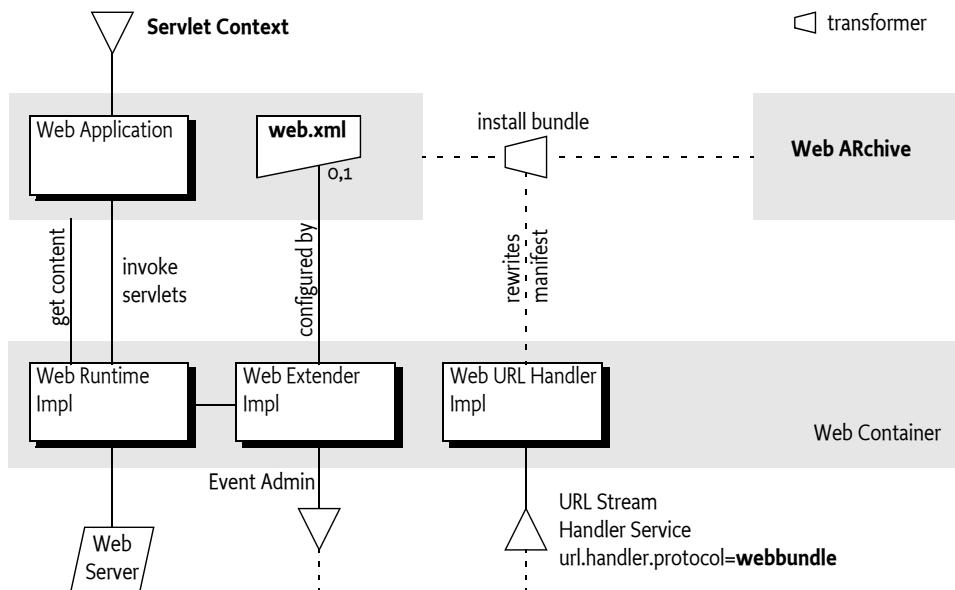
128.1.2   Entities

- *Web Container* - The implementation of this specification. Consists of a Web Extender, a Web URL Handler and a Servlet and Java Server Pages Web Runtime environment.
- *Web Application* - A program that has web accessible content. A Web Application is defined by [2] *Java EE Web Applications*.
- *Web Application Archive (WAR)* - The Java EE standard resource format layout of a JAR file that contains a deployable Web Application.
- *Web Application Bundle* - A Web Application deployed as an OSGi bundle, also called a WAB.
- *WAB* - The acronym for a Web Application Bundle.

- *Web Extender* - An extender bundle that deploys the Web Application Bundle to the Web Run-time based on the Web Application Bundle's state.
- *Web URL Handler* - A URL handler which transforms a Web Application Archive (WAR) to conform to the OSGi specifications during installation by installing the WAR through a special URL so that it becomes a Web Application Bundle.
- *Web Runtime* - A Java Server Pages and Servlet environment, receiving the web requests and translating them to servlet calls, either from Web Application servlets or other classes.
- *Web Component* - A Servlet or Java Server Page (JSP).
- *Servlet* - An object implementing the Servlet interface; this is for the request handler model in the Servlet Specification.
- *Servlet Context* - The model representing the Web Application in the Servlet Specification.
- *Java Server Page (JSP)* - A declarative, template based model for generating content through Servlets that is optionally supported by the Web Runtime.
- *Context Path* - The URI path prefix of any content accessible in a Web Application.

Figure 128.1   *Web Container Entities*

128.1.3   Dependencies

The package dependencies for the clients of this specification are listed in the following table.

Table 128.1   *Dependency versions*

Packages	Export Version	Client Import Range
javax.servlet	2.5	[2.5,3.0)
javax.servlet.http	2.5	[2.5,3.0)
javax.servlet.jsp.el	2.1	[2.1,3.0)
javax.servlet.jsp.jstl.core	1.2	[1.2,2.0)
javax.servlet.jsp.jstl.fmt	1.2	[1.2,2.0)
javax.servlet.jsp.jstl.sql	1.2	[1.2,2.0)
javax.servlet.jsp.jstl.tlv	1.2	[1.2,2.0)
javax.servlet.jsp.resources	2.1	[2.1,3.0)

Packages	Export Version	Client Import Range
javax.servlet.jsp.tagext	2.1	[2.1,3.0)
javax.servlet.jsp	2.1	[2.1,3.0)

JSP is optional for the Web Runtime.

128.1.4 Synopsis

The Web Application Specification is composed of a number of cooperating parts, which are implemented by a *Web Container*. A Web Container consists of:

- *Web Extender* - Responsible for deploying Web Application Bundles (WAB) to a Web Runtime,
- *Web Runtime* - Provides support for Servlet and optionally for JSPs, and
- *Web URL Handler* - Provides on-the-fly enhancements of non-OSGi aware Web ARchives (WAR) so that they can be installed as a WAB.

WABs are standard OSGi bundles with additional headers in the manifest that serve as deployment instructions to the Web Extender. WABs can also contain the Java EE defined web.xml descriptor in the WEB-INF/ directory. When the Web Extender detects that a WAB is ready the Web Extender deploys the WAB to the Web Runtime using information contained in the web.xml descriptor and the appropriate manifest headers. The Bundle Context of the WAB is made available as a Servlet Context attribute. From that point, the Web Runtime will use the information in the WAB to serve content to any requests. Both dynamic as well as static content can be provided.

The Web URL Handler allows the deployment of an unmodified WAR as a WAB into the OSGi framework. This Web URL Handler provides a URL stream handler with the webbundle: scheme. Installing a WAR with this scheme allows the Web URL Handler to interpose itself as a filter on the input stream of the contents of the WAR, transforming the contents of the WAR into a WAB. The Web URL Handler rewrites the manifest by adding necessary headers to turn the WAR into a valid WAB. Additional headers can be added to the manifest that serve as instructions to the Web Extender.

After a WAB has been deployed to the Web Runtime, the Web Application can interact with the OSGi framework via the provided Bundle Context. The Servlet Context associated with this WAB follows the same life cycle as the WAB. That is, when the underlying Web Application Bundle is started, the Web Application is deployed to the Web Runtime. When the underlying Web Application Bundle is stopped because of a failure or other reason, the Web Application is undeployed from the Web Run-time.

128.2 Web Container

A Web Container is the implementation of this specification. It consists of the following parts:

- *Web Extender* - Detects Web Application Bundles (WAB) and tracks their life cycle. Ready WABs are deployed to the Web Runtime.
- *Web Runtime* - A runtime environment for a Web Application that supports the [3] *Servlet 2.5 specification* and [4] *JSP 2.1 specification* or later. The Web Runtime receives web requests and calls the appropriate methods on servlets. Servlets can be implemented by classes or Java Server Pages.
- *Web URL Handler* - A URL stream handler providing the webbundle: scheme. This scheme can be used to install WARs in an OSGi framework. The Web URL Handler will then automatically add the required OSGi manifest headers.

The extender, runtime, and handler can all be implemented in the same or different bundles and use unspecified mechanisms to communicate. This specification uses the defined names of the subparts as the actor; the term Web Container is the general name for this collection of actors.

128.3   Web Application Bundle
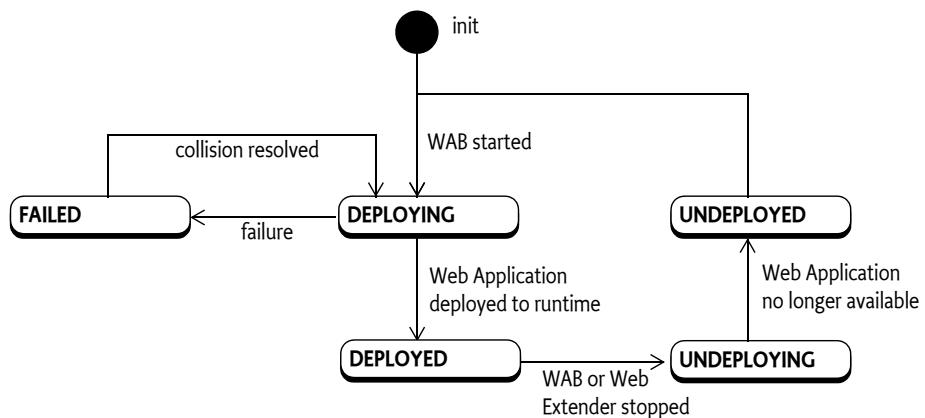
Bundles are the deployment and management entities under OSGi. A *Web Application Bundle* (WAB) is deployed as an OSGi bundle in an OSGi framework, where each WAB provides a single *Web Application*. A Web Application can make use of the [3] *Servlet 2.5 specification* and [4] *JSP 2.1 specification* programming models, or later, to provide content for the web.

A WAB is defined as a normal OSGi bundle that contains web accessible content, both static and dynamic. There are no restrictions on bundles. A Web Application can be packaged as a WAB during application development, or it can be transparently created at bundle install time from a standard Web Application aRchive (WAR) via transformation by the Web URL Handler, see *Web URL Handler* on page 542.

A WAB is a valid OSGi bundle and as such must fully describe its dependencies and exports (if any). As Web Applications are modularized further into multiple bundles (and not deployed as WAR files only) it is possible that a WAB can have dependencies on other bundles.

A WAB may be installed into the framework using the BundleContext.installBundle methods. Once installed, a WAB's life cycle is managed just like any other bundle in the framework. This life cycle is tracked by the Web Extender who will then deploy the Web Application to the Web Runtime when the WAB is ready and will undeploy it when the WAB is no longer ready. This state is depicted in Figure 128.2.

Figure 128.2   *State diagram Web Application*

128.3.1   WAB Definition

A WAB is differentiated from non Web Application bundles through the specification of the additional manifest header:

```
Web-ContextPath ::= path
```

The Web-ContextPath header specifies the value of the *Context Path* of the Web Application. All web accessible content of the Web Application is available on the web server relative to this Context Path. For example, if the context path is /sales, then the URL would be something like: http://www.acme.com/sales. The Context Path must always begin with a solidus ('/' \u002F).

The Web Extender must not recognize a bundle as a Web Application unless the Web-ContextPath header is present in its manifest and the header value is a valid path for the bundle.

A WAB can optionally contain a web.xml resource to specify additional configuration. This web.xml must be found with the Bundle findEntries method at the path:

```
WEB-INF/web.xml
```

The findEntries method includes fragments, allowing the web.xml to be provided by a fragment. The Web Extender must fully support a web.xml descriptor that specifies Servlets, Filters, or Listeners whose classes are required by the WAB.

128.3.2 Starting the Web Application Bundle

A WAB's Web Application must be *deployed* while the WAB is *ready*. Deployed means that the Web Application is available for web requests. Once deployed, a WAB can serve its web content on the given Context Path. Ready is when the WAB:

• Is in the ACTIVE state, or
• Has a lazy activation policy and is in the STARTING state.

The Web Extender should ensure that serving static content from the WAB does not activate the WAB when it has a lazy activation policy.

To deploy the WAB, the Web Extender must initiate the deploying of the Web Application into a Web Runtime. This is outlined in the following steps:

1. Wait for the WAB to become ready. The following steps can take place asynchronously with the starting of the WAB.
2. Post an org/osgi/service/web/DEPLOYING event. See *Events* on page 545.
3. Validate that the Web-ContextPath manifest header does not match the Context Path of any other currently deployed web application. If the Context Path value is already in use by another Web Application, then the Web Application must not be deployed, and the deployment fails, see *Failure* on page 540. The Web Extender should log the collision. If the prior Web Application with the same Context Path is undeployed later, this Web Application should be considered as a candidate, see *Stopping the Web Application Bundle* on page 541.
4. The Web Runtime processes deployment information by processing the web.xml descriptor, if present. The Web Container must perform the necessary initialization of Web Components in the WAB as described in the [3] *Servlet 2.5 specification.* This involves the following sub-steps in the given order:
   • Create a Servlet Context for the Web Application.
   • Instantiate configured Servlet event listeners.
   • Instantiate configured application filter instances etc.

   The Web Runtime is required to complete instantiation of listeners prior to the start of execution of the first request into the Web Application by the Web Runtime. Attribute changes to the Servlet Context and Http Session objects can occur concurrently. The Servlet Container is not required to synchronize the resulting notifications to attribute listener classes. Listener classes that maintain state are responsible for the integrity of the data and should handle this case explicitly.

   If event listeners or filters are used in the web.xml, then the Web Runtime will load the corresponding classes from the bundle activating the bundle if it was lazily started. Such a configuration will therefore not act lazily.

5. Publish the Servlet Context as a service with identifying service properties, see *Publishing the Servlet Context* on page 540.
6. Post an org/osgi/service/web/DEPLOYED event to indicate that the web application is now available. See *Events* on page 545.

If at any moment before the org/osgi/service/web/DEPLOYED event is published the deployment of the WAB fails, then the WAB deployment fails, see *Failure* on page 540.

128.3.3   Failure

Any validation failures must prevent the Web Application from being accessible via HTTP, and must result in a `org/osgi/service/web/FAILED` event being posted. See *Events* on page 545. The situation after the failure must be as if the WAB was never deployed.

128.3.4   Publishing the Servlet Context

To help management agents with tracking the state of Web Applications, the Web Extender must register the Servlet Context of the WAB as a service, using the Bundle Context of the WAB. The Servlet Context service must be registered with the service properties listed in the following table.

Table 128.2   *Servlet Context Service Properties*

Property Name	Type	Description
osgi.web.symbolicname	String	The symbolic name for the Web Application Bundle
osgi.web.version	String	The version of the Web Application Bundle. If no Bundle-Version is specified in the manifest then this property must not be set.
osgi.web.contextpath	String	The Context Path from which the WAB's content will be served.

128.3.5   Static Content

A deployed WAB provides content on requests from the web. For certain access paths, this can serve content from the resources of the web application: this is called *static content*. A Web Runtime must use the Servlet Context resource access methods to service static content, the resource loading strategy for these methods is based on the `findEntries` method, see *Resource Lookup* on page 546. For confidentiality reasons, a Web Runtime must not return any static content for paths that start with one of the following prefixes:

```
WEB-INF/
OSGI-INF/
META-INF/
OSGI-OPT/
```

These *protected directories* are intended to shield code content used for dynamic content generation from accidentally being served over the web, which is a potential attack route. In the servlet specification, the WEB-INF/ directory in the WAR is protected in such a way. However, this protection is not complete. A dependent JAR can actually be placed outside the WEB-INF directory that can then be served as static content. The same is true for a WAB. Though the protected directories must never be served over the web, there are no other checks required to verify that no content can be served that is also available from the Bundle class path.

It is the responsibility of the author of the WAB to ensure that confidential information remains confidential by placing it in one of the protected directories. WAB bundles should be constructed in such a way that they do not accidentally expose code or confidential information. The simplest way to achieve this is to follow the WAR model where code is placed in the WEB-INF/classes directory and this directory is placed on the Bundle's class path as the first entry. For example:

```
Bundle-ClassPath: WEB-INF/classes, WEB-INF/lib/acme.jar
```

128.3.6   Dynamic Content

Dynamic content is content that uses code to generate the content, for example a servlet. This code must be loaded from the bundle with the Bundle `loadClass` method, following all the Bundle class path rules.

Unlike a WAR, a WAB is not constrained to package classes and code resources in the WEB-INF/ classes directory or dependent JARs in WEB-INF/lib/ only. These entries can be packaged in any way that's valid for an OSGi bundle as long as such directories and JARs are part of bundle class path as set with the Bundle-ClassPath header and any attached fragments. JARs that are specified in the Bundle-ClassPath header are treated like JARs in the WEB-INF/lib/ directory of the Servlet specification. Similarly, any directory that is part of the Bundle-ClassPath header is treated like WEB-INF/classes directory of the Servlet specification.

Like WARs, code content that is placed outside the protected directories can be served up to clients as static content.

128.3.7 Content Serving Example

This example consists of a WAB with the following contents:

```
acme.jar:
    Bundle-ClassPath: WEB-INF/classes, LIB/bar.jar
    Web-ContextPath: /acme

    WEB-INF/lib/foo.jar
    LIB/bar.jar
    index.html
    favicon.ico
```

The content of the embedded JARs foo.jar and bar.jar is:

```
foo.jar:                  bar.jar:
    META-INF/foo.tld          META-INF/bar.tld
    foo/FooTag.class          bar/BarTag.class
```

Assuming there are no special rules in place then the following lists specifies the result of a number of web requests for static content:

```
/acme/index.html          acme.wab:index.html
/acme/favicon.ico         acme.wab:favicon.ico
/acme/WEB-INF/lib/foo.jar not found because protecteddirectory
/acme/LIB/bar.jar         acme.wab:LIB/bar.jar   (code, but not protected)
```

In this example, the tag classes in bar.jar must be found (if JSP is supported) but the tag classes in foo.jar must not because foo.jar is not part of the bundle class path.

128.3.8 Stopping the Web Application Bundle

A web application is stopped by stopping the corresponding WAB. In response to a WAB STOPPING event, the Web Extender must *undeploy* the corresponding Web Application from the Servlet Container and clean up any resources. This undeploying must occur synchronously with the WAB's stopping event. This will involve the following steps:

1. An org/osgi/service/web/UNDEPLOYING event is posted to signal that a Web Application will be removed. See *Events* on page 545.
2. Unregister the corresponding Servlet Context service
3. The Web Runtime must stop serving content from the Web Application.
4. The Web Runtime must clean up any Web Application specific resources as per servlet 2.5 specification.
5. Emit an org/osgi/service/web/UNDEPLOYED event. See *Events* on page 545.
6. It is possible that there are one or more *colliding* WABs because they had the same Context Path as this stopped WAB. If such colliding WABs exists then the Web Extender must attempt to deploy the colliding WAB with the lowest bundle id.

Any failure during undeploying should be logged but must not stop the cleaning up of resources and notification of (other) listeners as well as handling any collisions.

128.3.9 Uninstalling the Web Application Bundle

A web application can be uninstalled by uninstalling the corresponding WAB. The WAB will be uninstalled from the OSGi framework.

128.3.10 Stopping of the Web Extender

When the Web Extender is stopped all deployed WABs are undeployed as described in *Stopping the Web Application Bundle* on page 541. Although the WAB is undeployed it remains in the ACTIVE state. When the Web Extender leaves the STOPPING state all WABs will have been undeployed.
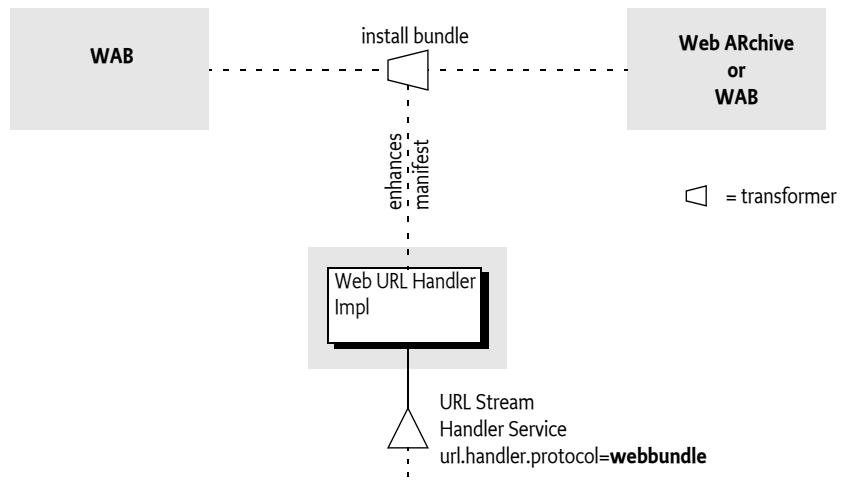
128.4 Web URL Handler

The Web URL Handler acts as a filter on the Input Stream of an install operation. It receives the WAB or WAR and it then generates a JAR that conforms to the WAB specification by rewriting the manifest resource. This process is depicted in Figure 128.3.

Figure 128.3   *Web URL Handler*

When the Web Container bundle is installed it must provide the webbundle: scheme to the URL class. The Web URL Handler has two primary responsibilities:

* *WAB* - If the source is already a bundle then only the Web-ContextPath can be set or overwritten.
* *WAR* - If the source is a WAR (that is, it must not contain any OSGi defined headers) then convert the WAR into a WAB.

The Web URL Handler can take parameters from the query arguments of the install URL, see *URL Parameters* on page 543.

The URL handler must validate query parameters, and ensure that the manifest rewriting results in valid OSGi headers. Any validation failures must result in Bundle Exception being thrown and the bundle install must fail.

Once a WAB is generated and installed, its life cycle is managed just like any other bundle in the framework.

128.4.1 URL Scheme

The Web URL Handler's scheme is defined to be:

```
scheme      ::= 'webbundle:' embedded '?' web-params
embedded    ::= <embedded URL according to RFC 1738>
web-params ::= ( web-param ( '&' web-param )* )?
web-param  ::= <key> '=' <value>
```

The web-param ‹key› and ‹value› as well as the ‹embedded url› must follow [6] *Uniform Resource Locators, RFC 1738* for their escaping and character set rules. A Web URL must further follow all the rules of a URL. Whitespaces are not allowed between terms.

An example for a webbundle: URL:

```
webbundle:http://www.acme.com:8021/sales.war?Web-ContextPath=/sales
```

Any URL scheme understood by the framework can be embedded, such as an http:, or file: URL. Some forms of embedded URL also contain URL query parameters and this must be supported. The embedded URL most be encoded as a standard URL. That is, the control characters like colon (':' \u003A), solidus ('/' \u002F), percent ('%' \u0025), and ampersand ('&' \u0026) must not be encoded. Thus the value returned from the getPath method may contain a query part. Any implementation must take care to preserve both the query parameters for the embedded URL, and for the complete webbundle: URL. A question mark must always follow the embedded URL to simplify this processing. The following example shows an HTTP URL with some query parameters:

```
webbundle:http://www.acme.com/sales?id=123?Bundle-SymbolicName=com.example&
                                          Web-ContextPath=/
```

128.4.2 URL Parsing

The URL object for a webbundle: URL must return the following values for the given methods:

- getProtocol - webbundle
- getPath - The complete embedded URL
- getQuery - The parameters for processing of the manifest.

For the following example:

```
webbundle:http://acme.com/repo?war=example.war?Web-ContextPath=/sales
```

The aforementioned methods must return:

- getProtocol - webbundle
- getPath - http://acme.com/repo?war=example.war
- getQuery - Web-ContextPath=/sales

128.4.3 URL Parameters

All the parameters in the webbundle: URL are optional except for the Web-ContextPath parameter. The parameter names are case insensitive, but their values must be treated as case sensitive. Table 128.3 describes the parameters that must be supported by any webbundle: URL Stream handler. A Web URL Handler is allowed to support additional parameters.

Table 128.3   *Web bundle URL Parameters*

Parameter Name	Description
Bundle-SymbolicName	The desired symbolic name for the resulting WAB.

Parameter Name	Description
Bundle-Version	The version of the resulting WAB. The value of this parameter must follow the OSGi versioning syntax.
Bundle-ManifestVersion	The desired bundle manifest version. Currently, the only valid value for this parameter is 2.
Import-Package	A list of packages that the war file depends on.
Web-ContextPath	The Context Path from which the Servlet Container should serve content from the resulting WAB. This is the only valid parameter when the input JAR is already a bundle. This parameter must be specified.

128.4.4   WAB Modification

The Web URL Handler can set or modify the Web-ContextPath of a WAB if the input source is already a bundle. It must be considered as a bundle when any of the OSGi defined headers listed in Table 128.3 is present in the bundle.

For WAB Modification, the Web URL Handler must only support the Web-ContextPath parameter and it must not modify any existing headers other than the Web-ContextPath. Any other parameter given must result in a Bundle Exception.

128.4.5   WAR Manifest Processing

The Web URL Handler is designed to support the transparent deployment of Java EE Web ARchives (WAR). Such WARs are ignorant of the requirements of the underlying OSGi framework that hosts the Web Runtime. These WARs are not proper OSGi bundles because they do not contain the necessary metadata in the manifest. For example, a WAR without a Bundle-ManifestVersion, Import-Package, and other headers cannot operate in an OSGi framework.

The Web URL Handler implementation copies the contents of the embedded URL to the output and rewrites the manifest headers based on the given parameters. The result must be a WAB.

Any parameters specified must be treated as manifest headers for the web. The following manifest headers must be set to the following values if not specified:

- Bundle-ManifestVersion - Must be set to 2.
- Bundle-SymbolicName - Generated in an implementation specific way.
- Bundle-ClassPath - Must consist of:
  - WEB-INF/classes
  - All JARs from the WEB-INF/lib directory in the WAR. The order of these embedded JARs is unspecified.
  - If these JARs declare dependencies in their manifest on other JARs in the bundle, then these jars must also be appended to the Bundle-ClassPath header. The process of detecting JAR dependencies must be performed recursively as indicated in the Servlet Specification.
- Web-ContextPath - The Web-ContextPath must be specified as a parameter. This Context Path should start with a leading solidus ('/' \u002F). The Web URL handler must add the preceding solidus it if it is not present.

The Web URL Handler is responsible for managing the import dependencies of the WAR. Implementations are free to handle the import dependencies in an implementation defined way. They can augment the Import-Package header with byte-code analysis information, add a fixed set of clauses, and/or use the DynamicImport-Package header as last resort.

Any other manifest headers defined as a parameter or WAR manifest header not described in this section must be copied to the WAB manifest by the Web URL Handler. Such an header must not be modified.

128.4.6   Signed WAR files

When a signed WAR file is installed using the Web URL Handler, then the manifest rewriting process invalidates the signatures in the bundle. The OSGi specification requires fully signed bundles for security reasons, security resources in partially signed bundles are ignored.

If the use of the signing metadata is required, the WAR must be converted to a WAB during development and then signed. In this case, the Web URL Handler cannot be used. If the Web URL Handler is presented with a signed WAR, the manifest name sections that contain the resource's check sums must be stripped out by the URL stream handler. Any signer files (∗.SF and their corresponding DSA/RSA signature files) must also be removed.

128.5   Events

The Web Extender must track all WABs in the OSGi framework in which the Web Extender is installed. The Web Extender must post Event Admin events, which is asynchronous, at crucial points in its processing. The topic of the event must be one of the following values:

- org/osgi/service/web/DEPLOYING - The Web Extender has accepted a WAB and started the process of deploying a Web Application.
- org/osgi/service/web/DEPLOYED - The Web Extender has finished deploying a Web Application, and the Web Application is now available for web requests on its Context Path.
- org/osgi/service/web/UNDEPLOYING - The web extender started undeploying the Web Application in response to its corresponding WAB being stopped or the Web Extender is stopped.
- org/osgi/service/web/UNDEPLOYED - The Web Extender has undeployed the Web Application. The application is no longer available for web requests.
- org/osgi/service/web/FAILED - The Web Extender has failed to deploy the Web Application, this event can be fired after the DEPLOYING event has fired and indicates that no DEPLOYED event will be fired.

For each event topic above, the following properties must be published:

- bundle.symbolicName - (String) The bundle symbolic name of the WAB.
- bundle.id - (Long) The bundle id of the WAB.
- bundle - (Bundle) The Bundle object of the WAB.
- bundle.version - (Version) The version of the WAB.
- context.path - (String) The Context Path of the Web Application.
- timestamp - (Long) The time when the event occurred
- extender.bundle - (Bundle) The Bundle object of the Web Extender Bundle
- extender.bundle.id - (Long) The id of the Web Extender Bundle.
- extender.bundle.symbolicName - (String) The symbolic name of the Web Extender Bundle.
- extender.bundle.version - (Version) The version of the Web Extender Bundle.

In addition, the org/osgi/service/web/FAILED event must also have the following property:

- exception - (Throwable) If an exception caused the failure, an exception detailing the error that occurred during the deployment of the WAB.
- collision - (String) If a name collision occurred, the Web-ContextPath that had a collision
- collision.bundles - (Collection<Long>) If a name collision occurred, a collection of bundle ids that all have the same value for the Web-ContextPath manifest header.

128.6　Interacting with the OSGi Environment

128.6.1　Bundle Context Access

In order to properly integrate in an OSGi environment, a Web Application can access the OSGi service registry for publishing its services, accessing services provided by other bundles, and listening to bundle and service events to track the life cycle of these artifacts. This requires access to the Bundle Context of the WAB.

The Web Extender must make the Bundle Context of the corresponding WAB available to the Web Application via the Servlet Context `osgi-bundlecontext` attribute. A Servlet can obtain a Bundle Context as follows:

```
BundleContext ctxt = (BundleContext)
    servletContext.getAttribute("osgi-bundlecontext");
```

128.6.2　Other Component Models

Web Applications sometimes need to inter-operate with services provided by other component models, such as a Declarative Services or Blueprint. Per the Servlet specification, the Servlet Container owns the life cycle of a Servlet; the life cycle of the Servlet must be subordinate to the life cycle of the Servlet Context, which is only dependent on the life cycle of the WAB. Interactions between different bundles are facilitated by the OSGi service registry. This interaction can be managed in several ways:

- A Servlet can obtain a Bundle Context from the Servlet Context for performing service registry operations.
- Via the JNDI Specification and the `osgi:service` JNDI namespace. The OSGi JNDI specification describes how OSGi services can be made available via the JNDI URL Context. It defines an `osgi:service` namespace and leverages URL Context factories to facilitate JNDI integration with the OSGi service registry.

Per this specification, it is not possible to make the Servlet life cycle dependent on the availability of specific services. Any synchronization and service dependency management must therefore be done by the Web Application itself.

128.6.3　Resource Lookup

The `getResource` and `getResourceAsStream` methods of the `ServletContext` interface are used to access resources in the web application. For a WAB, these resources must be found according to the `findEntries` method, this method includes fragments. For the `getResource` and `getResourceAsStream` method, if multiple resources are found, then the first one must be used.

Since the `getResource` and `getResourceAsStream` methods do not support wildcards while the `findEntries` method does it is necessary to escape the wildcard asterisk (`*` \u002A) with prefixing it with a reverse solidus (`\` \u005C). This implies that a reverse solidus must be escaped with an extra reverse solidus. For example, the path `foo\bar*` must be escaped to `foo\\bar\*`.

The `getResourcePaths` method must map to the Bundle `getEntryPaths` method, its return type is a Set and can not handle multiples. However, the paths from the `getEntryPaths` method are relative while the methods of the `getResourcePaths` must be absolute.

For example, assume the following manifest for a bundle:

```
Bundle-ClassPath: localized, WEB-INF
...
```

This WAB has an attached fragment `acme-de.jar` with the following content:

```
META-INF/MANIFEST.MF
localized/logo.png
```

The `getResource` method for `localized/logo.png` uses the `findEntries` method to find a resource in the directory `/localized` and the resource `logo.png`. Assuming the host bundle has no `localized/` directory, the Web Runtime must serve the `logo.png` resource from the `acme-de.jar`.

128.6.4 Resource Injection and Annotations

The Web Application `web.xml` descriptor can specify the `metadata-complete` attribute on the web-app element. This attribute defines whether the `web.xml` descriptor is *complete*, or whether the classes in the bundle should be examined for deployment annotations. If the `metadata-complete` attribute is set to `true`, the Web Runtime must ignore any servlet annotations present in the class files of the Web Application. Otherwise, if the `metadata-complete` attribute is not specified, or is set to `false`, the container should process the class files of the Web Application for annotations, if supported.

A WAB can make use of the annotations defined by [7] *JSR 250 Common Annotations for the Java Platform* if supported by the Web Extender. Such a WAB must import the packages the annotations are contained in. A Web Extender that does not support the use of JSR 250 annotations must not process a WAB that imports the annotations package.

128.6.5 Java Server Pages Support

Java Server Pages (JSP) is a rendering technology for template based web page construction. This specification supports [4] *JSP 2.1 specification* if available with the Web Runtime. The `servlet` element in a `web.xml` descriptor is used to describe both types of Web Components. JSP components are defined implicitly in the `web.xml` descriptor through the use of an implicit `.jsp` extension mapping, or explicitly through the use of a `jsp-group` element.

128.6.6 Compilation

A Web Runtime compiles a JSP page into a Servlet, either during the deployment phase, or at the time of request processing, and dispatches the request to an instance of such a dynamically created class. Often times, the compilation task is delegated to a separate JSP compiler that will be responsible for identifying the necessary tag libraries, and generating the corresponding Servlet. The container then proceeds to load the dynamically generated class, creates an instance and dispatches the servlet request to that instance.

Supporting in-line compilation of a JSP inside a bundle will require that the Web Runtime maintains a private area where it can store such compiled classes. The Web Runtime can leverage its private bundle storage area. The Web Runtime can construct a special class loader to load generated JSP classes such that classes from the bundle class path are visible to newly compiled JSP classes.

The JSP specification does not describe how JSP pages are dynamically compiled or reloaded. Various Web Runtime implementations handle the aspects in proprietary ways. This specification does not bring forward any explicit requirements for supporting dynamic aspects of JSP pages.

128.7 Security

The security aspects of this specification are defined by the [3] *Servlet 2.5 specification*.

128.8 References

[1]   *Jave Enterprise Edition Release 5*

Java 1.5.0 Packages. http://www.oracle.com/technetwork/java/javaee/tech/javaee5-jsp-135162.html

[2]   *Java EE Web Applications*
      http://www.oracle.com/technetwork/java/javaee/tech/webapps-138511.html

[3]   *Servlet 2.5 specification*
      http://jcp.org/aboutJava/communityprocess/mrel/jsr154/index.html

[4]   *JSP 2.1 specification*
      http://jcp.org/aboutJava/communityprocess/final/jsr245/index.html

[5]   *PAX Web Extender*
      http://team.ops4j.org/wiki/display/paxweb/Pax+Web

[6]   *Uniform Resource Locators, RFC 1738*
      http://www.ietf.org/rfc/rfc1738.txt

[7]   *JSR 250 Common Annotations for the Java Platform*
      http://jcp.org/aboutJava/communityprocess/pfd/jsr250/index.html

130   Coordinator Service Specification

Version 1.0

130.1   Introduction

The OSGi programming model is based on the collaboration of standard and custom components. In such a model there is no central authority that has global knowledge of the complete application. Though this lack of authority can significantly increase reusability (and robustness) there are times when the activities of the collaborators must be coordinated. For example, a service that is repeatedly called in a task could optimize performance by caching intermediate results until it *knew* the task was ended.

To know when a task involving multiple collaborators has ended is the primary purpose of the Coordinator service specification. The Coordinator service provides a rendezvous for an initiator to create a Coordination where collaborators can decide to participate. When the Coordination has ended, all participants are informed.

This Coordinator service provides an explicit Coordination model, the Coordination is explicitly passed as a parameter, and an implicit model where the Coordination is associated with the current thread. Implicit Coordinations can be nested.

Coordinators share the coordination aspects of the resource model of transactions. However, the model is much lighter-weight because it does not support any of the ACID properties.

130.1.1   Essentials
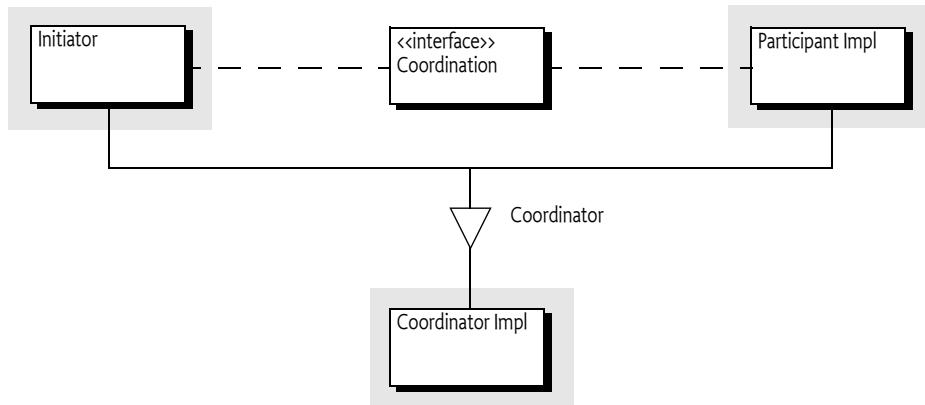
- *Coordination* - Provide a solution to allow multiple collaborators to coordinate the outcome of a task initiated by an initiator.
- *Initiator* - An initiator must be able to initiate a coordination and control the final outcome.
- *Participants* - Participants in the task must be informed when the coordination has ended or failed as well as being able to terminate the Coordination.
- *Time-out* - A Coordination should fail after a given time-out.
- *Blocking* - Provide support for blocking and serializing access to Participants.
- *Nesting* - It must be possible to nest Coordinations.
- *Per Thread Model* - Provide a per-thread current Coordination model.
- *Variables* - Provide a variable space per Coordination

130.1.2   Entities

- *Coordinator* - A service that can create and enumerate Coordinations.
- *Coordination* - Represents the ongoing Coordination.
- *Initiator* - The party that initiates a Coordination.
- *Participant* - A party that wants to be informed of the outcome of a Coordination.
- *Collaborator* - Either a participant or initiator.

Class and Service overview

130.2   Usage

This section is an introduction in the usage of the Coordinator service. It is not the formal specification, the normative part starts at *Coordinator Service* on page 559. This section leaves out some of the details for clarity.

130.2.1   Synopsis
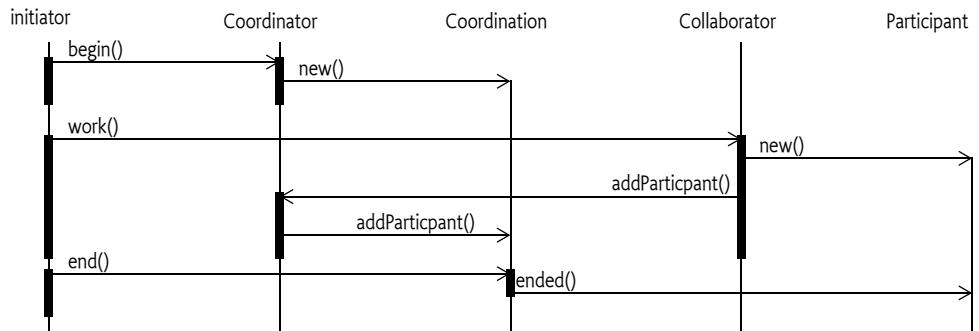
The Coordinator service provides a mechanism for multiple parties to *collaborate* on a common task without a priori knowledge of who will collaborate in that task. A collaborator can participate by adding a *Participant* to the Coordination. The Coordination will notify the Participants when the coordination is *ended* or when it is *failed*.

Each Coordination has an *initiator* that creates the Coordination object through the Coordinator service. The initiator can then push this object on a thread-local stack to make it an implicit Coordination or it can pass this object around as a parameter for *explicit* Coordinations. Collaborators can then use the *current* Coordination on the stack or get it from a parameter. Whenever a bundle wants to participate in the Coordination it adds itself to the Coordination as a participant. If necessary, a collaborator can initiate a new Coordination, which could be a nested Coordination for implicit Coordinations.

A Coordination must be *terminated*. Termination is either a normal end when the initiator calls the end method or it is failed when the fail method is called. A Coordination can be failed by any of the collaborators. A Coordination can also fail independently due to a *time-out* or when the initiator releases its Coordinator service. All participants in the Coordination are informed in reverse participation order about the outcome in a callback for ended or failed Coordinations.

A typical action diagram with a successful outcome is depicted in Figure 130.2.

Figure 130.2   *Action Diagram Implicit Coordination*

130.2.2   Explicit Coordination

The general pattern for an initiator is to create a Coordination through the Coordinator service, per-form the work in a try block, catch any exceptions and fail the Coordination in the catch block, and then ensure ending the Coordination in the finally block. The finally block can cause an exception. This is demonstrated in the following example:

```
Coordination c = coordinator.create("com.example.work",0);
try {
    doWork(c);
} catch( Exception e ) {
    c.fail(e);
} finally {
    c.end();
}
```
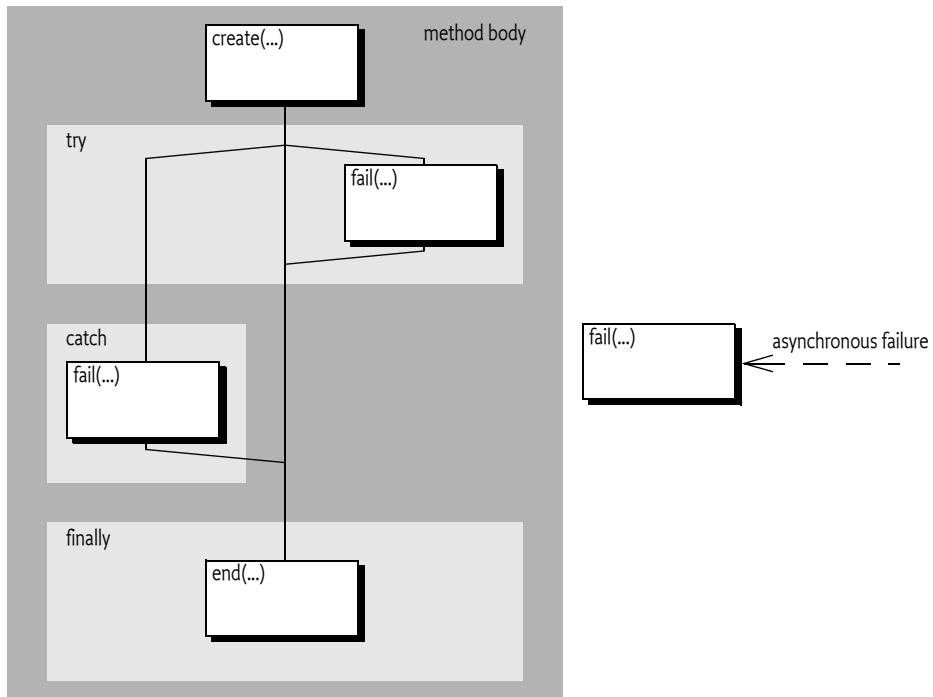
This deceptively small template is quite robust:

- If the doWork method throws an Exception then the template fails with a Coordination Excep-tion because it is failed in the try block.
- Any exceptions thrown in the try block are automatically causing the Coordination to fail.
- The Coordination is always terminated and removed from the stack due to the finally block.
- All failure paths, Coordinations that are failed by any of the collaborators, time-outs, or oth-er problems are handled by the end method in the finally block. It will throw a FAILED or PARTIALLY_ENDED Coordination Exception for any of the failures.

The different failure paths and their handling is pictured in Figure 130.3.

Figure 130.3   *Flow through the Coordination template*

The example shows an explicit Coordination because the create method is used, implicit Coordinations are used in *Implicit Coordinations* on page 553. The parameters of the create method are the name of the Coordination and its time-out. The name is used for informational purposes as well as security. For security reasons, the name must follow the same syntax as the Bundle Symbolic Name. In a secure environment the name can be used to limit Coordinations to a limited set of bundles. For example, a set of bundles signed by a specific signer can use names like com.acme.* that are denied to all other bundles.

The zero time-out specifies that the Coordination will not have a time-out. Otherwise it must be a positive long, indicating the number of milliseconds the Coordination may take. However, implementations should have a configurable time-out to ensure that the system remains alive.

In the doWork method the real work is done in conjunction with the collaborators. Explicit Coordinations can be passed to other threads if needed. Collaborators can decide to add participants whenever they require a notification when the Coordination has been terminated. For example, the following code could be called from the doWork method:

```
void foo(Coordination c) {
  doPrepare();
  c.addParticipant(this);
}
```

This method does the preparation work but does not finalize it so that next time it can use some intermediate results. For example, the prepare method could cache a connection to a database that should be reused during the Coordination. The collaborator can assume that it will be called back on either the failed or ended method. These methods could look like:

```
public void ended(Coordination c)  { doFinish(); }
public void failed(Coordination c) { doFailed(); }
```

The Coordinator provides the guarantee that this code will always call the doFinish method when the Coordination succeeds and doFailed method when it failed.

The Participant must be aware that the ended(Coordination) and failed(Coordination) methods can be called on any thread.

If the doWork method throws an exception it will end up in the catch block of the initiator. The catch block will then fail the Coordination by calling the fail method with the given exception. If the Coordination was already terminated because something else already had failed it then the method call is ignored, only the first fail is used, later fails are ignored.

In all cases, the finally block is executed last. The finally block ends the Coordination. If this coordination was failed then it will throw a Coordination Exception detailing the reason of the failure. Otherwise it will terminate it and notify all the participants.

The Coordination Exception is a Runtime Exception making it unnecessary to declare it.

130.2.3 Multi Threading

Explicit Coordinations allow the Coordination objects to be passed to many different collaborators who can perform the work on different threads. Each collaborator can fail the Coordination at any moment in time or the time-out can occur on yet another thread. Participants must therefore be aware that the callbacks ended and failed can happen on any thread. The following example shows a typical case where a task is parallelized. If any thread fails the Coordination, all other threads could be notified before they're finished.

```
Executor executor = ...
final CountDownLatch latch = new CountdownLatch(10);
final Coordination c = coordinator.create("parallel", 0);
for ( int i=0; i<10; i++) {
  executor.execute(
    new Runnable() {
        public void run() { baz(c); latch.countDown(); }
      });
  }
  latch.await();
  c.end();
```

The Coordination object is thread safe so it can be freely passed around.

130.2.4 Implicit Coordinations

An explicit Coordination requires that the Coordination is passed as a parameter to the doWork method. The Coordinator also supports *implicit* Coordinations. With implicit Coordinations the Coordinator maintains a thread local stack of Coordinations where the top of this stack is the *current* Coordination for that thread. The usage of the implicit Coordination is almost identical to the explicit Coordinations except that all the work occurs on a single thread. The control flow is almost identical to explicit Coordinations:

```
Coordination c = coordinator.begin("com.example.work",0);
try {
    doWork();
} catch( Exception e ) {
    c.fail(e);
} finally {
    c.end();
}
```

See also Figure 130.3. However, in this case the finally block with the call to the end method is even more important. With an implicit Coordination the Coordination is put on a thread local stack in the begin method and must therefore be popped when the Coordination is finished. The finally block ensures therefore the proper cleanup of this thread local stack.

The difference between implicit and explicit Coordinations is that the implicit Coordination is not passed as a parameter, instead, collaborators use the current Coordination. With implicit Coordinations all method invocations in a thread can always access the current Coordination, even if they have many intermediates on the stack. The implicit model allows a collaborator many levels down the stack to detect a current Coordination and register itself without the need to modify all intermediate methods to contain a Coordination parameter. The explicit model has the advantage of explicitness but requires all APIs to be modified to hold the parameter. This model does not support passing the parameter through layers that are not aware of the Coordination. For example, OSGi services in general do not have a Coordination parameter in their methods making the use of explicit Coordinations impossible.

Collaborators can act differently in the presence of a current Coordination. For example, a collaborator can optimize its work flow depending on the presence of a current Coordination.

```
Coordinator coordinator = ...
void foo() {
  doPrepare();
  if ( !coordinator.addParticipant(this))
      doFinish();
}
```

The Coordinator service has an addParticipant method that makes working with the current Coordination simple. If there is a current Coordination then the Coordinator service will add the participant and return true, otherwise it returns false. It is therefore easy to react differently in the presence of a current Coordination. In the previous example, the doFinish method will be called immediately if there was no current Coordination, otherwise it is delayed until the Coordination fails or succeeds. The participant callbacks look the same as in the previous section:

```
public void ended(Coordination c)  { doFinish(); }
public void failed(Coordination c) { doFailed(); }
```

Though the code looks very similar for the implicit and explicit Coordinations there are some additional rules for implicit Coordinations.

The end method must be called on the same thread as the begin method, trying to end it on another thread results in a WRONG_THREAD Coordination Exception being thrown.

Even though the end method must be called on the initiating thread, the callbacks to the Participants can be done on any thread as the specification allows the Coordinator to use multiple threads for all callbacks.

130.2.5 Partial Ending

The Coordination is a best effort mechanism to coordinate, not a transaction model with integrity guarantees. This means that users of the Coordinator service must understand that there are cases where a Coordination ends in limbo. This happens when one of the Participants throws an Exception in the ended callback. This is similar to a transactional resource manager failing to commit in a 2-phase commit after it has voted yes in the prepare phase; a problem that is the cause of much of the complexity of a transaction manager. The Coordinator is limited to use cases that do not require full ACID properties and can therefore be much simpler. However, users of the Coordinator service must be aware of this limitation.

If a Participant throws an exception in the ended method, the end method that terminated the Coordination must throw a PARTIALLY_ENDED Coordination Exception. It is then up to the initiator to

correct the situations. In most cases, this means allowing the exception to be re-thrown and handle the failure at the top level. Handling in those cases usually implies logging and continuing.

The following code shows how the PARTIALLY_ENDED case can be handled more explicitly.

```
Coordination c = coordinator.begin("work",0);
try {
  doWork();
} catch( Excption e ) {
  c.fail(e);
} finally {
  try {
    c.end();
  } catch( CoordinationException e ) {
    if ( e.getType() == CoordinationException.PARTIALLY_ENDED) {
      // limbo!
      ...
    }
  }
}
```

130.2.6   Locking

To participate in a Coordination and receive callbacks a collaborator must add a Participant object to the Coordination. The addParticipant(Participant) method blocks if the given Participant object is already used in another Coordination. This blocking facility can be used to implement a number of simple locking schemes that can simplify maintaining state in a concurrent environment.

Using the Participant object as the key for the lock makes it simple to do course grained locking. For example, a service implementation could use the service object as a lock, effectively serializing access to this service when it is used in a Coordination. Coarse grained locking allows all the state to be maintained in the coarse object and not having to worry about multiplexing simultaneous requests. The following code uses the coarse locking pattern because the collaborator implements the Participant interface itself:

```
public class Collaborator implements Participant{
  public void doWork(Coordination coordination ) {
    ...
    coordination.addParticipant(this);
  }

  public void ended(Coordination c) { ... }
  public void failed(Coordination c) { ... }
}
```

The simplicity of the coarse grained locking is at the expense of lower performance because tasks are serialized even if it would have no contention. Locks can therefore also be made more fine grained, allowing more concurrency. In the extreme case, creating a new object for each participation makes it possible to never lock. For example, the following code never locks because it always creates a new object for the Participant:

```
    public void doWork(Coordination coordination){
      final State state = ...
      coordination.addParticipant(
        new Participant() {
          public void ended(Coordination c) { state ... }
          public void failed(Coordination c) { state ...}
```

```
} ); }
```

130.2.7 Failing

Any collaborator can fail an ongoing Coordination by calling the fail(Throwable) method, the Throwable parameter must not be null. When the Coordination has already terminated then this is a no-op. The Coordinator service has a convenience method that fails the current Coordination if present. The fail methods return a boolean that is true when the method call causes the termination of the Coordination, in all other cases it is false.

Failing a Coordination will immediately perform the callbacks and reject any additional Participants by throwing an ALREADY_ENDED Coordination Exception. The asynchronous nature of the fail method implies that it is possible to have been called even before the addParticipant(Participant) method has returned. Anybody that has the Coordination object can check the failed state with the getFailure() method.

In general, the best and most robust strategy to handle failures is to throw an Exception from the collaborator, allowing the initiator to catch the exception and properly fail the Coordination.

130.2.8 Time-out

The time-out is specified in the Coordinator create(String,long) or begin(String,long) methods. A time-out of zero is indefinite, otherwise the time-out specifies the number of milliseconds the Coordination can take to terminate. A given time-out can be extended with the extendTimeout(long) method. This method will add an additional time-out to the existing deadline if a prior deadline was set. For example, the following code extends the time-out with 5 seconds whenever a message must be sent to a remote address:

```
Object sendMessage(Message m) {
  Coordination c = coordinator.peek();
  Address a = m.getDestination();
  if ( c != null && a.isRemote() ) {
    c.extendTimeout(5000);
  }
  return sendMessage0(m);
}
```

Applications should not rely on the exact time-out of the Coordination and only use it as a safety function against deadlocks and hanging collaborators.

130.2.9 Joining

When a Coordination is terminated it is not yet completely finished, the callback to the Participants happens after the atomic termination. In certain cases it is necessary to ensure that a method does not progress until all the participants have been notified. It is therefore possible to wait for the Coordination to completely finish with the join(long) method. This method can have a time-out. For example:

```
void collaborate( final Coordination c ) {
  doWork();
  Thread t = new Thread() {
    public void run(){
      try {
          c.join(0);
          ... // really terminated here, all participantscalled back
      } catch( Exception e) { ... }
    }
  };
```

```
    t.start();
  }
```

130.2.10   Variables

A Participant is likely to have to maintain state that is particular for the collaboration. This state is usually needed in the `ended` method to properly finalize the work. In general, the best place to store this state is in the `Participant` object itself, inner classes and final variables are a good technique for storing the state. However, the state can also be stored in a Coordination *variable*. Each Coordination has a private set of variables that can be obtained with the getVariables() method. The resulting map takes a class as the key and returns an Object. The map is not synchronized, any changes to the map must be synchronized on the returned Map object to ensure the visibility of the changes to other threads. The class used for the key is not related to the returned type, it is a `Class` object to provide a convenient namespace.

The following example shows how the state can be stored with variables.

```
public void doWork(Coordination coordination){
  Map<Class<?>,Object> map = coordination.getVariables();
  synchronized(map) {
    State state = (State) map.get( SharedWorker.class );
    if ( state == null ) {
      state = new State(this);
      map.put( state );
      ... do initial work
    }
  }
  ... do other work
  coordination.addParticipant( this );
}
public void ended(Coordination c) {
  Map<Class<?>,Object> map = coordination.getVariables();
  synchronized(map) {
    State state = (State) map.get( SharedWorker.class );
    .. finalize
  }
}
public void failed(Coordination c) {
  Map<Class<?>,Object> map = coordination.getVariables();
  synchronized(map) {
    State state = (State) map.get( SharedWorker.class );
    .. finalize
  }
}
```

130.2.11   Optimizing Example

For example, a web based system has a charge service:

```
public interface Charge {
  void charge( String reason, int amount );
}
```

This service is used throughout the system for charging the tasks the system performs. Each servlet request can actually create multiple Charge Data Records (CDR). For this reason, a Coordination is started before the page is constructed. Each part of the page that has an associated cost must create a CDR. There are the following issues at stake:

- Charging should not take place when failing, and
- Performance can be optimized to only persist the CDRs once, and
- The user must be passed to the Charge service.

To begin with the request code:

```
public void doGet(HttpServletRequest rq, HttpServletResponsersp) {
  Coordination c = coordinator.begin("com.acme.request", 30000);
  try {
    Principal p = rq.getUserPrincipal();
    Map<Class<?>,Object> map = c.getVariables();
    map.put( Principal.class, p );
    buildPage(rq,rsp);
  } catch( Exception e  ) { c.fail(e); }
    finally               { c.end(); }
}
```

Each method that has a charge will call the Charge service. The following code shows an implementation of this Charge service.

```
public class ChargeImpl implements Charge,Participant {
  final List<CDR> records = new ArrayList<CDR>();

  public void charge( String reason, int amount ) {
    Coordination c = coordinator.peek();
    if ( c == null ) {
       save( Arrays.asList( new CDR(null, reason, amount)));
    } else {
      Principal p = getPrincipal(c);
      records.add( new CDR(p, reason, amount ) );
      c.addParticipant( this );
    }
  }

  Principal getPrincipal(Coordination c) {
    if ( c == null )
      return null;

    Map<Class<?>,Object> map = c.getVariables();
    synchronized(map) {
      Principal p = (Principal) map.get( Principal.class );
      return p != null ? p : getPrincipal(c.getEnclosingCoordination());
    }
  }

  public void ended(Coordination c) {
    save(records);
    records.clear();
  }
  public void failed(Coordination c) {
    records.clear();
  }

  void save(List<CDR> records) { ... }
}
```

130.2.12 Security Example

The Coordination Permission is a filter based permission that is asserted for many of the methods in the API, the bundle that is checked is always the bundle that created the corresponding Coordination. For example:

```
ALLOW {
    [ BundleSignerCondition "cn=ACME" ]
    ( CoordinationPermission "(signer=cn=ACME)" "*" )
}
```

This example allows bundles signed by ACME to perform all Coordination actions on Coordinations created by bundles signed by ACME.

The filter can also assert the name of the Coordination:

```
coordination.name
```

It is therefore possible to create a name based protection scheme. By denying all bundles except a select group through the use of a name prefix, the use of Coordinations can be restricted to this select group:

```
DENY {
    [ BundleSignerCondition "cn=ACME" "!" ]
    ( CoordinationPermission "(coordination.name=com.acme.*)" "*" )
}
ALLOW {
    ( CoordinationPermission "(coordination.name=*)" "*" )
}
```

If a bundle is not signed by ACME it will be denied the use of Coordination names starting with com.acme. though it will be allowed to use any other name. This effectively enables only bundles signed by ACME to create Coordinations with this name prefix.

130.3 Coordinator Service

The Coordinator service is the entry point for the Coordination. It provides the following functions:

- Coordination creation
- Life cycle management of a Coordination
- Thread based Coordinations
- Introspection

130.3.1 Coordination Creation

A Coordination object is created by an *initiator*. An initiator can create a Coordination object with the Coordinator create(String,long) or begin(String,long) method. Each Coordination when created gets a positive long identity that is available with getId(). Ids are a unique identifier for a specific Coordinator service. The id is always increasing, that is, a Coordination with a higher id is created later.

The create methods specify the name of the Coordination. This name is a security concept, see *Security* on page 564, as well as used for debugging. The coordination name must therefore conform to the same syntax as a bundle symbolic name:

```
coordination-name ::= symbolic-name    // see OSGi Core Release 6
```

Passing a name that does not conform to this syntax must throw an Illegal Argument Exception. There are no constraints on duplicates, multiple different Coordinations can use the same name. The name of the Coordination is available with the getName() method.

130.3.2 Adding Participants

The Coordination object can be passed to *collaborators* as a parameter in a method call. Some of these collaborators might be interested in *participating* in the given Coordination, they can achieve this by adding a Participant object to the Coordination.

A Participant is a collaborator that requires a callback after the Coordination has been terminated, either when it ended or when it failed. To participate, it must add a Participant object to a Coordination with the addParticipant(Participant) method on Coordination. This method throws an ALREADY_ENDED or FAILED Coordination Exception when the Coordination has been terminated.

When a Participant is:

- *Not in any Coordination* - Add it to the given Coordination and return.
- *In target Coordination* - Ignore, participant is already present. A Participant can participate in the same Coordination multiple times by calling addParticipant(Participant) but will only be called back once when the Coordination is terminated. Its order must be defined by the first addition.
- *In another Coordination* - Lock until after the other Coordination has notified all the Participants. Implementations can detect deadlocks in certain cases and throw a Coordination Exception if a dead lock exist, otherwise the deadlock is solved when the Coordination times out.

Verifying if a Participant object is already in another Coordination must use identity and not equality.

130.3.3 Active
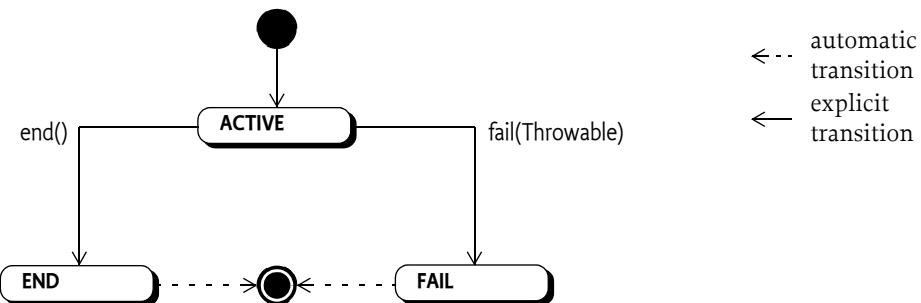
A Coordination is active until it is *terminated*. A Coordination can terminate because it is *ended*, or it is *failed*. The following methods cause a termination:

- end() - A normal end. All participants that were added before the end call are called back on their ended(Coordination) method.
- fail(Throwable) - The Coordination has failed, this will call back the failed(Coordination) method on the participants. This method can be called by the Coordinator, the initiator, or any of the collaborators. There are a number of failures that are built in to the Coordinator. These failures use singleton Exception instances defined in the Coordination interface:
  - TIMEOUT - If the Coordination times out the Coordination is failed with the TIMEOUT exception instance in Coordination.
  - RELEASED - If the Coordinator that created the Coordination was unget, all Coordinations created by it will fail with the RELEASED exception.

The state diagram for the Coordination is pictured in Figure 130.4.

Figure 130.4   *Coordination state diagram*

130.3.4   Explicit and Implicit Models

The Coordinator supports two very different models of usage: *explicit* and *implicit*. The explicit model is when a Coordination is created and passed around as a parameter. The second model is the implicit model where the Coordinator maintains a thread local stack of Coordinations. Any collaborator can then decide to use the top of the stack as the *current* Coordination. The peek() method provides access to the current Coordination.

The begin(String,long) method creates a new Coordination and pushes this on the stack, beginning an implicit Coordination. This is identical to:

```
coordinator.create("work",0).push();
```

Once a Coordination is pushed on a stack it is from that moment on associated with the current thread. A Coordination can only be pushed once, the ALREADY_PUSHED Coordination Exception must be thrown when the Coordination is already associated with one of the thread local stacks maintained by the Coordinator service.

The Coordination is removed from the stack in the end() method. The end() method must not only terminate itself but it must also terminate all nested Coordinations.

The current Coordination can also be explicitly removed with the Coordinator pop() method.

A Coordination that is pushed on a thread local stack returns the associated thread on the getThread() method. This method returns null for Coordinations not on any stack, that is, explicit Coordinations.

130.3.5   Termination

Both the end() and fail(Throwable) methods terminate the Coordination if it was not already terminated. Termination is atomic, only the end or the fail method can terminate the Coordination. Though this happens on different threads, a Coordination can never both end and fail from any perspective. That is, if a fail races with end then only one of them can win and the other provides the feedback that the Coordination was already terminated.

Terminating a Coordination has the following effects:

- It is atomic, it can only happen once in a Coordination
- It freezes the set of participants, no more participants can be added

130.3.6   Ending

The end() method should always be called at the end of a Coordination to ensure proper termination, notification, and cleanup. The end method throws a FAILED or PARTIALLY_ENDED Coordination Exception if the Coordination was failed before.

If the Coordination had already been ended before then this is a programming error and an ALREADY_ENDED Configuration Exception is thrown. The end() method should never be called twice on the same Coordination.

If the termination succeeds then the participants must be notified by calling the ended(Coordination) method on each Participant that had been successfully added to the Coordination. This callback can take place on any thread but must be in reverse order of adding. That is, the last added Participant is called back first.

Participants must never make any assumptions about the current Coordination in the callback. The Coordination it was added to is therefore given as an explicit parameter in the ended(Coordination) method.

If a Participant throws an Exception then this must not prevent the calling of the remaining participants. The Exception should be logged. If a Participant has thrown an Exception then the end()

method must throw a PARTIALLY_ENDED Coordination Exception after the last Participant has returned from its callback, otherwise the method returns normally. Participants should normally not throw Exceptions in their callbacks.

If the Coordination is implicit (it is pushed on a stack) then the Coordination must be removed from its stack after the participants have been called back. This requires that the ending thread is the same as the thread of the Coordination. The end thread is the thread of the end() method call. If the Coordination's thread is not the same as the ending thread then a WRONG_THREAD Coordination Exception is thrown.

If the ending Coordination is on the stack but it is not the current Coordination then each nested Coordination must be ended before the current Coordination, see *Nesting Implicit Coordinations* on page 562 for more information.

The fail(Throwable) method must not remove the current Coordination, it must remain on the stack. The initiator must always call the end() method. Always calling end() in a finally block is therefore paramount.

130.3.7 Failing, TIMEOUT, ORPHANED, and RELEASED

Failing can happen asynchronously during the time a Coordination is active. A Coordination is failed by calling fail(Throwable). The Throwable argument must not be null, it is the cause of the failure.

Failing a Coordination must first terminate it. If the Coordination was already terminated the fail(Throwable) method has no effect. Otherwise, it must callback all its added Participants on the failed(Coordination) callback method. Exceptions thrown from this method should be logged and further ignored. The callback can occur on any thread, including the caller's.

Implicit Coordinations must not be popped from its stack in a fail nor is it necessary to call the fail method from any particular thread. The removal of the Coordination from the stack must happen in the end method.

There are two asynchronous events that can also fail the Coordination. If the Coordination times out, it will be treated as a fail(TIMEOUT) and if the Coordinator is ungotten with active Coordinations then each of those Coordinations must fail as if fail(RELEASED) is called.

A Coordination can also be *orphaned*. An orphaned Coordination has no longer any outside references. This means that the Coordination can no longer be ended or failed. Such Coordinations must fail with an ORPHANED Exception.

130.3.8 Nesting Implicit Coordinations

Implicit Coordinations can be nested. For this reason, the Coordinator maintains a thread local stack of Coordinations where the top, accessible with the peek() method, is the current Coordination. Each time a new Coordination is begun with the begin(String,long) method, the current Coordination is replaced with the newly created Coordination. When that Coordination is ended, the previous current Coordination is restored. Nesting is always on the same thread, implicit Coordinations are always associated with a single thread, available through its getThread() method. The end method must be called on the same thread as the begin(String,long) or last push() method.

Using the standard model for implicit Coordinations, where the initiator always ends the Coordination on the same thread as it begun, ensures that nesting is properly handled. However, in certain cases it is necessary to manipulate the stack or make implicit Coordinations explicit or vice versa. For this reason, it is possible to pop Coordinations from the stack with the pop() method. This method disassociates the Coordination from the current thread and restores the previous (if any) Coordination as the current Thread. A Coordination can then be made the current Coordination for a thread by calling the push() method. However, a Coordination can be pushed on the stack at most once. If a Coordination is pushed a second time, in any thread, the ALREADY_PUSHED Coordination Exception must be thrown.

The Coordination is removed from its stack when the end() method is called. It is therefore highly recommended to always end a Coordination in the nesting order. However, it is possible that a Coordination is ended that is not the current Coordination, it has nested Coordinations that were not properly ended. In that case all nested Coordinations must be ended in reverse creation order, that is, the current Coordination first, by calling the end method on it.

If any Coordination fails to end properly (including PARTIALLY_ENDED) then the remaining Coordinations on the stack must fail and chain the exceptions. In pseudo code:

```
while (coordinator.peek() != this) {
 try {
     coordinator.peek().end();
 } catch (CoordinationException e) {
     coordinator.peek().fail(e);
 }
}
```

130.3.9   Time-outs

When a Coordination is created it will receive a time-out. A time-out is a positive value or zero. A zero value indicates that the Coordination should have no time-out. This does not imply that a Coordination will never time-out, implementations are allowed to be configured with a limit to the maximum active time for a Coordination.

Collaborators can extend the time out with the extendTimeout(long) method. If no time-out was set (0), this method will be ignored. Otherwise the given amount (which must be positive) is added to the existing deadline. A Coordinator implementation can fail the Coordination earlier, however, when configured to do so.

If a Coordination is timed out, the Coordination is failed with a fail(TIMEOUT) method call from an unspecified thread, see *Failing, TIMEOUT, ORPHANED, and RELEASED* on page 562.

130.3.10   Released

The Coordination's life cycle is bound to the Coordinator service that created it. If the initiator's bundle ungets this service then the Coordinator must fail all the Coordinations created by this Coordinator by calling the fail(RELEASED) method.

Participants from bundles that are stopped are not taken into account. This means that it is possible that a participant is called while its bundle is stopped. Stopped Participants should fail any Coordinations that they participate in.

130.3.11   Coordinator Convenience Methods

The Coordinator contains a number of convenience methods that can be used by collaborators to interact with the current Coordination.

- begin(String,long) - Is logically the same as create(String,long). push().
- addParticipant(Participant) - This method makes it easy to react differently to the presence of a current implicit Coordination. If a current Coordination exists, the participant is added and true is returned (or an exception thrown if the Coordination is already terminated), otherwise false is returned.
- fail(Throwable) - If there is no current Coordination, this method returns false. Otherwise it returns the result of calling fail(Throwable) on the current Coordination. This method therefore only returns true when a current Coordination was actually terminated due to this call.

130.3.12   Administrative Access

The Coordination objects provide a number of methods that are used for administrating the Coordinations and the Coordinator.

- getBundle() - Provide the bundle that created the Coordination. This bundle is the bundle belonging to the Bundle Context used to get the Coordinator service.
- getFailure() - The Exception that caused this Coordination to fail or null. There are two fixed exception instances for a time out (TIMEOUT), when the Coordination is orphaned (ORPHANED), and when the Coordinator service is released (RELEASED).
- getId() - The Coordination's id.
- getName() - The name of the Coordination.
- getParticipants() - The current list of participants. This is a mutable snapshot of the added participants. Changing the snapshot has no effect on the Coordination.
- getThread() - Answer the thread associated with an implicit Coordination. If the Coordination is not implicit then the answer is null.
- getEnclosingCoordination() - Return the enclosing Coordination.

And for the Coordinator:

- getCoordination(long) - Retrieve a Coordination by its id.
- getCoordinations() - Get a list of active Coordinations

130.3.13 Summary

A Coordination can exist in three different states *ACTIVE*, *END*, and *FAIL*. During its life it will transition from ACTIVE to either END or FAIL. The entry (when the state is entered) and exit (when the state is left) actions when this transition takes place and the effect on the different methods are summarized in the following table.

Table 130.1   *States and transitions*

State/Method	ACTIVE	END	FAIL
entry action		Notify all the participants by calling the ended(Coordination) method.	Notify all the participants by calling the failed(Coordination) method.
exit action	Terminate		
end()	-> END.	throws ALREADY_ENDED	throws FAILED
	Can throw PARTIALLY_ENDED		
fail(Throwable)	-> FAIL, return true.	return false.	return false.

130.4 Security

This specification provides a Coordination Permission. This permission can enforce the name of the coordination as well as assert the properties of the initiating bundle, like for example the signer or bundle symbolic name. The permission therefore uses a filter as name, as defined in the filter based permissions section in *OSGi Core Release 6*, see *OSGi Core Release 6*. There is one additional parameter for the filter:

coordination.name

The value is the given name of the Coordination. Restricting the name of a Coordination allows the deployer to limit the use of this name to a restricted set of bundles.

The following actions are defined:

- INITIATE - Required to initiate and control a Coordination.
- PARTICIPATE - Required to participate in a Coordination.

- ADMIN - Required to administrate a Coordinator.

The target bundle of the Coordination Permission is the initiator's bundle. This is the bundle that got the Coordinator service to create the Coordination. An initiator must therefore have permission to create Coordinations for itself.

There are two constructors available:

- CoordinationPermission(String,String) - The constructor for the granted permission. It is given a filter expression and the actions that the permission applies to.
- CoordinationPermission(String,Bundle,String) - The constructor for the requested permission. It is given the name of the permission, the bundle that created the corresponding coordination, and the requested actions.

130.5 org.osgi.service.coordinator

Coordinator Package Version 1.0.

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

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.coordinator; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.coordinator; version="[1.0,1.1)"
```

130.5.1 Summary

- Coordination - A Coordination object is used to coordinate a number of independent Participants.
- CoordinationException - Unchecked exception which may be thrown by a Coordinator implementation.
- CoordinationPermission - A bundle's authority to create or use a Coordination.
- Coordinator - A Coordinator service coordinates activities between different parties.
- Participant - A Participant participates in a Coordination.

130.5.2 public interface Coordination

A Coordination object is used to coordinate a number of independent Participants.

Once a Coordination is created, it can be used to add Participant objects. When the Coordination is ended, the participants are notified. A Coordination can also fail for various reasons. When this occurs, the participants are notified of the failure.

A Coordination must be in one of two states, either ACTIVE or TERMINATED. The transition between ACTIVE and TERMINATED must be atomic, ensuring that a Participant can be guaranteed of either receiving an exception when adding itself to a Coordination or of receiving notification the Coordination has terminated.

A Coordination object is thread safe and can be passed as a parameter to other parties regardless of the threads these parties use.

The following example code shows how a Coordination should be used.

```
void foo() {
```

```
  Coordination c = coordinator.create("work", 0);
  try {
    doWork(c);
  }
  catch (Exception e) {
    c.fail(e);
  }
  finally {
    c.end();
  }
}
```

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

130.5.2.1   **public static final Exception ORPHANED**

A singleton exception that will be the failure cause when a Coordination has been orphaned.

130.5.2.2   **public static final Exception RELEASED**

A singleton exception that will be the failure cause when the Coordinations created by a bundle are terminated because the bundle released the Coordinator service.

130.5.2.3   **public static final Exception TIMEOUT**

A singleton exception that will be the failure cause when a Coordination times out.

130.5.2.4   **public void addParticipant(Participant participant)**

participant The Participant to register with this Coordination. The participant must not be null.

☐ Register a Participant with this Coordination.

Once a Participant is registered with this Coordination, it is guaranteed to receive a notification for either normal or failure termination when this Coordination is terminated.

Participants are registered using their object identity. Once a Participant is registered with this Coordination, subsequent attempts to register the Participant again with this Coordination are ignored and the Participant is only notified once when this Coordination is terminated.

A Participant can only be registered with a single active Coordination at a time. If a Participant is already registered with an active Coordination, attempts to register the Participation with another active Coordination will block until the Coordination the Participant is registered with terminates. Notice that in edge cases the notification to the Participant that this Coordination has terminated can happen before this method returns.

Attempting to register a Participant with a terminated Coordination will result in a CoordinationException being thrown.

The ordering of notifying Participants must follow the reverse order in which the Participants were registered.

Throws CoordinationException– If the Participant could not be registered with this Coordination. This exception should normally not be caught by the caller but allowed to be caught by the initiator of this Coordination.

SecurityException– If the caller does not have CoordinationPermission[PARTICIPATE] for this Coordination.

130.5.2.5   **public void end()**

☐ Terminate this Coordination normally.

If this Coordination has been pushed on the thread local Coordination stack of another thread, this method does nothing except throw a CoordinationException of type CoordinationException.WRONG_THREAD.

If this Coordination has been pushed on the thread local Coordination stack of this thread but is not the current Coordination, then the Coordinations on the thread local Coordination stack above this Coordination must be terminated and removed from the thread local Coordination stack before this Coordination is terminated. Each of these Coordinations, starting with the current Coordination, will be terminated normally . If the termination throws a CoordinationException, then the next Coordination on the thread local Coordination stack will be terminated as a failure with a failure cause of the thrown CoordinationException. At the end of this process, this Coordination will be the current Coordination and will have been terminated as a failure if any of the terminated Coordinations threw a CoordinationException

If this Coordination is the current Coordination, then it will be removed from the thread local Coordination stack.

If this Coordination is already terminated, a CoordinationException is thrown. If this Coordination was terminated as a failure, the failure cause will be the cause of the thrown CoordinationException.

Otherwise, this Coordination is terminated normally and then all registered Participants are notified. Participants should finalize any work associated with this Coordination. The successful return of this method indicates that the Coordination has terminated normally and all registered Participants have been notified of the normal termination.

It is possible that one of the Participants throws an exception during notification. If this happens, this Coordination is considered to have partially failed and this method must throw a CoordinationException of type CoordinationException.PARTIALLY_ENDED after all the registered Participants have been notified.

Throws CoordinationException– If this Coordination has failed, including timed out, or partially failed or this Coordination is on the thread local Coordination stack of another thread.

SecurityException– If the caller does not have CoordinationPermission[INITIATE] for this Coordination.

130.5.2.6   **public long extendTimeout(long timeMillis)**

timeMillis The time in milliseconds to extend the current timeout. If the initial timeout was specified as 0, no extension must take place. A zero must have no effect.

□ Extend the time out of this Coordination.

Participants can call this method to extend the timeout of this Coordination with at least the specified time. This can be done by Participants when they know a task will take more than normal time.

This method will return the new deadline if an extension took place or the current deadline if, for whatever reason, no extension takes place. Note that if a maximum timeout is in effect, the deadline may not be extended by as much as was requested, if at all. If there is no deadline, zero is returned. Specifying a timeout extension of 0 will return the existing deadline.

Returns The new deadline in milliseconds. If the specified time is 0, the existing deadline is returned. If this Coordination was created with an initial timeout of 0, no timeout is set and 0 is returned.

Throws CoordinationException– If this Coordination is terminated.

IllegalArgumentException– If the specified time is negative.

SecurityException– If the caller does not have CoordinationPermission[PARTICIPATE] for this Coordination.

130.5.2.7   **public boolean fail(Throwable cause)**

cause   The failure cause. The failure cause must not be null.

☐   Terminate this Coordination as a failure with the specified failure cause.

If this Coordination is already terminated, this method does nothing and returns false.

Otherwise, this Coordination is terminated as a failure with the specified failure cause and then all registered Participants are notified. Participants should discard any work associated with this Coordination. This method will return true.

If this Coordination has been pushed onto a thread local Coordination stack, this Coordination is not removed from the stack. The creator of this Coordination must still call end() on this Coordination to cause it to be removed from the thread local Coordination stack.

Returns   true if this Coordination was active and was terminated by this method, otherwise false.

Throws   SecurityException– If the caller does not have CoordinationPermission[PARTICIPATE] for this Coordination.

130.5.2.8   **public Bundle getBundle()**

☐   Returns the bundle that created this Coordination. This is the bundle that obtained the Coordinator service that was used to create this Coordination.

Returns   The bundle that created this Coordination.

Throws   SecurityException– If the caller does not have CoordinationPermission[ADMIN] for this Coordination.

130.5.2.9   **public Coordination getEnclosingCoordination()**

☐   Returns the Coordination enclosing this Coordination if this Coordination is on the thread local Coordination stack.

When a Coordination is pushed onto the thread local Coordination stack, the former current Coordination, if any, is the enclosing Coordination of this Coordination. When this Coordination is removed from the thread local Coordination stack, this Coordination no longer has an enclosing Coordination.

Returns   The Coordination enclosing this Coordination if this Coordination is on the thread local Coordination stack or null if this Coordination is not on the thread local Coordination stack or has no enclosing Coordination.

Throws   SecurityException– If the caller does not have CoordinationPermission[ADMIN] for this Coordination.

130.5.2.10   **public Throwable getFailure()**

☐   Returns the failure cause of this Coordination.

If this Coordination has failed, then this method will return the failure cause.

If this Coordination timed out, this method will return TIMEOUT as the failure cause. If this Coordination was active when the bundle that created it released the Coordinator service, this method will return RELEASED as the failure cause. If the Coordination was orphaned, this method will return ORPHANED as the failure cause.

Returns   The failure cause of this Coordination or null if this Coordination has not terminated as a failure.

Throws   SecurityException– If the caller does not have CoordinationPermission[INITIATE] for this Coordination.

130.5.2.11   **public long getId()**

□ Returns the id assigned to this Coordination. The id is assigned by the Coordinator service which created this Coordination and is unique among all the Coordinations created by the Coordinator service and must not be reused as long as the Coordinator service remains registered. The id must be positive and monotonically increases for each Coordination created by the Coordinator service.

Returns The id assigned to this Coordination.

130.5.2.12   **public String getName()**

□ Returns the name of this Coordination. The name is specified when this Coordination was created.

Returns The name of this Coordination.

130.5.2.13   **public List<Participant> getParticipants()**

□ Returns a snapshot of the Participants registered with this Coordination.

Returns A snapshot of the Participants registered with this Coordination. If no Participants are registered with this Coordination, the returned list will be empty. The list is ordered in the order the Participants were registered. The returned list is the property of the caller and can be modified by the caller.

Throws SecurityException– If the caller does not have CoordinationPermission[INITIATE] for this Coordination.

130.5.2.14   **public Thread getThread()**

□ Returns the thread in whose thread local Coordination stack this Coordination has been pushed.

Returns The thread in whose thread local Coordination stack this Coordination has been pushed or null if this Coordination is not in any thread local Coordination stack.

Throws SecurityException– If the caller does not have CoordinationPermission[ADMIN] for this Coordination.

130.5.2.15   **public Map<Class<?>,Object> getVariables()**

□ Returns the variable map associated with this Coordination. Each Coordination has a map that can be used for communicating between different Participants. The key of the map is a class, allowing for private data to be stored in the map by using implementation classes or shared data by using shared interfaces. The returned map is not synchronized. Users of the map must synchronize on the Map object while making changes.

Returns The variable map associated with this Coordination.

Throws SecurityException– If the caller does not have CoordinationPermission[PARTICIPANT] for this Coordination.

130.5.2.16   **public boolean isTerminated()**

□ Returns whether this Coordination is terminated.

Returns true if this Coordination is terminated, otherwise false if this Coordination is active.

130.5.2.17   **public void join(long timeMillis) throws InterruptedException**

timeMillis Maximum time in milliseconds to wait. Specifying a time of 0 will wait until this Coordination is terminated.

□ Wait until this Coordination is terminated and all registered Participants have been notified.

Throws InterruptedException– If the wait is interrupted.

IllegalArgumentException– If the specified time is negative.

SecurityException– If the caller does not have CoordinationPermission[PARTICIPATE] for this Coordination.

130.5.2.18   **public Coordination push()**

▫ Push this Coordination object onto the thread local Coordination stack to make it the current Coordination.

Returns This Coordination.

Throws CoordinationException– If this Coordination is already on the any thread's thread local Coordination stack or this Coordination is terminated.

SecurityException– If the caller does not have CoordinationPermission[INITIATE] for this Coordination.

130.5.3   **public class CoordinationException**
extends RuntimeException

Unchecked exception which may be thrown by a Coordinator implementation.

130.5.3.1   **public static final int ALREADY_ENDED = 4**

The Coordination has already terminated normally.

130.5.3.2   **public static final int ALREADY_PUSHED = 5**

The Coordination was already on a thread's thread local Coordination stack.

130.5.3.3   **public static final int DEADLOCK_DETECTED = 1**

Registering a Participant with a Coordination would have resulted in a deadlock.

130.5.3.4   **public static final int FAILED = 2**

The Coordination has terminated as a failure with Coordination.fail(Throwable). When this exception type is used, the getCause() method must return a non-null value.

130.5.3.5   **public static final int LOCK_INTERRUPTED = 6**

The current thread was interrupted while waiting to register a Participant with a Coordination.

130.5.3.6   **public static final int PARTIALLY_ENDED = 3**

The Coordination has partially ended.

130.5.3.7   **public static final int UNKNOWN = 0**

Unknown reason for this exception.

130.5.3.8   **public static final int WRONG_THREAD = 7**

The Coordination cannot be ended by the calling thread since the Coordination is on the thread local Coordination stack of another thread.

130.5.3.9   **public CoordinationException(String message,Coordination coordination,int type,Throwable cause)**

message The detail message for this exception.

coordination The Coordination associated with this exception.

cause The cause associated with this exception.

type The type of this exception.

▫ Create a new Coordination Exception with a cause.

Throws IllegalArgumentException– If the specified type is FAILED and the specified cause is null.

130.5.3.10   **public CoordinationException(String message,Coordination coordination,int type)**

message   The detail message for this exception.

coordination   The Coordination associated with this exception.

type   The type of this exception.

☐ Create a new Coordination Exception.

Throws   IllegalArgumentException– If the specified type is FAILED .

130.5.3.11   **public long getId()**

☐ Returns the id of the Coordination associated with this exception.

Returns   The id of the Coordination associated with this exception or -1 if no Coordination is associated with this exception.

130.5.3.12   **public String getName()**

☐ Returns the name of the Coordination associated with this exception.

Returns   The name of the Coordination associated with this exception or "<>" if no Coordination is associated with this exception.

130.5.3.13   **public int getType()**

☐ Returns the type for this exception.

Returns   The type of this exception.

130.5.4   public final class CoordinationPermission extends BasicPermission

A bundle's authority to create or use a Coordination.

CoordinationPermission has three actions: initiate, participate and admin.

Concurrency   Thread-safe

130.5.4.1   **public static final String ADMIN = "admin"**

The action string admin.

130.5.4.2   **public static final String INITIATE = "initiate"**

The action string initiate.

130.5.4.3   **public static final String PARTICIPATE = "participate"**

The action string participate.

130.5.4.4   **public CoordinationPermission(String filter,String actions)**

filter   A filter expression. Filter attribute names are processed in a case sensitive manner. A special value of "∗" can be used to match all coordinations.

actions   admin, initiate or participate (canonical order).

☐ Creates a new granted CoordinationPermission object. This constructor must only be used to create a permission that is going to be checked.

Examples:

```
(coordination.name=com.acme.∗)
(&(signer=\∗,o=ACME,c=US) (coordination.name=com.acme.∗))
(signer=\∗,o=ACME,c=US)
```

When a signer key is used within the filter expression the signer value must escape the special filter chars ('∗', '(', ')').

The name is specified as a filter expression. The filter gives access to the following attributes:

- signer - A Distinguished Name chain used to sign the exporting bundle. Wildcards in a DN are not matched according to the filter string rules, but according to the rules defined for a DN chain.
- location - The location of the exporting bundle.
- id - The bundle ID of the exporting bundle.
- name - The symbolic name of the exporting bundle.
- coordination.name - The name of the requested coordination.

Filter attribute names are processed in a case sensitive manner.

Throws IllegalArgumentException– If the filter has an invalid syntax.

130.5.4.5 **public CoordinationPermission(String coordinationName,Bundle coordinationBundle,String actions)**

coordinationName The name of the requested Coordination.

coordinationBun- The bundle which created the requested Coordination.
dle

actions admin, initiate or participate (canonical order).

☐ Creates a new requested CoordinationPermission object to be used by the code that must perform checkPermission. CoordinationPermission objects created with this constructor cannot be added to an CoordinationPermission permission collection.

130.5.4.6 **public boolean equals(Object obj)**

obj The object to test for equality with this CoordinationPermission object.

☐ Determines the equality of two CoordinationPermission objects. This method checks that specified permission has the same name and CoordinationPermission actions as this CoordinationPermission object.

Returns true if obj is a CoordinationPermission, and has the same name and actions as this CoordinationPermission object; false otherwise.

130.5.4.7 **public String getActions()**

☐ Returns the canonical string representation of the CoordinationPermission actions.

Always returns present CoordinationPermission actions in the following order: admin, initiate, participate.

Returns Canonical string representation of the CoordinationPermission actions.

130.5.4.8 **public int hashCode()**

☐ Returns the hash code value for this object.

Returns A hash code value for this object.

130.5.4.9 **public boolean implies(Permission p)**

p The requested permission.

☐ Determines if the specified permission is implied by this object.

This method checks that the filter of the target is implied by the coordination name of this object. The list of CoordinationPermission actions must either match or allow for the list of the target object to imply the target CoordinationPermission action.

Returns true if the specified permission is implied by this object; false otherwise.

130.5.4.10   **public PermissionCollection newPermissionCollection()**

□ Returns a new PermissionCollection object suitable for storing CoordinationPermission objects.

Returns A new PermissionCollection object.

130.5.5   public interface Coordinator

A Coordinator service coordinates activities between different parties.

A bundle can use the Coordinator service to create Coordination objects. Once a Coordination object is created, it can be pushed on the thread local Coordination stack to be an implicit parameter as the current Coordination for calls to other parties, or it can be passed directly to other parties as an argument. The current Coordination, which is on the top of the current thread's thread local Coordination stack, can be obtained with peek().

Any active Coordinations created by a bundle must be terminated when the bundle releases the Coordinator service. The Coordinator service must fail these Coordinations with the RELEASED exception.

A Participant can register to participate in a Coordination and receive notification of the termination of the Coordination.

The following example code shows a example usage of the Coordinator service.

```
void foo() {
  Coordination c = coordinator.begin("work", 0);
  try {
    doWork();
  } catch (Exception e) {
    c.fail(e);
  } finally {
    c.end();
  }
}
```

In the doWork method, code can be called that requires notification of the termination of the Coordination. The doWork method can then register a Participant with the Coordination.

```
void doWork() {
  if (coordinator.addParticipant(this)) {
    beginWork();
  } else {
    beginWork();
    finishWork();
  }
}

void ended(Coordination c) {
  finishWork();
}

void failed(Coordination c) {
  undoWork();
}
```

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

130.5.5.1   **public boolean addParticipant(Participant participant)**

participant The Participant to register with the current Coordination. The participant must not be null.

□ Register a Participant with the current Coordination.

If there is no current Coordination, this method does nothing and returns false.

Otherwise, this method calls Coordination.addParticipant(Participant) with the specified Participant on the current Coordination and returns true.

Returns false if there was no current Coordination, otherwise returns true.

Throws CoordinationException– If the Participant could not be registered with the current Coordination. This exception should normally not be caught by the caller but allowed to be caught by the initiator of this Coordination.

SecurityException– If the caller does not have CoordinationPermission[PARTICIPATE] for the current Coordination.

See Also Coordination.addParticipant(Participant)

130.5.5.2   **public Coordination begin(String name,long timeMillis)**

name The name of this coordination. The name does not have to be unique but must follow the symbolic-name syntax from the Core specification.

timeMillis Timeout in milliseconds. A value of 0 means no timeout is required. If the Coordination is not terminated within the timeout, the Coordinator service will fail the Coordination with a TIMEOUT exception.

□ Create a new Coordination and make it the current Coordination.

This method does that same thing as calling create(name, timeMillis).push()

Returns A new Coordination object

Throws IllegalArgumentException– If the specified name does not follow the symbolic-name syntax or the specified time is negative.

SecurityException– If the caller does not have CoordinationPermission[INITIATE] for the specified name and creating bundle.

130.5.5.3   **public Coordination create(String name,long timeMillis)**

name The name of this coordination. The name does not have to be unique but must follow the symbolic-name syntax from the Core specification.

timeMillis Timeout in milliseconds. A value of 0 means no timeout is required. If the Coordination is not terminated within the timeout, the Coordinator service will fail the Coordination with a TIMEOUT exception.

□ Create a new Coordination.

Returns The new Coordination object.

Throws IllegalArgumentException– If the specified name does not follow the symbolic-name syntax or the specified time is negative.

SecurityException– If the caller does not have CoordinationPermission[INITIATE] for the specified name and creating bundle.

130.5.5.4   **public boolean fail(Throwable cause)**

cause The failure cause. The failure cause must not be null.

□ Terminate the current Coordination as a failure with the specified failure cause.

If there is no current Coordination, this method does nothing and returns false.

Otherwise, this method returns the result from calling Coordination.fail(Throwable) with the specified failure cause on the current Coordination.

Returns false if there was no current Coordination, otherwise returns the result from calling Coordination.fail(Throwable) on the current Coordination.

Throws SecurityException– If the caller does not have CoordinationPermission[PARTICIPATE] for the current Coordination.

See Also Coordination.fail(Throwable)

130.5.5.5   **public Coordination getCoordination(long id)**

id The id of the requested Coordination.

☐ Returns the Coordination with the specified id.

Returns A Coordination having with specified id or null if no Coordination with the specified id exists, the Coordination with the specified id is terminated or the caller does not have CoordinationPermission[ADMIN] for the Coordination with the specified id.

130.5.5.6   **public Collection<Coordination> getCoordinations()**

☐ Returns a snapshot of all active Coordinations.

Since Coordinations can be terminated at any time, Coordinations in the returned collection can be terminated before the caller examines the returned collection.

The returned collection must only contain the Coordinations for which the caller has CoordinationPermission[ADMIN].

Returns A snapshot of all active Coordinations. If there are no active Coordinations, the returned list will be empty. The returned collection is the property of the caller and can be modified by the caller.

130.5.5.7   **public Coordination peek()**

☐ Returns the current Coordination.

The current Coordination is the Coordination at the top of the thread local Coordination stack. If the thread local Coordination stack is empty, there is no current Coordination. Each Coordinator service maintains thread local Coordination stacks.

This method does not alter the thread local Coordination stack.

Returns The current Coordination or null if the thread local Coordination stack is empty.

130.5.5.8   **public Coordination pop()**

☐ Remove the current Coordination from the thread local Coordination stack.

The current Coordination is the Coordination at the top of the thread local Coordination stack. If the thread local Coordination stack is empty, there is no current Coordination. Each Coordinator service maintains its own thread local Coordination stacks.

This method alters the thread local Coordination stack, if it is not empty, by removing the Coordination at the top of the thread local Coordination stack.

Returns The Coordination that was the current Coordination or null if the thread local Coordination stack is empty.

Throws SecurityException– If the caller does not have CoordinationPermission[INITIATE] for the current Coordination.

130.5.6   **public interface Participant**

A Participant participates in a Coordination.

A Participant can participate in a Coordination by registering itself with the Coordination. After successfully registering itself, the Participant is notified when the Coordination is terminated.

If a Coordination terminates normally, then all registered Participants are notified on their ended(Coordination) method. If the Coordination terminates as a failure, then all registered Participants are notified on their failed(Coordination) method.

Participants are required to be thread safe as notification can be made on any thread.

A Participant can only be registered with a single active Coordination at a time. If a Participant is already registered with an active Coordination, attempts to register the Participation with another active Coordination will block until the Coordination the Participant is registered with terminates. Notice that in edge cases the notification to the Participant that the Coordination has terminated can happen before the registration method returns.

Concurrency Thread-safe

130.5.6.1 **public void ended(Coordination coordination) throws Exception**

coordination The Coordination that has terminated normally.

☐ Notification that a Coordination has terminated normally.

This Participant should finalize any work associated with the specified Coordination.

Throws Exception– If this Participant throws an exception, the Coordinator service should log the exception. The Coordination.end() method which is notifying this Participant must continue notification of other registered Participants. When this is completed, the Coordination.end() method must throw a CoordinationException of type CoordinationException.PARTIALLY_ENDED.

130.5.6.2 **public void failed(Coordination coordination) throws Exception**

coordination The Coordination that has terminated as a failure.

☐ Notification that a Coordination has terminated as a failure.

This Participant should discard any work associated with the specified Coordination.

Throws Exception– If this Participant throws an exception, the Coordinator service should log the exception. The Coordination.fail(Throwable) method which is notifying this Participant must continue notification of other registered Participants.

130.6   References

[1] *OSGi Core Release 6*
http://www.osgi.org/Specifications/HomePage

132   Repository Service Specification

Version 1.1

132.1   Introduction

The guiding force behind the OSGi Specifications is a reusable component model. The *OSGi Core Release 6* provides a solid foundation for such a component model by providing a component collaboration framework with a comprehensive management model. The service specifications provide the abstract APIs to allow many different collaborations between components. This Repository Service Specification provides the capability to manage the external access to components and other resources.

Though the Repository service can be used as a standalone service to search and retrieve general binary artifacts, called resources, it is intended to be used in conjunction with the *Resolver Service Specification* on page 691.

The model of the Repository is based on the generic Requirement-Capability model defined in [3] *Resource API Specification*, this chapter relies on the definitions of the generic model.

132.1.1   Essentials
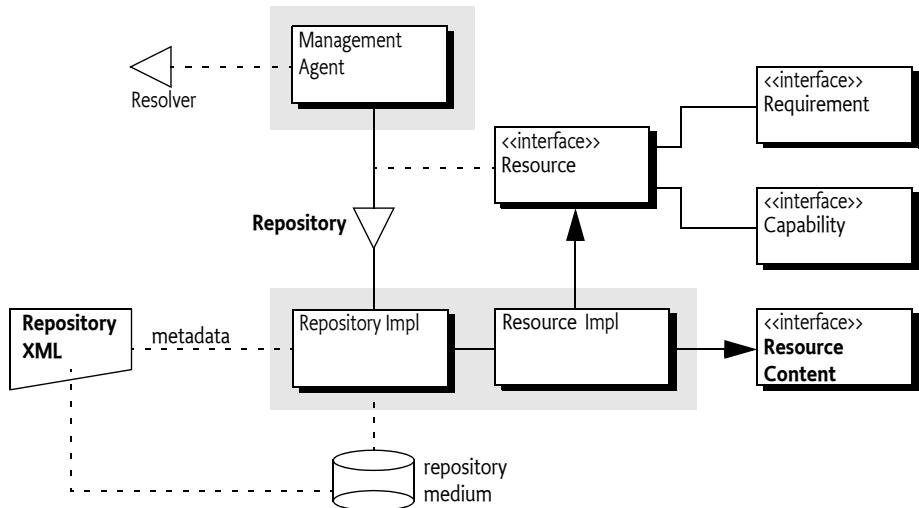
- *External* - Provide access to external components and resources.
- *Resolve* - The Repository API must be closely aligned with the Resolver API since they are intended to be used in conjunction.
- *Searching* - Support general queries.
- *Metadata* - Allow resources to provide content information.
- *Retrieval* - Allow the retrieval of Resources from remote locations.
- *Batching* - Repositories must be able to batch queries.
- *Distribution* - Allow Repositories to be defined with a simple storage scheme such that Repositories can be distributed on a removable media like a CD/DVD.
- *Mirroring* - Repositories must be able to support selecting a remote site based on the local situation.

132.1.2   Entities

- *Repository* - A facade to a (remote) set of resources described by capabilities.
- *Resource* - An artifact that has requirements that must be satisfied before it is available but provides capabilities when it becomes available.
- *Requirement* - An expression that asserts a capability.
- *Capability* - Describes a feature of the resource so that it can be required by a requirement.
- *Resource Content* - Provides access to the underlying bytes of the resource in the default format.

132.1.3   Synopsis

There are many different repositories available on the Internet or on fixed media. A repository can be made available to bundles by providing a Repository service. If such a bundle, for example a Management Agent performing a provisioning operation, finds that it has an unmatched requirement then it can query the repository services to find matching capabilities. The Repository service can implement the query in many different ways. It can ship the requirement to a remote side to be processed or it can process the query locally.

This specification also provides an XML schema that can be used to describe a Repository. Instances of this schema can be downloaded from a remote repository for local indexing or they can be stored for example on a DVD together with the resources.

132.2   Using a Repository

The Repository service provides an abstraction to a, potentially remote, set of resources. In the generic Capability-Requirement model, resources are modeled to declare capabilities and requirements. The primary purpose of a Repository is to enable a management agent that uses the Resolver API to leverage a wide array of repositories. This Repository service specification allows different Repository providers to be installed as bundles, and each bundle can register multiple Repository services. The Repository is sufficiently abstract to allow many different implementations.

Repository services are identified by a number of service properties:

- service.pid - A mandatory unique identity for this Repository service.
- service.description - An optional human readable name for this Repository.
- repository.url - Optional URLs to landing pages of the repository, if they exist.

In general, the users of the Repository service should aggregate all services in the service registry. This strategy allows the deployer to control the available Repositories. The following example, using Declarative Service annotations to show the dependencies on the service registry, shows how to aggregate the different Repository services.

```
List<Repository> repos = new CopyOnWriteArrayList<Repository>();
```

```
@Reference(
cardinality = ReferenceCardinality.MULTIPLE,
policy = ReferencePolicy.DYNAMIC)
void addRepository( Repository repo )    { repos.add(repo); }
void removeRepository( Repository repo ) { repos.remove(repo); }
```

To access a resource in a Repository service it is necessary to construct a requirement, pass this to the Repository service, and then use the returned capabilities to satisfy the resolver or to get the resource from the capability. The Repository then returns all matching capabilities. The requirement matches the capability if their namespaces match and the requirement's filter is absent or matches the attributes.

The findProviders(Collection) method takes a Collection of requirements. The reason for this collection is that it allows the caller to specify multiple requirements simultaneously so that Repositories can batch requests, the requirements in this collection are further unrelated. That is, they do not form an expression in any way. Multiple requirements as the parameter means that the result must be a map so that the caller can find out what requirement matched what capabilities. For example:

```
List<Capability> find( Requirement r ){
  List<Capability> result = new ArrayList<Capability>();

  for ( Repository repo : repos ) {
    Map<Requirement,Collection<Capability>> answer =
        repo.findProviders( Collections.singleton( r ) );
     result.addAll( answer.get( r ) );
  }
  return result;
}
```

Access to resources is indirect since the Repository returns capabilities. Each capability is declared in a resource and the getResource() method provides access to the underlying resource. Since each resource declares an osgi.identity capability it is possible to retrieve a resource from a repository if the identity name, type, and version are known. For example, to get a bundle resource:

```
Resource getResource( String type, String name, Version version ) {
  String filter = String.format(
    "(&(type=%s)(osgi.identity=%s)(version=%s))",
    type,
    name,
    version );

  RequirementBuilder builder = repo.newRequirementBuilder("osgi.identity");
  builder.addDirective("filter", filter);
  Requirement r = builder.build();

  List<Capability> capabilities = find( r );
  if ( capabilities.isEmpty() )
    return null;
  return capabilities.get( 0 ).getResource();
}
```

Resources that originate from Repository services must implement the RepositoryContent interface, this interface provides stream access to the default storage format. It is therefore possible to get the content with the following code.

```
InputStream getContent( String type, Stringname, Version version ) {
```

```
  Resource r = getResource( type, name, version );
  if ( r == null )
    return null;
  return ((RepositoryContent)r).getContent();
}
```

The getContent() method returns an Input Stream in the default format for that resource type. Resources from a Repository should also have one or more osgi.content capabilities that advertise the same resource in the same or different formats. The osgi.content capability has a number of attributes that provide information about the resource's download format:

- osgi.content - A unique SHA-256 for the content as read from the URL.
- url - A URL to the content.
- mime - An IANA MIME type for the content.
- size - Size in bytes of the content.

It is therefore possible to search for a specific MIME type and download that format. For example:

```
String getURL( String type, String name, Versionversion, String mime )
    throws Exception {
    Resource r = getResource( type, name, version );
    for ( Capability cap : r.getCapabilities( "osgi.content") ) {
        Map<String,Object> attrs = cap.getAttributes();
        String actual = (String) attrs.get("mime");
        if ( actual!=null && mime.equalsIgnoreCase( actual) ) {
            String url = (String) attrs.get( "url" );
            if ( url != null )
                return url;
        }
    }
    return null;
}
```

Since the osgi.content capability contains the SHA-256 digest as the osgi.content attribute it is possible to verify the download that it was correct.

Every resource has an osgi.identity capability. This namespace defines, in [2] *Framework Namespaces*, the possibility to add related resources, for example *javadoc* or *sources*. A resource then has informational requirements to osgi.identity capabilities; these requirements are marked with a classifier directive that holds the type of *relation*. The following example shows how it would be possible to find such a related resource:

```
InputStream getRelated(Resource resource,String classifier)
    throws Exception {
    for ( Requirement r : resource.getRequirements( "osgi.identity") ) {
        if ( classifier.equals( r.getDirectives().get( "classifier") ) ) {
            Collection<Capability> capabilities =
                repository.findProviders( Collections.singleton( r )).get( r );

            if ( capabilities.isEmpty())
                continue;

            Capability c = capabilities.iterator().next();
            Resource related = c.getResource();
            return ((RepositoryContent)related).getContent();
        }
```

```
        }
        return null;
}
```

132.2.1   Combining Requirements

In some cases it may be useful to find resources in the repository that satisfy criteria across multiple namespaces.

A simple Requirement object can contain a filter that makes assertions about capability attributes within a single namespace. So for example, a single requirement can state that a package org.example.mypkg must be exported in a version between 3.1 inclusive and 4.0 exclusive:

```
RequirementBuilder rb = repo.newRequirementBuilder("osgi.wiring.package");
String rf = "(&(osgi.wiring.package=org.example.mypkg)"
            + "(version>=3.1)(!(version>=4.0)))";
rb.addDirective("filter", rf);
Requirement r = rb.build();
```

This requirement contains three conditions on the osgi.wiring.package capability.

In some situations it may be needed to specify requirements that cover multiple namespaces. For example a bundle might be needed that exports the above package, but the bundle must also have the Apache License, Version 2.0 license. A resource's license is available as an attribute on the osgi.identity namespace. Constructing a constraint that combines requirements from multiple namespaces can be done by using an Expression Combiner, which can be obtained from the Repository service. The Repository service provides a findProviders(RequirementExpression) overload that can take a requirement expression and returns a Promise to a collection of matching resources.

```
RequirementBuilder lb = repo.newRequirementBuilder("osgi.identity");
String lf = "(license=http://opensource.org/licenses/Apache-2.0)";
lb.addDirective("filter", lf);

RequirementExpression expr = repo.getExpressionCombiner().and(
  lb.buildExpression(), rb.buildExpression());

Promise<Collection<Resource>> p = repo.findProviders(expr);

// Let findProviders() do its work async and update a ui component
// once the result is available
p.then(new Success<Collection<Resource>, Void>() {
  public Promise<Void> call(Promise<Collection<Resource>> resolved)
      throws Exception {
    ui.update(resolved.getValue());
    return null;
  }
});

// Instead of the async chain above its also possiblye to
// wait for the promise value synchronously:
//   Collection<Resource> resources = p.getValue();
```

For more details on OSGi Promises, see the *Promises Specification* on page 805.

132.3 Repository

A Repository service provides access to capabilities that satisfy a given requirement. A Repository can be the facade of a remote server containing a large amount of resources, a repository on removable media, or even a collection of bundles inside a ZIP file. A Repository communicates in terms of requirements and capabilities as defined in [3] *Resource API Specification*. This model is closely aligned with the *Resolver Service Specification* on page 691.

A Repository service must be registered with the service properties given in the following table.

Table 132.1 *Repository Service Properties*

Attribute	Opt	Type	Description
service.pid	mandatory	String	A globally unique identifier for this Repository.
service.description	optional	String	The Repository Name
repository.url	optional	String+	URLs related to this Repository.

The Repository implements the following methods:

- findProviders(Collection) - For each requirement find all the capabilities that match that requirement and return them as a Map<Requirement,Collection<Capability>>.
- findProviders(RequirementExpression) - Find all resources that match the requirement expression. The requirement expression is used to combine multiple requirements using the and, or and not operators.
- getExpressionCombiner() - Obtain an expression combiner. This expression combiner is used to produce requirement expressions from simple requirements or other requirement expressions.
- newRequirementBuilder(String) - Obtain a convenience builder for Requirement objects.

A Repository must not perform any namespace specific actions or matching. The Repository must therefore match a requirement to a capability with the following rules:

- The namespace must be identical, and
- The requirement's filter is absent or it must match the capability's attributes.

Resources originating from a Repository service must additionally:

- Implement the RepositoryContent interfaces, see *Repository Content* on page 582.
- Provide at least one osgi.content Capability, see *osgi.content Namespace* on page 582.

132.3.1 Repository Content

Resources originating from a Repository must implement the RepositoryContent interface. The purpose of this interface is to allow users of the Repositories access to an Input Stream that provides access to the resource.

The RepositoryContent interface provides a single method:

- getContent() - Return an Input Stream for the resource, if more than one osgi.content capability is present the content associated with the first capability is returned.

132.4 osgi.content Namespace

A resource is a logical concept, to install a resource in an environment it is necessary to get access to its *contents*. A resource can be formatted in different ways. It is possible to deliver a bundle as a JAR file, a Pack200 file, or some other format. In general, the RepositoryContent interface provides access to the default format.

The Repository can advertise the different formats with `osgi.content` capabilities. Each of those capabilities is identified with a unique SHA-256 checksum and has a URL for the resource in the specified format. The `size` and `mime` attributes provide information the download format, this can be used for selection. If more than one `osgi.content` capability is associated with a resource, the first capability must represent the default format. If the resource has a standard or widely used format (e.g., JAR for bundles and ESA for subsystems), and that format is provided as part of the repository, then that format should be the default format.

The `osgi.content` Namespace supports the attributes defined in the following table and Content-Namespace.

Table 132.2   osgi.content definition

Name	Kind	M/O	Type	Syntax	Description
osgi.content	CA	M	String	[0-9a-fA-F]{64}	The SHA-256 hex encoded digest for this resource
url	CA	M	String	<url>	The URL to the bytes. This must be an absolute URL.
size	CA	M	Long	[0-9]+	The size of the resource in bytes as it will be read from the URL.
mime	CA	M	String	<mime type>	An IANA defined MIME type for the format of this content.
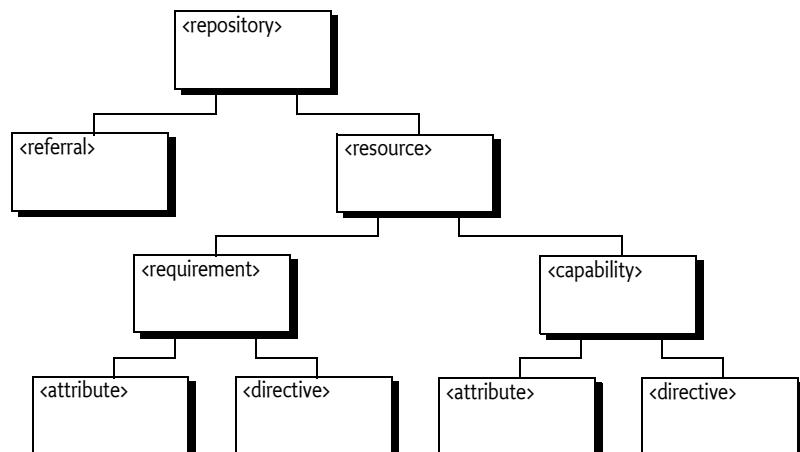
132.5   XML Repository Format

This is an optional part of the specification since the Repository interface does not provide access how the Repository obtains its information. However, the purpose of this part of the specification is to provide a commonly recognized format for interchanging Repository metadata.

This section therefore describes an XML schema to represent Repository content. It is expected that Internet based Repositories can provide such an XML file to clients. A Repository XML file can be used as a common interchange format between multiple Repository implementations.

The Repository XML describes a number of resources with their capabilities and requirements. Additionally the XML can refer to other Repository XML files. The XML Schema can be found at its XML namespace, see *XML Repository Schema* on page 586. The XML structure, which closely follows the Requirement-Capability model, is depicted in Figure 132.2.

Figure 132.2   XML Structure

The different elements are discussed in the following sections. All types are derived from the XML Schema types, see [4] *XML Schema Part 2: Data types Second Edition*.

132.5.1   Repository Element

The repository element is the root of the document. The repository element has the following child elements:

- referral* - Referrals to other repositories for a federated model, see *Referral Element* on page 584.
- resource* - Resource definitions, see *Resource Element* on page 584.

The repository element has the attributes defined in the following table.

Table 132.3   *repository element attributes*

Attribute	Type	Description
name	NCName	The name of this Repository. For informational purposes.
increment	long	Counter which increments every time the repository is changed. Can be used by clients to check for changes. The counter is not required to increase monotonically.

132.5.2   Referral Element

The purpose of the referral element is to allow a Repository to refer to other Repositories, allowing for federated Repositories. Referrals are applied recursively. However, this is not always desired. It is therefore possible to limit the depth of referrals. If the depth attribute is >= 1, the referred repository must be included but it must not follow any referrals from the referred repository. If the depth attribute is more than one, referrals must be included up to the given depth. Depths of referred repositories must also be obeyed, where referred repositories may reduce the effective depth but not increase it. For example if a top repository specifies a depth of 5 and a level 3 repository has a depth of 1 then the repository on level 5 must not be used. If not specified then there is no limit to the depth. Referrals that have cycles must be ignored, a resource of a given Repository must only occur once in a Repository.

The referral element has the attributes defined in the following table.

Table 132.4   *referral element attributes*

Attribute	Type	Description
depth	int	The max depth of referrals
url	anyURI	A URL to where the referred repository XML can be found. The URL can be absolute or relative to the URI of the current XML resource.

132.5.3   Resource Element

The resource element defines a Resource. The resource element has the following child elements:

- requirement* - The requirements of this resource, see *Requirement Element* on page 585.
- capability* - The capabilities of this resource, see *Capability Element* on page 584.

The Resource element has no attributes.

132.5.4   Capability Element

The capability element maps to a capability, it holds the attributes and directives. The capability element has the following child elements:

- directive* - The directives for the capability, see *Directive Element* on page 586.

- attribute∗ - The attributes for the capability, see *Attribute Element* on page 585.

The capability element has the attributes defined in the following table.

Table 132.5　　　　　*capability element attributes*

Attribute	Type	Description
namespace	token	The namespace of this capability

132.5.5　Requirement Element

The requirement element maps to a requirement, it holds the attributes and directives. The require-ment element has the following child elements:

- directive∗ - The directives for the requirement, see *Directive Element* on page 586.
- attribute∗ - The attributes for the requirement, see *Attribute Element* on page 585.

The requirement element has the attributes defined in the following table.

Table 132.6　　　　　*requirement element attributes*

Attribute	Type	Description
namespace	token	The namespace of this requirement

132.5.6　Attribute Element

An attribute element describes an attribute of a capability or requirement. Attributes are used to convey information about the Capability-Requirement. Attributes for the capability are used for matching the requirement's filter. The meaning of attributes is described with the documentation of the namespace in which they reside.

Attributes are optionally typed according to the [1] *Framework Module Layer* specification. The de-fault type is String, the value of the value attribute. However, if a type attribute is specified and it is not String then the value attribute must be converted according to the type attribute specifier. The syntax of the type attribute is as follows:

```
type    ::= list | scalar
list    ::= 'List<' scalar '>'    // no spaces between terminals
scalar  ::= 'String' | 'Version' | 'Long' | 'Double'
```

A list conversion requires the value to be broken in tokens separated by comma (',' \u002C). White-space around the list and around commas must be trimmed for non-String types. Each token must then be converted to the given type according to the scalar type specifier. The exact rules for the comma separated lists are defined in [1] *Framework Module Layer*, see *Bundle Capability Attributes*.

The conversion of value s, when scalar, must take place with the following methods:

- String - No conversion, use s
- Version - Version.parseVersion(s)
- Long - After trimming whitespace, Long.parseLong(s)
- Double - After trimming whitespace, Double.parseDouble(s)

The attribute element has the attributes defined in the following table.

Table 132.7　　　　　*attribute element attributes*

Attribute	Type	Description
name	token	The name of the attribute
value	string	The value of the attribute.
type		The type of the attribute, the syntax is outlined in the previ-ous paragraphs.

132.5.7 Directive Element

A directive element describes a directive of a capability or a requirement. Directives are used to convey information about the Capability-Requirement. The meaning of directives is described with the documentation of the namespace in which they reside.

The directive element has the attributes defined in the following table.

Table 132.8   *directive element attributes*

Attribute	Type	Description
name	token	The name of the attribute
value	string	The value of the attribute.

132.5.8 Sample XML File

The following example shows a very small XML file. The file contains one resource.

```
<repository name='OSGiRepository'
            increment='13582741'
            xmlns='http://www.osgi.org/xmlns/repository/v1.0.0'>
  <resource>

    <requirement namespace='osgi.wiring.package'>
      <directive name='filter' value=
                            '(&amp;(osgi.wiring.package=org.apache.commons.pool)(version&gt;=1.5.6))'/>
    </requirement>

    <requirement namespace='osgi.identity'>
      <directive name='effective' value='meta'/>
      <directive name='resolution' value='optional'/>
      <directive name='filter' value=
            '(&(version=1.5.6)(osgi.identity=org.acme.pool-src))'
        <directive name='classifier' value='sources'/>
    </requirement>

    <capability namespace='osgi.identity'>
      <attribute name='osgi.identity' value='org.acme.pool'/>
      <attribute name='version'type='Version' value='1.5.6'/>
      <attribute name='type' value='osgi.bundle'/>
    </capability>

    <capability namespace='osgi.content'>
      <attribute name='osgi.content' value='e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855'
      <attribute name='url' value='http://www.acme.com/repository/org/acme/pool/org.acme.pool-1.5.6.jar'/>
      <attribute name='size' type='Long' value='4405'/>
      <attribute name='mime' value='application/vnd.osgi.bundle'/>
    </capability>

    <capability namespace='osgi.wiring.bundle'>
      <attribute name='osgi.wiring.bundle' value='org.acme.pool'/>
      <attribute name='bundle-version' type='Version' value='1.5.6'/>
    </capability>

    <capability namespace='osgi.wiring.package'>
      <attribute name='osgi.wiring.package' value='org.acme.pool'/>
      <attribute name='version' type='Version' value='1.1.2'/>
      <attribute name='bundle-version' type='Version' value='1.5.6'/>
      <attribute name='bundle-symbolic-name' value='org.acme.pool'/>
      <directive name='uses' value='org.acme.pool,org.acme.util'/>
    </capability>

  </resource>
</repository>
```

132.6 XML Repository Schema

The namespace of this schema is:

```
http://www.osgi.org/xmlns/repository/v1.0.0
```

The schema for this namespace can be found at the location implied in its name. The recommended prefix for this namespace is repo.

```xml
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:repo="http://www.osgi.org/xmlns/repository/v1.0.0"
    targetNamespace="http://www.osgi.org/xmlns/repository/v1.0.0"
    elementFormDefault="unqualified"
    attributeFormDefault="unqualified"
    version="1.0.1">

    <element name="repository" type="repo:Trepository" />
    <complexType name="Trepository">
        <sequence>
            <choice minOccurs="0" maxOccurs="unbounded">
                <element name="resource" type="repo:Tresource" />
                <element name="referral" type="repo:Treferral" />
            </choice>
            <!-- It is non-deterministic, per W3C XML Schema 1.0:
            http://www.w3.org/TR/xmlschema-1/#cos-nonambig
            to use name space="##any" below. -->
            <any namespace="##other" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="name" type="string">
            <annotation>
                <documentation xml:lang="en">
                    The name of the repository. The name may contain
                    spaces and punctuation.
                </documentation>
            </annotation>
        </attribute>
        <attribute name="increment" type="long">
            <annotation>
                <documentation xml:lang="en">
                    An indication of when the repository was last changed. Client's can
                    check if a
                    repository has been updated by checking this increment value.
                </documentation>
            </annotation>
        </attribute>
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Tresource">
        <annotation>
            <documentation xml:lang="en">
                Describes a general resource with
                requirements and capabilities.
            </documentation>
        </annotation>
        <sequence>
            <element name="requirement" type="repo:Trequirement" minOccurs="0" maxOccurs="unbounded"/>
            <element name="capability" type="repo:Tcapability" minOccurs="1" maxOccurs="unbounded"/>
            <!-- It is non-deterministic, per W3C XML Schema 1.0:
            http://www.w3.org/TR/xmlschema-1/#cos-nonambig
            to use name space="##any" below. -->
            <any namespace="##other" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Treferral">
        <annotation>
            <documentation xml:lang="en">
                A referral points to another repository XML file. The
                purpose of this element is to create a federation of
                repositories that can be accessed as a single
                repository.
            </documentation>
        </annotation>
        <attribute name="depth" type="int" use="optional">
            <annotation>
```

```
                <documentation xml:lang="en">
                    The depth of referrals this repository acknowledges.
                </documentation>
            </annotation>
        </attribute>
        <attribute name="url" type="anyURI" use="required">
            <annotation>
                <documentation xml:lang="en">
                    The URL to the referred repository. The URL can be
                    absolute or relative from the given repository's
                    URL.
                </documentation>
            </annotation>
        </attribute>
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Tcapability">
        <annotation>
            <documentation xml:lang="en">
                A named set of type attributes and directives. A capability can be
                used to resolve a requirement if the resource is included.
            </documentation>
        </annotation>
        <sequence>
            <choice minOccurs="0" maxOccurs="unbounded">
                <element name="directive" type="repo:Tdirective" />
                <element name="attribute" type="repo:Tattribute" />
            </choice>
            <!-- It is non-deterministic, per W3C XML Schema 1.0:
            http://www.w3.org/TR/xmlschema-1/#cos-nonambig
            to use name space="##any" below. -->
            <any namespace="##other" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="namespace" type="string">
            <annotation>
                <documentation xml:lang="en">
                    Name space of the capability. Only requirements with the
                    same name space must be able to match this capability.
                </documentation>
            </annotation>
        </attribute>
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Trequirement">
        <annotation>
            <documentation xml:lang="en">
                A filter on a named set of capability attributes.
            </documentation>
        </annotation>
        <sequence>
            <choice minOccurs="0" maxOccurs="unbounded">
                <element name="directive" type="repo:Tdirective" />
                <element name="attribute" type="repo:Tattribute" />
            </choice>
            <!-- It is non-deterministic, per W3C XML Schema 1.0:
            http://www.w3.org/TR/xmlschema-1/#cos-nonambig
            to use name space="##any" below. -->
            <any namespace="##other" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <attribute name="namespace" type="string">
            <annotation>
                <documentation xml:lang="en">
                    Name space of the requirement. Only capabilities within the
                    same name space must be able to match this requirement.
                </documentation>
            </annotation>
        </attribute>
        <anyAttribute processContents="lax" />
    </complexType>

    <complexType name="Tattribute">
```

```
<annotation>
    <documentation xml:lang="en">
        A named value with an optional type that decorates
        a requirement or capability.
    </documentation>
</annotation>
<sequence>
    <any namespace="##any" processContents="lax" minOccurs="0"
        maxOccurs="unbounded" />
</sequence>
<attribute name="name" type="string">
    <annotation>
        <documentation xml:lang="en">
            The name of the attribute.
        </documentation>
    </annotation>
</attribute>
<attribute name="value" type="string">
    <annotation>
        <documentation xml:lang="en">
            The value of the attribute.
        </documentation>
    </annotation>
</attribute>
<attribute name="type" type="repo:TpropertyType" default="String">
    <annotation>
        <documentation xml:lang="en">
            The type of the attribute.
        </documentation>
    </annotation>
</attribute>
<anyAttribute processContents="lax" />
</complexType>

<complexType name="Tdirective">
    <annotation>
        <documentation xml:lang="en">
            A named value of type string that instructs a resolver
            how to process a requirement or capability.
        </documentation>
    </annotation>
    <sequence>
        <any namespace="##any" processContents="lax" minOccurs="0"
            maxOccurs="unbounded" />
    </sequence>
    <attribute name="name" type="string">
        <annotation>
            <documentation xml:lang="en">
                The name of the directive.
            </documentation>
        </annotation>
    </attribute>
    <attribute name="value" type="string">
        <annotation>
            <documentation xml:lang="en">
                The value of the directive.
            </documentation>
        </annotation>
    </attribute>
    <anyAttribute processContents="lax" />
</complexType>

<simpleType name="TpropertyType">
    <restriction base="string">
        <enumeration value="String" />
        <enumeration value="Version" />
        <enumeration value="Long" />
        <enumeration value="Double" />
        <enumeration value="List&lt;String&gt;" />
        <enumeration value="List&lt;Version&gt;" />
        <enumeration value="List&lt;Long&gt;" />
        <enumeration value="List&lt;Double&gt;" />
    </restriction>
</simpleType>
<attribute name="must-understand" type="boolean" default="false">
```

```
        <annotation>
            <documentation xml:lang="en">
                This attribute should be used by extensions to documents to require that
                the document consumer understand the extension. This attribute must be
                qualified when used.
            </documentation>
        </annotation>
    </attribute>
</schema>
```

132.7 Capabilities

Implementations of the Repository Service specification must provide the capabilities listed in this section.

132.7.1 osgi.implementation Capability

The Repository Service implementation bundle must provide the osgi.implementation capability with name osgi.repository. This capability can be used by provisioning tools and during resolution to ensure that a Repository Service implementation is present. The capability must also declare a uses constraint for the org.osgi.service.repository package and provide the version of this specification:

```
Provide-Capability: osgi.implementation;
        osgi.implementation="osgi.repository";
        uses:="org.osgi.service.repository";
        version:Version="1.1"
```

This capability must follow the rules defined for the *osgi.implementation Namespace* on page 685.

132.7.2 osgi.service Capability

The Repository Service implementation must provide a capability in the osgi.service namespace representing the Repository service. This capability must also declare a uses constraint for the org.osgi.service.repository package. For example:

```
Provide-Capability: osgi.service;
        objectClass:List<String>="org.osgi.service.repository.Repository";
        uses:="org.osgi.service.repository"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 685.

132.8 Security

132.8.1 External Access

Repositories in general will get their metadata and artifacts from an external source, which makes them an attack vector for a malevolent Bundle that needs unauthorized external access. Since a Bundle using a Repository has no knowledge of what sources the Repository will access it will be necessary for the Repository to implement the external access in a doPrivileged block. Implementations must ensure that callers cannot influence/modify the metadata in such a way that the getContent() method could provide access to arbitrary Internet resources. This could for example happen if:

- The implementation relies on the osgi.content namespace to hold the URL
- The attributes Map from the osgi.content Capability is modifiable

If the malevolent Bundle could change the osgi.content attribute it could change it to arbitrary URLs. This example should make it clear that Repository implementations must be very careful.

132.8.2   Permissions

Implementations of this specification will need the following minimum permissions.

```
ServicePermission[...Repository, REGISTER ]
SocketPermission[ ... carefully restrict external access...]
```

Users of this specification will need the following minimum permissions.

```
ServicePermission[...Repository, GET ]
```

132.9   org.osgi.service.repository

Repository Service Package Version 1.1.

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

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

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

132.9.1   Summary

- AndExpression - A RequirementExpression representing the and of a number of requirement expressions.
- ContentNamespace - Content Capability and Requirement Namespace.
- ExpressionCombiner - An ExpressionCombiner can be used to combine requirement expressions into a single complex requirement expression using the and, or and not operators.
- IdentityExpression - A RequirementExpression representing a requirement.
- NotExpression - A RequirementExpression representing the not (negation) of a requirement expression.
- OrExpression - A RequirementExpression representing the or of a number of requirement expressions.
- Repository - A repository service that contains resources.
- RepositoryContent - An accessor for the content of a resource.
- RequirementBuilder - A builder for requirements.
- RequirementExpression - The super interface for all requirement expressions.

132.9.2   public interface AndExpression
extends RequirementExpression

A RequirementExpression representing the and of a number of requirement expressions.

Since 1.1

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

132.9.2.1　　　　**public List<RequirementExpression> getRequirementExpressions()**

□　Return the requirement expressions that are combined by this AndExpression.

Returns　An unmodifiable list of requirement expressions that are combined by this AndExpression. The list contains the requirement expressions in the order they were specified when this requirement expression was created.

132.9.3　public final class ContentNamespace extends Namespace

Content Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace. All unspecified capability attributes are of type String and are used as arbitrary matching attributes for the capability. The values associated with the specified directive and attribute keys are of type String, unless otherwise indicated.

Concurrency　Immutable

132.9.3.1　　　　**public static final String CAPABILITY_MIME_ATTRIBUTE = "mime"**

The capability attribute that defines the IANA MIME Type/Format for this content.

132.9.3.2　　　　**public static final String CAPABILITY_SIZE_ATTRIBUTE = "size"**

The capability attribute that contains the size, in bytes, of the content. The value of this attribute must be of type Long.

132.9.3.3　　　　**public static final String CAPABILITY_URL_ATTRIBUTE = "url"**

The capability attribute that contains the URL to the content.

132.9.3.4　　　　**public static final String CONTENT_NAMESPACE = "osgi.content"**

Namespace name for content capabilities and requirements.

Also, the capability attribute used to specify the unique identifier of the content. This identifier is the SHA-256 hash of the content.

132.9.4　public interface ExpressionCombiner

An ExpressionCombiner can be used to combine requirement expressions into a single complex requirement expression using the and, or and not operators.

Since　1.1

Concurrency　Thread-safe

Provider Type　Consumers of this API must not implement this type

132.9.4.1　　　　**public AndExpression and(RequirementExpression expr1,RequirementExpression expr2)**

expr1　The first requirement expression to combine into the returned requirement expression.

expr2　The second requirement expression to combine into the returned requirement expression

□　Combine two RequirementExpressions into a requirement expression using the and operator.

Returns　An AndExpression representing an and of the specified requirement expressions.

132.9.4.2　　　　**public AndExpression and(RequirementExpression expr1,RequirementExpression expr2,RequirementExpression ... moreExprs)**

expr1　The first requirement expression to combine into the returned requirement expression.

expr2 The second requirement expression to combine into the returned requirement expression

moreExprs Optional, additional requirement expressions to combine into the returned requirement expression.

☐ Combine multiple RequirementExpressions into a requirement expression using the and operator.

Returns An AndExpression representing an and of the specified requirement expressions.

132.9.4.3 **public IdentityExpression identity(Requirement req)**

req The requirement to wrap in a requirement expression.

☐ Wrap a Requirement in an IdentityExpression. This can be useful when working with a combination of Requirements and RequirementExpresions.

Returns An IdentityExpression representing the specified requirement.

132.9.4.4 **public NotExpression not(RequirementExpression expr)**

expr The requirement expression to negate.

☐ Return the negation of a RequirementExpression.

Returns A NotExpression representing the not of the specified requirement expression.

132.9.4.5 **public OrExpression or(RequirementExpression expr1,RequirementExpression expr2)**

expr1 The first requirement expression to combine into the returned requirement expression.

expr2 The second requirement expression to combine into the returned requirement expression

☐ Combine two RequirementExpressions into a requirement expression using the or operator.

Returns An OrExpression representing an or of the specified requirement expressions.

132.9.4.6 **public OrExpression or(RequirementExpression expr1,RequirementExpression expr2,RequirementExpression ... moreExprs)**

expr1 The first requirement expression to combine into the returned requirement expression.

expr2 The second requirement expression to combine into the returned requirement expression

moreExprs Optional, additional requirement expressions to combine into the returned requirement expression.

☐ Combine multiple RequirementExpressions into a requirement expression using the or operator.

Returns An OrExpression representing an or of the specified requirement expressions.

132.9.5 **public interface IdentityExpression**
extends RequirementExpression

A RequirementExpression representing a requirement.

Since 1.1

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

132.9.5.1 **public Requirement getRequirement()**

☐ Return the Requirement contained in this IdentityExpression.

Returns The requirement contained in this IdentityExpression.

132.9.6 **public interface NotExpression**
extends RequirementExpression

A RequirementExpression representing the not (negation) of a requirement expression.

Since	1.1
Concurrency	Thread-safe
Provider Type	Consumers of this API must not implement this type

132.9.6.1   **public RequirementExpression getRequirementExpression()**

□ Return the requirement expression that is negated by this NotExpression.

Returns   The requirement expression that is negated by this NotExpression.

132.9.7   public interface OrExpression
extends RequirementExpression

A RequirementExpression representing the or of a number of requirement expressions.

Since	1.1
Concurrency	Thread-safe
Provider Type	Consumers of this API must not implement this type

132.9.7.1   **public List<RequirementExpression> getRequirementExpressions()**

□ Return the requirement expressions that are combined by this OrExpression.

Returns   An unmodifiable list of requirement expressions that are combined by this OrExpression. The list contains the requirement expressions in the order they were specified when this requirement expression was created.

132.9.8   public interface Repository

A repository service that contains resources.

Repositories may be registered as services and may be used as by a resolve context during resolver operations.

Repositories registered as services may be filtered using standard service properties.

Concurrency	Thread-safe
Provider Type	Consumers of this API must not implement this type

132.9.8.1   **public static final String URL = "repository.url"**

Service property to provide URLs related to this repository.

The value of this property must be of type String, String[], or Collection<String>.

132.9.8.2   **public Map<Requirement,Collection<Capability>> findProviders(Collection<? extends Requirement> requirements)**

requirements   The requirements for which matching capabilities should be returned. Must not be null.

□ Find the capabilities that match the specified requirements.

Returns   A map of matching capabilities for the specified requirements. Each specified requirement must appear as a key in the map. If there are no matching capabilities for a specified requirement, then the value in the map for the specified requirement must be an empty collection. The returned map is the property of the caller and can be modified by the caller. The returned map may be lazily populated, so calling size() may result in a long running operation.

132.9.8.3   **public Promise<Collection<Resource>> findProviders(RequirementExpression expression)**

expression   The RequirementExpression for which matching capabilities should be returned. Must not be null.

□ Find the resources that match the specified requirement expression.

Returns A promise to a collection of matching `Resources`. If there are no matching resources, an empty collection is returned. The returned collection is the property of the caller and can be modified by the caller. The returned collection may be lazily populated, so calling `size()` may result in a long running operation.

Since 1.1

132.9.8.4 **public ExpressionCombiner getExpressionCombiner()**

☐ Return an expression combiner. An expression combiner can be used to combine multiple requirement expressions into more complex requirement expressions using and, or and not operators.

Returns An `ExpressionCombiner`.

Since 1.1

132.9.8.5 **public RequirementBuilder newRequirementBuilder(String namespace)**

namespace The namespace for the requirement to be created.

☐ Return a new `RequirementBuilder` which provides a convenient way to create a requirement.

For example:

```
 Requirement myReq = repository.newRequirementBuilder("org.foo.ns1").
   addDirective("filter", "(org.foo.ns1=val1)").
   addDirective("cardinality", "multiple").build();
```

Returns A new requirement builder for a requirement in the specified namespace.

Since 1.1

132.9.9 **public interface RepositoryContent**

An accessor for the content of a resource. All Resource objects which represent resources in a Repository must implement this interface. A user of the resource can then cast the Resource object to this type and then obtain an `InputStream` to the content of the resource.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

132.9.9.1 **public InputStream getContent()**

☐ Returns a new input stream to the content of this resource. The content is represented on the resource through the `osgi.content` capability. If more than one such capability is associated with the resource, the first such capability is returned.

Returns A new input stream for associated content.

132.9.10 **public interface RequirementBuilder**

A builder for requirements.

Since 1.1

Provider Type Consumers of this API must not implement this type

132.9.10.1 **public RequirementBuilder addAttribute(String name,Object value)**

name The attribute name.

value The attribute value.

☐ Add an attribute to the set of attributes.

Returns This requirement builder.

132.9.10.2   **public RequirementBuilder addDirective(String name,String value)**

name The directive name.

value The directive value.

☐ Add a directive to the set of directives.

Returns This requirement builder.

132.9.10.3   **public Requirement build()**

☐ Create a requirement based upon the values set in this requirement builder.

Returns A requirement created based upon the values set in this requirement builder.

132.9.10.4   **public IdentityExpression buildExpression()**

☐ Create a requirement expression for a requirement based upon the values set in this requirement builder.

Returns A requirement expression created for a requirement based upon the values set in this requirement builder.

132.9.10.5   **public RequirementBuilder setAttributes(Map<String,Object> attributes)**

attributes The map of attributes.

☐ Replace all attributes with the attributes in the specified map.

Returns This requirement builder.

132.9.10.6   **public RequirementBuilder setDirectives(Map<String,String> directives)**

directives The map of directives.

☐ Replace all directives with the directives in the specified map.

Returns This requirement builder.

132.9.10.7   **public RequirementBuilder setResource(Resource resource)**

resource The resource.

☐ Set the Resource.

A resource is optional. This method will replace any previously set resource.

Returns This requirement builder.

132.9.11   public interface RequirementExpression

The super interface for all requirement expressions. All requirement expressions must extend this interface.

Since 1.1

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

132.10   References

[1]   *Framework Module Layer*
OSGi Core, Chapter 3 Module Layer

[2]   *Framework Namespaces*
        OSGi Core, Chapter 8, osgi.identity Namespace

[3]   *Resource API Specification*
        OSGi Core, Chapter 6 Resource API Specification

[4]   *XML Schema Part 2: Data types Second Edition*
        http://www.w3.org/TR/xmlschema-2/

132.11   Changes

- Introduction of *Requirement Expressions* to represent combined requirements.
- Introduction of *Requirement Builder* to facilitate the creation of Requirement objects.
- Changes to default content for the *Repository Content* API and `osgi.content` capability.
- Introduction of `osgi.implementation` and `osgi.service` *capabilities*.

133 Service Loader Mediator Specification

Version 1.0

133.1 Introduction

Java SE 6 introduced the *Service Loader*, a simple service-provider loading facility, that attempted to unify the different ad-hoc mechanisms used by Java's many factories and builders. The design allows a JAR to advertise the name of one or more embedded classes that implement a given interface and consumers to obtain instances of these implementation classes through the Service Loader API.

Though the Service Loader is about extensibility, its own design is closed and therefore not extendable. It does not support a provider model that would allow different ways of finding interface implementations; its classes are final and its policy is fixed. Unfortunately, the Service Loader's fixed design uses a non-modular class loading policy; it defines its visibility scope with a class loader, which in general requires full visibility of the application's class path. The Service Loader can therefore in OSGi not find implementations from other bundles. Additionally, the Service Loader also does not enforce a life cycle; objects are handed out forever.

Since the Service Loader is the only standardized plugin mechanism in the JRE it is necessary that the mechanism is supported in OSGi with as few changes as possible from the consumer's authors. This specification therefore defines a *mediator* that ensures that the Service Loader is useful in an OSGi Framework, allowing programs that leverage the Service Loader to be used in OSGi frameworks almost as-is.

133.1.1 Essentials

- *Compatibility* - Allow JARs that run in a classic Java SE environment that leverage the Service Loader to run in OSGi with only manifest modifications.
- *Services* - Register services for Service Provider bundles that opt-in.
- *Security* - Enforce service permissions for the Service Loader objects.
- *Life Cycle* - Manage the life cycle mismatch between OSGi bundles and the Service Loader's create only model.
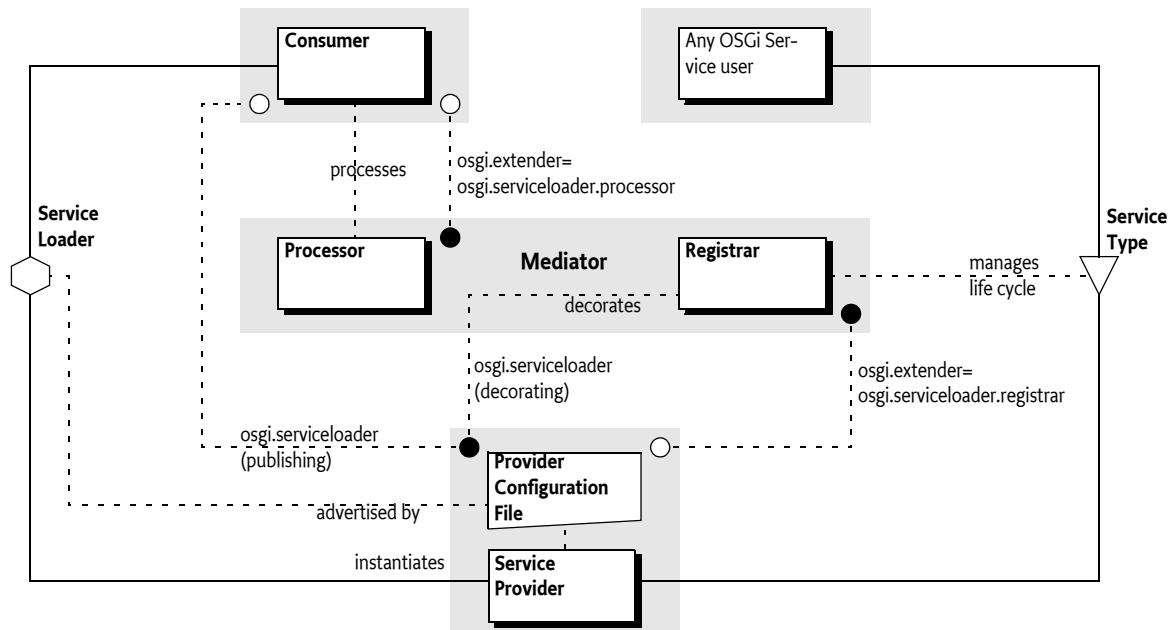
133.1.2 Entities

- *Service Loader* - An API in Java SE that allows a Consumer to find an implementation of a Service Type from a Service Provider by searching a class loader for Service Providers.
- *Service Type* - The interface or class that the Service Provider must implement/extend.
- *Provider Configuration File* - A resource in the META-INF/services directory that has the fully qualified name of the Service Type and contains one ore more fully qualified names of Service Providers.
- *Service Provider* - An implementation class that implements or extends the Service Type.
- *Consumer* - A class that uses the Java SE Service Loader inside an OSGi framework.
- *Mediator* - An extender that mediates between Consumer bundles, the Service Loader API, and Service Provider bundles in an OSGi environment. It consists of a Processor and a Registrar.

- *Processor* - Modifies a bundle that uses the Service Loader API so that it works in an OSGi environment.
- *Registrar* - Registers services on behalf of a bundle that contains Service Providers.

Figure 133.1   *Entities*

133.1.3   Synopsis

This specification defines two different functions that are provided by a Mediator extender:

- Register OSGi services for each Service Provider.
- Allow Consumers that uses the Service Loader API to access Service Providers from other bundles that would normally not be visible from a bundle.

A Service Provider bundle can provide access to all its Service Providers through OSGi services by declaring a requirement on the osgi.serviceloader.registrar extender. This requirement activates a Mediator to inspect the osgi.serviceloader capabilities. If no register directive is used then all Service Providers for the given Service Type must be registered. Otherwise, each capability can select one Service Provider with the register directive. The fully qualified name selects a specific Service Provider, allowing different Service Providers to be registered with different service properties. The Mediator will then register an OSGi service factory for each selected capability. The osgi.serviceloader capability's attributes are used to decorate the OSGi service registration with service properties. The service factory returns a new instance for each service get.

Consumers are classes that use the Service Loader API to find Service Provider instances. Since the Service Loader API requires full visibility the Service API fails to work inside an OSGi bundle. A osgi.serviceloader.processor extender, which is the Mediator, processes bundles that require this capability by modifying calls to the Service Loader API to ensures that the Service Loader has visibility to published Service Providers.

A Consumer's bundle by default receives visibility to all published Service Providers. Service Providers are published when a bundle declares one or more osgi.serviceloader capabilities for a Service Type. If the Consumer has an osgi.serviceloader requirement for the given Service Type then the Mediator must only expose the bundles that are wired to those requirements and for each bundle provide all its Service Providers.

133.2   Java Service Loader API

Java is quite unique with its focus on separation of *specification* and *implementation*. Virtually all Java Specification Requests (JSR) provide a specification that can be implemented independently by different parties. Though this is one of the industry's best practices it raises a new problem: how to find the implementation available in a Java environment from only the *Service Type*. A Service Type is usually an interface but a base class can also be used.

Finding a Service Provider (the implementation class) from a Service Type is the so called *instance coupling* problem. The use of Service Types removed the type coupling between the Consumer of the contract and the *Service Provider* of the contract (the implementation) but to make things work there is a need of at least one place where the Service Provider is instantiated. The very large number of factories in Java reflects that this is a very common problem.

The general pattern for factories to find Service Providers was to search the class loaders for classes with constant names, varying the package names, often using System properties to extend the different areas to be sought. Though a general pattern based on class loading tricks emerged in the Java VM and application programs, all these patterns differed in details and places where they looked. This became harder and harder to maintain and often caused unexpected instances to be found.

The java.util.ServiceLoader class was therefore first introduced in Java SE 6 to provide a generic solution to this problem, see [1] *Java Service Loader API*. With this API Service Providers of a specification can now *advertise* their availability by creating a *Provider Configuration File* in their JAR in the META-INF/services directory. The name of this resource is the fully qualified name of the Service Type, the Service Provider provides when instantiated.

The Provider Configuration File contains a number of lines with comments or a class name that implements/extends the Service Type. For example:

```
org.example.Foo
```

A Service Provider must then advertise itself like:

```
META-INF/services/org.example.Foo:
    # Foo implementation
    org.acme.impl.FooImplementation
```

The Service Loader API finds all advertisers by constructing the name of the Provider Configuration File from the Service Type and then calling the getResources method on the provided class loader. This returns an enumeration of URLs to the advertisements. It then parses the contents of the resources; that will provide it with a list of Service Providers for the sought Service Type without duplicates. The API will return an iterator that will instantiate an object for the next available Service Provider.

To find the Configuration files for a given Service Type, the Service Loader uses a class loader. The Consumer can select the following class loaders:

- A given class loader as an argument in the call to the constructor
- The Thread Context Class Loader (TCCL)
- The system loader (when null is passed or no TCCL is set)

The class loader restricts the visibility of the Service Loader to only the resources to which the class loader has visibility. If the Service Loader has no access to the advertisement of a Service Provider then it cannot detect it and it will thus not be found.

The Service Provider is loaded from the given class loader, however, the Class.forName method is used, which stores it in the cache of the initiating class loader. This means that Service Providers are

not garbage collected as long as there is a resolved bundle that used the Service Loader to get that Service Provider.

In the Service Loader API, the class does not have to originate from the same JAR file as the advertisement. In OSGi this is more restricted, the advertisement must come from the same bundle or must be explicitly imported.

For example, to load a Foo instance the following code could be used:

```
ServiceLoader<Foo> sl =
    ServiceLoader.load( Foo.class );
Iterator<Foo> it = sl.iterator();
if ( it.hasNext() ) {
    Foo foo = it.next();
    ...
}
```

Though the Service Loader API is about extensibility and contract based programming it is in itself not extendable nor replaceable. The ServiceLoader class is final, it comes from a sealed JAR, and is in a java package. It also does not provide an API to provide alternate means to find implementations for a Service Type.

133.3 Consumers

Consumers are classes that are not OSGi aware and directly use the *Service Loader API*. The Service Loader has a non-modular design and Consumers therefore run into many issues when running in an OSGi framework. Consumers should therefore in general be converted to use the OSGi service layer since this solves the visibility issues, life cycle impedance mismatch, and other problems. The Consumer part of this specification is therefore a last resort to use when existing code uses the Service Loader API and cannot be modified to leverage the OSGi service layer.

133.3.1 Processing

The Service Loader Mediator can *process* the Consumer by modifying calls to the Service Loader API. This specification does not detail how the Mediator ensures that the Consumer has visibility to other Service Providers. However, a Mediator could for example set an appropriate Thread Context Class Loader during the call to the Service Loader's constructor by weaving the Consumer's byte codes.

133.3.2 Opting In

Processing is an opt-in process, the Consumer bundle must declare that it is willing to be processed. The opt-in is handled by a requirement to the osgi.serviceloader.processor extender. This requirement must have a single cardinality (the default) since the Mediator uses the wiring to select the Consumer to process when multiple Mediators are present.

For example, the following requirement in a manifest enables a bundle to be processed:

```
Require-Capability:
    osgi.extender;
        filter:="(&(osgi.extender=osgi.serviceloader.processor)
                   (version>=1.0)(!(version>=2.0)))"
```

If the extender osgi.serviceloader.processor requirement is satisfied then the wired Mediator must process the Consumer.

The Mediator must give visibility to all bundles with *published* Service Providers unless the Consumer restricts the visibility by having osgi.serviceloader requirements. Bundles publish a Service

Type, meaning all their Service Providers for that type, by having at least one `osgi.serviceloader` capability for that Service Type.

133.3.3   Restricting Visibility

A Consumer's bundle can restrict its visibility to certain bundles by declaring an `osgi.serviceloader` requirement for each Service Type it wants to use. Only bundles wired from those requirement provide their advertised Service Providers. If no such requirements are declared then all bundles with the published Service Type become available.

The cardinality can be used to select a single Service Provider's bundle or multiple bundles if it needs to see all Service Provider bundles. The requirement can be made optional if the Consumer's bundle can work also when no Service Provider bundle is available. See *osgi.serviceloader Namespace* on page 609 for more details.

For example, a requirement that restricts visibility to the `org.example.Foo` Service Providers could look like:

```
Require-Capability:
    osgi.serviceloader;
    filter:="(osgi.serviceloader=org.example.Foo)";
    cardinality:=multiple
```

In this example, any bundle that publishes the `org.example.Foo` Service Type will contribute its Service Providers.

Visibility can also be restricted to bundles that publish with capability's attributes. Any bundle that has at least one matching capability will then be able to contribute all its Service Providers. For example, the following example selects only bundles that have the `classified` property set:

```
osgi.serviceloader; filter:="(classified=*)"
```

With Service Registrations, see *Registering Services* on page 606, the capability can discriminate between multiple Service Providers in the same bundle. The Service Loader API does not have this feature: any wired requirement has visibility to all Service Providers in the wired bundle, regardless of the `registered` directive.

133.3.4   Life Cycle Impedance Mismatch

A Consumer can only see Service Provider instances of bundles that are active during the time the next instance is created. That is, the Mediator must treat the life cycle of the Service Provider as if it was a service. However, the Service Loader implementations perform extensive class loader techniques and cache results. The exact life cycle of the Service Provider bundle with respect to the Consumer is therefore impossible to enforce.

The Service Loader API does not have a life cycle, objects are assumed to stay alive during the duration of the VM's process and due to the use of `Class.forName` in the Service Loader implementations. Therefore a Mediator should refresh a Consumer bundle when it is using a Service Provider and that Service Provider's bundle becomes stopped otherwise long running applications can run out of memory when bundles are regularly updated.

133.3.5   Consumer Example

A legacy JAR for which there is no more source code uses the Service Loader API to get access to `com.example.Codec` instances through the Service Loader API.

It is wrapped in a bundle that then has the following manifest:

```
Manifest-Version:     1.0
Bundle-ManifestVersion: 2
```

```
Bundle-SymbolicName:     com.example.impl
Bundle-Version:          23.98.1.v199101232310.02011
Import-Package:          com.example; version=3.45
Bundle-ClassPath:        legacy.jar
```

The manifest must then declare that the bundle must be processed, this is triggered by requiring the osgi.serviceloader.processor extender:

```
Require-Capability:
  osgi.extender;
    filter:="(&(osgi.extender=osgi.serviceloader.processor)
             (version>=1.0)(!(version>=2.0)))"
```

With this manifest, the Consumer bundle has full visibility to all Service Provider bundles that are published. The following lines can be added to restrict the visibility to codecs that have support for WAVE formats (although all Service Providers in that bundle will be visible to the consumer).

```
,
    osgi.serviceloader;
     filter:="(&(format=WAVE)(osgi.serviceloader=com.example.Codec))"
```

133.4   Service Provider Bundles

A *Service Provider bundle* is a bundle that contains one or more Service Providers that are usable by the Service Loader API. This section shows how Service Provider bundles should be constructed and what options they have.

133.4.1   Advertising

Service Providers are implementation classes that are *advertised* under a Service Type according to the rules in the Service Loader API. A Service Provider is advertised with a *Provider Configuration File* in a JAR. In an OSGi environment the Service Provider must reside in the same bundle as the advertisement or be imported. A single Provider Configuration File can contain multiple Service Providers. See *Java Service Loader API* on page 601.

133.4.2   Publishing the Service Providers

Service Providers can be used in two different scenarios:

- A Service Provider can be used by a processed Consumer as a Service Type, or
- It can be registered as a service.

A Service Type must be *published* to allow its use it in these scenarios. Publishing a Service Type consists of providing one or more osgi.serviceloader capabilities for an advertised Service Type, see *osgi.serviceloader Namespace* on page 609. These osgi.serviceloader capabilities must specify a fully qualified class name of the Service Type, there is no wildcarding allowed. Therefore, publishing a service implicitly makes all corresponding Service Providers available to Consumers.

If a bundle does not provide osgi.serviceloader capabilities then it does not publish any Service Providers and its Service Providers can therefore not be used by Consumers. They can then also not be registered as OSGi services, see *OSGi Services* on page 605. Tools can use the advertisement of the Service Provider in the JAR to automatically generate the osgi.serviceloader capabilities in the manifest.

For example, the following capability publishes all the Service Providers in its bundle that advertise the com.example.Codec interface:

```
Provide-Capability:
  osgi.serviceloader;
    osgi.serviceloader=com.example.Codec;
    uses:="com.example"
```

A Service Provider bundle must not require the osgi.serviceloader.processor extender unless it needs to be processed; publishing a Service Type is sufficient to allow Consumers to use the published Service Types.

133.4.3 OSGi Services

The Service Provider can have its osgi.serviceloader capabilities be registered as services that provide instances from the Service Providers. For this, the Service Provider bundle must require the osgi.serviceloader.registrar extender, which is the Mediator. For example:

```
Require-Capability:
    osgi.extender;
        filter:="(&(osgi.extender=osgi.serviceloader.registrar)
                   (version>=1.0)(!(version>=2.0)))"
```

The registrar must then inspect each osgi.serviceloader capability and register an associated OSGi Service for each Service Provider *selected* by that capability. A Service Provider is selected when:

- The capability has no register directive, or
- The register directive matches the fully qualified name of the Service Provider.

A register directive selects a Service Provider if it contains the fully qualified name of the Service Provider, that is, the implementation class. Selection only works for services, Consumer will always see all Service Providers regardless of the register directive due to limitations in the Service Loader API.

For example, the following manifest selects all Service Providers of the com.example.Foo Service Type since no register directive is present:

```
Provide-Capability:
    osgi.serviceloader;
        uses:="com.example";
        osgi.serviceloader=com.example.Foo
```

Selected Service Providers must be registered as defined in *Registering Services* on page 606, with the capability's attributes as *decorating* service properties. Private service properties (attributes that start with a full stop ('.' \u002E) and the defined capability attributes in the osgi.serviceloader namespace are not registered as service properties.

The following example would register the format service property but not the .hint service property for the com.acme.impl.WaveFoo Service Provider.

```
    osgi.serviceloader;
        osgi.serviceloader=com.example.Foo;
        uses:="com.example";
        format=WAVE;
        .hint=E5437Qy7;
        register:="com.acme.impl.WaveFoo"
```

The Mediator must only register OSGi services for selected Service Providers; the Service Provider bundle can therefore decide not to register certain Service Providers and register them with another mechanism, for example Declarative Services or in a bundle activator.

Since the Mediator must use the bundle context of the Service Provider to register the OSGi service the Service Provider bundle must have the proper Service Permission REGISTER for the Service Type.

133.4.4 Service Provider Example

A Foo Codecs JAR needs to be ported to OSGi, it provides a Service Provider for the org.example.Codec Service Type. In this example the JAR is given a new manifest:

```
Manifest-Version:       1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName:    com.example.foo.codecs
Import-Package:         com.example; version=3.45
```

To ensure that the bundle opts in to registering its services it must require the osgi.serviceloader.registrar extender.

```
Require-Capability:
    osgi.extender;
        filter:="(&(osgi.extender=osgi.serviceloader.registrar)
                  (version>=1.0)(!(version>=2.0)))"
```

To publish two Service Providers for the same type, two capabilities must be declared:

```
Provide-Capability:
    osgi.serviceloader;
        osgi.serviceloader="com.example.Codec";
        format:List<String>="WAVE,WMF";
      register:="com.acme.impl.FooWaveCodec";
      uses:="com.example,org.apache.common.codecs",
    osgi.serviceloader;
        osgi.serviceloader="com.example.Codec";
        format:List<String>=SINUS;
      register:="com.acme.impl.sinus.FooSinusCodec";
      uses:="com.example"
```

This example implicitly publishes the Service Type com.example.Codec multiple times with different attributes. Consumers that match any of these capabilities will however have visibility to all Service Providers since the Service Loader API cannot discriminate between different Service Providers from the same bundle.

133.5 Service Loader Mediator

A Mediator is the osgi.serviceloader.processor and osgi.serviceloader.registrar extender bundle that has the following responsibilities:

- It registers selected Service Providers as OSGi services.
- It processes any Consumers so that Service Loader API calls have proper visibility to published Service Provider bundles.

133.5.1 Registering Services
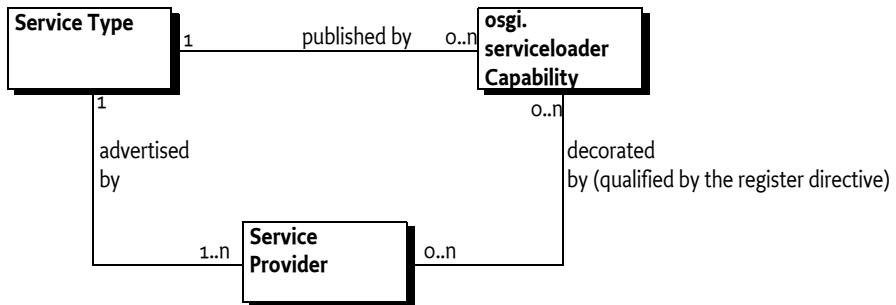
The Mediator must track bundles that are wired to its osgi.extender=osgi.serviceloader.registrar capability. These are called the *managed* bundles. For all managed bundles the Mediator must enumerate all osgi.serviceloader capabilities and register *selected* Service Providers as OSGi services. A Service Provider is selected by an osgi.serviceloader capability when:

- The advertised Service Type matches the corresponding `osgi.serviceloader` capability's Service Type, and
- The register directive is absent, or
  - The `register` directive contains the fully qualified name of the Service Provider.

An `osgi.serviceloader` capability that selects a Service Provider is said to *decorate* that Service Provider. A capability can decorate multiple Service Providers of the same Service Type and the same Service Provider can be decorated by different capabilities. Figure 133.2 depicts the resulting relations and their cardinalities since the relations are non-trivial.

Figure 133.2   *Cardinality Service Type*

The OSGi service for each selected Service Provider must be registered under the advertised Service Type of the Service Provider, which must match the Service Type specified in the capability.

133.5.2   OSGi Service Factory

The Mediator must register an OSGi service factory with the bundle context of the Service Provider's bundle. The OSGi service factory must be implemented such that it creates a new instance for each bundle that gets the service. This behavior is similar, though not quite identical, to the `ServiceLoader.load()` method that gives each consumer a separate instance of the service. The difference is that different users inside a bundle will share the same instance.

Each service registration is controlled by a decorating `osgi.serviceloader` capability. The attributes on this capability must be registered with the OSGi service as service properties, except for:

- *Private* - Private properties, property names that start with a full stop ('.' \u002E) must not be registered.

The following service property must be registered, overriding any identical named properties in the decorating capability:

- `serviceloader.mediator` - (`Long`) The bundle id of the mediator.

The Mediator should not verify class space consistency since the OSGi framework already enforces this as long as the publishing capability specifies the `uses` directive.

Any services registered in the OSGi Service Registry must be unregistered when the Service Provider's bundle is stopped or the Mediator is stopped.

133.5.3   Service Loader and Modularity

The Service Loader API causes issues in a modular environment because it requires a class loader that has wide visibility. In a modular environment like OSGi the Consumer, the Service Type, and the Service Provider can, and should, all reside in different modules because they represent different concerns. Best practice requires that only the Service Type is shared between these actors. However,

for the Service Loader to work as it was designed the Consumer must provide a class loader that has visibility of the Service Provider. The Service Provider is an implementation class, exporting such classes is the anathema of modularity. However, since the standard JRE provides application wide visibility this was never a major concern.

The simplest solution is to make the Service Loader aware of OSGi, its API clear is mappable to the OSGi service layer. However, the Service Loader is not extensible. The result is that using the Service Loader in OSGi fails in general because the Service Loader is unable to find the Service Providers. The issues are:

- The use of the Thread Context Class Loader (TCCL) is not defined in an OSGi environment. It should be set by the caller and this cannot be enforced. The multi threaded nature of OSGi makes it hard to predict what thread a Consumer will use, making it impossible to set an appropriate TCCL outside the Consumer.
- A bundle cannot import `META-INF/services` since the name is not a package name. Even if it could, the OSGi framework can only bind a single exporter to an importer for a given package. The Service Loader API requires access to all these pseudo-packages via the Class Loader's `getResources` method, the technique used to find Service Providers.
- Instantiating a Service Provider requires access to internal implementation classes, by exporting these classes, an implementing bundle would break its encapsulation.
- If a Service Provider was exported then importing this class in a Consumer bundle would couple it to a specific implementation package; this also violates the principle of loose coupling.
- The Service Loader API does assume an eternal life cycle, there is no way to signal that a Service Provider is no longer available. This is at odds with the dynamic bundle life cycle.

133.5.4 Processing Consumers

Consumers are not written for OSGi and require help to successfully use the Service Loader API. It is the Mediator's responsibility to ensure that bundles that are wired to published Service Types have access to these Service Provider's instances through the Service Loader API.

This specification does not define how this is done. There are a number of possibilities and it is up to the Mediator to provide the guarantee to the Consumer that it has been properly processed.

A Mediator must only process Consumer's bundles that are wired to the `osgi.extender` capability for the `osgi.serviceloader.processor` extender. Since Consumers must require this extender capability with the default cardinality of 1 there can at most be one extender wired to a Consumer.

133.5.5 Visibility

The Mediator must process the Consumer bundle in such a way that when the Consumer uses the Service Loader API it receives all the Service Providers of bundles that:

- Provide one or more `osgi.serviceloader` capabilities for the requested Service Type, and
- Are not type space incompatible with the requester for the given Service Type, and
- Either the Consumer has no `osgi.serviceloader` requirements or one of its requirements is wired to one of the `osgi.serviceloader` capabilities.

The Mediator must verify that the Consumer has Service Permission GET for the given Service Type since the Consumer uses the Service Type as a service. This specification therefore reuses the Service Permission for this purpose. The check must be done with the `ServicePermission(String,String)` constructor using the bundle's Access Control Context or the bundle's `hasPermission` method.

133.5.6 Life Cycle

There is a life cycle mismatch between the Service Loader API and the dynamic OSGi world. A Service Loader provides a Consumer with an object that could come from a bundle that is later stopped

and/or refreshed. Such an object becomes *stale*. Mediators should attempt to refresh bundles that have access to these stale objects.

133.6 osgi.serviceloader Namespace

The osgi.serviceloader Namespace:

- Allows the Consumer's bundle to require the presence of a Service Provider for the required Service Type.
- Provides the service properties for the service registration.
- Indicates which Service Providers should be registered as an OSGi service.

The namespace is defined in the following table and ServiceLoaderNamespace, see *Common Namespaces Specification* on page 681 for the legend of this table.

Table 133.1   osgi.serviceloader namespace definition

Name	Kind	M/O	Type	Syntax	Description
osgi.serviceloader	CA	M	String	qname	The Service Type's fully qualified name.
*	CA	O	*	*	Additional matching attributes are permitted. These attributes will be registered as custom service properties unless they are private (start with a full stop).
register	CD	O	String	qname	Use this capability to register a different Service Factory under the Service Type for each selected Service Provider.
					A Service Provider is selected if the Service Type is the advertising Service Type and the Service Provider's fully qualified name matches the given name. If no register directive is present all advertised Service Providers must be registered. To register no Service Providers, because the capability must only be used to publish, provide an empty string.

133.7 Use of the osgi.extender Namespace

This section specifies the extender names for Mediators. They are used by both by Consumer and Service Provider bundles to ensure that a Mediator is present. Both names are defined for the general osgi.extender namespace in *osgi.extender Namespace* in *OSGi Core Release 6*.

The osgi.extender namespace requires the use of an *extender name*, the name of the Mediator extenders is:

```
osgi.serviceloader.processor
osgi.serviceloader.registrar
```

The version is for this specification is in both cases:

```
1.0.0
```

133.8   Security

133.8.1   Mediator

The Mediator will require significant permissions to perform its tasks. First, it will require access to the Bundle Context of the Service Provider bundle, which means it must have Admin Permission:

```
AdminPermission[<Service Provider Bundles>,CONTEXT|METADATA|CLASS]
```

Since it will have to register on behalf of the Service Provider bundle it must have complete liberty to register services:

```
ServicePermission[<Service Type>,REGISTER]
```

Depending on the way the Consumers are processed additional requirements may be necessary.

The Mediator connects two parties; it must ensure that neither party will receive additional permissions.

133.8.2   Consumers

Consumers must have:

```
ServicePermission[<Service Type>,GET]
PackagePermission[<Service Type's package>,IMPORT]
CapabilityPermission["osgi.extender", REQUIRE]
CapabilityPermission["osgi.serviceloader", REQUIRE]
```

The Mediator must ensure that the Consumer has the ServicePermission before it provides the instance. It must use the Bundle Context hasPermission method or the bundle's Access Control Context to verify this.

133.8.3   Service Providers

Service Providers must have:

```
ServicePermission[<Service Type>,REGISTER]
PackagePermission[<Service Type's package>,IMPORT]
CapabilityPermission["osgi.extender", REQUIRE]
CapabilityPermission["osgi.serviceloader", PROVIDE]
```

The Mediator must ensure that the Service Provider has the ServicePermission before it provides the instance. It must use the Bundle Context hasPermission method or the bundle's Access Control Context to verify this.

133.9   org.osgi.service.serviceloader

Service Loader Mediator Package Version 1.0.

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

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.serviceloader; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.serviceloader; version="[1.0,1.1)"

133.9.1 Summary

- ServiceLoaderNamespace - Service Loader Capability and Requirement Namespace.

133.9.2 public final class ServiceLoaderNamespace extends Namespace

Service Loader Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace.

All unspecified capability attributes are of one of the following types:

- String
- Version
- Long
- Double
- List<String>
- List<Version>
- List<Long>
- List<Double>

and are used as arbitrary matching attributes for the capability. The values associated with the specified directive and attribute keys are of type String, unless otherwise indicated.

All unspecified capability attributes, unless the attribute name starts with full stop ('.' \u002E), are also used as service properties when registering a Service Provider as a service.

Concurrency Immutable

133.9.2.1 public static final String CAPABILITY_REGISTER_DIRECTIVE = "register"

The capability directive used to specify the implementation classes of the service. The value of this attribute must be of type List<String>.

If this directive is not specified, then all advertised Service Providers that match the service type name must be registered. If this directive is specified, then only Service Providers that match the service type name whose implementation class is contained in the value of this attribute must be registered. To not register a service for this capability use an empty string.

133.9.2.2 public static final String SERVICELOADER_NAMESPACE = "osgi.serviceloader"

Namespace name for service loader capabilities and requirements.

Also, the capability attribute used to specify the fully qualified name of the service type.

133.10 References

[1]   *Java Service Loader API*
      http://docs.oracle.com/javase/6/docs/api/java/util/ServiceLoader.html

134   Subsystem Service Specification

Version 1.1

134.1   Introduction

The *OSGi Core Release 6* specifies a life-cycle model where bundles can be installed, resolved, and started in order to provide their own classes and services as well as use those provided by other bundles in the system. In the core framework, the bundle is the coarsest deployment unit that a management agent is able to work with; however, oftentimes it is necessary to work with collections of bundles and other types of *resources*, such as subsystems and implementation specific resources. For example, a collection of bundles may correspond to a particular feature of a middleware product, such as a Web container. The applications deployed to that container may also be developed as a collection of bundles that an administrator is required to manage as a consistent whole. The Subsystems specification provides a declarative model for defining resource collections, including bundles, and an API for installing and managing those collections of resources.

Many use cases only require *unscoped* resource collections where all provided capabilities are freely exported to and all required capabilities are freely imported from the system. However, in some cases, it is important to allow the exporting of provided capabilities to be *scoped* such that they can only be used by a subset of resources in the system. It may also be necessary to restrict the importing of required capabilities from outside the collection to ensure its internal capabilities are always preferred over capabilities outside the collection. For example, applications running on a Web application server or in a cloud environment may be deployed to the same server instance. The side-effects of co-locating applications on the same server must be minimized, and scoping is used to ensure each application does not use the classes and services of the others.

The framework provides hooks for influencing resolution, and access to bundles and services. These framework hooks may be used to implement scoping for a collection of bundles. The Subsystems specification provides a higher-level declarative model for defining scoping for collections of resources, including bundles.

When deploying a collection of bundles in an OSGi framework, gaps can exist between the requirements of the bundles and the capabilities provided by the target runtime. Management agents are responsible for ensuring additional bundles are installed to plug these gaps such that the collection of bundles will resolve and run. The *Resolver Service Specification* on page 691 and *Repository Service Specification* on page 577 help management agents address these needs but do not cover how deployment works for resource collections, especially when those collections are scoped. Scoping affects requirement and capability resolution and therefore affects the choice of resources. The Subsystems specification defines resolution and provisioning rules to help management agents consistently deploy collections of resources. The specification also defines a format for developers and testers to provide predetermined deployment resolutions to help ensure consistency between test and production environments.

134.1.1   Essentials

- *Collections* - Allow the management of a collection of resources as a whole.
- *Scoping* - Provide support for isolating resources in the collection such that a subset of their capabilities (for example packages and services) are available to satisfy requirements outside the Sub-

system and a subset of their requirements are able to resolve to capabilities provided outside the Subsystem.
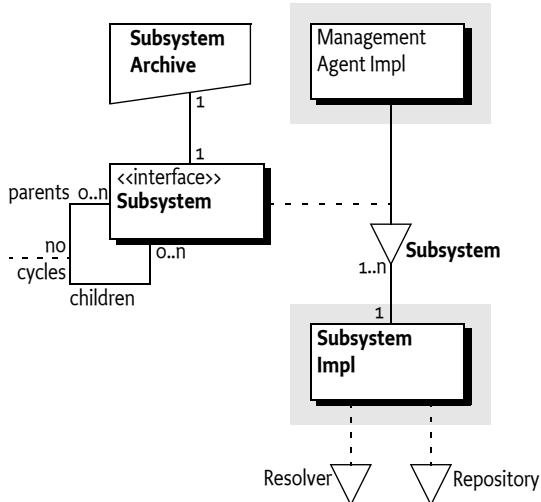
- *Sharing* - Allow Scoped Subsystems to share their resources with others and share resources from others.
- *Dynamic* - Provide life cycle information to users of Subsystems and be able to react to changes in the state of the environment in which a Subsystem is deployed.
- *Flexible* - Enable a flexible definition with subsequent resolution to determine the resources to be used.
- *Deterministic* - Enable the choice of resources deployed for a Subsystem to be determined ahead of deployment.
- *Life-cycle* - Define a life cycle for a Subsystem describing how it affects the Subsystem's resources and allow the life cycle to be observed.
- *Reflective* - Allow discovery of runtime structural and state information.
- *Resolution* - Allow the resolution of a flexible definition during installation to determine the resources to be used.
- *Recursive* - Allow Subsystems to be defined in terms of other Subsystems.

134.1.2   Entities

- *Subsystem* - A collection of resources, such as bundles, or other Subsystems, administered as a whole through a Subsystem service.
- *Subsystem Manifest* - A manifest used to provide a Subsystem definition.
- *Deployment Manifest* - A manifest used to provide a deployment definition for a Subsystem. The definition identifies the exact resources to be deployed for the Subsystem.
- *Subsystem Archive* - A zip file with an `.esa` extension that describes a Subsystem definition. It may include the Subsystem Manifest, Deployment Manifest, or resource files that constitute the Subsystem.
- *Resource* - An element which may be used in the composition of a Subsystem, such as a bundle or another Subsystem.
- *Repository* - A service that is used to discover a Subsystem's content and dependencies. The repository service is described in the *Repository Service Specification* on page 577.
- *Resolver* - A service used to resolve requirements against capabilities to determine the resources required by a Subsystem. The resolver service is described in the *Resolver Service Specification* on page 691.
- *Constituent* - A resource that belongs to one or more Subsystems.

Figure 134.1   *Entities org.osgi.service.subsystem*

134.1.3   Synopsis

The OSGi framework does not provide any support for managing collections of resources. Management of collections of resources is enabled by a Subsystems implementation. When a Subsystems implementation is installed into the framework, it registers a Subsystem service. This service represents the framework as the *Root Subsystem*, which is a Subsystem that provides the capability to install and manage other *child Subsystems*, and is the parent of those Subsystems, but does not itself have a parent.

A Subsystem Archive provides a definition of a Subsystem that is read by the Subsystem implementation as part of installation. The Subsystem is packaged in a Subsystem Archive (.esa) file which is the Subsystem equivalent of a bundle .jar file. The Subsystem definition can be described using a Subsystem Manifest or defaulted based on the name and contents of the Subsystem Archive. Installing a new Subsystem results in another Subsystem service being registered to represent that Subsystem in the runtime. Each Subsystem service enables management and reflection of the Subsystem it represents.

A Subsystem Manifest allows flexibility in the identification of the Subsystem's content resources through version ranges and optionality. The exact versions to be deployed and any required *dependencies* (resources required to satisfy unresolved requirements of the Subsystem's content resources) can be identified in an optional Deployment Manifest. The corresponding resource binaries can be packaged in the Subsystem Archive, or found in a repository. Depending on the type of Subsystem the Subsystem Manifest may describe a *sharing policy* for the Subsystem, such as the packages or services the Subsystem exports or imports. The Deployment Manifest also describes the *sharing policy* details for the Subsystem and is defined by the type of Subsystem and the *sharing policy* described in the Subsystem Manifest.

A Subsystem that does not have a Deployment Manifest has its deployment details calculated during installation. This may be done using the *Resolver Service Specification* on page 691, if available. The starting set of requirements to be resolved are those identifying the Subsystem content (that is, requirements for content resources). The Subsystems implementation provides a resolve context that implements the policy for the Subsystem and consults the configured Repository services to find candidates to satisfy requirements.

This resolve context can also represent the target deployment environment, which might be a live framework, or a static definition of a target runtime. The resulting resolution is used to determine the exact resources to provision, equivalent to those identified in the deployment manifest. If any

of the Repository or Resolver services are unavailable, then a Subsystem implementation can use its own means to determine the deployment, or fail the installation.

A Subsystem definition includes sharing policy configuration to scope requirements and capabilities visibility into and out of a Subsystem. The Subsystems specification defines the concept of Subsystem types to help simplify the configuration of sharing policies. Each type has its own default sharing policy, for example, to forbid the sharing of capabilities out, or to share all capabilities in. Three Subsystem types are defined in the Subsystems specification: *application*, *composite* and *feature*.

An *Application Subsystem* is a Scoped Subsystem with a sharing policy associated with what is often considered to be an application. An application does not share (export) any capabilities. Any requirements that are not satisfied by the application's contents are automatically imported from outside the application.

A *Composite Subsystem* is a Scoped Subsystem with a fully explicit sharing policy. Capabilities may be explicitly imported into, or exported out of, the Composite Subsystem.

A *Feature Subsystem* is an Unscoped Subsystem and so all its requirements and capabilities are shared.

This specification allows for other types to be defined, including ones outside this specification.

134.2   Subsystems

This specification defines a unit of installation called a *Subsystem*. A Subsystem is comprised of resources, including OSGi bundles and other Subsystems, which together can provide functions to end users.

A Subsystem is deployed as a Subsystem Archive (`.esa`) file. Subsystem Archives are used to store Subsystems and optionally their resources in a standard ZIP-based file format. This format is defined in [4] *Zip File Format*. Subsystems normally use the Subsystem Archive extension of `.esa` but are not required to. However there is a special MIME type reserved for OSGi Subsystems that can be used to distinguish Subsystems from normal ZIP files. This MIME type is:

```
application/vnd.osgi.subsystem
```

The type is defined in [5] *IANA application/vnd.osgi.subsystem*. A Subsystem is a ZIP file that:

- Contains zero or more resources. These resources may be OSGi bundles or other Subsystems. Subsystems may be nested or included to any depth.
- Contains an optional Subsystem Manifest named `OSGI-INF/SUBSYSTEM.MF`. The Subsystem Manifest describes the contents of the Subsystem Archive and provides information about the Subsystem. The Subsystem Archive uses headers to specify information that the Subsystems implementation needs to install, resolve and start the Subsystem correctly. For example, it can state the list of content resources that comprise the Subsystem and the Subsystem's type.
- Contains an optional Deployment Manifest file named `OSGI-INF/DEPLOYMENT.MF`. The Deployment Manifest describes the content resources, dependencies, and sharing policy that need to be provisioned to satisfy the Subsystem definition and ultimately allow it to resolve at runtime.

The Subsystem and Deployment Manifest follow the JAR manifest format (version 1.0), but with the following relaxed rules:

- No limit on the line length. Lines are allowed to exceed the JAR manifest maximum of 72 bytes.
- The last line is not required to be a carriage-return new-line combination.
- There is only one section in the manifest (the main section). A Subsystems implementation is free to ignore other sections of the manifest.

Once a Subsystem is started, its functionality is provided. Depending on the type of Subsystem it may expose capabilities, such as packages and services, to other resources installed in the OSGi framework.

134.2.1 Subsystem Manifest Headers

A Subsystem can carry descriptive information about itself in the Subsystem manifest file contained in its Subsystem Archive under the name OSGI-INF/SUBSYSTEM.MF. This specification defines Subsystem manifest headers, such as Subsystem-SymbolicName and Subsystem-Version, which Subsystem developers use to supply descriptive information about a Subsystem. A Subsystems implementation must:

- Process the main section of the manifest. Any other sections of the manifest can be ignored.
- Ignore unknown manifest headers. The Subsystem developer can define additional manifest headers as needed.
- Ignore unknown attributes and directives.

All specified manifest headers are listed in the following sections. All headers are optional. Example values are provided to help explain each header (e.g. Export-Package: org.acme.logging; version=1.0).

134.2.1.1 Export-Package: org.acme.logging; version=1.0

The Export-Package header declares the exported packages for a Scoped Subsystem. See *Export-Package* on page 654.

134.2.1.2 Import-Package: org.osgi.util.tracker; version="[1.4, 2.0)"

The Import-Package header declares the imported packages for a Scoped Subsystem. See *Import-Package* on page 654.

134.2.1.3 Preferred-Provider: com.acme.logging

The Preferred-Provider header declares a list bundles and Subsystems which are the providers of capabilities that are preferred when wiring the requirements of a Scoped Subsystem. See *Preferred-Provider Header* on page 629.

134.2.1.4 Provide-Capability: com.acme.dict; from=nl; to=de; version:Version=1.2

The Provide-Capability header declares the capabilities exported for a Scoped Subsystem. See [3] *Resource and Wiring*.

134.2.1.5 Require-Bundle: com.acme.chess; version= "[1.0, 2.0)"

The Require-Bundle header declares the required bundles for a Scoped Subsystem. See *Require-Bundle* on page 655.

134.2.1.6 Require-Capability: osgi.ee; filter:="(osgi.ee=*)"

The Require-Capability header declares the required capabilities for a Scoped Subsystem. See [3] *Resource and Wiring*.

134.2.1.7 Subsystem-Category: osgi, test, nursery

The Subsystem-Category header identifies the categories of the subsystem as a comma-delimited list.

134.2.1.8 Subsystem-ContactAddress: 2400 Oswego Road, Austin, TX 74563

The Subsystem-ContactAddress header identifies the contact address where problems with the subsystem may be reported; for example, an email address.

134.2.1.9   **Subsystem-Content: com.acme.logging**

The Subsystem-Content header lists requirements for resources that are considered to be the contents of this Subsystem. See *Subsystem-Content Header* on page 627.

134.2.1.10   **Subsystem-Copyright: OSGi (c) 2014**

The Subsystem-Copyright header identifies the subsystem's copyright information.

134.2.1.11   **Subsystem-Description: The ACME Account Admin Application**

The Subsystem-Description header defines a human-readable description for this Subsystem, which can potentially be localized.

134.2.1.12   **Subsystem-DocURL: http://www.example.com/Firewall/doc**

The Subsystem-DocURL header identifies the subsystem's documentation URL, from which further information about the subsystem may be obtained.

134.2.1.13   **Subsystem-ExportService: org.acme.billing.Account; filter:="(user=bob)"**

The Subsystem-ExportService header specifies the exported services for a Scoped Subsystem. See *Subsystem-ExportService* on page 655.

134.2.1.14   **Subsystem-Icon: /icons/acme-logo.png; size=64**

The optional Subsystem-Icon header provides a list of URLs to icons representing this subsystem in different sizes. The following attribute is permitted:

- size - (integer) Specifies the size of the icon in pixels horizontal. It is recommended to always include a 64x64 icon.

The URLs are interpreted as relative to the subsystem archive. That is, if a URL with a scheme is provided, then this is taken as an absolute URL. Otherwise, the path points to an entry in the subsystem archive file.

134.2.1.15   **Subsystem-ImportService: org.acme.billing.Account; filter:="(type=premium)"**

The Subsystem-ImportService header specifies the imported services for a Scoped Subsystem. See *Subsystem-ImportService* on page 655.

134.2.1.16   **Subsystem-License: http://www.opensource.org/licenses/jabberpl.php**

The Subsystem-License header provides an optional machine readable form of license information. The purpose of this header is to automate some of the license processing required by many organizations like for example license acceptance before a subsystem is used. The header is structured to provide the use of unique license naming to merge acceptance requests, as well as links to human readable information about the included licenses. This header is purely informational for management agents and must not be processed by the Subsystems implementation.

The syntax for this header is as follows:

```
Subsystem-License ::= '<<EXTERNAL>>' |
                      ( license ( ',' license ) * )
license        ::= name ( ';' license-attr ) *
license-attr   ::= description | link
description    ::= 'description' '=' string
link           ::= 'link' '=' <url>
```

This header has the following attributes:

- name - Provides a globally unique name for this license, preferably world wide, but it should at least be unique with respect to the other clauses. The magic name <<EXTERNAL>> is used to indi-

cate that this artifact does not contain any license information but that licensing information is provided in some other way. This is also the default contents of this header.

Clients of this subsystem can assume that licenses with the same name refer to the same license. This can for example be used to minimize the click through licenses. This name should be the canonical URL of the license, it must not be localized by the translator. This URL does not have to exist but must not be used for later versions of the license. It is recommended to use URLs from [9] *Open Source initiative*. Other licenses should use the following structure, but this is not mandated:

```
http://<domain-name>/licenses/
            <license-name>-<version>.<extension>
```

- description - (optional) Provide the description of the license. This is a short description that is usable in a list box on a UI to select more information about the license.
- link - (optional) Provide a URL to a page that defines or explains the license. If this link is absent, the name field is used for this purpose. The URL is relative to the root of the bundle. That is, it is possible to refer to a file inside the bundle.

If the Subsystem-License statement is absent, then this does not mean that the subsystem is not licensed. Licensing could be handled outside the subsystem and the <<EXTERNAL>> form should be assumed. This header is informational and may not have any legal bearing. Consult a lawyer before using this header to automate licensing processing.

134.2.1.17   Subsystem-Localization: OSGI-INF/l10n/subsystem

The Subsystem-Localization header identifies the default base name of the localization properties files contained in the subsystem archive. The default value is OSGI-INF/l10n/subsystem. Translations are therefore, by default, OSGI-INF/l10n/subsystem_de.properties , OSGI-INF/l10n/subsystem_nl.properties, and so on. The location is relative to the root of the subsystem archive. See *Subsystem-Localization Header* on page 622

134.2.1.18   Subsystem-ManifestVersion: 1

The Subsystem-ManifestVersion header defines that the Subsystem follows the rules of a Subsystems Specification. It is 1 (the default) for this version of the specification. Future versions of the Subsystems Specification can define higher numbers for this header.

134.2.1.19   Subsystem-Name: Account Application

The Subsystem-Name header defines a short, human-readable name for this Subsystem which may be localized. This should be a short, human-readable name that can contain spaces.

134.2.1.20   Subsystem-SymbolicName: com.acme.subsystem.logging

The Subsystem-SymbolicName header specifies a non-localizable name for this Subsystem. The Subsystem symbolic name together with a version identify a Subsystem Definition though a Subsystem can be installed multiple times in a framework. The Subsystem symbolic name should be based on the reverse domain name convention. See *Subsystem-SymbolicName Header* on page 620.

134.2.1.21   Subsystem-Type: osgi.subsystem.application

The Subsystem-Type header specifies the type for this Subsystem. Three types of Subsystems must be supported: osgi.subsystem.application, osgi.subsystem.composite and osgi.subsystem.feature. See *Subsystem Identifiers and Type* on page 620 for details about the three different types of Subsystems. See *Subsystem-Type Header* on page 620 for more information about the values for the Subsystem-Type header.

134.2.1.22   Subsystem-Vendor: OSGi Alliance

The Subsystem-Vendor header contains a human-readable description of the subsystem vendor.

134.2.1.23   **Subsystem-Version: 1.0**

The Subsystem-Version header specifies the version of this Subsystem. See *Subsystem-Version Header* on page 620.

134.2.2   Subsystem Identifiers and Type

A Subsystem is identified by a number of names that vary in their Scope:

- *Subsystem identifier* - A long that is a Subsystems implementation assigned unique identifier for the full lifetime of an installed Subsystem, even if the framework or the Subsystem's implementation is restarted. Its purpose is to distinguish Subsystems installed in a framework. Subsystem identifiers are assigned in ascending order to Subsystems when they are installed. The getSubsystemId() method returns a Subsystem's identifier.
- *Subsystem location* - A name assigned by a management agent to a Subsystem during the installation. This string is normally interpreted as a URL to the Subsystem Archive but this is not mandatory. Within a particular framework, a Subsystem location must be unique. A location string uniquely identifies a Subsystem. The getLocation() method returns a Subsystem's location.
- *Subsystem Symbolic Name and Subsystem Version* - A name and version assigned by the developer. The combination of a Subsystem symbolic name and Subsystem version is intended to provide a globally unique identifier for a Subsystem Archive or Subsystem definition. The getSymbolicName() method returns the assigned Subsystem name. The getVersion() method returns the assigned version. Though this pair is intended to be unique, it is developer assigned and there is no verification at runtime that the pair uniquely identifies a Subsystem Archive. It is possible to install a Subsystem multiple times as long as the multiple Subsystem symbolic name and version pairs are isolated from each other by Subsystem sharing policies.

134.2.3   Subsystem-SymbolicName Header

The Subsystem-SymbolicName header specifies the symbolic name of the Subsystem. The Subsystem-SymbolicName header may also specify arbitrary matching attributes. Subsystem-SymbolicName is an optional header; the default value is derived as described in *Deriving the Subsystem Identity* on page 621.

The Subsystem-SymbolicName header must conform to the following syntax:

```
Subsystem-SymbolicName ::= symbolic-name(';' parameter )*
```

No directives are defined by this specification for the Subsystem-SymbolicName header. The header allows the use of arbitrary attributes that can be required by the Subsystem-Content header.

134.2.4   Subsystem-Version Header

The Subsystem-Version header is optional and must conform to the following syntax:

```
Subsystem-Version ::= version
```

If the Subsystem-Version header is not specified then the default value is derived as described in *Deriving the Subsystem Identity* on page 621.

134.2.5   Subsystem-Type Header

The Subsystem-Type header specifies the *type* of the Subsystem. Three types of Subsystems are defined by this specification:

- osgi.subsystem.application
- osgi.subsystem.composite
- osgi.subsystem.feature

See *Subsystem Types* on page 656 for details about the three different types of Subsystems. Subsystem-Type is an optional header; the default value is osgi.subsystem.application.

The following directive must be recognized for the Subsystem-Type header:

- provision-policy - (rejectDependencies|acceptDependencies) Directive used to declare if the Subsystem is willing to accept dependencies as *constituents*. A *constituent* is the term used to refer to a resource that belongs to one or more Subsystems. It can belong to a Subsystem as a result of being listed as content or, as described here, can have been provisioned into the Subsystem as a dependency. The default policy is rejectDependencies. See *Accepting Dependencies* on page 633 for installing and tracking dependencies. The value acceptDependencies must not be used for Feature Subsystems. If a Feature Subsystem attempts to use the acceptDependencies policy then the Subsystem installation must fail. If the provision-policy directive is declared and has any other value besides acceptDependencies or rejectDependencies then the Subsystem installation must fail.

134.2.6 Deriving the Subsystem Identity

If the Subsystem-SymbolicName and Subsystem-Version are not specified, then the following rules are defined for deriving the values of the Subsystem's symbolic name and version. If not otherwise specified, the default value of the version is 0.0.0.

When installing a Subsystem, the following URI syntax must be used as the location string in order to specify default values.

```
subsystem-uri ::= 'subsystem://' url? '?' params?
params        ::= param ( '&' param )*
param         ::= key '=' value
key           ::= unreserved | escaped
value         ::= unreserved | escaped
url           ::= ‹see [6] RFC 1738 Uniform Resource Locators›
escaped       ::= ‹see [7] Uniform Resource Identifiers (URI): Generic Syntax›
unreserved    ::= ‹see [7] Uniform Resource Identifiers (URI): Generic Syntax›
```

The query parameters represent Subsystem Manifest header names and values. Implementations must support the Subsystem-SymbolicName and Subsystem-Version parameters. Implementations can support additional parameters but must fail the installation if any unsupported parameters are included.

As an example, the following Subsystem URI has an embedded URL of:

http://www.foo.com/sub#system.esa

It includes a default symbolic name of com.acme.foo and default version of 1.0.0.

```
subsystem://http%3A%2F%2Fwww.foo.com%2Fsub%2523system.esa? «
    Subsystem-SymbolicName=com.acme.foo&Subsystem-Version=1.0.0
```

When installing a Subsystem containing other Subsystem Archives with no symbolic name or version, values will be derived from the Subsystem Archive file or resource name. The syntax is as follows:

```
subsystem-archive ::= symbolic-name ( '@'version ) '.esa'
```

If the symbolic name is not provided in the manifest and cannot be computed by any other means then the Subsystem must fail to install.

134.2.7 Subsystem Identity Capability

The Subsystem's symbolic name, version, type and the arbitrary matching attributes specified on the Subsystem-SymbolicName header compose the osgi.identity capability for a Subsystem re-

source. The osgi.identity capability is provided by a Subsystem resource when contained within a Repository service, see *Repository Service Specification* on page 577. For example:

```
Subsystem-SymbolicName: org.acme.billing;category=banking
Subsystem-Version:      1.0.0
Subsystem-Type:         osgi.subsystem.composite
```

The above headers are used to declare a Subsystem of type osgi.subsystem.composite with the symbolic name of org.acme.billing, version of 1.0.0 and arbitrary matching attribute category of banking. This information will also be reflected in the osgi.identity capability of the Subsystem's Resource (org.osgi.resource.Resource). The following osgi.identity capability would be generated for a Subsystem resource from the above headers using the Provide-Capability header syntax. For example:

```
osgi.identity;
osgi.identity=org.acme.billing;
 version:Version=1.0.0;
 type=osgi.subsystem.composite;
 category=banking
```

This allows for requirements to be used to search a repository for Subsystems. The following requirement could be used to search for all Subsystems of type osgi.subsystem.composite using the Require-Capability header syntax. For example:

```
osgi.identity; filter:="(type=osgi.subsystem.composite)"
```

134.2.8   Subsystem-Localization Header

For consistency and ease of comprehension, the design for localizing subsystem manifest headers follows the approach used by bundles.

134.2.8.1   Localization Properties

A localization entry contains key/value entries for localized information. All headers in a subsystem's manifest can be localized. However, the subsystems implementation must always use the non-localized versions of headers that have subsystem semantics. Note that the use of localization on certain such headers, such as Subsystem-SymbolicName, may cause errors as a value with a % sign will not be valid.

A localization key can be specified as the value of a subsystem's manifest header using the following syntax:

```
header-value ::= '%' text
text ::= < any value which is both a valid manifest header value and a valid
    property key name >
```

For example, consider the following subsystem manifest entries:

```
Subsystem-Name: %acme subsystem
Subsystem-Description: %acme description
Subsystem-SymbolicName: acme.Subsystem
Acme-Defined-Header: %acme special header
```

User-defined headers can also be localized. Spaces in the localization keys are explicitly allowed.

The previous example manifest entries could be localized by the following entries in the manifest localization entry OSGI-INF/l10n/subsystem.properties.

```
# subsystem.properties
acme\ subsystem=The ACME Subsystem
acme\ description=The ACME Subsystem provides all of the ACME \ services
```

```
acme\ special\ header=user-defined Acme Data
```

The above manifest entries could also have French localizations in the manifest localization entry:

```
OSGI-INF/l10n/subsystem_fr_FR.properties.
```

134.2.8.2   Locating Localization Entries

The Subsystems implementation must search for localization entries by appending suffixes to the localization base name according to a specified locale and finally appending the .properties suffix. If a translation is not found, the locale must be made more generic by first removing the variant, then the country and finally the language until an entry is found that contains a valid translation. For example, looking up a translation for the locale en_GB_welsh will search in the following order:

```
OSGI-INF/l10n/subsystem_en_GB_welsh.properties
OSGI-INF/l10n/subsystem_en_GB.properties
OSGI-INF/l10n/subsystem_en.properties
OSGI-INF/l10n/subsystem.properties
```

134.3   Subsystem Region

A *Region* provides isolation for a group of one or more Subsystems. Each Subsystem installed must be a member of one and only one Region. A Region consists of one and only one Scoped Subsystem and optionally a set of Unscoped Subsystems. Every Region has one and only one Parent Region, with the exception of the Root Region which has no Parent Region. The Region parent → child connections form the *Region Tree*, which by definition contains no cycles.
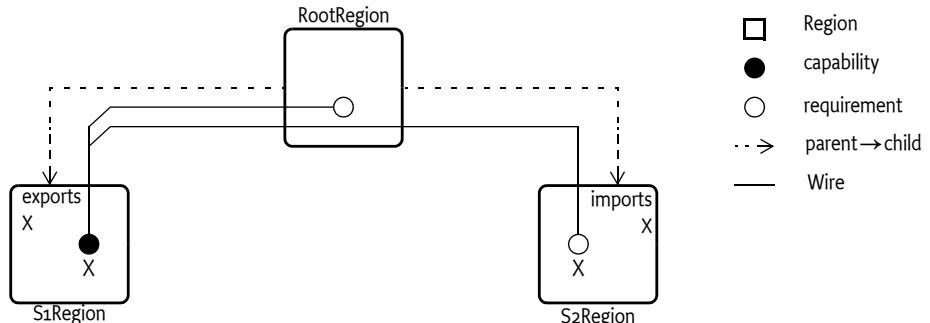
Each Region, except the Root Region, has a *sharing policy* associated with it which is defined by a Scoped Subsystem. A *sharing policy* consists of two parts:

- *Export Policy* - Defines the set of capabilities provided by the constituents contained in the Region that are made available to the parent Region.
- *Import Policy* - Defines the set of capabilities which are available in the parent Region that are made available to the child Region.

Figure 134.2 illustrates a set of Regions that contain capabilities and requirements for a capability. For the purposes of this illustration the Subsystems and resources have been omitted.

*Figure 134.2*   *Regions and Import/Export*
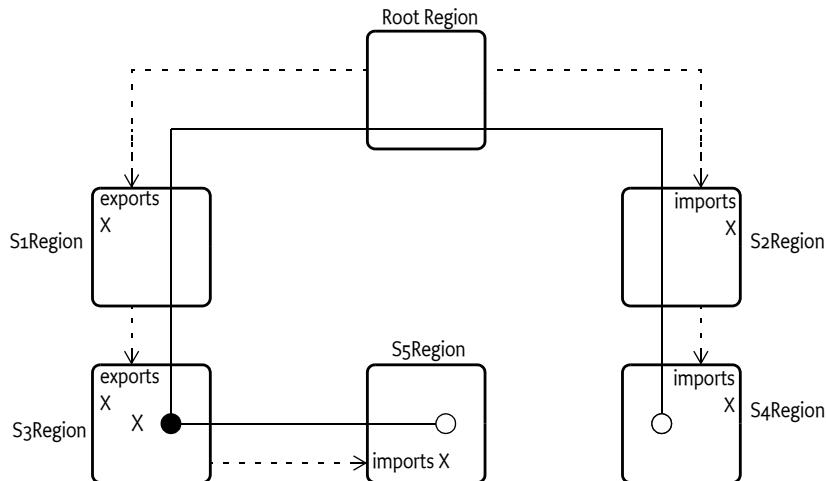
In this example some constituent of Region S1Region provide a capability S1→X. The S1Region's sharing policy exports the capability S1→X to its parent RootRegion. The RootRegion contains a constituent which has a requirement Root→X. The export sharing policy of S1Region allows visibility to the capability S1→X from the RootRegion which allows requirement Root→X to be satisfied by the capability S1→X. The S2Region also contains a constituent which has a requirement on S2→X.

The sharing policy of S2Region imports the capability X from its parent Region RootRegion. Since RootRegion has visibility to the capability S1→X this allows S2Region to also have visibility to capability S1→X through its import sharing policy which allows requirement S2→X to be satisfied by the capability S1→X.

Sharing policies of the Regions allow for a capability to be shared across an arbitrary number of Regions. For those familiar with the Region digraph, see [8] *Equinox Region Digraph*, the connections between Subsystem Regions is more restrictive than what the full Region digraph specification allows. A *visibility path* is the path over the sharing policies of the Region tree from a requirement to a capability that allows a requirement to get wired to a capability. Since all (non-Root) Regions have one and only one Parent Region the *visibility paths* over the sharing policies between a requirement and a capability is limited to 0 or 1. Figure 134.3 is another figure that illustrates a capability being shared across many different Regions.

Figure 134.3   *Regions and Sharing Capabilities*

In this example the capability S3→X is exported by the S3Region sharing policy to its parent S1Region. S1Region also exports X to its parent RootRegion. Then S2Region imports X from its parent RootRegion and finally S4Region imports X from its parent S2Region. The visibility path from requirement S4→X to capability S3→X is the following: S4 → S2 → Root → S1 → S3.
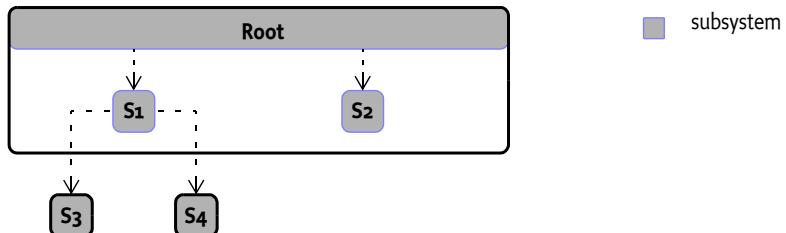
Notice that in this example the S5Region also has a sharing policy that imports X from its parent S3Region. Child Regions are allowed to import any capability to which the Parent Region has visibility. This is true even if the Parent Region does not export the capability. Regions can selectively choose what capabilities they want to expose (or export) to their Parent Region. Child Regions also can selectively choose what capabilities they want to be exposed to (or import) from their Parent Region. A Parent Region has no control over what capabilities its children have visibility. Similarly a Parent Region has no control over what capabilities a Child Region is allowed to export to the Parent Region. In other words, a Parent Region must give a Child Region everything the Child Region asks for (if the Parent Region has access to it) and a Parent Region must accept everything a Child Region offers to the Parent Region.

134.4 Subsystem Relationships

Subsystems installed into a framework become part of the *Subsystem graph*. The Subsystem graph may be thought of as is directed acyclic graph with one and only one source vertex, which is the Root Subsystem. The edges have the child as the head and parent as the tail (parent→child). This is depicted in Figure 134.4.

Figure 134.4   *Parent Child Relationship*

A Subsystem installed into or included within one or more Subsystems is called a *child Subsystem*. A Subsystem which has one or more child Subsystem(s) installed or included in it is called a *parent Subsystem*. Note that a Subsystem may be both a parent and child Subsystem. The Subsystem graph has the following rules:
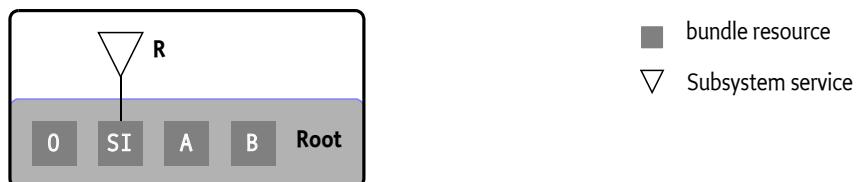
• There is one and only one source vertex (i.e. a Subsystem with no parents), which is the *Root Subsystem*.
• The Root Subsystem is considered a Scoped Subsystem of type application with a provision-policy of acceptDependencies.
• The Root Subsystem has a symbolic name of org.osgi.service.subsystem.root, version 1.0.0, Subsystem identifier of 0, and a location of

      subsystem://?Subsystem-SymbolicName=org.osgi.service.subsystem.root& «
          Subsystem-Version=1.0.0

• The Root Subsystem always exists when a Subsystems implementation is active, even if no other Subsystems are installed and all initial bundles installed into the framework along with the Subsystems implementation are considered content resources of the Root Subsystem.
• All other (non-root) Subsystems must have one or more parent Subsystems. This implies that there are no orphan Subsystems (except the Root Subsystem) and the Subsystem graph is fully connected.
• All parents of a Subsystem belong to the same Region.
• An Unscoped Subsystem must belong to the same Region to which its parents belong.
• A Scoped Subsystem (other than the Root Subsystem) must belong to a child Region of the Region to which the Subsystem's parents belong.

When a Subsystem is installed using a Subsystem service install(String) or install(String,InputStream) method the *Subsystem resource* becomes a constituent of the Subsystem which the install method was called on. The *Subsystem resource* is the Subsystem Archive and may be retrieved by calling the Subsystem service getConstituents() method. Figure 134.4 illustrates the Root Subsystem with initial bundles A, B, SI (Subsystems implementation, may be multiple bundles), and the system bundle (identifier 0).

Figure 134.5   *Subsystem resources*
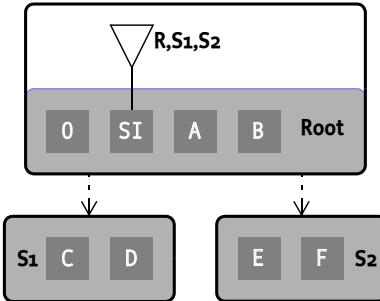
In Figure 134.5 Bundles A, B and SI are considered constituents of the Root Subsystem. The system bundle is also considered to be a constituent of the Root Subsystem (bundle zero). A Subsystem ser-

vice R is registered that represents the Root Subsystem. When Subsystems are installed using the Root Subsystem then these Subsystem resources become constituents of the Root Subsystem and the Subsystems become child Subsystems of the Root Subsystem. For example, Figure 134.6 illustrates the Root Subsystem with Scoped Subsystem S1 with constituent bundles C and D and Scoped Subsystem S2 with constituent bundles E and F:
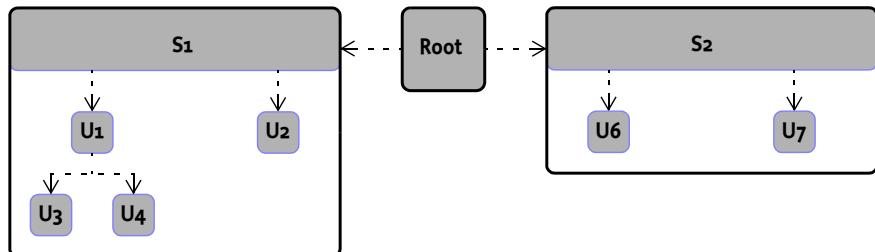
Figure 134.6   *Subsystems*

The two Subsystems S1 and S2 have the same parent and Subsystems S1 and S2 are children of the Root Subsystem. Figure 134.7 shows a more complicated tree that has both Scoped and Unscoped Subsystems installed. This figure omits the constituent resources and Subsystem services:

Figure 134.7   *Parent Child Relationship with Unscoped Subsystems*
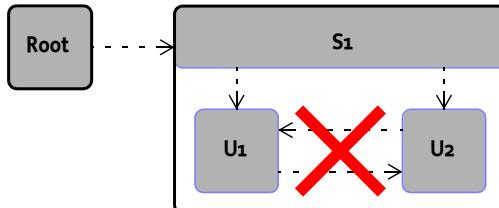
134.4.1   Prevent Cycles and Recursion

It is possible to end up with cycles in the parent → child relationships between Subsystems contained in the same Region. Figure 134.8 illustrates this.
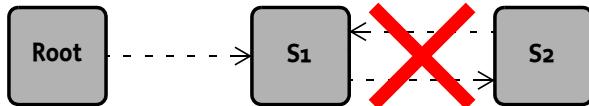
Figure 134.8   *Subsystems and cycles*

In this example Subsystem S1 has been installed. The Scoped Subsystem S1 has included in its constituents the Unscoped Subsystems U1 and U2. Furthermore U1 has included the Unscoped Subsystem U2 as a constituent and U2 has included the Unscoped Subsystem U1 as a constituent. This causes Unscoped Subsystem U1 to have parents S1 and U2 and Unscoped Subsystem U2 to have parents S1 and U1. There is now a cycle between the Subsystems U1 and U2. Subsystems implementations must detect this cycle and fail the installation of such a degenerative Subsystem. The top level Subsystem being installed must fail the install operation by throwing a Subsystem Exception. In this case the install operation of the S1 Subsystem must fail with a Subsystem Exception being thrown.

Cycles may also exist in the definition of Scoped Subsystems which includes other child Subsystems. Figure 134.9 illustrates this.

Figure 134.9   *Scoped Subsystems and cycles*

In this example the Scoped Subsystem S1 includes as a child the Scoped Subsystem S2. The S2 Subsystem also includes as a child the Scoped Subsystem S1. Subsystems implementations must detect this and fail the installation of such a degenerative Subsystem. The top level Subsystem being installed must fail the install operation by throwing a Subsystem Exception. In this case the install operation of the first S1 Subsystem must fail by throwing a Subsystem Exception.

134.5   Determining Content

A Subsystem definition may declare different types of *content resources*. A Subsystems implementation may support additional types, but the following types must be supported:

- osgi.bundle
- osgi.fragment
- osgi.subsystem.application
- osgi.subsystem.feature
- osgi.subsystem.composite

A Subsystems implementation is free to support additional content types as value-add, but an implementation is required to fail the installation of a Subsystem which declares content resource types which are not recognized by the implementation.

The individual content resources may be specified in the following ways:

- The Subsystem-Content header, or
- The entries of the Subsystem Local Repository, see *Resource Repositories* on page 630.

134.5.1   Subsystem-Content Header

The Subsystem-Content header contains a list of symbolic names, with optional attribute and directive assertions. Each element specifies a single resource that is to be a content resource of the Subsystem. See also *Discovering Content Resources* on page 631. The Subsystem-Content header must conform to the following syntax:

```
Subsystem-Content ::= resource ( ','resource )*
resource          ::= symbolic-name ( ';' parameter )*
```

The Subsystem-Content header may specify the following directives:

- resolution - (mandatory|optional) A mandatory content resource prevents the Subsystem from successfully installing when the constituent cannot be found (or satisfied); an optional content resource allows a Subsystem to successfully install even if the content cannot be found (or satisfied). The default value is mandatory.
- start-order - (Integer >= 1) Specifies the start order of the content resource in relation to other content resources of the Subsystem. See *Start Order* on page 641.

The Subsystem-Content header may specify the following architected matching attributes as well as any arbitrary matching attributes:

- version - (Version) A version range used to select the version of the resource to use. This follows the OSGi version range scheme, including the default value of 0.0.0.
- type - Indicates the type of the content. It is recommended that a reverse domain name convention is used unless those types and their processing is standardized by the OSGi Alliance, for example bundles. The default type is `osgi.bundle`. A Subsystems implementation may support additional types, but the following types must be supported:
  - `osgi.bundle`
  - `osgi.fragment`
  - `osgi.subsystem.application`
  - `osgi.subsystem.composite`
  - `osgi.subsystem.feature`

For example, the following header specifies three Subsystem constituents:

```
Subsystem-Content:
    org.acme.billing.impl;
        type=osgi.bundle;
        version=1.0,
    org.acme.billing.frag;
        type=osgi.fragment;
        version=1.0,
    org.acme.billing.credit.subsystem;
        type=osgi.subsystem.composite;
        version=1.0
```

The above header specifies three content resources of a Subsystem:

- A bundle resource with the symbolic name org.acme.billing.impl at version 1.0 or greater
- A fragment resource with the symbolic name org.acme.billing.frag at version 1.0 or greater
- A child composite Subsystem resource with the symbolic name org.acme.billing.credit.subsystem at version 1.0 or greater

134.5.2 Subsystem-Content Requirements

Each element of the Subsystem-Content header is used to locate a resource that is to be used as content of the Subsystem. One way of describing the elements of the Subsystem-Content header is in terms of a Requirement using the `osgi.identity` namespace. The Requirement is defined in [3] *Resource and Wiring*. To illustrate, a single Subsystem-Content element:

```
org.acme.billing.impl;
    type=osgi.bundle;
    version=1.0
```

This Subsystem-Content header can be converted into the following `osgi.identity` Requirement with the Require-Capability syntax for illustration:

```
osgi.identity;
    filter:="(&
        (osgi.identity=org.acme.billing.impl)
        (type=osgi.bundle)
        (version>=1.0)
    )"
```

All directives specified on the Subsystem-Content header, except `start-order`, should be included in the Requirement. All attributes should be included in the `filter` directive of the Requirement. Notice that the version attribute is a range and must be converted into a proper filter. The

VersionRange.toFilter method can be used to do this conversion. All other matching attributes are treated as type String and use an equality operation in the filter. Here is an example of a more complex transformation to Requirement:

```
org.acme.billing.credit.subsystem;
    type=osgi.subsystem.composite;
    version="[1.0, 1.1)";
    category=banking;
    resolution:=optional;
    start-order:=1
```

The above Subsystem-Content element converts into the following osgi.identity Requirement:

```
osgi.identity;
    filter:="(&
        (osgi.identity=org.acme.billing.impl)
        (type=osgi.subsystem.composite)
        (&(version>=1.0)(!(version>=1.1))
            (category=banking)
        )"
    resolution:=optional
```

134.5.3   Preferred-Provider Header

The Preferred-Provider header contains a list of symbolic names, with optional attributes assertions. Each element specifies a single bundle or Subsystem resource that is to be preferred when resolving the requirements of the Subsystem content resources. The Preferred-Provider header must conform to the following syntax:

```
Preferred-Provider ::= resource (',' resource )*
resource           ::= symbolic-name ( ';' attribute )*
```

The Preferred-Provider header may specify the following architected matching attributes:

- version - (Version) A version range used to select the version of the bundle or Subsystem to use. This follows the OSGi version range scheme, including the default value of 0.0.0.
- type - (String) Indicates the type of the provider. Valid types are:
  - osgi.bundle
  - osgi.subsystem.composite
  - osgi.subsystem.feature

  The default type is osgi.subsystem.composite. Specifying an unsupported type results in an installation failure.

Each element of the Preferred-Provider header is used to locate a resource that is to be used as a preferred provider of the Subsystem. The Preferred-Provider header elements are converted to Requirements using the osgi.identity namespace just like the Subsystem-Content header except the default type is osgi.subsystem.composite. See *Subsystem-Content Requirements* on page 628.

Because this header influences resolution, it is only valid for it to be used on a Scoped Subsystem. If a Subsystems implementation encounters this header on an Unscoped Subsystem, it must fail the installation of the Subsystem.

The Preferred-Provider header has the effect of influencing the import policy into the Region representing the Scoped Subsystem that specified the header. If there are multiple candidate capabilities for a requirement and one or more of those capabilities is from a bundle or Subsystem identified in the Preferred-Provider header, then the Region import policy must prefer the capabilities from the preferred bundle or Subsystem.

A resource may be considered as a preferred provider only if it is a constituent of either the Scoped Subsystem's or any ancestor's Region.

134.5.4 Resource Repositories

When a Subsystem is installed the Subsystems implementation is responsible for provisioning resources that are associated with the Subsystem. For example, the Subsystem's content resources as well as any resources that are needed to satisfy dependencies of a Subsystem's content resources. During the Subsystem install process the Subsystems implementation uses a defined set of repositories to find the required resources needed to install a Subsystem. This set of repositories includes the following:

- *Local Repository* - Contains the resources included in the Subsystem Archive, see *Local Repository* on page 630.
- *System Repository* - Contains the resources currently installed, see *System Repository* on page 630
- *Repository Services* - The set of repositories registered as OSGi services, see *Repository Services* on page 630.
- *Content Repository* - The set of resources that comprise the Subsystem content, see *Content Repository* on page 630
- *Preferred Repository* - The set of resources that are considered preferred providers, see *Preferred Repository* on page 631.

134.5.4.1 Local Repository

The Root of the Subsystem Archive contains 0 or more resources. The Subsystems implementation must read all entries (except directory entries) in the Root of the Subsystem Archive and treat each entry as a potential resource. One way of describing the resource entries contained in the Root of the Subsystem Archive is in the terms of an Repository implementation. For the purpose of this specification these resources are referred to as the Subsystem's *Local Repository*. The Local Repositories must not be registered as an OSGi Repository service. Also, it is not required that the Subsystem implementation actually implement a Local Repository as a concrete implementation of the Repository service interface.

134.5.4.2 System Repository

The term *System Repository* is used to describe the set of resources that are constituents of one or more of the currently installed Subsystems. The System Repository must not be registered as an OSGi service. Also it is not required that System Repository be implemented as a concrete implementation of the Repository service. There is a single System Repository representing the resources installed in the OSGi framework.

134.5.4.3 Repository Services

The repositories which are registered as Repository services, see *Repository Service Specification* on page 577. These Repositories are used to discover Subsystem content resources and dependencies. A Subsystems implementation searches registered Repository services by service ranking order.

134.5.4.4 Content Repository

The set of content resources for a Subsystem is referred to as the Subsystem's Content Repository. Similar to the Local and System Repositories, the Content Repositories must not be registered as an OSGi service and it is not required that the Subsystems implementation actually implement a Content Repository as a concrete implementation of the Repository service. There are two types of resources that can exist in a Subsystem's Content Repository:

- *Installable Content* - A content resource which must be installed and result in a distinct resource at runtime. That is, a distinct bundle or Subsystem installation.

- *Shared Content* - A content resource which is already installed and is a constituent of one or more already installed Subsystems that belong to the same Region as the Subsystem that the Subsystem content repository is for. This resource must be reused, the Subsystems implementation must not install another instance of the resource.

Details on how the content resources are discovered for the Content Repository are discussed in *Discovering Content Resources* on page 631.

134.5.4.5 Preferred Repository

The set of resources which are considered preferred providers of capabilities required by a Subsystem is referred to as the Preferred Provider Repository for the Subsystem. The Preferred Provider Repository for a Subsystem must not be registered as an OSGi service and it is not required that the Subsystems implementation actually implement the Preferred Provider Repository as a concrete implementation of the Repository service.

The following steps must be followed to discover the resources of the preferred provider repository for a Subsystem:

1. The Preferred-Provider header is parsed into a list of elements where each element specifies a single `osgi.identity` requirement, see *Preferred-Provider Header* on page 629.
2. For each `osgi.identity` requirement a `Requirement` object is created and used to search Repositories for preferred provider resources.
3. The System Repository is searched. For each capability found in the System Repository; if the resource providing the `osgi.identity` capability is a constituent contained in the parent Region of the Scoped Subsystem's Region then the providing Resource of the Capability is considered a preferred provider and the search stops; otherwise continue to the next step.
4. The Subsystem's Local Repository is searched. If a capability is found then the providing resource is used as a preferred provider and the search stops; otherwise continue to the next step.
5. The registered Repository services are searched. If a repository service finds a capability then the providing resource is used as a preferred provider and the search stops; otherwise the preferred provider is not found.

134.5.5 Discovering Content Resources

When a Subsystem is installed the Subsystems implementation must determine the set of resources that compose the content of the Subsystem. The content resources of a Subsystem may be specified in the following ways:

- The values of the Subsystem-Content header must be used if it is present. See *Subsystem-Content Header* on page 627.
- The content of the Subsystem's Local Repository, if the Subsystem-Content header is not present.

When a Deployment Manifest is not present, *Pre-Calculated Deployment* on page 651, the Subsystems implementation must use this information to discover the content resources for a Subsystem as described in the following sections.

134.5.5.1 Declared Subsystem–Content

If the Subsystem-Content manifest header is declared then the following steps must be followed to discover the Subsystem's contents:

- The Subsystem-Content header is parsed into a list of elements where each element specifies a single `osgi.identity` requirement. For each `osgi.identity` requirement element a Requirement is created and used to search Repositories for content resources.
  - If the Subsystem is a Scoped Subsystem then continue to the next step; otherwise if the Subsystem is an Unscoped Subsystem then the System Repository must be searched in order to

discover any currently installed resources that match the content Requirement. For each matching capability found it must be determined if the capability provider Resource is a constituent of a Subsystem which is in the same Region as the installing Subsystem; if so then the provider Resource must be used as an *shared content* resource. If no *shared content* resource is found then continue to the next step; otherwise the search stops.

- The Subsystem's Local Repository is searched to find a matching Capability for the content Requirement. If a Capability is found then the providing Resource of the Capability is used as an *installable content* resource of the Subsystem. If no *installable content* resource is found then continue to the next step, otherwise the search stops.
- The registered Repository services are searched to find a matching capability for the content Requirement. If a Repository finds a provider for the content requirement then the provider Resource of the capability is used as an *installable content* resource of the Subsystem. If no matching provider is found then the discovery of the content resource has failed.

134.5.5.2   Use Subsystem Local Repository

If the Subsystem-Content header is not declared then the list of content resources is defined as all the Resources within the Subsystem's Local Repository which provide an `osgi.identity` capability with the type attribute of:

- `osgi.bundle`
- `osgi.fragment`
- `osgi.subsystem.application`
- `osgi.subsystem.composite`
- `osgi.subsystem.feature`
- Any other type that is supported by the implementation.

If a resource is found to be an unsupported type then installation of the Subsystem must fail.

For Scoped Subsystems this list is used as is and each Resource is considered an installable content resource. For Unscoped Subsystems the System Repository must be searched in order to determine if there are any already installed contents resources which may be used as a *shared content* resources. If no shared content resource can be found then the resource is considered an *installable content* resource.

134.6   Determining Dependencies

When a Subsystem is installed the Subsystems implementation determines the set of resources that compose the content of the Subsystem. Content resources may have requirements on capabilities that are not provided by any of the content resources for the Subsystem. When a Subsystem is installed the Subsystems implementation must determine the set of additional resources that are required in order to allow the Subsystem's content resources to resolve. These additional resources are called *dependencies*. When a Subsystem is installed the set of dependencies contains two types of resources:

- *Installable Dependency* - A resource which must be installed and result in a distinct resource at runtime. That is, a distinct bundle or Subsystem installation.
- *Preinstalled Dependency* - A resource which is already installed and is a constituent of one or more already installed Subsystems and the required capabilities provided by the resource are accessible according to the sharing policies of the Subsystems. This resource must be reused, the Subsystems implementation must not install another instance of the resource.

When a Deployment Manifest is not present, see *Pre-Calculated Deployment* on page 651, the Subsystems implementation must determine the set of dependencies for the Subsystem. To determine

the set of dependencies the Subsystems implementation should attempt to resolve the Subsystem content resources before installing the content resources. One possible way of resolving the content resources, before installing them, is to use a Resolver service, see *Resolver Service Specification* on page 691. This specification illustrates the behavior of dependency resource discovery using terms defined by the Resolver service. A Subsystems implementation is not required to use the Resolver service to accomplish dependency resource discovery. Other mechanisms may be used to accomplish the same goal as long as the resolution results in a valid class space. Subsystems implementations need not guarantee to find a solution to every resolution problem, but if a valid solution is not found, then discovery of the dependencies must fail, resulting in an installation failure.

In order to use the Resolver service the Subsystems implementation has to provide a `ResolveContext` object that represents the currently installed Subsystems and their constituent resources. This resolve context must search Repositories in the following order when searching for capabilities to satisfy content dependencies within the `ResolveContext.findProviders` method. The Content Repository, Preferred Provider Repository, System Repository and Local Repository must all be searched and all the results presented to the Resolver with a corresponding preference. If a result was found in these repositories, searching the registered Repository services is optional, but if no result was found yet, the registered Repository services must be searched. The order of matching capabilities presented to the Resolver must coincide with the Repository search order.

1. The Content Repository.
2. The Preferred Provider Repository.
3. The System Repository. For each matching capability found in the System Repository the Subsystems implementation must determine if the capability is accessible to the content resources of the installing Subsystem according to the sharing policy of the Subsystem. See *Sharing Capabilities* on page 635 for more details on Subsystem types and their sharing policies.
4. The Local Repository. This allows a Subsystem Archive to optionally include dependencies.
5. The registered Repository services.

Any matching capabilities found in the steps after step 1 above are considered to be provided by potential dependencies for the Subsystem. The capabilities found in the System Repository are provided by already installed resources, referred to as potential *pre-installed dependencies*. The capabilities found which are provided by other potential dependencies must be installed in order to resolve the Subsystem content resources, referred to as *installable dependencies*.

The Resolver's job is to select one of the potential capabilities returned by the `findProviders` method as the capability to satisfy a Requirement. At the end of a resolve operation a result (`Map<Resource, List<Wire>>`) is returned which contains the Resources that got resolved and a list containing the Wires for the resolved Resource. The Subsystems implementation uses this resolution result in order to determine which resolved Resources are content resources, pre-installed dependencies, or installable dependencies. The installable dependencies must be installed as described in *Accepting Dependencies* on page 633. The pre-installed dependencies must have their reference count incremented as described in *Reference Count* on page 640.

134.7 Accepting Dependencies

When a Subsystem is installed the Subsystems implementation must determine the set of installable dependencies as described in *Determining Dependencies* on page 632. The Subsystems implementation must also determine what Subsystem is willing to accept the installable dependency as a constituent, referred to as an *accepted dependency constituent*. A Subsystem declares that it is willing to accept dependencies as constituents by specifying the provision-policy directive with the `acceptDependencies` value on the Subsystem-Type header, see *Subsystem-Type Header* on page 620.

The `acceptDependencies` provision policy is useful for creating isolation layers that do not pollute parent Regions with dependencies. For example, an application container may be implemented as
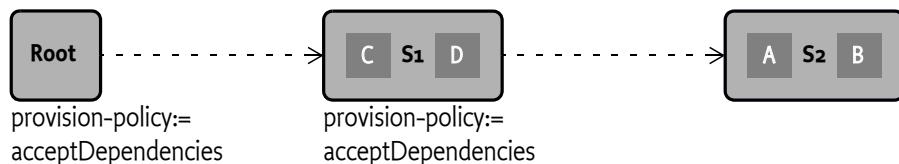
a Subsystem. Such a container Subsystem could be installed into something called a kernel Subsystem. Applications are installed as Subsystems into the container Subsystem. In this case the container Subsystem would likely use the `acceptDependencies` provision policy so that any applications installed into the container Subsystem do not end up polluting the kernel Subsystem with the application's dependencies.

A dependency becomes an accepted dependency constituent of the Subsystem with a provision policy of `acceptDependencies` and that lies on the longest path between the Root Subsystem and the Subsystem being installed, inclusively. Note that a Subsystem that has `acceptDependencies` provision policy will accept its own installable dependencies as constituents since it lies on the longest path between the Root Subsystem and itself, inclusively.

The following figure illustrates a simple example of accepting dependency constituents. A Scoped Subsystem S2 is being installed into another Scoped Subsystem S1 and S1 has a provision-policy of `acceptDependencies`. When S2 is being installed the Subsystems implementation discovers content resources A and B and installable dependencies C and D. This is depicted in Figure 134.10.

Figure 134.10   *Provision Policy*
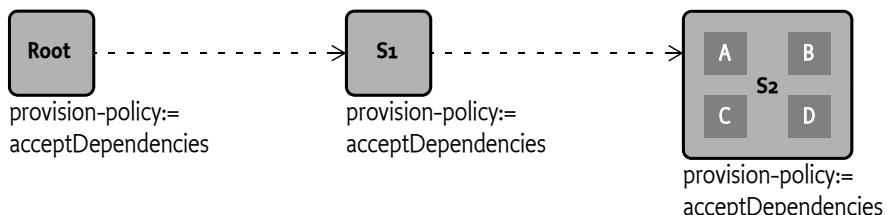
In the previous example the Subsystem with a provision-policy of `acceptDependencies` and that lies on the longest path between the Root Subsystem and the Subsystem being installed is S1. Therefore the installable dependencies C and D become accepted dependency constituents of S1.

Figure 134.11 illustrates the same example but with S2 also having a provision-policy of `acceptDependencies`
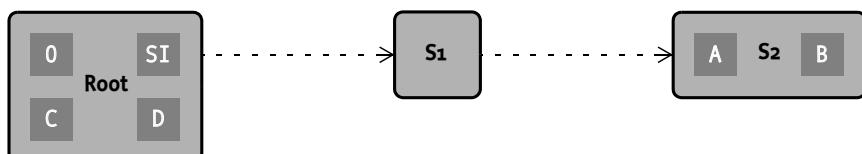
Figure 134.11   *Provision Policy*

In this example the Subsystem with a provision-policy of `acceptDependencies` and that lies on the longest path between the Root Subsystem and the Subsystem being installed is S2 itself. Therefore the installable dependencies C and D become accepted dependency constituents of S2.

Figure 134.12 illustrates the same example but with S1 and S2 not defining the provision-policy (default is `rejectDependencies`). The Root Subsystem always has a provision-policy of `acceptDependencies`:

Figure 134.12   *Subsystems and acceptDependencies*

In this example the Subsystem with a provision policy of acceptDependencies and that lies on the longest path between the Root Subsystem and the Subsystem being installed is the Root Subsystem. Therefore the installable dependencies C and D become accepted dependency constituents of the Root Subsystem.
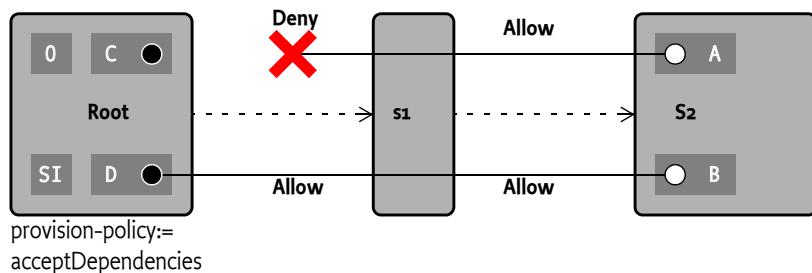
134.8 Sharing Capabilities

Scoped Subsystems define a sharing policy for the Region they are contained in. A sharing policy controls access to capabilities between parent → child Region boundaries. For Subsystems, a sharing policy is composed of two parts: an import policy and an export policy:

- *Export Policy* - Defines the set of capabilities provided by the constituents contained in the Region that are made available to the parent Region.
- *Import Policy* - Defines the set of capabilities which are available in the parent Region that are made available to the child Region.

The import sharing policies of a Subsystem's ancestor parent chain may prevent a Subsystem from accessing the required capabilities provided by a dependency. Figure 134.13 illustrates this.

Figure 134.13   *Sharing Capabilities*

provision-policy:=
acceptDependencies
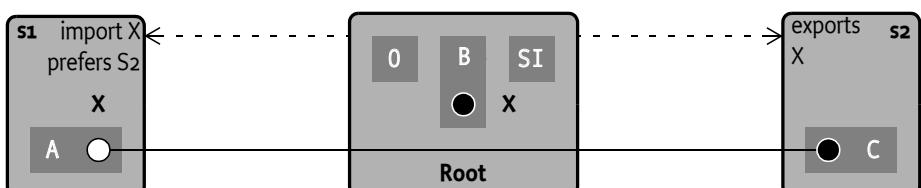
In this example the C and D provide capabilities that are required by A and B respectively. The import sharing policy of S2 allows the capabilities to be imported into S2, but the import sharing policy for S1 denies the import of one of the capabilities and allows the other. In order for A or B to access capabilities provided by C or D they must traverse both the import sharing policy of S2 and S1. Ultimately S1 sharing policy prevents the necessary access to the dependencies necessary to resolve S2. Such a situation must be detected during the discovery of the installable dependencies and result in a failure to install the dependencies. This must result in the failure to install the Subsystem that requires the dependency, in this case S2.

134.8.1 Preferred Provider

The sharing policy for a Scoped Subsystem may specify a set of preferred providers. If a capability is provided by a preferred provider then that capability must be used to resolve a Scoped Subsystem's import policy. Figure 134.13 illustrates this.

Figure 134.14   *Preferred Provider*

In this example constituent B of the Root Region provides the capability X (called Root→X). Also constituent C of S2Region provides the capability S2→X. The export sharing policy of S2 policy exports the capability S2→X to its parent Region, the RootRegion. The S1Region contains a constituent A that has a requirement on S1→X. The two capabilities, Root→X and S2→X, are available to satisfy the requirement S1→X. The S1Region's import sharing policy imports capability X and has a preferred provider of S2. This means that the capability S2→X must be used to satisfy the requirement S1→X.

134.8.2   System Capabilities

The osgi.ee and osgi.native namespaces are used by the System Bundle to describe capabilities for the Java execution environment and the native environment. These capabilities must only be provided by the System Bundle. A Subsystems implementation must allow access to the osgi.ee and osgi.native capabilities provided by the System Bundle to every Subsystem installed. This includes scoped subsystems which may or may not already have an import sharing policy configured to import these namespaces from the System Bundle.

134.9   Region Context Bundle

The Region context bundle provides a perspective from a constituent of a Subsystem contained in the Region. When a Scoped Subsystem is installed the Subsystems implementation must generate the Region context bundle and install it as a constituent bundle of the Subsystem. The Region context bundle has the following characteristics:

- Has a symbolic name of org.osgi.service.subsystem.region.context.<subsystem id>
- Version 1.0.0
- Has a location string of <subsystem location>/<subsystem id>
- Must always be allowed to resolve and start (i.e. has no requirements, imports or bundle activator).
- Has a start-level of 1 and is persistently started.
- Is installed and started before the Subsystem service is registered.

This bundle is installed and must remain active as long as the Subsystem is installed. If the Region context bundle is stopped, updated or uninstalled then the Subsystem runtime should log an error and may ensure the context bundle is installed and restarted.

When the Subsystems implementation is active it must establish the Root Subsystem, see *Subsystem Service* on page 643. In establishing the Root Subsystem the Subsystems implementation must ensure that there is a Region context bundle available for the Root Region. This Root Region context bundle will have a symbolic name of org.osgi.service.subsystem.region.context.0.

Typically the Region's context bundle would be used to obtain a bundle context with the getBundle-Context() method, which has a perspective as a constituent of the Region. This is useful in the following ways:

- Implementing Subsystem aware extenders. Such extenders need to be able to register listeners and monitor the inside of a Region in order to react to the constituent bundles of a Region.
- Monitoring of internal events.

134.10   Explicit and Implicit Resources

Depending on how a resource is installed the Subsystems implementation considers the resource to be either an *explicit resource* or *implicit resource*. An *explicit resource* is a resource whose installation was initiated by an agent outside of the Subsystems implementation. An *implicit resource* is a re-

source whose installation was initiated by the Subsystems implementation during the explicit installation of a Subsystem, including the content and dependencies of the explicitly installed Subsystem.

134.10.1 Explicit Resources

An explicit resource is a resource that was installed programmatically, by an agent outside of the Subsystems implementation, using some resource specific API. This specification defines two types of resources that can be installed explicitly:

- *Subsystem resource* - A Subsystem resource may be installed explicitly by using the Subsystem service. Note that content and dependencies of an explicitly installed Subsystem are not considered to be explicit resources themselves since they were implicitly installed by the Subsystems implementation.
- *Bundle resource* - A bundle resource may be installed explicitly by using a bundle context. This includes fragments.

134.10.1.1 Explicit Bundle Resources

When a bundle is installed explicitly with a bundle context, the Subsystems implementation must determine the Subsystem of which a bundle becomes a constituent. The following rules are followed when a bundle is installed explicitly with a bundle context:

1. Determine the bundle performing the install. This is the bundle whose Bundle Context is performing the install operation.
2. Determine the target Region. This is the Region to which the bundle performing the install operation belongs.
3. If a bundle with the same symbolic name and version already exists in the target Region then the bundle installation must fail unless the same location string is used. If the same location is used then the existing bundle is returned. This may be accomplished by the use of a bundle collision hook.
4. Determine the Subsystem(s) of which the bundle performing the install is a constituent. The bundle performing the install may be a shared resource. In that case the bundle performing the install is a constituent of two or more Subsystems.
5. The newly installed bundle must become a constituent of all the Subsystems of which the bundle performing the install is a constituent.

134.10.1.2 Explicit Subsystem Resources

When a Subsystem is installed explicitly with a Subsystem service, the Subsystems implementation must determine what Subsystem(s) the Subsystem resource and its children will become a constituent of. The following rules are followed when a Subsystem is installed:

1. Determine the target Subsystem. This is the Subsystem service which is performing the install operation or the Subsystem which includes another Subsystem as part of its content, see *Determining Content* on page 627.
2. Determine the target Region. This is the Region to which the target Subsystem belongs.
3. If no Subsystem resource with the same location exists then continue to the next step; otherwise do the following:
   - If the existing Subsystem is not a part of the target Region then fail the install operation by throwing a Subsystem Exception; otherwise continue to the next step.
   - If the existing Subsystem symbolic name, version and type is not the same as the Subsystem being installed then fail the install operation by throwing a Subsystem Exception; otherwise continue to the next step.
   - If the existing Subsystem is already a constituent of the target Subsystem then return the existing Subsystem from the install method; otherwise the existing Subsystem resource be-

comes a shared resource by increasing the reference count of the existing Subsystem by one, see *Reference Count* on page 640, and the existing Subsystem becomes a constituent of the target Subsystem; finally, the existing Subsystem is returned from the install method.

4. If no Subsystem resource with the same symbolic-name and version already exists in the target Region then the Subsystem resource being installed becomes a constituent of the Subsystem target; otherwise do the following:

- If the existing Subsystem type is not the same as the type of the Subsystem being installed then fail the install operation by throwing a Subsystem Exception; otherwise continue to the next sub-step.

- If the existing Subsystem is already a constituent of the target Subsystem then return the existing Subsystem from the install method; otherwise the existing Subsystem resource becomes a shared resource by increasing the reference count of the existing Subsystem by one and the existing Subsystem becomes a constituent of the target Subsystem; finally, the existing Subsystem is returned from the install method.
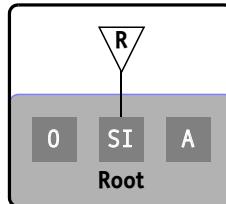
134.10.2 Explicit Resource Example

A scenario is used to illustrate the rules for determining which Subsystem an explicitly installed resource is a constituent. Figure 134.15 illustrates the Root Subsystem with initial content bundles A, SI (Subsystems implementation) and the system bundle (id = 0) installed.
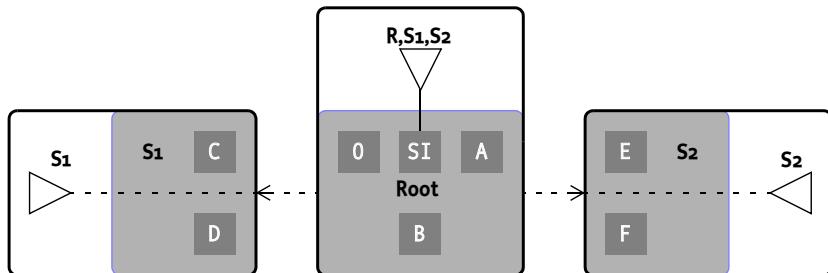
Figure 134.15   *Explicit Resource Example*

If bundle A uses its own Bundle Context to explicitly install bundle B then bundle B becomes a constituent of the Root Subsystem. If bundle A uses Subsystem R to install Scoped Subsystem S1 then the S1 resource becomes a constituent of the Root Subsystem and S1 Subsystem becomes a child of the Root Subsystem. S1 contains constituent bundles C and D. Also, if bundle B uses Subsystem R to install Scoped Subsystem S2 then the S2 resource becomes a constituent of the Root Subsystem and the S2 Subsystem becomes a child of the Root Subsystem. S2 contains constituent bundles E and F. Figure 134.16 illustrates this.
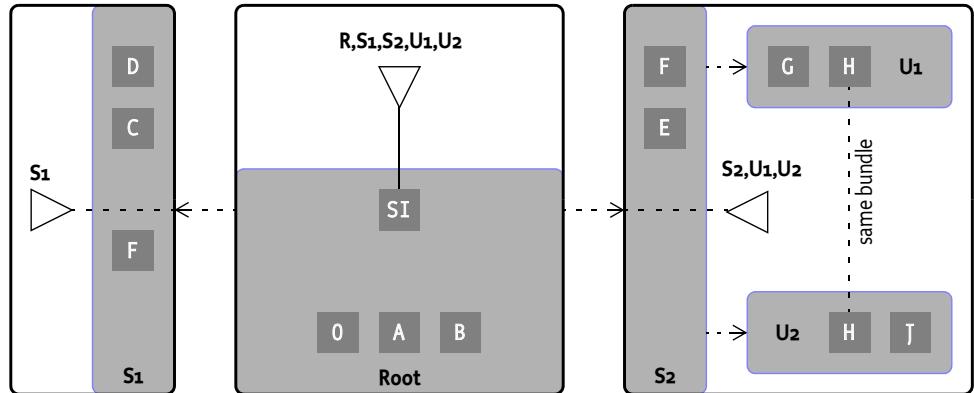
Figure 134.16   *Subsystems and Resources*

Then if bundle C uses its own Bundle Context to install bundle F (using a different location string from constituent bundle S2→F) then the bundle becomes a constituent of S1. If bundle E uses Sub-

system service S2 to install Unscoped Subsystem U1 (with constituents G and H) and installs Unscoped Subsystem U2 (with constituents H and J) then both Subsystem bundles U1 and U2 become constituents of S2. The S2 Subsystem also becomes the parent Subsystem for both U1 and U2 Subsystems, see Figure 134.17.
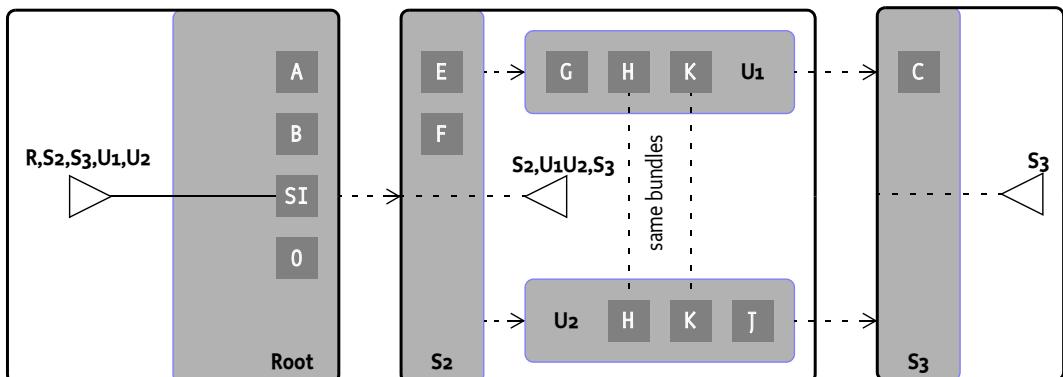
Figure 134.17   Subsystems and Resources

In this scenario bundle H is a shared constituent of both U1 and U2 Subsystems. If bundle H installs a bundle K with its bundle context then bundle K becomes a shared constituent of both U1 and U2 Subsystems. Also, if Subsystem service U1 is used to install Scoped Subsystem S3 and Subsystem service U2 is also used to install Subsystem S3 then S3 resource becomes a shared constituent of both Unscoped Subsystems U1 and U2. The following illustrates this:
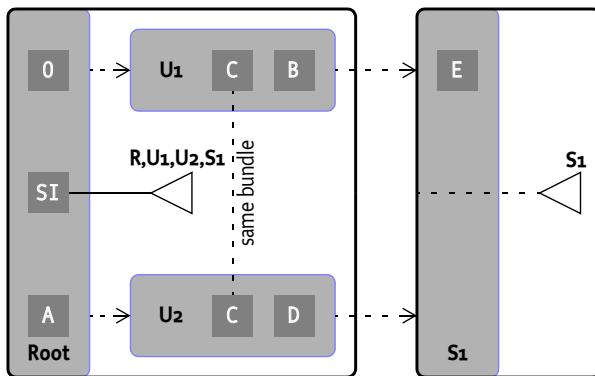
Figure 134.18   Subsystems and Resources

Since the S3 Subsystem resource is a shared constituent of both Subsystems U1 and U2 the S3 Subsystem has two parents: U1 and U2. In this case S3 Subsystem has two parent Subsystems but the S3Region still must only have one parent of S2Region. This is enforced by the rule that requires all of the parents of a Subsystem to belong to the same Region. For Scoped Subsystems the Region which contains all of the Subsystem's parents is parent Region.

So far the examples have illustrated cases where the Root Subsystem has Scoped Subsystem children. It is also acceptable for an Unscoped Subsystem to be installed into the Root Region as the following figure illustrates:

Subsystems and Resources

134.11 Resource References

A Subsystems implementation must track the resources which are installed and determine which Subsystems reference a resource. The *reference count* indicates the number of installed Subsystems which reference an installed resource. The resource references and reference counts are used by the Subsystems implementation to determine if an installed resource is eligible for garbage collection and also plays a role in determining when a resource should be started and stopped, see *Starting and Stopping Resources* on page 641 ; the term reference count is only used to illustrate these concepts. The reference count of a resource is not exposed in the API of Subsystems. The following types of resources are referenced by a Subsystem:

- *Content Resources* - These are the content resources which were installed when the Subsystem was installed, that is the resources declared in the Subsystem-Content header or from the Local Repository when the Subsystem-Content header is not specified, see *Determining Content* on page 627. Content Resources are considered to be implicit resources.
- *Explicit Resources* - These are constituent resources which are installed explicitly, see *Explicit Resources* on page 637.
- *Dependencies* - These resources provide capabilities required to satisfy requirements for a Subsystem's content resources, see *Determining Dependencies* on page 632. Dependencies are considered to be implicit resources.

Accepted dependency constituents are not defined as being referenced by the Subsystem of which they are a constituent unless constituent resource is a dependency for that Subsystem. Parent Subsystems are also not defined as being referenced by a child Subsystem.

134.11.1 Reference Count

When a Subsystem is being installed the Subsystems implementation must determine what resources are referenced by the Subsystem being installed. Each resource that is referenced by the Subsystem being installed will have its reference count incremented by 1. A top-level Subsystem being installed may contain child Subsystems. Each resource that is referenced by the child Subsystem being installed will have its reference count incremented by 1.

When a Subsystem is being uninstalled the Subsystems implementation must determine what resources are referenced by the Subsystem being uninstalled. Each resource that is referenced by the Subsystem being uninstalled will have its reference count decremented by 1. A top-level Subsystem being uninstalled may contain child Subsystems. Each resource that is referenced by each child Subsystem being uninstalled will have its reference count decremented by 1.

When a reference count gets set to zero then the resource is eligible for garbage collection and will be uninstalled. A Subsystems implementation may perform the garbage collection immediately or postpone the garbage collection for later. If garbage collection is postponed then the Subsystems implementation must prevent any additional usage of capabilities provided by the resource which is to be uninstalled. The garbage collection must occur in a reasonable period of time.

Bundle resources (including fragments) and Subsystem resources may be uninstalled explicitly. For example, uninstalling a Subsystem resource through the Subsystem service, or by other means outside of the Subsystems API such as uninstalling a bundle using a Bundle object. Each of the following must occur when a resource is explicitly uninstalled:

- If the resource being explicitly uninstalled was not itself installed explicitly then an error must be logged indicating that the explicitly uninstalled resource still has one or more Subsystems referencing it.
- If the resource being explicitly uninstalled was itself installed explicitly and the reference count is greater than 1 then an error must be logged indicating that the explicitly uninstalled resource still has one or more Subsystems referencing it.
- The resource being explicitly uninstalled has its reference count set to 0 and finally the resource is uninstalled.

134.12 Starting and Stopping Resources

A Subsystem can be started by calling the Subsystem *start* method or the Subsystems implementation can automatically start the Subsystem if the Subsystem is *ready* and the *autostart setting* of the Subsystem indicates that it must be started.

A Subsystem is *ready* if the Subsystem's parent is in the process of starting or is active. A started Subsystem may need to be automatically started again by the Subsystems implementation after a restart. The Subsystems implementation therefore maintains a persistent *autostart setting* for each Subsystem. This *autostart setting* can have the following values:

- *Stopped* - The Subsystem should not be started.
- *Started* - The Subsystem must be started once it is ready.

Subsystem resources which are installed as content resources, see *Discovering Content Resources* on page 631, of one or more Subsystems must have their *autostart setting* set to *started*.

When a Subsystem is started and stopped then the resources the Subsystem references may be started and stopped. See for details *Starting* on page 649 and *Stopping* on page 649.

The Subsystems implementation must track the resources which are installed and be able to determine when a resource must be started and stopped. To describe this behavior the term *active use count* is used. A active use count indicates the number of active Subsystems which reference a resource. The active use count is used by the Subsystems implementation in order to determine when a resource is started and stopped. The term active use count is only used to illustrate the starting and stopping of resources. The active use count of a resource is not exposed in the API of Subsystems.

Resource starting and stopping only applies to resources for which it is valid to start and stop. For example, it is not valid to start or stop resources of type osgi.fragment and a Subsystems implementation must not attempt to start or stop such resources.

134.12.1 Start Order

A Subsystem's Subsystem-Content header, see *Subsystem-Content Header* on page 627, can use the optional start-order directive for each content resource it declares. The start-order directive specifies the start order of the content resource in relation to other content resources of the Subsystem. Content resources are started in ascending order and stopped in descending order according to the

start-order directive values. Content resources with the same start-order value may be started and stopped in any order in relation to each other. There is no default value for `start-order`. If the `start-order` is not specified then a Subsystem implementation is free to start the resource in any order. For example, the following header specifies four Subsystem content resources:

```
Subsystem-SymbolicName: S1
Subsystem-Type: osgi.subsystem.composite
Subsystem-Content:
    A;
        type=osgi.bundle;
        version=1.0;
        start-order:=3,
    B;
        type=osgi.bundle;
        version=1.0;
        start-order:=2,
    C;
        type=osgi.bundle;
        version=1.0;
        start-order:=1,
    D;
        type=osgi.bundle;
        version=1.0;
        start-order:=2
```

The above headers specify a Subsystem S1 with four content resources: A, B, C and D. The `start-order` directive is used to sort the content resources to determine the order to start or stop them when the Subsystem is started or stopped. The content resources are sorted from lowest to highest start-order. Content resources that have the same `start-order` value may be started and stopped in any order in relation to each other. In this example the content resources are sorted into the list [C], [B, D], [A]. C has the lowest start-order, therefore it is the first in the list. B and D have the same `start-order` and therefore can be started in any order in relation to each other. Finally A is last in the list because it has the highest start-order.

When the Subsystem S1 is started the content resource C must be started first, followed by the starting of B and D in any order, finally resource A is started last. When the Subsystem S1 is stopped the content resource A must be stopped first, followed by the stopping of B and D in any order, finally resource C is stopped last.

Resources that do not specify a start-order can be started and stopped in any order.

134.12.2   Active Use Count

When a Subsystem is being started the Subsystems implementation must increment the active use count of every resource which is referenced by the Subsystem being started, see *Resource References* on page 640. After incrementing the active use counts of the resources referenced by a Subsystem, the Subsystems implementation must determine which referenced resources need to be started. For each resource referenced by the Subsystem which is valid to be started; if the active use count is greater than zero and the resource is not currently active then the resource must be started. The collection of dependencies are started before the Subsystem's content resources. The start order for the individual resources contained in the collection of dependencies is not specified. See *Start Order* on page 641.

When a Subsystem is being stopped the Subsystems implementation must decrement the active use count of every resource which is referenced by the Subsystem being stopped. After decrementing the active use counts of the resources referenced by a Subsystem, the Subsystems implementation must determine which referenced resources need to be stopped. For each resource referenced by the Subsystem which is valid to be started; if the active use count equals zero and the resource is

currently active then the resource must be stopped. The Subsystem content resources are stopped before the collection of dependencies. *Start Order* on page 641 describes the stop order of the Subsystem's content resources. The stop order of the individual resources contained in the collection of dependencies is not specified.

When starting the resource types supported by this specification the following rules apply:

- `osgi.bundle` - The bundle must be transiently started using the activation policy of the bundle, that is with the `Bundle.START_ACTIVATION_POLICY`.
- `osgi.fragment` - fragments cannot be started, this is a no-op.
- `osgi.subsystem.application`, `osgi.subsystem.composite`, `osgi.subsystem.feature` - The Subsystem must be transiently started if its *autostart setting* is set to *started*.

When stopping the resource types supported by this specification the following rules apply:

- `osgi.bundle` - The bundle must be persistently stopped.
- `osgi.fragment` - Fragments cannot be stopped, this is a no-op.
- `osgi.subsystem.application`, `osgi.subsystem.composite`, `osgi.subsystem.feature` - The Subsystem must be transiently stopped. Its *autostart setting* must not be changed.

Note that for resources referenced by a stopped Subsystem; bundle resources are persistently stopped and Subsystem resources are transiently stopped. This is a safeguard to handle cases where a constituent bundle is eagerly started by some other agent outside of the Subsystems implementation. Persistently started bundles will get auto started by the framework according to the start-level of the bundle. This can cause a constituent bundle to be stopped even though the Subsystem it is a constituent of is not active. To avoid this situation the Subsystems implementation always clears the persistent autostart setting of the bundle resources.

Subsystem resources which are referenced by a Subsystem are started or stopped transiently. There is no API to transiently start or stop a Subsystem. The Subsystems implementation must perform the starting or stopping of a referenced Subsystem normally except the starting or stopping process does not change the *autostart setting* for the referenced Subsystem.

134.13 Subsystem Service

The Subsystem service represents an Subsystem Archive resource that is installed in an OSGi Framework. The installation of a Subsystem can only be performed by using the Subsystem service API or through implementation specific means. The Subsystem interface's methods and service properties can be divided into the following categories:

- *Information* - Access to information about the Subsystem itself as well as other Subsystems that are installed.
- *Life Cycle* - The possibility to install other Subsystems and start, stop, and uninstall Subsystems.

For each Subsystem installed, the Subsystems implementation must register an associated Subsystem object as a service. The Subsystem service is used for monitoring the state of the Subsystem, for controlling the life cycle of the installed Subsystem and for installing child Subsystems.

134.13.1 Root Subsystem

A Subsystems implementation must register the Root Subsystem service. When a Subsystems implementation is started for the first time it must establish the Root Subsystem. The following steps are required to establish the Root Subsystem.

1. The Root Subsystem has a symbolic name `org.osgi.service.subsystem.root`, version `1.0.0` (the version of the Subsystems specification), a Subsystem id of 0 and a location of

```
subsystem://?Subsystem-SymbolicName=org.osgi.service.subsystem.root& «
 Subsystem-Version=1.0.0
```

2. The Root Subsystem has no parent Subsystem. More specifically the Root Subsystem is the only source vertex in the Subsystem graph.

3. The Root Subsystem is considered a Scoped Subsystem of type application, with provision-policy of acceptDependencies. Since the Root Subsystem has no parent it does not import or export any capabilities.

4. The Subsystem content is the set of bundles installed in the framework that do not belong to any other Subsystem.

5. The root subsystem has a region context bundle as described in *Region Context Bundle* on page 636.

The Root Subsystem always exists when a Subsystems implementation is present and active, even if no other Subsystems are installed The Root Subsystem is used as the starting point for installing Subsystems as child Subsystems. The Root Subsystem cannot be stopped or uninstalled by calling the Subsystem service stop or uninstall methods. Any attempt to do so must result in a Subsystem Exception.

134.13.2 Subsystem Service Properties

The primary means of discovering and monitoring a Subsystem is the Subsystem service. A Subsystems implementation must register one Subsystem service for each Subsystem installed. The Subsystem service is used for monitoring and controlling the life-cycle of the installed Subsystem. Service properties on the Subsystem service carry most of the information required to monitor Subsystem life cycle operations and the current state of a Subsystem. The following table describes the service properties of a Subsystem:

Table 134.1   *Subsystem Mandatory Service Properties*

Key Name	Type	Description
subsystem.id	Long	The Subsystem id of the Subsystem
subsystem.symbolicName	String	The symbolic name of the Subsystem
subsystem.version	Version	The version of the Subsystem
subsystem.type	String	The type of Subsystem.
subsystem.state	Subsystem.State	Contains the current state of the Subsystem

134.13.3 Subsystem States

The Subsystem service property subsystem.state contains the current state of the Subsystem (this is referred to as the *subsystem state*). All Subsystem states are defined by the Subsystem.State enum, for example, INSTALLED. The possible values of a subsystem.state are shown in the table below:

Table 134.2   *Subsystem States*

subsystem.state	Description
INSTALLING	When a Subsystem is first installed the Subsystems implementation must register a Subsystem service with the initial subsystem.state of INSTALLING. The subsystem.state must remain in the INSTALLING state until all of the Subsystem constituents are installed successfully.
INSTALLED	When all contents of a Subsystem has been successfully provisioned then the subsystem.state is set to INSTALLED.
INSTALL_FAILED	Indicates that some failure occurred while attempting to install the Subsystem's contents.

subsystem.state	Description
RESOLVING	Starting a Subsystem triggers the resolution of a Subsystem if the subsystem.state is INSTALLED. A RESOLVING state indicates that a resolve process is occurring in an attempt to resolve all of the subsystem's content resources.
RESOLVED	Indicates that the Subsystem is resolved. A Subsystem is resolved if all of its content resources are resolved.
STARTING	Indicates that the Subsystem is in the process of being started. During this state the resources the Subsystem references which are eligible for starting are started, see *Starting and Stopping Resources* on page 641. Once all the eligible resources are successfully started then the subsystem.state is set to ACTIVE.
ACTIVE	The ACTIVE state indicates that all eligible resources referenced by the Subsystem were successfully started during the starting process.
STOPPING	Indicates that the Subsystem is in the process of being stopped. During this state the resources referenced by the Subsystem are stopped if appropriate.
UNINSTALLING	Indicates that the Subsystem is in the process of being uninstalled. During this state the resources referenced by the Subsystem are marked for garbage collection if they are eligible, see *Resource References* on page 640.
UNINSTALLED	When all of the resources referenced by the Subsystem which are eligible for garbage collection have been uninstalled then the subsystem.state is set to UNINSTALLED.

134.13.4 Subsystem Service Registrations

The Subsystems implementation must register one Subsystem service for each Subsystem installed. The Subsystems implementation must provide access to every Subsystem service from the Root Region. Every other Region must have access to the following Subsystem Services:

- Subsystem service representing the Scoped Subsystem of the Region.
- All Unscoped Subsystem services contained in the Region.
- All Subsystems which are children of a Subsystem contained in the Region.

A Region is granted access to the Subsystem services listed above automatically by the Subsystems implementation regardless of the sharing policy defined by the Scoped Subsystem of that Region. Additional Subsystem services may be imported into a Region from its parent Region by the sharing policy defined by the Scoped Subsystem of that Region.

For example, a Root Subsystem and Root Region that has two Scoped Subsystem children, S1 and S2. All Subsystem services are registered by the Subsystems implementation and are visible in the Root Region. The S1 Subsystem service is also implicitly visible in the S1 Region because it represents the Scoped Subsystem S1 contained in that Region. Similarly the S2 Subsystem service is also implicitly visible from the S2 Region. This example is depicted in Figure 134.20.

Figure 134.20   *Root, attached to Scoped Subsystems S1, S2*
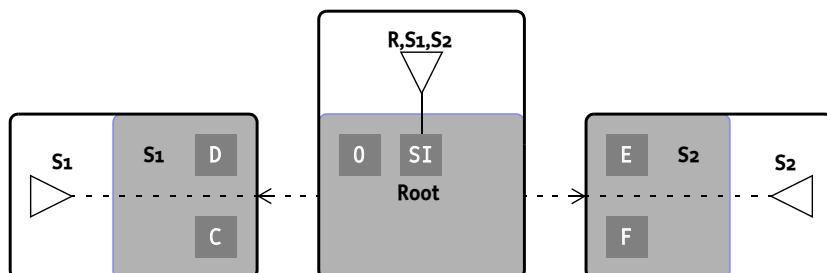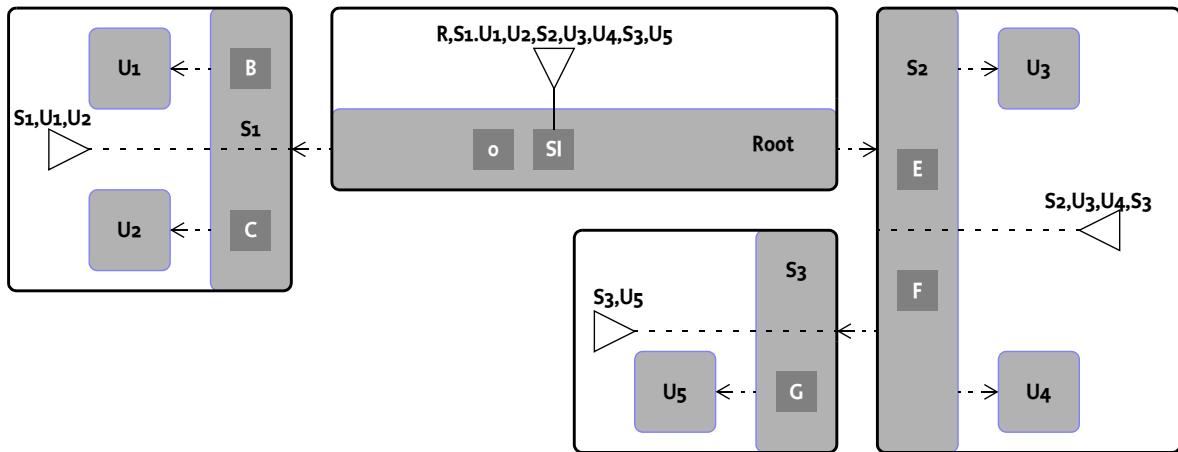
Figure 134.21 defines a more complicated scenario where Subsystems and multiple children are involved.

Figure 134.21   *Complex example Subsystem scoping*

All Subsystem services are visible in the Root Region. The S1 Subsystem service is also implicitly visible in the S1 Region because it represents the Scoped Subsystem S1 contained in that Region. The S1 Region also has visibility to the U1 and U2 Subsystem services because these Unscoped Subsystems are contained in the S1 Region. Similarly the S2 Subsystem service is also implicitly visible from the S2 Region. The S2 Region also has visibility to the U3 and U4 Subsystem services because these Unscoped Subsystems are contained in the S2 Region. The S2 Region also has visibility to the S3 Subsystem service because the S3 Subsystem is a child of a Subsystem contained in the S2 Region. Finally, the S3 Region has implicit visibility to the S3 Subsystem service and it has visibility to the U5 Subsystem service because the Unscoped Subsystem is contained in the S3 Region.

Note that a Scoped Subsystem's import sharing policy may grant its Region visibility to additional Subsystem services.

134.13.5 Subsystem Manifest Headers

The Subsystem service interface has the getSubsystemHeaders(Locale) method which returns the values of the Subsystem's manifest headers. The headers returned by this method includes the values specified in the Subsystem manifest file and the values derived by the Subsystems implementation. Certain manifest headers may be derived at install time by the Subsystems implementation if they were not specified in the Subsystem manifest file. When a Subsystem manifest value is derived then the derived value must be included in the headers returned by the method getSubsystemHeaders(Locale). The following Subsystem manifest headers may be derived by the Subsystems implementation:

- Subsystem-SymbolicName
- Subsystem-Version
- Subsystem-Content

134.14 Subsystem Life Cycle

The Subsystems specification provides an API to control the life cycle operations of a Subsystem. For each Subsystem installed there is an associated Subsystem object (also registered as a Subsystem service). A Subsystem's life-cycle is controlled by operations performed on the Subsystem object. Operations performed on the Subsystem may also cause equivalent operations on the resources referenced by the Subsystem. For example starting a Subsystem will cause all of its content resources to start if appropriate.
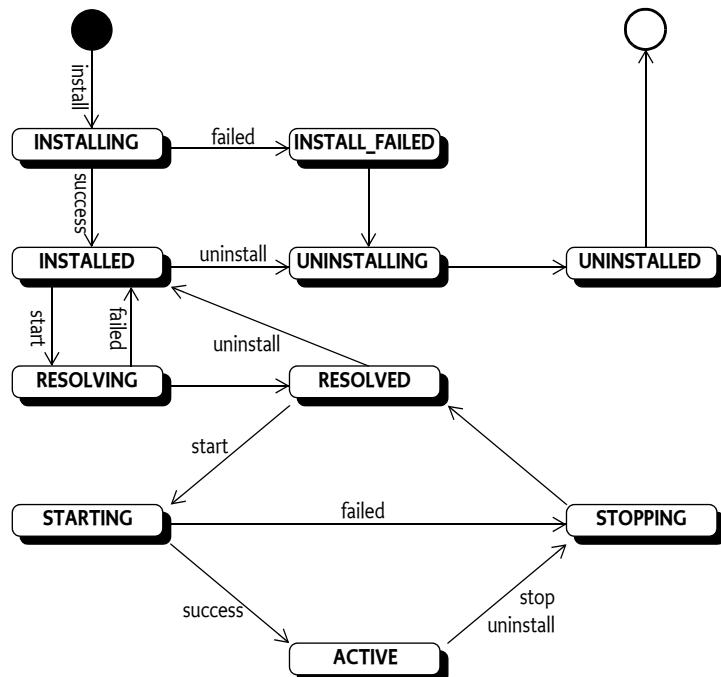
For Scoped Subsystems the export and import sharing policies are initially disabled at runtime and get enabled at runtime by the Subsystems implementation depending on the state of the Scoped Subsystem which defines the sharing policy. When an import sharing policy is disabled at runtime, none of the installed resources contained in the Region associated with the Scoped Subsystem have visibility to capabilities available in the parent Region. Once an import policy is enabled at runtime the installed resources contained in the Region have visibility to capabilities available in the parent Region according to what the import sharing policy specifies. When an export sharing policy is disabled at runtime, none of the capabilities provided by installed resources contained in the Region associated with the Scoped Subsystem are visible in the parent Region. Once an export policy is enabled at runtime the capabilities provided by installed resources contained in the Region are visible in the parent Region according to what the export sharing policy specifies.

The `subsystem.state` is a reflection of the last action performed on the Subsystem through the Subsystem service. The use of any other API to change the state of a resource referenced by a Subsystem directly does not result in a change of the `subsystem.state` (i.e. calling stop on a bundle). For example, uninstalling a Subsystem content resource which is a bundle does not cause the Subsystem to be uninstalled, but it does result in an error being logged.

All references to changing the state of this Subsystem include both changing the state of the Subsystem object as well as the state property of the Subsystem service.

The following figure illustrates the life cycle of a Subsystem:

Figure 134.22   *State diagram Subsystems*

134.14.1 Installing

A Subsystem's install process is initiated using one of the Subsystem service's *install* methods. The Subsystems implementation must assign a unique Subsystem identifier that is higher than any previous installed Subsystem identifier. Previously installed Subsystem identifiers include Subsystems which were uninstalled in a previous session of the framework. The installation of a Subsystem must be:

- *Persistent* - The Subsystem must remain installed across framework and Java VM invocations until the Subsystem is explicitly uninstalled.
- *Atomic* - The install method must completely install the Subsystem or, if installation fails, the Subsystems implementation must leave the framework in the same state as it was before the method was called.

Once a Subsystem has been installed, a `Subsystem` object is created and all remaining life cycle operations for the installed Subsystem must be performed upon this object. The returned `Subsystem` object can be used to start, stop, and uninstall the Subsystem as well as install child Subsystems.

When a Subsystem is being installed the Subsystems implementation must perform the following operations synchronously before returning from the install method:

1. Determine the symbolic name, version, and type for the Subsystem being installed as defined in *Subsystems* on page 616. If the Subsystem name, version or type are invalid then the install fails and a Subsystem Exception is thrown.
2. Determine the Subsystems for which the Subsystem being installed will become a constituent of by following the steps in *Explicit and Implicit Resources* on page 636.
3. Determine the Subsystem identifier. Subsystem identifiers are unique and assigned by the Subsystems implementation.
4. If the Subsystem is a Scoped Subsystem then create the new Region for the Subsystem and install and start the Region context bundle. See *Region Context Bundle* on page 636 for the Region context bundle.
5. Register a Subsystem service with the initial `subsystem.state` service property set to INSTALLING. This Subsystem service represents the Subsystem resource. See *Subsystem Service Properties* on page 644 and *Subsystem Service Registrations* on page 645 for more details.
6. Determine the Subsystem content resources. See *Determining Content* on page 627 for details on how the Subsystem contents are determined. If the contents cannot be discovered successfully and the content is not optional then an installation failure occurs and a Subsystem Exception is thrown. Otherwise continue to the next step.
7. Determine the Subsystem dependencies. See *Determining Dependencies* on page 632 for details on determining the Subsystem's dependencies. If the dependencies cannot be determined successfully then an installation failure occurs and a Subsystem Exception is thrown. Otherwise continue to the next step.
8. Install the dependencies. The Subsystems implementation must prevent resolution of dependency wires to the capabilities provided by the installed dependencies until the Subsystem has successfully entered INSTALLED state. See *Explicit and Implicit Resources* on page 636 for details on where dependencies are installed and see *Resource References* on page 640 for how they are tracked. If any dependency fails to install then an installation failure occurs and a Subsystem Exception is thrown. Otherwise continue to the next step.
9. Install content resources. The content resources must be disabled from resolving until the Subsystem has successfully entered INSTALLED state. If any content resource fails to install then and installation failure occurs and a Subsystem Exception is thrown. Otherwise continue to the next step.
10. If the Subsystem is scoped, enable the import sharing policy for the Region. See *Sharing Capabilities* on page 635.
11. Enable resolution for all of the Subsystem content and any dependencies installed. Set the `subsystem.state` to INSTALLED and return the installed `Subsystem` object.

The state INSTALL_FAILED is used to inform about an installation failure. All installation failures use the following steps:

1. When a Subsystem fails to install it enters the INSTALL_FAILED state.
2. Immediately transition the Subsystem to the UNINSTALLING state.

3. All content and dependencies which may have been installed by the Subsystem installing process must be uninstalled.

4. Transition the Subsystem to the UNINSTALLED state.

5. Unregister the Subsystem service.

6. If the Subsystem is scoped then, uninstall the Region context bundle.

7. Throw a Subsystem Exception indicating an install failure.

134.14.2   Resolving

A Subsystem's resolve process is initiated by performing a start operation on a Subsystem whose subsystem.state is currently set to INSTALLED. There is no explicit operation for initiating the resolve process of a Subsystem. The Subsystems implementation is free to initiate the resolve process for a Subsystem for any reason. For example, the Subsystems implementation may choose to try to resolve all currently installed Subsystems when the start operation is performed on a single Subsystem.

134.14.3   Starting

A Subsystem can be started by calling the Subsystem start() method or the Subsystems implementation can automatically start the Subsystem if the Subsystem is *ready* and the *autostart setting* of the Subsystem indicates that it must be started. When a Subsystem is being started the Subsystems implementation must perform the following operations synchronously before returning from the start() method:

1. If the subsystem.state is INSTALL_FAILED, UNINSTALLED, or UNINSTALLING, then an Illegal State Exception is thrown.

2. Set the Subsystems *autostart setting* to *started*.

3. If the subsystem.state is ACTIVE then the start method returns immediately.

4. If the Subsystem is not *ready* to be started then the start method returns immediately.

5. If this subsystem.state is RESOLVING, STARTING or STOPPING, then the start method must wait for starting or stopping to complete before continuing. If this does not occur in a reasonable time, a Subsystem Exception is thrown to indicate the Subsystem was unable to be started.

6. If the subsystem.state is RESOLVED then continue to the next step; otherwise if the subsystem.state is INSTALLED then the subsystem.state is set to RESOLVING and an attempt is made to resolve all of the Subsystem's content resources. If all contents are resolved then set the subsystem.state to RESOLVED, enable the export sharing policy and continue to the next step; otherwise a starting failure occurs and a Subsystem Exception is thrown.

7. Set the subsystem.state to STARTING.

8. Start all resources referenced by the Subsystem according to *Starting and Stopping Resources* on page 641. If all of the resources start successfully then continue to the next step; otherwise a start failure occurs.

9. Set the subsystem.state to ACTIVE and return.

All start failures use the following steps:

1. If the subsystem state is STARTING then change the state to STOPPING.

2. Stop all resources that were started as part of this operation.

3. Change the state to INSTALLED or RESOLVED depending on if the Subsystem was resolved.

4. Throw a Subsystem Exception indicating the cause of the start failure.

134.14.4   Stopping

A Subsystem's stop process is initiated using the Subsystem service's stop() method. When a Subsystem is being stopped the Subsystems implementation must perform the following operations synchronously before returning from the stop() method:

1. If the subsystem.state is UNINSTALLED, INSTALL_FAILED, or UNINSTALLING, then an Illegal State Exception is thrown.
2. Set the Subsystems *autostart setting* to *stopped*.
3. If the subsystem.state is RESOLVED or INSTALLED then the stop() method returns immediately.
4. If this subsystem.state is STARTING or STOPPING, then the stop method must wait for starting or stopping to complete before continuing. If this does not occur in a reasonable time, a Subsystem Exception is thrown to indicate the Subsystem was unable to be stopped.
5. Set the subsystem.state to STOPPING.
6. Stop all resources referenced by the Subsystem according to *Starting and Stopping Resources* on page 641. If any error occurs while stopping a resource the Subsystems implementation must continue to stop the remaining resources that are eligible to stop.
7. Set the subsystem.state to RESOLVED.

With regard to error handling while stopping resources referenced by the Subsystem, errors subsequent to the first should be logged. Once the stop process has completed, a Subsystem Exception must be thrown with the initial error as the specified cause.

134.14.5 Uninstalling

A Subsystem's uninstall process is initiated using the Subsystem service's uninstall() method. To whatever extent possible, the Subsystems implementation must determine the resources referenced by the Subsystem which are eligible for garbage collection, *Reference Count* on page 640. This method must always uninstall the Subsystem from the persistent storage of the Subsystems implementation.

Once this method returns, the state of the platform must be the same as if the Subsystem had never been installed, unless some bundle resource which was uninstalled has exported package which are being used by other bundles still installed in the platform. All old exports must remain available for existing bundles and future resolves until the uninstalled bundle is refreshed or the framework is restarted.

When a Subsystem is being uninstalled the Subsystems implementation must perform the following operations before returning from the uninstall() method:

1. If the subsystem.state is UNINSTALLED then this method returns immediately.
2. If the subsystem.state is STARTING, STOPPING or ACTIVE then the Subsystem is stopped according to *Stopping* on page 649. Otherwise continue to the next step.
3. If the subsystem.state is INSTALLING and the installing process is interruptible, fail the install process; otherwise, wait until the installation is complete.
4. If the subsystem.state is in the INSTALL_FAILED state then skip to step 6.
5. Set the subsystem.state to INSTALLED.
6. Set the subsystem.state to UNINSTALLING.
7. Determine the resources referenced by the Subsystem which are eligible for garbage collection according to *Reference Count* on page 640. If a Subsystems implementation does garbage collection synchronously and any error occurs while uninstalling a resource the Subsystems implementation must continue to uninstall the remaining resources that are eligible to garbage collect.
8. Set the subsystem.state to UNINSTALLED.
9. Unregister the Subsystem service.
10. If the Subsystem is a Scoped Subsystem then uninstall the Region context bundle. At this point the Region no longer exists.

With regard to error handling while synchronously uninstalling resources eligible for garbage collection, errors subsequent to the first should be logged. Once the uninstall process has completed, a Subsystem Exception must be thrown with the initial error as the specified cause.

134.15 Pre-Calculated Deployment

A pre-calculated deployment in the form of a deployment manifest can be included as part of a Subsystem Archive or provided by a deployer at installation time. Manifests provided at install time override those included within an archive, and those within an archive override calculated ones. The deployment manifest defines the precise deployment of the Subsystem. Providing a deployment manifest means a Subsystem can be deployed and the exact resources that are installed are known ahead of time. This allows test teams to test specific deployments and these same deployments can then be used in production. The deployment manifest is a locking down of the variability in a Subsystem manifest (or the equivalent if the Subsystem definition is calculated during deployment based on the Subsystem Archive). The deployment manifest follows the same syntax rules as the Subsystem manifest but uses different headers for deployment-specific information. A deployment manifest describes the following:

- The exact versions for content resources
- Any dependencies required to resolve the Subsystem's content that are not satisfied by the target runtime
- Sharing policy for requirements and capabilities shared into or out of the Subsystem.

Because a Deployment Manifest's dependencies bridge between the requirements of the Subsystem and the capabilities of the target runtime, it is not guaranteed to be portable. If available, the Subsystem service implementation must first attempt to use the Deployment Manifest to deploy the Subsystem. If the Deployment Manifest is found not to work, for example, the chosen resources do not resolve for the target runtime, then the Subsystem's implementation must fail the installation of the Subsystem.

134.15.1 Deployment Headers

A Subsystem can carry descriptive information about its deployment in the Deployment Manifest file contained in its Subsystem Archive under the name OSGI-INF/DEPLOYMENT.MF. This specification defines Deployment Manifest headers such as Deployed-Content, which Subsystem deployers (typically tools) use to supply deployment information about a Subsystem. A Subsystems implementation must:

- Process the main section of the manifest. Individual section of the manifest are ignored.
- Ignore unknown manifest headers. The Subsystem deployer can define additional manifest headers as needed.
- Ignore unknown attributes and directives.

All specified manifest headers are listed in the following sections, and include example values. All headers are optional, unless specifically indicated.

134.15.1.1 Deployment-ManifestVersion: 1

The Deployment-ManifestVersion header defines that the deployment manifest follows the rules of a Subsystems Specification. It is 1 (the default) for this version of the specification. Future versions of the Subsystems Specification can define higher numbers for this header.

134.15.1.2 Subsystem-SymbolicName: com.acme.subsystem.logging

The Subsystem-SymbolicName header specifies a non-localizable name for the Subsystem that the deployment manifest is for. The Subsystem symbolic name together with a version must identify a unique Subsystem though it can be installed multiple times in a framework. See *Validating Subsystem Identity* on page 652.

134.15.1.3   **Subsystem-Version: 1.0**

The Subsystem-Version header specifies the version of this Subsystem that the deployment manifest is for. See *Validating Subsystem Identity* on page 652.

134.15.1.4   **Deployed-Content: com.acme.logging;type=osgi.bundle;deployed-version=1.0.0**

The Deployed-Content header lists requirements for the exact resources that are considered to be the contents of this Subsystem. This header identifies the exact versions of the resources listed in the Subsystem-Content header. See *Deployed-Content* on page 653.

134.15.1.5   **Provision-Resource: com.acme.logging;type=osgi.bundle;deployed-version=1.0.0**

The Provision-Resource header lists requirements for the exact resources to be installed in order to satisfy requirements from the Deployed-Content resources that are not satisfied by the capabilities of the target runtime. See *Provision-Resource* on page 653.

134.15.1.6   **Import-Package: com.acme.api;version="[1.0,1.1)"**

The Import-Package header lists package requirements for capabilities that are to be imported into a Scoped Subsystem. See *Import-Package* on page 654.

134.15.1.7   **Export-Package: com.acme.api;version=1.0.1**

The Export-Package header lists package capabilities that are to be exported out of a Scoped Subsystem. See *Export-Package* on page 654.

134.15.1.8   **Require-Bundle: com.acme.logging;version="[1.0, 1.1)"**

The Require-Bundle header lists bundle requirements for bundle capabilities that are to be imported into a Scoped Subsystem. See *Require-Bundle* on page 655.

134.15.1.9   **Provide-Capability: com.acme.dict; from=nl; to=de; version:Version=1.2**

The Provide-Capability header declares the capabilities exported for a Scoped Subsystem. See [3] *Resource and Wiring*.

134.15.1.10   **Require-Capability: osgi.ee; filter:="(osgi.ee=*)"**

The Require-Capability header declares the required capabilities for a Scoped Subsystem. See [3] *Resource and Wiring*.

134.15.1.11   **Subsystem-ImportService: com.acme.service.Logging**

The Subsystem-ImportService header lists service requirements for service capabilities that are to be imported into a Scoped Subsystem. See *Services* on page 655.

134.15.1.12   **Subsystem-ExportService: com.acme.service.Logging**

The Subsystem-ExportService header lists service requirements that are matched against service capabilities provided by the Deployed-Content resources. Any matching capabilities are exported out of the Scoped Subsystem.

134.15.2   Validating Subsystem Identity

The Subsystem to which the deployment manifest applies is identified by the Subsystem's symbolic name and version headers. These headers are identical to those specific in the Subsystem manifest. A Subsystem runtime must validate that the headers specified in the deployment manifest match those of the Subsystem manifest, taking into account Subsystem manifest defaulting rules. This allows the two manifests to be managed by teams separately during development or testing whilst ensuring no mistakes have been made when they are brought together for deployment. If the headers do not match, then the runtime must not use the deployment manifest and must fail the installation.

134.15.3   Deployed-Content

The Deployed-Content header lists the exact constituents to be installed for the Subsystem. For each mandatory entry in the Subsystem-Content header, there must be a corresponding Deployed-Content entry. If a content resources is identified as optional and there is a corresponding entry in the deployment manifest, then it must be deployed. If there is no corresponding entry in the deployment manifest then no resource must be deployed for it. The Deployed-Content entry identifies the exact version of the constituent whereas the Subsystem-Content entry may specify a version range. Each Deployed-Content entry is identified by symbolic name, version and type (an osgi identity).

```
Deployed-Content:
    com.acme.logging;
        deployed-version=1.0,
    com.acme.persistence;
        deployed-version=1.1;
        type=osgi.subsystem.composite
```

Each entry must uniquely identify the resource to be provisioned as a constituent of the Subsystem.

The following mandatory matching attributes must be applied to each entry:

- deployed-version - The exact version of the resource to be deployed. Deployed version is a specific version, not a version range, hence the use of a new attribute name. There is no default value for this attribute.

The following architected matching attribute as well as any arbitrary matching attributes can be applied to each entry:

type - The type of the constituent. It is recommended that a reverse domain name convention is used unless those types and their processing is standardized by the OSGi Alliance (e.g. bundles). The default value is osgi.bundle. A Subsystems implementation may support additional types, but the following types must be supported:

- osgi.bundle
- osgi.fragment
- osgi.subsystem.application
- osgi.subsystem.composite
- osgi.subsystem.feature

The value of this directive must match the type directive for the corresponding entry in the Subsystem-Content header, including taking into account defaulting. If the type does not match, then the installation must fail.

The following directive can be applied to each entry:

- start-order - The precedence the constituent should have during the start sequence. Resources with lower start-order values are started before resources with higher values. Resources with the same start-order value may be started sequentially or in parallel. The value of this directive must match the start-order directive for the corresponding entry in the Subsystem-Content header, including taking into account defaulting.

134.15.4   Provision-Resource

The Provision-Resource header lists the resources to be provisioned in support of the Subsystem's dependencies. The exact location in the Subsystem hierarchy where the resources are installed is determined by the provision-policy of the Subsystem or its parents.

The Provision-Resource header must result in a transitively complete deployment. For example, if a resource added to Provision-Resource brings in additional unsatisfied requirements, further resources must be added to satisfy these, until there are no unresolved requirements remaining.

Provision resource has one required matching attribute:

- deployed-version - The exact version of the resource to be deployed. Deployed version is a specific version, not a version range, hence the use of a new attribute name. There is no default value for this attribute.

The following architected matching attributes as well as any arbitrary matching attributes can be applied to each entry:

type - The type of the resource. It is recommended that a reverse domain name convention is used unless those types and their processing is standardized by the OSGi Alliance (e.g. bundles). The default type is osgi.bundle. A Subsystems implementation may support additional types, but the following types must be supported:

- osgi.bundle
- osgi.fragment
- osgi.subsystem.application
- osgi.subsystem.composite
- osgi.subsystem.feature

The list of the Provision-Resource entries is determined by resolving the Subsystem's requirements. The way in which the Subsystem's requirements are resolved is dependent on the Subsystem's sharing policy.

For a Scoped Subsystem the provision resources header must identify a set of resources necessary to satisfy the requirements into the Subsystem that are not satisfied by the target deployment environment. These requirements may be for packages, services, or other types of requirements, and are those identified in the deployment manifest using headers such as Import-Package and Subsystem-ImportService.

For an Unscoped Subsystem any mandatory requirements that are not satisfied by capabilities provided by the target environment may be satisfied by other constituents or a resource added to the Provision-Resource header. The resolution process for Unscoped Subsystems has no propensity to resolve to capabilities provided by the Subsystem's constituents and so a resource listed in Provision-Resource may provide capabilities that are also provided by a constituent resource.

134.15.5 Import-Package

Scoped Subsystems describe the exact packages they import in their Deployment Manifests. They do this using the bundle Import-Package header. Any packages that match the Import-Package statement must be allowed into the Scoped Subsystem by its associated Region's sharing policy.

Unscoped Subsystems have a sharing policy that shares all packages and therefore their deployment manifests do not use this header to describe the sharing of individual packages. If this header is present and the Subsystem is unscoped, then the runtime must fail the installation of the Subsystem.

134.15.6 Export-Package

Scoped Subsystems describe the exact packages they export in their deployment manifests. They do this using the bundle Export-Package header. Any packages that match the Export-Package statement must be made available outside the Subsystem by its associated Region's sharing policy.

Unscoped Subsystems have a sharing policy that shares all packages and therefore their deployment manifests do not use this header to describe the sharing of individual packages. If this header

is present and the Subsystem is unscoped, then the runtime must fail the installation of the Subsystem.

134.15.7   Require-Bundle

Scoped Subsystems can have Require-Bundle requirements satisfied by bundles outside the Subsystem. These bundle requirements are described using the bundle Require-Bundle header. Any bundles that match the Require-Bundle statement must be allowed into the Scoped Subsystem by its associated Region's sharing policy. If a bundle matches the Require-Bundle requirement then it becomes available as a candidate for wiring any Require-Bundle requirements inside the Subsystem. However, any packages the matching bundle provides are not made available to satisfy Import-Package requirements by the Region's sharing policy. If the packages are also required then they must be listed in the deployment manifest's Import-Package header.

Unscoped Subsystems have a sharing policy that shares all bundles and therefore their deployment manifests do not use this header to describe the sharing of specific bundles. If this header is present and the Subsystem is unscoped, then the runtime must fail the installation of the Subsystem.

134.15.8   Services

Scoped Subsystems can import and export services using the Subsystem-ImportService and Subsystem-ExportService headers respectively. These two headers must conform to the following syntax:

```
Subsystem-ImportService ::= service( ',' service )*
Subsystem-ExportService ::= service ( ',' service )*
service                 ::= qname ( ';' parameter )*
```

Both headers support the following directive:

- filter - A filter expression that is used to match against the service properties of services registered using the specified qname of the service's object class. The filter directive is optional. If no filter directive is defined then all services registered using the specified qname match the service statement.

134.15.9   Subsystem-ImportService

Scoped Subsystems describe the services they import in their deployment manifests. They do this using the Subsystem-ImportService header. Subsystem-ImportService header defines a list of OSGi service filters that are matched against the services visible inside the Scoped Subsystem's parent Region. Each service visible in the Subsystem's parent Region that matches one or more Subsystem-ImportService statements must be allowed into the Scoped Subsystem by its associated Region's sharing policy. The following example imports services registered under the com.acme.logging.Loginterface with a service property threshold=error.

```
Subsystem-ImportService: com.acme.logging.Log;filter:="(threshold=error)"
```

Unscoped Subsystems have a sharing policy that shares all services and therefore their deployment manifests do not use this header to describe the sharing of specific services. If this header is present and the Subsystem is unscoped, then the runtime must fail the installation of the Subsystem.

134.15.10   Subsystem-ExportService

Scoped Subsystems describe the services they export in their deployment manifests. They do this using the Subsystem-ExportService header. The Subsystem-ExportService header defines a list of OSGi service filters that are matched against the services visible inside the Scoped Subsystem's Region. Each service visible in the Scoped Subsystem's Region that matches one or more Subsystem-ExportService statements must be allowed by its associated Region's sharing policy into the

Scoped Subsystem's parent Region. The following example exports services registered under the com.acme.logging.Log interface with a service property threshold=error.

```
Subsystem-ExportService: com.acme.logging.Log;filter:="(threshold=error)"
```

Unscoped Subsystems have a sharing policy that shares all services and therefore their Deployment Manifests do not use this header to describe the sharing of specific services. If this header is present and the Subsystem is unscoped, then the runtime must fail the installation of the Subsystem.

134.16 Subsystem Types

Subsystem types simplify the configuration of sharing policies. The type of Subsystem is specified using the Subsystem-Type header. Each type has its own default sharing policy, for example, to forbid the sharing of capabilities out, or to share all capabilities in. This specification defines three Subsystem types:

- osgi.subsystem.application
- osgi.subsystem.composite
- osgi.subsystem.feature

Other, non-standard, types are permitted. The specifics of each standard type are describe below.

134.16.1 Application

An application is a Scoped Subsystem with a sharing policy associated with what is often considered to be an application. An application does not share (export) any capabilities to other bundles or Subsystems. It also does not explicitly import any capabilities. Any required capabilities that are not satisfied by the application's constituents are automatically shared in (imported) from the parent Subsystem.

A Subsystem is identified as an application by specifying a Subsystem type value of osgi.subsystem.application in the Subsystem manifest.

```
Subsystem-Type: osgi.subsystem.application
```

134.16.2 Application Deployment

Application Subsystems are not configured using additional requirement or capability headers, such as Import-Package. Applications do not export any capabilities. If an application Subsystem contains any capability exports then the Subsystem runtime should log an error and must fail.

Any imported capabilities are derived from the application Subsystem content. An application Subsystem implicitly imports any capabilities required to satisfy requirements from the Subsystem contents that are not satisfied by the capabilities of the Subsystem content.

Unsatisfied mandatory requirements result in a subsystem installation failure. Unsatisfied optional requirements do not. However, implementations must ensure any unsatisfied optional requirements are added to the sharing policy.

134.16.2.1 Package Imports

Application resolution is required to prefer packages provided by content bundles over those provided outside the application. For this reason, the application Subsystem sharing policy only imports packages corresponding to Import-Package statements from the content bundles that are not satisfied when resolving the application contents in isolation. This is equivalent to first resolving the Subsystem-Content requirements to determine the Deployed-Content and then based on this set of resources, determining which Import-Package requirements remain unsatisfied.

A deployment manifest for an application Subsystem would list these package imports using the Import-Package header.

134.16.2.2   Service Imports

Application resolution is required to prefer services provided by content bundles over those provided outside the application. For this reason, the application Subsystem sharing policy only imports services required by the Subsystem's content bundles that are not also provided by the content bundles. This specification provides a means of declaratively identifying the services a bundle provides or requires using the Provide-Capability and Require-Capability headers with the osgi.service namespace. See *osgi.service Namespace* on page 685

An example of a bundle providing the service and declaring it using the Provide-Capability header is as follows:

```
Provide-Capability: osgi.service;
 objectClass:List<String>="com.foo.MyService";
 uses:="com.foo"
```

Note that declaring a provided service in this manner only affects resolution. It does not affect service visibility at runtime. In other words, a subsystem that imports service com.acme.Foo will see all of the corresponding service registrations that its parent sees regardless of whether or not the provider declared this service in the Provide-Capability header.

An example of a bundle requiring a service and declaring the requirement using the Require-Capability header is as follows:

```
Require-Capability: osgi.service;
 filter:="(objectClass=com.foo.MyService)";
 effective:="active"
```

These headers can be hand-written (e.g., to declare programmatic use of an OSGi service) or generated by a tool (e.g., BND) based on a declarative component model configuration (e.g., Declarative Services or Blueprint). A Subsystems implementation must assume these headers, if present, declare all of the service dependencies. Implementations must therefore not search the bundle for additional dependencies from other sources.

A deployment manifest for an application Subsystem would list these service imports using the Subsystem-ImportService header.

134.16.2.3   Bundle Requirements

Application resolution is required to prefer bundle capabilities provided by content bundles over those provided outside the application. For this reason, the application Subsystem sharing policy only requires bundle capabilities corresponding to Require-Bundle statements from the content bundles that are not satisfied when resolving the application contents in isolation. This is equivalent to first resolving the Subsystem-Content requirements to determine the Deployed-Content and then based on this set of resources, determining which Require-Bundle requirements remain unsatisfied.

A Deployment Manifest for an application Subsystem would list these bundle requirements using the Require-Bundle header.

134.16.2.4   Generic Requirements

Application resolution is required to prefer generic capabilities provided by content bundles over those provided outside the application. For this reason, the application Subsystem sharing policy only generic requirements corresponding to Require-Capability statements from the content bundles that are not satisfied by Provide-Capability statements of the content bundles when resolving the application contents in isolation. This is equivalent to first resolving the Subsystem-Content re-

quirements to determine the Deployed-Content and then based on this set of resources, determining which Require-Capability statements remain unsatisfied.

An deployment manifest for an application Subsystem would list these generic requirements using the Require-Capability header.

134.16.2.5   Dependencies

Application Subsystems' implicit requirements are determined as described in the Application Deployment section in *Determining Dependencies* on page 632. Any mandatory requirements from constituents that are not satisfied by capabilities provided by the target environment or other constituents must be satisfied by additional dependencies. The Subsystem runtime is responsible for provisioning these based on the Subsystem's provision policy or those of its scoped parents. If the application Subsystem has an associated deployment manifest, then these resources are described in the Provision-Resource header.

134.16.3   Composite

A composite is a Scoped Subsystem with a sharing policy that by default does not share anything with its parent and therefore all sharing is fully explicit. Capabilities, such as packages and services, may be explicitly imported into or exported out of the composite.

A Subsystem is identified as an composite by specifying a Subsystem type value of osgi.subsystem.composite in the Subsystem manifest.

```
Subsystem-Type: osgi.subsystem.composite
```

134.16.3.1   Subsystem Content

The Subsystem-Content header allows version ranges for content resources. For composite Subsystems, this value must be a fixed version range (e.g. [1.0, 1.0]) for resources of type osgi.bundle, osgi.fragment, osgi.subsystem.application, osgi.subsystem.composite, and osgi.subsystem.feature. This is due to the fact that there is an inextricable link between the versions on the explicit import and export statements made on a composite and the chosen versions of the content bundles. Allowing variability in the content versions for these types of resources risks introducing incompatibilities with sharing policy for the composite. If a composite Subsystem does not use strict version ranges then the composite Subsystem must fail to install.

134.16.3.2   Package Imports

A composite Subsystem explicitly states the packages it imports using the Import-Package header. If the composite includes a deployment manifest then the Import-Package header is used to describe these and they must be identical (logically, not syntactically) to the Import-Package headers in the composite's Subsystem manifest. If the imports are not the same then the Subsystem runtime should log an error and must fail the installation.

134.16.3.3   Package Exports

A composite Subsystem explicitly states the packages it exports using the Export-Package header. If the composite includes a deployment manifest then the Export-Package header is used to describe these and they must be identical (logically, not syntactically) to the Export-Package headers in the composite's Subsystem manifest. If the exports are not the same then the Subsystem runtime should log an error and must fail the installation.

134.16.3.4   Service Imports

A composite Subsystem explicitly states the services it imports using the Subsystem-ImportService header (see *Subsystem-ImportService* on page 655). For example:

```
Subsystem-ImportService: com.acme.logging.Log
```

If the composite includes a deployment manifest then the Subsystem-ImportService header is used to describe these and they must be identical (logically, not syntactically) to the Subsystem-ImportService headers in the composite's Subsystem manifest. If the imports are not the same then the Subsystem runtime should log an error and must fail the installation.

134.16.3.5   Service Exports

A composite Subsystem explicitly states the services it exports using the Subsystem-ExportService header (see *Subsystem-ExportService* on page 655). For example:

```
Subsystem-ServiceExport: com.acme.logging.Log
```

If the composite includes a deployment manifest then the Subsystem-ExportService header is used to describe these and they must be identical (logically, not syntactically) to the Subsystem-ExportService headers in the composite's Subsystem manifest. If the exports are not the same then the Subsystem runtime should log an error and must fail the installation.

134.16.3.6   Bundle Requirements

A composite Subsystem explicitly states the bundles it requires using the Require-Bundle header.

If the composite includes a deployment manifest then the Require-Bundle header is used to describe these and the requirements must be identical (logically, not syntactically) to the Require-Bundle requirements in the composite's Subsystem manifest. If the requirements are not the same then the Subsystem runtime should log an error and must fail the installation.

134.16.3.7   Generic Requirements

A composite Subsystem explicitly states the generic capabilities it requires using the Require-Capability header.

If the composite includes a deployment manifest then the Require-Capability header is used to describe these and they must be identical (logically, not syntactically) to the Require-Capability headers in the composite's Subsystem manifest. If the capability requirements are not the same then the Subsystem runtime should log an error and must fail the installation.

134.16.3.8   Generic Capabilities

A composite Subsystem explicitly states the generic capabilities it provides using the Provide-Capability header.

If the composite includes a deployment manifest then the Provide-Capability header is used to describe these and they must be identical (logically, not syntactically) to the Provide-Capability headers in the composite's Subsystem manifest. If the capabilities are not the same then the Subsystem runtime should log an error and must fail the installation.

134.16.3.9   Dependencies

A composite Subsystem's explicit requirements are stated in the Subsystem manifest. Any mandatory requirements that are not satisfied by capabilities provided by the target environment must be satisfied by additional dependencies. The Subsystem runtime is responsible for provisioning these based on the Subsystem's provision policy or the provision policy of its scoped parents. If the composite Subsystem has an associated deployment manifest, then these resources are described in the Provision-Resource header.

134.16.4   Feature

A feature is an Unscoped Subsystem and therefore provides no isolation of its own. A feature does however always exist in the context of one and only one Region which can restrict the capabilities a feature can see and the extent to which a feature's capabilities are shared.

A Subsystem is identified as a feature by specifying a Subsystem type value of
osgi.subsystem.feature in the Subsystem manifest.

```
Subsystem-Type: osgi.subsystem.feature
```

134.16.4.1   Explicit Requirements and Capabilities

A feature Subsystem implicitly imports and exports all requirements and capabilities. If the feature Subsystem include any headers designed to modify the sharing policy of a Subsystem, such as Import-Package or Subsystem-ImportService, then the Subsystem runtime should log an error and must fail the installation of the Subsystem.

134.16.4.2   Dependencies

Feature Subsystems implicitly import all capabilities. A Subsystem runtime is responsible for provisioning any dependencies necessary for the Subsystem's constituents to resolve. The calculation of the dependencies can also take into account capabilities provided by the target runtime. The dependencies can include resources that provide capabilities equivalent to those provided by one or more of the constituent resources where the dependency's capability is a considered a better match in the context of some resolution. The Subsystem runtime is responsible for provisioning the dependencies based on the Subsystem's provision policy or the provision policy of its scoped parents. If the feature Subsystem has an associated deployment manifest, then these dependencies are described in the Provision-Resource header.

134.17   Weaving Hooks

Subsystems implementations must ensure that dynamic package imports added by weaving hooks are available to subsystems whose classes have been woven by updating the sharing policies.

Dynamic package imports added by weaving hooks are observed by registering a WovenClassListener service and receiving notifications via the WovenClassListener.modified(WovenClass) method. The sharing policy must be updated while the woven class is in the TRANSFORMED state so that it takes effect before the bundle wiring is updated during the transition to DEFINED; otherwise, the class would fail to load.

The bundle containing the woven class can be obtained by calling the WovenClass.getBundleWiring().getBundle() method. A bundle might be a constituent of multiple subsystems, but never more than one scoped subsystem. The rest are features, which have no sharing policies to update. It's possible the bundle will not be a constituent of a scoped subsystem. The scoped subsystem, if any, containing the bundle as a constituent is retrieved.

It's possible for a class load request to occur on a bundle in an unresolved subsystem because the framework is free to resolve bundles whenever it desires. A resolved bundle can potentially receive a class load request. For example, a BundleEventListener registered with the system bundle context could receive the RESOLVED event and, for whatever reason, load a class. Also, a resolved bundle in an unresolved feature might get wired to another bundle. If this is the case, the subsystem must be resolved in order to guarantee the dynamic imports will not effect the resolution and, therefore, potentially create a wiring inconsistent with the deployment manifest. Just as the framework is free to resolve bundles at anytime and for whatever reason, Subsystems implementations are free to resolve subsystems.

The sharing policy is only updated if the dynamic import cannot be completely satisfied from within the subsystem. Note that all dynamic imports with a wildcard must always be added to the sharing policy.

134.18   Stopping and Uninstalling Subsystems Implementation

When the Subsystems implementation is stopped all of the installed Subsystems must be persistently stored and present when the Subsystems implementation becomes active again. This includes any bundles that got installed as part of a Subsystem installation. The Subsystems implementation is not required to do any additional cleanup when the Subsystems implementation is stopped or uninstalled. All bundles that got installed as a result of installing a Subsystem may still be installed after stopping or uninstalling the Subsystems implementation bundle. If it is important to clean up the bundles associated with a Subsystem installation then the Subsystem should be uninstalled before uninstalling the Subsystems implementation.

134.19   Capabilities

Implementations of the Subsystem Service specification must provide the following capabilities.

- A capability in the osgi.implementation namespace declaring the implemented specification to be osgi.subsystem. This capability must also declare a uses constraint for the org.osgi.service.subsystem package. For example:

```
Provide-Capability: osgi.implementation;
    osgi.implementation="osgi.subsystem";
    version:Version="1.1";
    uses:="org.osgi.service.subsystem"
```

This capability must follow the rules defined for the *osgi.implementation Namespace* on page 685.

- A capability in the osgi.service namespace representing the Subsystem service. This capability must also declare a uses constraint for the org.osgi.service.subsystem package. For example:

```
Provide-Capability: osgi.service;
    objectClass:List<String>="org.osgi.service.subsystem.Subsystem";
    uses:="org.osgi.service.subsystem"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 685.

134.20   Security

134.20.1   Subsystem Permission

The Subsystem Permission is a permission used to grant the right to manage Subsystems with the option to restrict this right to a subset of Subsystems, called targets. For example, an operator can give a bundle the right to only manage Subsystems with a symbolic-name prefix of com.acme.:

```
...SubsystemPermission("(name=com.acme.*)",
...)
```

The actions of Subsystem Permission are fine-grained. They allow a deployer to assign only the permissions that are necessary for a bundle. For example, a bundle may be granted only the permission to start and stop all Subsystems:

```
...SubsystemPermission["*", EXECUTE]
```

Code that needs to check Subsystem Permission must always use the constructor that takes a Subsystem as a parameter: SubsystemPermission(Subsystem, String) with a single action.

For example, the implementation of Subsystem.start method must check that the caller has access to execute the Subsystem:

```
public class SubsystemImpl implements Subsystem{
    public void start() {
        securityManager.checkPermission(new SubsystemPermission(this,"execute"));
    }
}
```

The Subsystem Permission takes a Filter as its name argument. Filter based permissions are described in [1] *Filter Based Permissions*. Subsystem Archives are not signed and therefore the signer key is not supported. The keys have the following meaning for the Subsystem Permission:

- id - The Subsystem ID of a Subsystem. For example (id=23)
- location - The location of a Subsystem. For example (location=https://www.acme.com/download/*)
- name - The symbolic name of a Subsystem. For example (name=com.acme.*)

The name parameter of the permission can also be a single wildcard character ('*' \u002a). In that case all Subsystems must match.

134.20.2   Actions

The action parameter of Subsystem Permission will specify the subset of privileged Subsystem management operations that are allowed. The actions that are architected are listed below. Future versions of the specification can add additional actions. The given set should therefore not be assumed to be a closed set.

Table 134.3   Actions

Action	Used in
CONTEXT	Subsystem.getBundleContext
METADATA	Subsystem.getSubsystemHeaders
	Subsystem.getLocation
LIFECYCLE	Subsystem.install
	Subsystem.uninstall
EXECUTE	Subsystem.start
	Subsystem.stop

134.20.3   Required Permissions

A Subsystems implementation must check the caller for the appropriate Subsystem Permission before initiating a Subsystem management operation (e.g. install, start, stop, uninstall). Once the Subsystem Permission is checked against the caller the Subsystems implementation will proceed with the actual Subsystem operation. This operation will require a number of other permissions to complete. For example, the Admin Permission will be needed to install, start, stop, and uninstall resources of type osgi.bundle for a Subsystem. The Subsystems implementation must isolate the caller from such permission checks by use of a proper doPrivileged block.

134.21   org.osgi.service.subsystem

Subsystem Service Package Version 1.1.

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

Example import for consumers using the API in this package:

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

Example import for providers implementing the API in this package:

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

134.21.1   Summary

- Subsystem - A subsystem is a collection of resources constituting a logical, possibly isolated, unit of functionality.
- Subsystem.State - An enumeration of the possible states of a subsystem.
- SubsystemConstants - Defines the constants used by Subsystem service property, manifest header, attribute and directive keys.
- SubsystemException - A Subsystem exception used to indicate a problem.
- SubsystemPermission - A bundle's authority to perform specific privileged administrative operations on or to get sensitive information about a subsystem.

134.21.2   public interface Subsystem

A subsystem is a collection of resources constituting a logical, possibly isolated, unit of functionality.

A subsystem may be *scoped* or *unscoped*. Scoped subsystems are isolated by implicit or explicit sharing policies. Unscoped subsystems are not isolated and, therefore, have no sharing policy. There are three standard types of subsystems.

- Application - An implicitly scoped subsystem. Nothing is exported, and imports are computed based on any unsatisfied content requirements.
- Composite - An explicitly scoped subsystem. The sharing policy is defined by metadata within the subsystem archive.
- Feature - An unscoped subsystem.

Conceptually, a subsystem may be thought of as existing in an isolated region along with zero or more other subsystems. Each region has one and only one scoped subsystem, which dictates the sharing policy. The region may, however, have many unscoped subsystems. It is, therefore, possible to have shared constituents across multiple subsystems within a region. Associated with each region is a bundle whose context may be retrieved from any subsystem within that region. This context may be used to monitor activity occurring within the region.

A subsystem may have children and, unless it's the root subsystem, must have at least one parent. Subsystems become children of the subsystem in which they are installed. Unscoped subsystems have more than one parent if they are installed in more than one subsystem within the same region. The subsystem graph may be thought of as an acyclic digraph [http://en.wikipedia.org/wiki/Directed_acyclic_graph] with one and only one source vertex, which is the root subsystem. The edges have the child as the head and parent as the tail.

A subsystem has several identifiers.

- Location - An identifier specified by the client as part of installation. It is guaranteed to be unique within the same framework.
- ID - An identifier generated by the implementation as part of installation. It is guaranteed to be unique within the same framework.
- Symbolic Name/Version - The combination of symbolic name and version is guaranteed to be unique within the same region. Although type is not formally part of the identity, two subsystems with the same symbolic names and versions but different types are not considered to be equal.

A subsystem has a well-defined life cycle. Which stage a subsystem is in may be obtained from the subsystem's state and is dependent on which life cycle operation is currently active or was last invoked.

A subsystem archive is a ZIP file having an .esa extension and containing metadata describing the subsystem. The form of the metadata may be a subsystem or deployment manifest, as well as any content resource files. The manifests are optional and will be computed if not present. The subsystem manifest headers may be retrieved in raw or localized forms. There are five standard types of resources that may be included in a subsystem.

- Bundle - A bundle that is not a fragment.
- Fragment - A fragment bundle.
- Application Subsystem - An application subsystem.
- Composite Subsystem - A composite subsystem.
- Feature Subsystem - A feature subsystem.

Resources contained by a subsystem are called constituents. There are several ways a resource may become a constituent of a subsystem:

- A resource is listed as part of the subsystem's content.
- A subsystem resource is a child of the subsystem.
- The subsystem has a provision policy of accept dependencies.
- A bundle resource is installed using the region bundle context.
- A bundle resource is installed using the bundle context of another resource contained by the subsystem.

In addition to invoking one of the install methods, a subsystem instance may be obtained through the service registry. Each installed subsystem has a corresponding service registration. A subsystem service has the following properties.

- ID - The ID of the subsystem.
- Symbolic Name - The symbolic name of the subsystem.
- Version - The version of the subsystem.
- Type - The type of the subsystem.
- State - The state of the subsystem.

Because a subsystem must be used to install other subsystems, a root subsystem is provided as a starting point. The root subsystem may only be obtained as a service and has the following characteristics.

- The ID is 0.
- The symbolic name is org.osgi.service.subsystem.root.
- The version matches this specification's version.
- It has no parents.
- All existing bundles, including the system and subsystem implementation bundles, are constituents.

- The type is osgi.subsystem.application with no imports.
- The provision policy is acceptDependencies.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

134.21.2.1   public BundleContext getBundleContext()

☐ Returns the bundle context of the region within which this subsystem resides.

The bundle context offers the same perspective of any resource contained by a subsystem within the region. It may be used, for example, to monitor events internal to the region as well as external events visible to the region. All subsystems within the same region have the same bundle context. If this subsystem is in a state where the bundle context would be invalid, null is returned.

Returns The bundle context of the region within which this subsystem resides or null if this subsystem's state is in INSTALL_FAILED, UNINSTALLED.

Throws SecurityException– If the caller does not have the appropriate SubsystemPermission[this,CONTEXT], and the runtime supports permissions.

134.21.2.2   public Collection<Subsystem> getChildren()

☐ Returns the child subsystems of this subsystem.

Returns The child subsystems of this subsystem. The returned collection is an unmodifiable snapshot of all subsystems that are installed in this subsystem. The collection will be empty if no subsystems are installed in this subsystem.

Throws IllegalStateException– If this subsystem's state is in INSTALL_FAILED, UNINSTALLED.

134.21.2.3   public Collection<Resource> getConstituents()

☐ Returns the constituent resources of this subsystem.

Returns The constituent resources of this subsystem. The returned collection is an unmodifiable snapshot of the constituent resources of this subsystem. If this subsystem has no constituents, the collection will be empty.

Throws IllegalStateException– If this subsystem's state is in INSTALL_FAILED, UNINSTALLED.

134.21.2.4   public Map<String,String> getDeploymentHeaders()

☐ Returns the headers for this subsystem's deployment manifest.

Each key in the map is a header name and the value of the key is the corresponding header value. Because header names are case-insensitive, the methods of the map must treat the keys in a case-insensitive manner. If the header name is not found, null is returned. Both original and derived headers will be included in the map.

This method must continue to return the headers while this subsystem is in the INSTALL_FAILED or UNINSTALLED states.

Returns The headers for this subsystem's deployment manifest. The returned map is unmodifiable.

Throws SecurityException– If the caller does not have the appropriate SubsystemPermission[this,METADATA], and the runtime supports permissions.

Since 1.1

134.21.2.5   public String getLocation()

☐ Returns the location identifier of this subsystem.

The location identifier is the location that was passed to the install method of the parent subsystem. It is unique within the framework.

This method must continue to return this subsystem's headers while this subsystem is in the INSTALL_FAILED or UNINSTALLED states.

Returns   The location identifier of this subsystem.

Throws   `SecurityException`– If the caller does not have the appropriate SubsystemPermission[this,METADATA], and the runtime supports permissions.

134.21.2.6   public Collection<Subsystem> getParents()

☐ Returns the parent subsystems of this subsystem.

Returns   The parent subsystems of this subsystem. The returned collection is an unmodifiable snapshot of all subsystems in which this subsystem is installed. The collection will be empty for the root subsystem; otherwise, it must contain at least one parent. Scoped subsystems always have only one parent. Unscoped subsystems may have multiple parents.

Throws   `IllegalStateException`– If this subsystem's state is in INSTALL_FAILED, UNINSTALLED.

134.21.2.7   public Subsystem.State getState()

☐ Returns the current state of this subsystem.

This method must continue to return this subsystem's state while this subsystem is in the INSTALL_FAILED or UNINSTALLED states.

Returns   The current state of this subsystem.

134.21.2.8   public Map<String,String> getSubsystemHeaders(Locale locale)

locale   The locale for which translations are desired. The header values are translated according to the specified locale. If the specified locale is `null` or not supported, the raw values are returned. If the translation for a particular header is not found, the raw value is returned.

☐ Returns the headers for this subsystem's subsystem manifest.

Each key in the map is a header name and the value of the key is the corresponding header value. Because header names are case-insensitive, the methods of the map must treat the keys in a case-insensitive manner. If the header name is not found, `null` is returned. Both original and derived headers will be included in the map.

This method must continue to return the headers while this subsystem is in the INSTALL_FAILED or UNINSTALLED states.

Returns   The headers for this subsystem's subsystem manifest. The returned map is unmodifiable.

Throws   `SecurityException`– If the caller does not have the appropriate SubsystemPermission[this,METADATA], and the runtime supports permissions.

134.21.2.9   public long getSubsystemId()

☐ Returns the identifier of this subsystem.

The identifier is a monotonically increasing, non-negative integer automatically generated at installation time and guaranteed to be unique within the framework. The identifier of the root subsystem is zero.

This method must continue to return this subsystem's identifier while this subsystem is in the INSTALL_FAILED or UNINSTALLED states.

Returns   The identifier of this subsystem.

134.21.2.10   public String getSymbolicName()

☐ Returns the symbolic name of this subsystem.

The subsystem symbolic name conforms to the same grammar rules as the bundle symbolic name and is derived from one of the following, in order.

- The value of the Subsystem-SymbolicName header, if specified.
- The subsystem URI if passed as the location along with the content to the install method.
- Optionally generated in an implementation specific way.

The combination of subsystem symbolic name and version is unique within a region. The symbolic name of the root subsystem is org.osgi.service.subsystem.root.

This method must continue to return this subsystem's symbolic name while this subsystem is in the INSTALL_FAILED or UNINSTALLED states.

Returns The symbolic name of this subsystem.

134.21.2.11   **public String getType()**

☐ Returns the type of this subsystem.

This method must continue to return this subsystem's type while this subsystem is in the INSTALL_FAILED or UNINSTALLED states.

Returns The type of this subsystem.

134.21.2.12   **public Version getVersion()**

☐ Returns the version of this subsystem.

The subsystem version conforms to the same grammar rules as the bundle version and is derived from one of the following, in order.

- The value of the Subsystem-Version header, if specified.
- The subsystem URI if passed as the location along with the content to the install method.
- Defaults to 0.0.0.

The combination of subsystem symbolic name and version is unique within a region. The version of the root subsystem matches this specification's version.

This method must continue to return this subsystem's version while this subsystem is in the INSTALL_FAILED or UNINSTALLED states.

Returns The version of this subsystem.

134.21.2.13   **public Subsystem install(String location)**

location The location identifier of the subsystem to install.

☐ Installs a subsystem from the specified location identifier.

This method performs the same function as calling install(String, InputStream) with the specified location identifier and null as the content.

Returns The installed subsystem.

Throws IllegalStateException– If this subsystem's state is in INSTALLING, INSTALL_FAILED, UNINSTAL-LING, UNINSTALLED.

SubsystemException– If the installation failed.

SecurityException– If the caller does not have the appropriate SubsystemPermission[installed subsystem,LIFECYCLE], and the runtime supports permissions.

See Also install(String, InputStream)

134.21.2.14   **public Subsystem install(String location,InputStream content)**

location The location identifier of the subsystem to be installed.

content The input stream from which this subsystem will be read or `null` to indicate the input stream must be created from the specified location identifier. The input stream will always be closed when this method completes, even if an exception is thrown.

☐   Installs a subsystem from the specified content.

The specified location will be used as an identifier of the subsystem. Every installed subsystem is uniquely identified by its location, which is typically in the form of a URI. If the specified location conforms to the `subsystem-uri` grammar, the required symbolic name and optional version information will be used as default values.

If the specified content is `null`, a new input stream must be created from which to read the subsystem by interpreting, in an implementation dependent manner, the specified location.

A subsystem installation must be persistent. That is, an installed subsystem must remain installed across Framework and VM restarts.

All references to changing the state of this subsystem include both changing the state of the subsystem object as well as the state property of the subsystem service registration.

The following steps are required to install a subsystem.

1. If an installed subsystem with the specified location identifier already exists, return the installed subsystem.
2. Read the specified content in order to determine the symbolic name, version, and type of the installing subsystem. If an error occurs while reading the content, an installation failure results.
3. If an installed subsystem with the same symbolic name and version already exists within this subsystem's region, complete the installation with one of the following.
    -   If the installing and installed subsystems' types are not equal, an installation failure results.
    -   If the installing and installed subsystems' types are equal, and the installed subsystem is already a child of this subsystem, return the installed subsystem.
    -   If the installing and installed subsystems' types are equal, and the installed subsystem is not already a child of this subsystem, add the installed subsystem as a child of this subsystem, increment the installed subsystem's reference count by one, and return the installed subsystem.
4. Create a new subsystem based on the specified location and content.
5. If the subsystem is scoped, install and start a new region context bundle.
6. Change the state to INSTALLING and register a new subsystem service.
7. Discover the subsystem's content resources. If any mandatory resource is missing, an installation failure results.
8. Discover the dependencies required by the content resources. If any mandatory dependency is missing, an installation failure results.
9. Using a framework `ResolverHook`, disable runtime resolution for the resources.
10. For each resource, increment the reference count by one. If the reference count is one, install the resource. If an error occurs while installing a resource, an install failure results with that error as the cause.
11. If the subsystem is scoped, enable the import sharing policy.
12. Enable runtime resolution for the resources.
13. Change the state of the subsystem to INSTALLED.
14. Return the new subsystem.

Implementations should be sensitive to the potential for long running operations and periodically check the current thread for interruption. An interrupted thread should result in a SubsystemException with an InterruptedException as the cause and be treated as an installation failure.

All installation failure flows include the following, in order.

1. Change the state to INSTALL_FAILED.

2. Change the state to UNINSTALLING.

3. All content and dependencies which may have been installed by the installing process must be uninstalled.

4. Change the state to UNINSTALLED.

5. Unregister the subsystem service.

6. If the subsystem is a scoped subsystem then, uninstall the region context bundle.

7. Throw a SubsystemException with the cause of the installation failure.

Returns The installed subsystem.

Throws IllegalStateException– If this subsystem's state is in INSTALLING, INSTALL_FAILED, UNINSTAL-LING, UNINSTALLED.

SubsystemException– If the installation failed.

SecurityException– If the caller does not have the appropriate SubsystemPermission[installed subsystem,LIFECYCLE], and the runtime supports permissions.

134.21.2.15   **public Subsystem install(String location,InputStream content,InputStream deploymentManifest)**

location The location identifier of the subsystem to be installed.

content The input stream from which this subsystem will be read or null to indicate the input stream must be created from the specified location identifier. The input stream will always be closed when this method completes, even if an exception is thrown.

deploymentMani-fest The deployment manifest to use in lieu of the one in the archive, if any, or a computed one.

☐ Installs a subsystem from the specified content according to the specified deployment manifest.

This method installs a subsystem using the provided deployment manifest instead of the one in the archive, if any, or a computed one. If the deployment manifest is null, the behavior is exactly the same as in the install(String, InputStream) method. Implementations must support deployment manifest input streams in the format described by section 134.2 of the Subsystem Service Specification. If the deployment manifest does not conform to the subsystem manifest (see 134.15.2), the installation fails.

Returns The installed subsystem.

Throws IllegalStateException– If this subsystem's state is in INSTALLING, INSTALL_FAILED, UNINSTAL-LING, UNINSTALLED.

SubsystemException– If the installation failed.

SecurityException– If the caller does not have the appropriate SubsystemPermission[installed subsystem,LIFECYCLE], and the runtime supports permissions.

Since 1.1

134.21.2.16   **public void start()**

☐ Starts this subsystem.

The following table shows which actions are associated with each state. An action of Wait means this method will block until a state transition occurs, upon which the new state will be evaluated in order to determine how to proceed. If a state transition does not occur in a reasonable time while waiting then no action is taken and a SubsystemException is thrown to indicate the subsystem was unable to be started. An action of Return means this method returns immediately without taking any other action.

State	Action
INSTALLING	Wait

INSTALLED	Resolve, Start
INSTALL_FAILED	IllegalStateException
RESOLVING	Wait
RESOLVED	Start
STARTING	Wait
ACTIVE	Return
STOPPING	Wait
UNINSTALLING	IllegalStateException
UNINSTALLED	IllegalStateException

All references to changing the state of this subsystem include both changing the state of the subsystem object as well as the state property of the subsystem service registration.

A subsystem must be persistently started. That is, a started subsystem must be restarted across Framework and VM restarts, even if a start failure occurs.

The following steps are required to start this subsystem.

1. Set the subsystem *autostart setting* to *started*.
2. If this subsystem is in the RESOLVED state, proceed to step 7.
3. Change the state to RESOLVING.
4. Resolve the content resources. A resolution failure results in a start failure with a state of INSTALLED.
5. Change the state to RESOLVED.
6. If this subsystem is scoped, enable the export sharing policy.
7. Change the state to STARTING.
8. For each eligible resource, increment the active use count by one. If the active use count is one, start the resource. All dependencies must be started before any content resource, and content resources must be started according to the specified start order. If an error occurs while starting a resource, a start failure results with that error as the cause.
9. Change the state to ACTIVE.

Implementations should be sensitive to the potential for long running operations and periodically check the current thread for interruption. An interrupted thread should be treated as a start failure with an InterruptedException as the cause.

All start failure flows include the following, in order.

1. If the subsystem state is STARTING then change the state to STOPPING and stop all resources that were started as part of this operation.
2. Change the state to either INSTALLED or RESOLVED.
3. Throw a SubsystemException with the specified cause.

Throws SubsystemException– If this subsystem fails to start.

IllegalStateException– If this subsystem's state is in INSTALL_FAILED, UNINSTALLING, or UNINSTALLED, or if the state of at least one of this subsystem's parents is not in STARTING, ACTIVE.

SecurityException– If the caller does not have the appropriate SubsystemPermission[this,EXECUTE], and the runtime supports permissions.

134.21.2.17   **public void stop()**

☐ Stops this subsystem.

The following table shows which actions are associated with each state. An action of Wait means this method will block until a state transition occurs, upon which the new state will be evaluated

in order to determine how to proceed. If a state transition does not occur in a reasonable time while waiting then no action is taken and a SubsystemException is thrown to indicate the subsystem was unable to be stopped. An action of Return means this method returns immediately without taking any other action.

State	Action
INSTALLING	Wait
INSTALLED	Return
INSTALL_FAILED	IllegalStateException
RESOLVING	Wait
RESOLVED	Return
STARTING	Wait
ACTIVE	Stop
STOPPING	Wait
UNINSTALLING	IllegalStateException
UNINSTALLED	IllegalStateException

A subsystem must be persistently stopped. That is, a stopped subsystem must remain stopped across Framework and VM restarts.

All references to changing the state of this subsystem include both changing the state of the subsystem object as well as the state property of the subsystem service registration.

The following steps are required to stop this subsystem.

1. Set the subsystem *autostart setting* to *stopped*.
2. Change the state to STOPPING.
3. For each eligible resource, decrement the active use count by one. If the active use count is zero, stop the resource. All content resources must be stopped before any dependencies, and content resources must be stopped in reverse start order.
4. Change the state to RESOLVED.

With regard to error handling, once this subsystem has transitioned to the STOPPING state, every part of each step above must be attempted. Errors subsequent to the first should be logged. Once the stop process has completed, a SubsystemException must be thrown with the initial error as the specified cause.

Implementations should be sensitive to the potential for long running operations and periodically check the current thread for interruption, in which case a SubsystemException with an InterruptedException as the cause should be thrown. If an interruption occurs while waiting, this method should terminate immediately. Once the transition to the STOPPING state has occurred, however, this method must not terminate due to an interruption until the stop process has completed.

Throws SubsystemException– If this subsystem fails to stop cleanly.

IllegalStateException– If this subsystem's state is in INSTALL_FAILED, UNINSTALLING, or UNINSTALLED.

SecurityException– If the caller does not have the appropriate SubsystemPermission[this,EXECUTE], and the runtime supports permissions.

134.21.2.18   public void uninstall()

□ Uninstalls this subsystem.

The following table shows which actions are associated with each state. An action of Wait means this method will block until a state transition occurs, upon which the new state will be evaluated in order to determine how to proceed. If a state transition does not occur in a reasonable time while

waiting then no action is taken and a SubsystemException is thrown to indicate the subsystem was unable to be uninstalled. An action of Return means this method returns immediately without taking any other action.

State	Action
INSTALLING	Wait
INSTALLED	Uninstall
INSTALL_FAILED	Wait
RESOLVING	Wait
RESOLVED	Uninstall
STARTING	Wait
ACTIVE	Stop, Uninstall
STOPPING	Wait
UNINSTALLING	Wait
UNINSTALLED	Return

All references to changing the state of this subsystem include both changing the state of the subsystem object as well as the state property of the subsystem service registration.

The following steps are required to uninstall this subsystem after being stopped if necessary.

1. Change the state to INSTALLED.
2. Change the state to UNINSTALLING.
3. For each referenced resource, decrement the reference count by one. If the reference count is zero, uninstall the resource. All content resources must be uninstalled before any dependencies.
4. Change the state to UNINSTALLED.
5. Unregister the subsystem service.
6. If the subsystem is scoped, uninstall the region context bundle.

With regard to error handling, once this subsystem has transitioned to the UNINSTALLING state, every part of each step above must be attempted. Errors subsequent to the first should be logged. Once the uninstall process has completed, a SubsystemException must be thrown with the specified cause.

Implementations should be sensitive to the potential for long running operations and periodically check the current thread for interruption, in which case a SubsystemException with an InterruptedException as the cause should be thrown. If an interruption occurs while waiting, this method should terminate immediately. Once the transition to the UNINSTALLING state has occurred, however, this method must not terminate due to an interruption until the uninstall process has completed.

Throws SubsystemException– If this subsystem fails to uninstall cleanly.

SecurityException– If the caller does not have the appropriate
SubsystemPermission[this,LIFECYCLE], and the runtime supports permissions.

134.21.3   enum Subsystem.State

An enumeration of the possible states of a subsystem.

These states are a reflection of what constituent resources are permitted to do and not an aggregation of constituent resource states.

134.21.3.1   INSTALLING

The subsystem is in the process of installing.

A subsystem is in the INSTALLING state when the install method of its parent is active, and attempts are being made to install its content resources. If the install method completes without exception, then the subsystem has successfully installed and must move to the INSTALLED state. Otherwise, the subsystem has failed to install and must move to the INSTALL_FAILED state.

134.21.3.2   INSTALLED

The subsystem is installed but not yet resolved.

A subsystem is in the INSTALLED state when it has been installed in a parent subsystem but is not or cannot be resolved. This state is visible if the dependencies of the subsystem's content resources cannot be resolved.

134.21.3.3   INSTALL_FAILED

The subsystem failed to install.

A subsystem is in the INSTALL_FAILED state when an unrecoverable error occurred during installation. The subsystem is in an unusable state but references to the subsystem object may still be available and used for introspection.

134.21.3.4   RESOLVING

The subsystem is in the process of resolving.

A subsystem is in the RESOLVING state when attempts are being made to resolve its content resources. If the resolve process completes without exception, then the subsystem has successfully resolved and must move to the RESOLVED state. Otherwise, the subsystem has failed to resolve and must move to the INSTALLED state.

134.21.3.5   RESOLVED

The subsystem is resolved and able to be started.

A subsystem is in the RESOLVED state when all of its content resources are resolved. Note that the subsystem is not active yet.

134.21.3.6   STARTING

The subsystem is in the process of starting.

A subsystem is in the STARTING state when its start method is active, and attempts are being made to start its content and dependencies. If the start method completes without exception, then the subsystem has successfully started and must move to the ACTIVE state. Otherwise, the subsystem has failed to start and must move to the RESOLVED state.

134.21.3.7   ACTIVE

The subsystem is now running.

A subsystem is in the ACTIVE state when its content and dependencies have been successfully started.

134.21.3.8   STOPPING

The subsystem is in the process of stopping.

A subsystem is in the STOPPING state when its stop method is active, and attempts are being made to stop its content and dependencies. When the stop method completes, the subsystem is stopped and must move to the RESOLVED state.

134.21.3.9   UNINSTALLING

The subsystem is in the process of uninstalling.

A subsystem is in the UNINSTALLING state when its uninstall method is active, and attempts are being made to uninstall its constituent and dependencies. When the uninstall method completes, the subsystem is uninstalled and must move to the UNINSTALLED state.

134.21.3.10 **UNINSTALLED**

The subsystem is uninstalled and may not be used.

The UNINSTALLED state is only visible after a subsystem's constituent and dependencies are uninstalled. The subsystem is in an unusable state but references to the subsystem object may still be available and used for introspection.

134.21.4   public class SubsystemConstants

Defines the constants used by Subsystem service property, manifest header, attribute and directive keys.

The values associated with these keys are of type String, unless otherwise indicated.

Concurrency Immutable

134.21.4.1 **public static final String DEPLOYED_CONTENT = "Deployed-Content"**

Manifest header identifying the resources to be deployed.

134.21.4.2 **public static final String DEPLOYED_VERSION_ATTRIBUTE = "deployed-version"**

Manifest header attribute identifying the deployed version.

134.21.4.3 **public static final String DEPLOYMENT_MANIFESTVERSION = "Deployment-ManifestVersion"**

Manifest header identifying the deployment manifest version. If not present, the default value is 1.

134.21.4.4 **public static final String PREFERRED_PROVIDER = "Preferred-Provider"**

Manifest header used to express a preference for particular resources to satisfy implicit package dependencies.

134.21.4.5 **public static final String PROVISION_POLICY_ACCEPT_DEPENDENCIES = "acceptDependencies"**

A value for the provision-policy directive indicating the subsystem accepts dependency resources. The root subsystem has this provision policy.

134.21.4.6 **public static final String PROVISION_POLICY_DIRECTIVE = "provision-policy"**

Manifest header directive identifying the provision policy. The default value is rejectDependencies

See Also PROVISION_POLICY_ACCEPT_DEPENDENCIES, PROVISION_POLICY_REJECT_DEPENDENCIES

134.21.4.7 **public static final String PROVISION_POLICY_REJECT_DEPENDENCIES = "rejectDependencies"**

A value for the provision-policy directive indicating the subsystem does not accept dependency resources. This is the default value.

134.21.4.8 **public static final String PROVISION_RESOURCE = "Provision-Resource"**

Manifest header identifying the resources to be deployed to satisfy the dependencies of a subsystem.

134.21.4.9 **public static final String ROOT_SUBSYSTEM_SYMBOLICNAME = "org.osgi.service.subsystem.root"**

The symbolic name of the root subsystem.

134.21.4.10 **public static final String START_ORDER_DIRECTIVE = "start-order"**

Manifest header directive identifying the start order of subsystem contents. There is no default value. Specified values are of type String and must represent an integer.

134.21.4.11   **public static final String SUBSYSTEM_CATEGORY = "Subsystem-Category"**

Manifest header identifying the categories of a subsystem as a comma-delimited list.

Since 1.1

134.21.4.12   **public static final String SUBSYSTEM_CONTACTADDRESS = "Subsystem-ContactAddress"**

Manifest header identifying the contact address where problems with a subsystem may be reported; for example, an email address.

Since 1.1

134.21.4.13   **public static final String SUBSYSTEM_CONTENT = "Subsystem-Content"**

Manifest header identifying the list of subsystem contents identified by a symbolic name and version.

134.21.4.14   **public static final String SUBSYSTEM_COPYRIGHT = "Subsystem-Copyright"**

Manifest header identifying a subsystem's copyright information.

Since 1.1

134.21.4.15   **public static final String SUBSYSTEM_DESCRIPTION = "Subsystem-Description"**

Manifest header identifying the human readable description.

134.21.4.16   **public static final String SUBSYSTEM_DOCURL = "Subsystem-DocURL"**

Manifest header identifying a subsystem's documentation URL, from which further information about the subsystem may be obtained.

Since 1.1

134.21.4.17   **public static final String SUBSYSTEM_EXPORTSERVICE = "Subsystem-ExportService"**

Manifest header identifying services offered for export.

134.21.4.18   **public static final String SUBSYSTEM_ICON = "Subsystem-Icon"**

Manifest header identifying the icon URL for the subsystem.

Since 1.1

134.21.4.19   **public static final String SUBSYSTEM_ID_PROPERTY = "subsystem.id"**

The name of the service property for the subsystem ID. The value of this property must be of type Long.

134.21.4.20   **public static final String SUBSYSTEM_IMPORTSERVICE = "Subsystem-ImportService"**

Manifest header identifying services required for import.

134.21.4.21   **public static final String SUBSYSTEM_LICENSE = "Subsystem-License"**

Manifest header identifying a subsystem's license.

Since 1.1

134.21.4.22   **public static final String SUBSYSTEM_LOCALIZATION = "Subsystem-Localization"**

Manifest header identifying the base name of a subsystem's localization entries.

Since 1.1

134.21.4.23   **public static final String SUBSYSTEM_LOCALIZATION_DEFAULT_BASENAME = "OSGI-INF/l1on/subsystem"**

Default value for the Subsystem-Localization manifest header.

Since 1.1

134.21.4.24   **public static final String SUBSYSTEM_MANIFESTVERSION = "Subsystem-ManifestVersion"**

Manifest header identifying the subsystem manifest version. If not present, the default value is 1.

134.21.4.25   **public static final String SUBSYSTEM_NAME = "Subsystem-Name"**

Manifest header identifying the human readable subsystem name.

134.21.4.26   **public static final String SUBSYSTEM_STATE_PROPERTY = "subsystem.state"**

The name of the service property for the subsystem state. The value of this property must be of type Subsystem.State.

134.21.4.27   **public static final String SUBSYSTEM_SYMBOLICNAME = "Subsystem-SymbolicName"**

Manifest header value identifying the symbolic name for the subsystem. Must be present.

134.21.4.28   **public static final String SUBSYSTEM_SYMBOLICNAME_PROPERTY = "subsystem.symbolicName"**

The name of the service property for the subsystem symbolic name.

134.21.4.29   **public static final String SUBSYSTEM_TYPE = "Subsystem-Type"**

Manifest header identifying the subsystem type.

See Also SUBSYSTEM_TYPE_APPLICATION, SUBSYSTEM_TYPE_COMPOSITE,
SUBSYSTEM_TYPE_FEATURE

134.21.4.30   **public static final String SUBSYSTEM_TYPE_APPLICATION = "osgi.subsystem.application"**

The resource type value identifying an application subsystem.

This value is used for the osgi.identity capability attribute type, the SUBSYSTEM_TYPE manifest header and the SUBSYSTEM_TYPE_PROPERTY service property.

134.21.4.31   **public static final String SUBSYSTEM_TYPE_COMPOSITE = "osgi.subsystem.composite"**

The resource type value identifying an composite subsystem.

This value is used for the osgi.identity capability attribute type, the SUBSYSTEM_TYPE manifest header and the SUBSYSTEM_TYPE_PROPERTY service property.

134.21.4.32   **public static final String SUBSYSTEM_TYPE_FEATURE = "osgi.subsystem.feature"**

The resource type value identifying an feature subsystem.

This value is used for the osgi.identity capability attribute type, the SUBSYSTEM_TYPE manifest header and the SUBSYSTEM_TYPE_PROPERTY service property.

134.21.4.33   **public static final String SUBSYSTEM_TYPE_PROPERTY = "subsystem.type"**

The name of the service property for the subsystem type.

See Also SUBSYSTEM_TYPE_APPLICATION, SUBSYSTEM_TYPE_COMPOSITE,
SUBSYSTEM_TYPE_FEATURE

134.21.4.34   **public static final String SUBSYSTEM_VENDOR = "Subsystem-Vendor"**

Manifest header identifying a subsystem's vendor.

Since 1.1

134.21.4.35   **public static final String SUBSYSTEM_VERSION = "Subsystem-Version"**

Manifest header value identifying the version of the subsystem. If not present, the default value is 0.0.0.

134.21.4.36   **public static final String SUBSYSTEM_VERSION_PROPERTY = "subsystem.version"**

The name of the service property for the subsystem version. The value of this property must be of type Version.

134.21.5   public class SubsystemException extends RuntimeException

A Subsystem exception used to indicate a problem.

134.21.5.1   **public SubsystemException()**

☐ Construct a Subsystem exception with no message.

134.21.5.2   **public SubsystemException(String message)**

message   The message to include in the exception.

☐ Construct a Subsystem exception specifying a message.

134.21.5.3   **public SubsystemException(Throwable cause)**

cause   The cause of the exception.

☐ Construct a Subsystem exception specifying a cause.

134.21.5.4   **public SubsystemException(String message,Throwable cause)**

message   The message to include in the exception.

cause   The cause of the exception.

☐ Construct a Subsystem exception specifying a message and a cause.

134.21.6   public final class SubsystemPermission extends BasicPermission

A bundle's authority to perform specific privileged administrative operations on or to get sensitive information about a subsystem. The actions for this permission are:

```
Action     Methods
context    Subsystem.getBundleContext
execute    Subsystem.start
           Subsystem.stop
lifecycle  Subsystem.install
           Subsystem.uninstall
metadata   Subsystem.getSubsystemHeaders
           Subsystem.getLocation
```

The name of this permission is a filter expression. The filter gives access to the following attributes:

- location - The location of a subsystem.
- id - The subsystem ID of the designated subsystem.
- name - The symbolic name of a subsystem.

Filter attribute names are processed in a case sensitive manner.

Concurrency   Thread-safe

134.21.6.1   **public static final String CONTEXT = "context"**

The action string context.

134.21.6.2   **public static final String EXECUTE = "execute"**

The action string execute.

134.21.6.3   **public static final String LIFECYCLE = "lifecycle"**

The action string lifecycle.

134.21.6.4   **public static final String METADATA = "metadata"**

The action string metadata.

134.21.6.5   **public SubsystemPermission(String filter,String actions)**

filter   A filter expression that can use, location, id, and name keys. Filter attribute names are processed in a case sensitive manner. A special value of "∗" can be used to match all subsystems.

actions   execute, lifecycle, metadata, or context.

□   Create a new SubsystemPermission. This constructor must only be used to create a permission that is going to be checked.

Examples:

```
(name=com.acme.*)(location=http://www.acme.com/subsystems/*))
(id>=1)
```

Throws   IllegalArgumentException– If the filter has an invalid syntax.

134.21.6.6   **public SubsystemPermission(Subsystem subsystem,String actions)**

subsystem   A subsystem.

actions   execute, lifecycle, metadata, or context.

□   Creates a new requested SubsystemPermission object to be used by the code that must perform checkPermission. SubsystemPermission objects created with this constructor cannot be added to an SubsystemPermission permission collection.

134.21.6.7   **public boolean equals(Object obj)**

obj   The object being compared for equality with this object.

□   Determines the equality of two SubsystemPermission objects.

Returns   true if obj is equivalent to this SubsystemPermission; false otherwise.

134.21.6.8   **public String getActions()**

□   Returns the canonical string representation of the SubsystemPermission actions.

Always returns present SubsystemPermission actions in the following order: execute, lifecycle, metadata, context.

Returns   Canonical string representation of the SubsystemPermission actions.

134.21.6.9   **public int hashCode()**

□   Returns the hash code value for this object.

Returns   Hash code value for this object.

134.21.6.10   **public boolean implies(Permission p)**

p   The requested permission.

    ☐ Determines if the specified permission is implied by this object. This method throws an exception if the specified permission was not constructed with a subsystem.

This method returns true if the specified permission is a SubsystemPermission AND

- this object's filter matches the specified permission's subsystem ID, subsystem symbolic name, and subsystem location OR
- this object's filter is "∗"

AND this object's actions include all of the specified permission's actions.

Special case: if the specified permission was constructed with "∗" filter, then this method returns true if this object's filter is "∗" and this object's actions include all of the specified permission's actions

Returns true if the specified permission is implied by this object; false otherwise.

134.21.6.11 **public PermissionCollection newPermissionCollection()**

    ☐ Returns a new PermissionCollection object suitable for storing SubsystemPermissions.

Returns A new PermissionCollection object.

134.22 References

[1] *Filter Based Permissions*
OSGi Core, Chapter 2, Filter Based Permissions

[2] *Core Service Hooks*
OSGi Core, Chapter 55 Service Hook Service Specification

[3] *Resource and Wiring*
OSGi Core, Chapter 6 Resource API Specification

[4] *Zip File Format*
The Zip file format as defined by the java.util.zip package.

[5] *IANA application/vnd.osgi.subsystem*
www.iana.org/assignments/media-types/application/vnd.osgi.subsystem

[6] *RFC 1738 Uniform Resource Locators*
http://www.ietf.org/rfc/rfc1738.txt

[7] *Uniform Resource Identifiers (URI): Generic Syntax*
http://www.ietf.org/rfc/rfc2396.txt

[8] *Equinox Region Digraph*
http://underlap.blogspot.com/2011/02/stumbling-towards-better-design.html

[9] *Open Source initiative*
http://www.opensource.org/

134.23 Changes

- Specified the translation of subsystem manifest headers.
- Added more informational subsystem manifest headers.
- Specified the interaction between Subsystems and Weaving Hooks for the purposes of supporting dynamically added import packages.

- Added the ability to override the archive's deployment manifest at install time.
- Added the ability to retrieve deployment manifest headers.
- Specified a means for declaratively identifying the services a bundle provides.
- Modified the procedure for determining dependencies to include the capabilities, in a specified order, from all repositories in order to increase the likelihood that a resolution will be found such as when a uses constraint violation would otherwise result.

135   Common Namespaces Specification

Version 1.1

135.1   Introduction

A key aspect of the OSGi general dependency model based on requirements and capabilities is the concept of a *Namespace*. A Namespace defines the semantics of a Requirement-Capability pair. The generic model is defined in the [3] *Resources API Specification*. This section defines a number of Namespaces that are not part of the *OSGi Core Release 6* specification. Unless an attribute is specifically overridden, all Namespaces inherit the attributes and directives of the default Namespace as defined [4] *Framework Namespaces Specification*.

Each Namespace is defined with the following items:

- *Name* - the name of an attribute or directive
- *Kind* - Defines where the attribute or directive can be used
  - CA - Capability Attribute
  - CD - Capability Directive
  - RA - Requirement Attribute
  - RD - Requirement Directive
- *M/O* - Mandatory (M) or Optional (O)
- *Type* - The data type
- *Syntax* - Any syntax rules. The syntax refers in general to the syntaxes defined in [5] *General Syntax Definitions* and [6] *Common Headers*.

135.1.1   Versioning

In general, capabilities in a Namespace are versioned using Semantic Versioning. See [7] *Semantic Versioning*. Therefore, a capability will specify a single version and a requirement will specify a version range. See *osgi.extender Namespace* for an example.

For some Namespaces, capabilities are not versioned using Semantic Versioning. The versioning scheme used in those Namespaces will be described in the specification for the Namespace.

135.2   osgi.extender Namespace

An *Extender* is a bundle that uses the life cycle events from another bundle, the *extendee*, to extend that bundle's functionality when that bundle is active. It can use metadata (headers, or files inside the extendee) to control its functionality. Extendees therefore have a dependency on the Extender that can be modeled with the `osgi.extender` Namespace. The definition for this Namespace can be found in the following table and the `ExtenderNamespace` class.

Table 135.1   *osgi.extender Namespace*

Name	Kind	M/O	Type	Syntax	Description
osgi.extender	CA	M	String	symbolic-name	A symbolic name for the extender. These names are defined in their respective specifications and should in general use the specification top level package name. For example, org.acme.foo. The OSGi Alliance reserves names that start with "os-gi.".
version	CA	M	Version	version	A version. This version must correspond to the specification of the extender.

Specifications for extenders (Blueprint, Declarative Services, etc.) should specify the values for these attributes. Extenders that provide such a capability should list the packages that they use in their specification in the uses directive of that capability to ensure class space consistency. For example a Declarative Services implementation could declare its capability with the following manifest header:

```
Provide-Capability: osgi.extender;
   osgi.extender="osgi.component";
   uses:="org.osgi.service.component";
   version:Version="1.3"
```

A bundle that depends on a Declarative Services implementation should require such an extender with the following manifest header:

```
Require-Capability: osgi.extender;
   filter:="(&(osgi.extender=osgi.component)(version>=1.3)(!(version>=2.0)))"
```

Extenders can extend an extendee bundle even if that bundle does not require the extender, unless the extender's specification explicitly forbids this. It is recommended that an extender should only extend a bundle if one of the following is true:

- The bundle's wiring has a required wire for at least one osgi.extender capability with the name of the extender and the first of these required wires is wired to the extender.
- The bundle's wiring has no required wire for an osgi.extender capability with the name of the extender.

Otherwise, the extender should not extend the bundle.

135.2.1 Extenders and Framework Hooks

The Framework provides a number of hooks that allow groups of bundles to be scoped. For example, the *Subsystem Service Specification*. An extender may want to extend the complete set of bundles installed in the Framework even when extendee bundles are hidden from the extender. The system bundle context provides a complete view of the bundles and services available in the Framework even if Framework hooks are used to scope groups of bundles. The system bundle context can be used by an extender to track all bundles installed in the Framework regardless of how Framework hooks are used to scope groups of bundles. This is useful in scenarios where several scoped groups contain bundles that require an extender. Instead of requiring an extender to be installed in each scoped group of bundles, a single extender that uses the system bundle context to track extendees can be installed to extend all scoped groups of bundles.

135.3 osgi.contract Namespace

Products or technologies often have a number of related APIs consisting of a large set of packages. Some IDEs have not optimized for OSGi and requires work for each imported package. In these development environments using modularized systems tends to require a significant amount of manual effort to manage the imported packages.

The `osgi.contract` Namespace addresses this IDE deficiency. It allows a developer to specify a single name and version for a contract that can then be expanded to a potentially large number of packages. For example, a developer can then specify a dependency on Java Enterprise Edition 6 contract that can be provided by an application server.

The `osgi.contract` Namespace provides such a name and binds it to a set of packages with the uses constraint. The bundle that declares this contract must then import or export each of the listed packages with the correct versioning. Such a bundle is called a *contract bundle*. The contract bundle must ensure that it is bound to the correct versions of the packages contained within the contract it is providing. If the contract bundle imports the packages which are specified as part of the contract then proper matching attributes must be used to make sure it is bound to the correct versions of the packages.

Additionally, the `osgi.contract` Namespace can be used in cases where API is defined by parties that do not use Semantic Versioning. In those cases, the version of the exported package can be unclear and so it is difficult to specify a meaningful version range for the package import. In such cases, importing the package *without* specifying a version range and specifying a requirement in the `osgi.contract` Namespace can provide a way to create portable bundles that use the API. OSGi has defined contract names for a number of such APIs. See [2] *Portable Java Contract Definitions* for more information.

An `osgi.contract` capability can then be used in the following ways:

- IDEs can use the information in the `uses` directive to make all those packages available on the build path. In this case the developer no longer has to specify each package separately.
- During run time the `uses` clause is used to enforce that all packages in the contract form a consistent class space.

The `uses` directive will make it impossible to get wired to packages that are not valid for the contract. Since the uses constrains enforce the consistency, it is in principle not necessary to version the imported packages on client bundles since only the correctly versioned packages can be used. Contracts are aggregates and therefore make clients depend on the whole and all their transitive dependencies, even if the client only uses a single package of the contract.

The recommended way of using contracts is to create a contract bundle that provides the `osgi.contract` capability and imports the packages with their required version range. For example:

```
Provide-Capability: osgi.contract;
    osgi.contract=JavaServlet;
    version:Version=2.5;
    uses:="javax.servlet,javax.servlet.http"
Export-Package:
    javax.servlet;       version="2.5",
    javax.servlet.http;  version="2.5"
```

A contract may support multiple versions of a named contract. Such a contract must use a single capability for the contract name that specifies a list of all the versions that are supported. For example, the JavaServlet 3.1 contract capability would be specified with the following:

```
Provide-Capability: osgi.contract;
```

```
        osgi.contract=JavaServlet;
        version:List<Version>="2.5,3.0,3.1";
        uses:=
            "javax.servlet,
            javax.servlet.annotation,
            javax.servlet.descriptor,
            javax.servlet.http"
Export-Package:
        javax.servlet;              version="3.1",
        javax.servlet.annotation;   version="3.1",
        javax.servlet.descriptor;   version="3.1",
        javax.servlet.http;         version="3.1"
```

A client bundle that requires the Servlet 2.5 contract can then have the following manifest:

```
Require-Capability: osgi.contract;
        filter:="(&(osgi.contract=JavaServlet)(version=2.5))",
Import-Package:
        javax.servlet, javax.servlet.http
```

The client bundle will be constrained by the contract's uses constraints and automatically gets the correct packages. In this example, no semantic versioning is used for the contract because the Servlet Specifications do not use semantic versioning (version 3.0 is backward compatible with 2.X).

In this model it is even possible to use the normally not recommended DynamicImport-Package header with a wild card since also this header is constrained by the uses constraints. However, using a full wildcard can also dynamically import packages that are not part of the contract. To prevent these unwanted dynamic imports, the exporter could include an attribute on the exports. For example:

```
Require-Capability: osgi.contract;
        filter:="(&(osgi.contract=JavaServlet)(version=2.5))"
DynamicImport-Package:
        *;JavaServlet=contract
```

However, this model requires the exporter to specify an agreed attribute. The contract bundle does not require such coordination; it also allows the package exporters to reside in different and unrelated bundles.

The definition of the osgi.contract Namespace is in the following table and in the ContractNamespace class. See [2] *Portable Java Contract Definitions*.

Table 135.2   *osgi.contract Namespace*

Name	Kind	M/O	Type	Syntax	Description
osgi.contract	CA	M	String	symbolic-name	A symbolic name for the contract.
version	CA	O	Version+	version	A list of versions for the contract. A contract that supports multiple versions must use a single capability with a version attribute that lists all versions supported.
uses	CD	O	String	package-name (',' package-name)	For a contract, the standard uses clause is used to indicate which packages are part of the contract. The imports or exports of those packages link these packages to a particular version.

135.3.1   Versioning

As the osgi.contract Namespace follows the versioning of the associated contract, capabilities in this Namespace are *not* semantically versioned. The associated contracts are often versioned using

marketing or other versioning schemes and therefore the version number cannot be used as an indication of backwards compatibility.

As a result, capabilities in the osgi.contract Namespace use a *discrete* versioning scheme. In such a versioning scheme, each version is treated as separate without any implied relation to another version. A capability lists *all* compatible versions. A requirement only selects a single version.

135.4 osgi.service Namespace

The Service Namespace is intended to be used for:

- Preventing a bundle from resolving if there is not at least one bundle that potentially can register a specific service.
- Providing a hint to the provisioning agent that the bundle requires a given service.
- Used as template for specifications like Blueprint and Declarative Services to express their provided and referenced services in the Repository model, see the *Repository Service Specification*.

A bundle providing this capability indicates that it can register such a service with at least the given custom attributes as service properties. At resolve time this is a promise since there is no guarantee that during runtime the bundle will actually register such a service; clients must handle this with the normal runtime dependency managers like Blueprint, Declarative Services, or others.

See the following table and the ServiceNamespace class for this Namespace definition.

Table 135.3   *osgi.service Namespace*

Name	Kind	M/O	Type	Syntax	Description
objectClass	CA	M	List <String>	qname (',' qname)*	The fully qualified name of the object class of the service.
*	CA	O	*	*	Custom attributes that will be provided as service properties if they do not conflict with the service properties rules and are not private service properties. Private properties start with a full stop ('.' \u002E).

135.4.1 Versioning

Capabilities in the osgi.service Namespace are *not* versioned. The package of a service's object class is generally versioned and the package can be associated with the capability via the uses directive.

135.5 osgi.implementation Namespace

The Implementation Namespace is intended to be used for:

- Preventing a bundle from resolving if there is not at least one bundle that provides an implementation of the specified specification or contract.
- Providing uses constraints to ensure that bundles which require an implementation of a specification or contract will be wired appropriately by the framework.
- Providing a hint to the provisioning agent that the bundle requires a given specification or contract implementation.
- Used as a general capability Namespace for specifications or contracts to express their provided function in the Repository model, see the *Repository Service Specification*.

A bundle providing this capability indicates that it implements a specification or contract with the specified name and version. For example, the *Asynchronous Service Specification* would provide the following capability:

```
Provide-Capability: osgi.implementation;
    osgi.implementation="osgi.async";
    version:Version="1.0";
    uses:="org.osgi.service.async"
```

See the following table and the ImplementationNamespace class for this Namespace definition.

Table 135.4   *osgi.implementation Namespace*

Name	Kind	M/O	Type	Syntax	Description
osgi.implementation	CA	M	String	symbolic-name	The symbolic name of the specification or contract. The OSGi Alliance reserves names that start with "osgi.".
version	CA	M	Version	version	The version of the implemented specification or contract.
*	CA	O	*	*	Custom attributes that can be used to further identify the implementation

135.6   org.osgi.namespace.contract

Contract Namespace Package Version 1.0.

Bundles should not need to import this package at runtime since all the types in this package just contain constants for capability and requirement namespaces specified by the OSGi Alliance.

135.6.1   Summary

- ContractNamespace - Contract Capability and Requirement Namespace.

135.6.2   public final class ContractNamespace extends Namespace

Contract Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace. All unspecified capability attributes are of type String and are used as arbitrary matching attributes for the capability. The values associated with the specified directive and attribute keys are of type String, unless otherwise indicated.

Concurrency Immutable

135.6.2.1   public static final String CAPABILITY_VERSION_ATTRIBUTE = "version"

The capability attribute contains the Version of the specification of the contract. The value of this attribute must be of type Version.

135.6.2.2   public static final String CONTRACT_NAMESPACE = "osgi.contract"

Namespace name for contract capabilities and requirements.

Also, the capability attribute used to specify the name of the contract.

135.7   org.osgi.namespace.extender

Extender Namespace Package Version 1.0.

Bundles should not need to import this package at runtime since all the types in this package just contain constants for capability and requirement namespaces specified by the OSGi Alliance.

135.7.1   Summary

- `ExtenderNamespace` - Extender Capability and Requirement Namespace.

135.7.2   public final class ExtenderNamespace
extends Namespace

Extender Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace. All unspecified capability attributes are of type `String` and are used as arbitrary matching attributes for the capability. The values associated with the specified directive and attribute keys are of type `String`, unless otherwise indicated.

Concurrency   Immutable

135.7.2.1   public static final String CAPABILITY_VERSION_ATTRIBUTE = "version"

The capability attribute contains the Version of the specification of the extender. The value of this attribute must be of type Version.

135.7.2.2   public static final String EXTENDER_NAMESPACE = "osgi.extender"

Namespace name for extender capabilities and requirements.

Also, the capability attribute used to specify the name of the extender.

135.8   org.osgi.namespace.service

Service Namespace Package Version 1.0.

Bundles should not need to import this package at runtime since all the types in this package just contain constants for capability and requirement namespaces specified by the OSGi Alliance.

135.8.1   Summary

- `ServiceNamespace` - Service Capability and Requirement Namespace.

135.8.2   public final class ServiceNamespace
extends Namespace

Service Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace.

All unspecified capability attributes are of one of the following types:

- `String`
- `Version`
- `Long`

- Double
- List<String>
- List<Version>
- List<Long>
- List<Double>

and are used as arbitrary matching attributes for the capability. The values associated with the speci-fied directive and attribute keys are of type String, unless otherwise indicated.

Concurrency   Immutable

135.8.2.1   **public static final String CAPABILITY_OBJECTCLASS_ATTRIBUTE = "objectClass"**

The capability attribute used to specify the types of the service. The value of this attribute must be of type List<String>.

A ServiceNamespace capability should express a uses constraint for all the packages mentioned in the value of this attribute.

135.8.2.2   **public static final String SERVICE_NAMESPACE = "osgi.service"**

Namespace name for service capabilities and requirements.

135.9   org.osgi.namespace.implementation

Implementation Namespace Package Version 1.0.

Bundles should not need to import this package at runtime since all the types in this package just contain constants for capability and requirement namespaces specified by the OSGi Alliance.

135.9.1   Summary

- ImplementationNamespace - Implementation Capability and Requirement Namespace.

135.9.2   public final class ImplementationNamespace
extends Namespace

Implementation Capability and Requirement Namespace.

This class defines the names for the attributes and directives for this namespace.

This class defines the names for the attributes and directives for this namespace. All unspecified ca-pability attributes are of type String and are used as arbitrary matching attributes for the capability. The values associated with the specified directive and attribute keys are of type String, unless other-wise indicated.

Concurrency   Immutable

135.9.2.1   **public static final String CAPABILITY_VERSION_ATTRIBUTE = "version"**

The capability attribute contains the Version of the specification or contract being implemented. The value of this attribute must be of type Version.

135.9.2.2   **public static final String IMPLEMENTATION_NAMESPACE = "osgi.implementation"**

Namespace name for "implementation" capabilities and requirements. This is also the capability at-tribute used to specify the name of the specification or contract being implemented.

A ImplementationNamespace capability should express a uses constraint for the appropriate pack-ages defined by the specification/contract the packages mentioned in the value of this attribute.

135.10   References

[1]   *Specification References*
http://www.osgi.org/Specifications/Reference

[2]   *Portable Java Contract Definitions*
http://www.osgi.org/Specifications/ReferenceContract

[3]   *Resources API Specification*
OSGi Core, Chapter 6 Resource API Specification

[4]   *Framework Namespaces Specification*
OSGi Core, Chapter 8 Framework Namespaces Specification

[5]   *General Syntax Definitions*
OSGi Core, General Syntax Definitions

[6]   *Common Headers*
OSGi Core, Chapter 3, Common Header Syntax

[7]   *Semantic Versioning*
OSGi Core, Chapter 3, Semantic Versioning

135.11   Changes

- Clarified osgi.extender Namespace and added a section on the relationship with framework hooks.
- Clarified osgi.contract Namespace and examples.
- Added *osgi.implementation Namespace* on page 685.
- Clarified versioning of capabilities in a namespace.

136 Resolver Service Specification

Version 1.0

136.1 Introduction

Today very few applications are self contained, the predominant development model is that applications are built from (external) components, which are often open source. Application developers add business logic, glue code, and assemble the diverse components into a resource that provides the desired capabilities when installed in an environment. Designing the assembly has long been a manual and error prone process, partly due to the complexity of external dependencies. Although the direct dependencies are often given, the largest number of dependencies are usually the transitive dependencies: the dependencies of the dependencies. Modern applications can end up with hundreds to thousands of external dependencies. Numbers that make tooling inevitable.

The OSGi framework is the first specification that provides a foundation for automating a significant part of this assembly process. The Requirement-Capability model defined in [3] *Framework Resource API (Core)* provides a dependency model that allows resources to express dependencies, constraints, and capabilities. If a resource's constraints are met it provides capabilities that can satisfy further requirements. The OSGi dependency model is fully generic and is not limited to bundles. Resources can be bundles but also certificates, plugged in devices, etc.

Resolving transitive dependencies is a non-trivial process that requires careful design to achieve the required performance since the underlying problem is NP-complete. OSGi frameworks have always included such resolvers but these were built into the frameworks. They were not usable outside the framework for tooling, for example automatically finding the dependencies of a bundle that needs to be installed.

The number of dependencies is rapidly reaching a threshold where manual methods no longer can provide reliable results. This specification therefore provides the *Resolver service*, a service that can be the base for provisioning, deployment, build, and diagnostic tooling. The service can take a requirement and resolve it to a wiring of resources. For example, with cloud computing a new requirement can be translated into a new OSGi framework instance being started on a node and provisioned with the set of bundles that satisfy the given requirement. The OSGi Resolver service is intended be a corner stone of such an auto-provisioning tool.

However, the OSGi Resolver service is not limited to these higher end schemes. Build tools can use the Resolver to find components for the build path and/or run time environment and predict the results of installing a set of bundles on a target environment. The OSGi Resolver service is an essential part of a software model where applications are built out of independent components.

This specification is based on the concepts and API defined in the [3] *Framework Resource API (Core)*, [2] *Framework Wiring API (Core)*, and the [1] *Framework Module Layer*. These specifications are required reading for understanding this specification. This specification is for highly specialized use, it is not intended to be used in applications, the Resolver API is a low level service intended for system developers with deep knowledge of the OSGi module layer.

136.1.1 Essentials

- *Transitive* - From a requirement, find a consistent set of resources that satisfy that requirement.
- *Diagnostics* - Provide diagnostic information when no resolution can be found.

- *Scoped Repositories* - Allow the environment to control the repositories to use.
- *Build Tools* - Must be useful in establishing build and run time class paths.
- *Provisioning* - Must be useful to find a set of bundles that can be installed in a system without running into unresolved dependencies.
- *OSGi* - Provide the semantics of all the OSGi namespaces, including the uses constraints.
- *API* - The API for the Resolver must provide the base for the Framework Bundle Wiring API.
- *Performant* - Enable highly performant implementations.
- *Frameworks* - Allow Frameworks to provide their resolver as a service.
- *Scalable* - Allow access to, and use of, very large repositories.
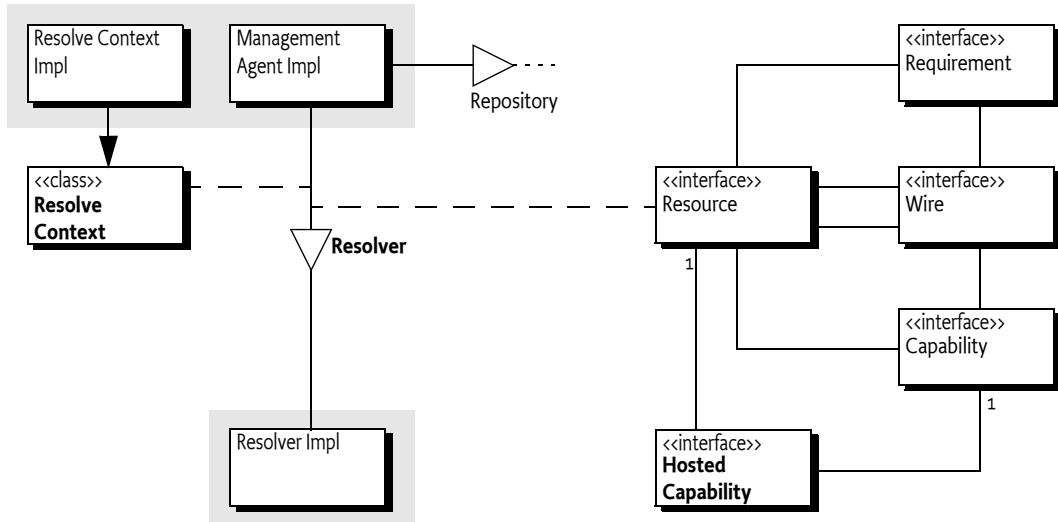
136.1.2   Entities

- *Environment* - A container or framework that can install resources and uses a Resolver to wire these resources.
- *Resolve Context* - An interface implemented by the management agent to provide the context of the resolution.
- *Wiring* - Represents the state of a resource's wires, requirements, and capabilities in an environment.
- *Resolver* - A service that can find a set of wires that can be applied to an existing wiring state where a set of initial resources have all their mandatory requirements satisfied.
- *Wire* - Links requirement to a capability.
- *Resource* -An artifact with requirements that need to be provisioned and resolved to provide its capabilities.
- *Requirement* - A necessity for a given resource before it can provide its capabilities; expressed as a filter expression on the attributes of a capability in a given namespace.
- *Capability* - A set of attributes and directives defined by a namespace, describes what a resource provides when resolved.
- *Hosted Capability* - Pairs a resource with a capability to model hosting capabilities in another resource.
- *Namespace* - The type for a capability and requirement.
- *Resolution* - The result of a resolve operation.

Figure 136.1   Class and Service overview

136.1.3   Synopsis

The Resolver service can find a complete and consistent set of transitive dependencies starting with an initial set of mandatory and optional resources. Such a set can be used to install resources in the environment from local or remote repositories. To use the Resolver service, the client must provide a `ResolveContext` object. This complex object represents the context of a resolution; it provides the initial resources (optional and mandatory), defines a number of namespaces specific policies, and provides the state of the environment.

A resolution can be based on an existing wiring in the environment, for example the current framework state. For the framework, the Resolve Context can find this existing state via the [2] *Framework Wiring API (Core)*. The Resolver must then satisfy all the requirements of the mandatory resources. The Resolver must always ask the Resolve Context to find additional capabilities for the unsatisfied requirements. A capability is always associated with a resource, which is subsequently associated with additional requirements. The final resolution must contain a set of resources that include the initial set of mandatory resources, has no unsatisfied mandatory requirements, and is consistent with the implied constraints. Otherwise it fails.

The Requirement-Capability model is fully generic but provides special semantics through the use of namespaces. The Resolver must implement the standard OSGi namespaces as described in [2] *Framework Wiring API (Core)*, which includes the `uses` constraints. Part of the semantics of the OSGi namespaces are delegated to the Resolve Context so that it can implement different policies. Singletons, ordering of capabilities, and matching are the responsibility of the Resolve Context; the Resolver never matches a requirement to a capability.

Requirements have an `effective` directive that indicates in what situations the requirement must be resolved. Also here, the Resolve Context decides if a particular requirement is effective or not. Only effective requirements are wired in the resolution.

Since capabilities are declared by resources that have additional requirements, the Resolver must create a set of resources where all transitive requirements are satisfied or fail with a Resolution Exception. This Resolution Exception can provide additional details why the resolution failed, if possible.

At the end of a successful resolve operation the Resolver returns a `Map<Resource,List<Wire>>`. These wires are a delta on the existing state, if any. The wires can then be used to provision missing resources or to provide diagnostic feedback.

136.2   The Resolve Context

Provisioning is the process of providing a framework with the necessary resources to allow it to operate according to set goals. In OSGi terms, this consists of installing bundles and ensuring that the configuration is set up correctly. With OSGi, bundles explicitly describe their capabilities and requirements as manifest headers. This can range from Export-Package (a capability) to a generic Provide-Capability header.

OSGi Frameworks have a *resolving* stage that ensures requirements are satisfied before a bundle is allowed to provide code to the shared space. As long as the requirements are not met, the bundle remains in the INSTALLED state and is thus prohibited from contributing capabilities. Once all the mandatory requirements are met, the bundle becomes RESOLVED. That is, a framework combines two decisions when it resolves bundles:

- Find a resolution based on the existing set of installed bundles.
- Move the bundles that have all their mandatory requirements satisfied to the RESOLVED state.

The *Resolver service* separates these two stages and thus allows a third party, the *management agent*, to define the *environment* of the resolution. A management agent can interact with the Resolver service

while it is searching for a resolution because the Resolver service calls back the management agent through a ResolveContext object. The Resolver service will therefore allow the management agent to handle more scenarios, better diagnostics, etc.

The Resolve Context is provided by the management agent, it is an abstract base class and must therefore be extended. It is a non-trivial class to implement since it is tightly coupled to the rules of the environment; it represents the policies of the management agent for that environment. For OS-Gi framework resolvers, the Resolve Context must understand and implement a part of the OSGi framework namespaces.

With the Resolver service, a management agent can try out resolutions before any bundle can see the result of such a resolution but it can also include extra bundles or other resources on demand. The Resolver service will also allow resolutions to be calculated for other frameworks.

For example, a management agent could use a Resolver service to find missing dependencies and install them on demand from a local directory with bundles. Such a Provisioner could have the following skeleton:

```
public class Provisioner {
  File                  bundles   = ...;
  Map<String,Resource>  resources = ...;
  Resolver              resolver  = ...;
  BundleContext         context   = ...;

  public void install(String location) {
    Resource resource = resources.get( location );
    if ( resource == null ) error(...);

    try {
       ResolveContextImpl rc = ...
       rc.addMandatory( resource );
      Set<Resource> provision = resolver.resolve( rc ).keySet();

      for ( Resource rb : provision ) {
        String location = getLocation( rb );

        Bundle bundle = context.installBundle( location );
            if ( !isFragment( bundle ) )
            bundle.start();
      }
    } catch(ResolutionException re) {
      ... // diagnostics
    } catch(BundleException be) {
      ... // diagnostics
    }
  }
}
```

136.2.1 Mandatory and Optional Resources

The Resolve Context provides all the parameters for the resolve operation, the Resolver does not maintain any state between invocations. The Resolve Context must therefore provide the mandatory and optional resources, which are essentially the input parameters to the resolve operation. The resolver must find a solution that includes at least the initial mandatory resources and should include the optional resources.

136.2.2   Finding Capabilities

The Resolve Context's findProviders(Requirement) method must be implemented in such a way that it returns an ordered list of capabilities. The Resolver will treat the order of the capabilities as preferences, the first element is more preferred than a later element. The Resolver cannot guarantee that the wiring obeys this preference since there can be other constraints. However, a Resolver must use this preference order for simple cases and try to use it in more constrained situations.

The Resolver does not make any assumptions, this means that the findProviders(Requirement) method must do all the matching. Even though the Resolver gets the mandatory and optional resources it will not search these for capabilities to satisfy requirements. If the findProviders(Requirement) method does not search these resources then their capabilities will not be used. The same is true for the existing wiring state used.

Since this section describes the Resolver with respect to a provisioning agent, the set of resources is not limited to the installed set. That is, normally when a framework is resolved the Resolver only has to include installed resources. However, for a provisioning agent it is possible to retrieve external resources. The *Repository Service Specification* on page 577 provides access to resource repositories but a management agent is free to find capabilities by any alternative means.

For resolving an OSGi framework the specifications outlines a number of heuristics that guide the order of wiring bundles and packages:

1. A resource that is already resolved, that is, it is already wired
2. The highest version
3. The lowest bundle id

The Resolver can, and likely will, use the returned list to maintain its internal state during the resolve operation while trying out different potential solutions. It can add and remove capabilities at will. The returned list must therefore be mutable and not reused, after it is returned; it becomes owned by the Resolver. However, the Resolver can call back later with the insertHostedCapability(List,HostedCapability) method, giving back the returned list as the first parameter, see *Insert Hosted Capabilities* on page 698.

For example, assume that all possible resources in the previous example can be gotten with the getSortedResources method in the preferred resource order. This list contains all installed resources as well as any potentially installable resources. This list is sorted once on the given ordering criteria, this only has to be done once and not for each findProviders(Requirement) method invocation. The following code, which does not order by capability versions, could then be a simple skeleton for the findProviders(Requirement) method on the ResolveContextImpl inner class:

```
public List<Capability> findProviders(Requirement requirement ) {
  List<Capability> result = new ArrayList<Capability>();

  for ( Resource r : getSortedResources() )
    for ( Capability c : r.getCapabilities( null ) )
      if ( match( requirement, c ) )
          result.add( c );

  return result;
}
```

136.2.3   Matching

The findProviders(Requirement) method is responsible for only returning the capabilities that the management agent wants to be considered by the Resolver. Since the Resolver must never match any requirements to capabilities it allows the management agent to fully control the matching. However, in an OSGi environment the following matching behavior is expected:

- Requirements and capabilities must be in the same namespace.
- Only requirements and capabilities that have no effective directive or have the directive set to resolve should be considered.
- The requirement's filter must match the capability's attributes.
- If the namespace is an osgi.wiring.* namespace then the mandatory directive on the capability must be supported. Mandatory attributes are defined with a mandatory directive on a capability, they contain a list of attribute names. Each of these attributes must be used in the filter. Since the filter must be constructed from the corresponding manifest header it is sufficient to search the filter string with a regular expression that detects the usage of an attribute name.

The following example shows a skeleton match method that provides OSGi semantics:

```
boolean match(Requirement r, Capability c){
  if ( !r.getNamespace().equals( c.getNamespace() ) )
    return false;

  String effective = c.getDirectives().get("effective");
  if ( !(effective == null || effective.equals( "resolve") ) )
    return false;

  String f = r.getDirectives().get( "filter" );
  if ( f  != null ) {
    Filter filter = context.createFilter( f );
    if ( !filter.matches( c.getAttributes() ) )
      return false;
  }

  if ( !c.getNamespace().startsWith( "osgi.wiring." ) )
    return true;

  String mandatory = c.getDirectives().get("mandatory");
  if ( mandatory == null)
    return true;

  List<String> attrs =
    Arrays.asList( mandatory.toLowerCase().split( "\\s*,\\s*") );

  Matcher m = FILTER_ASSERT_PATTERN.matcher( f == null ? "": f );
  while( m.find() )
    attrs.remove(m.group(1)); // the attribute name

  return mandatory.isEmpty();
}
```

136.2.4 Repositories

Resolving to provision a framework is different than a framework resolving itself. During provisioning remote repositories can be consulted to find external resources while the framework only resolves a limited set (the installed bundles). These repositories generally contain magnitudes more bundles than what is installed in a framework.

Repositories do not implement any namespace specific semantics and therefore do not understand any directives. Repositories only verify the requirement's filter (if any) against the capability's attributes. The Resolver expects the Resolve Context to do the namespace specific matching. The *Repository Service Specification* on page 577 provides the details for a service that abstracts a Requirement-Capability aware repository.

With such a repository service the findProviders(Requirement) method can be implemented as follows:

```
List<Repository> repositories = new CopyOnWriteArrayList<Repository>();

void addRepository(  Repository repository) { repositories.add(repository);}
void removeRepository(Repository repository){ repositories.remove(repository);}

public List<Capability> findProviders( Requirement requirement) {
  List<Capability> result = new ArrayList<Capability>();

  // previous findProviders that searches the initial resources

  for ( Repository repository : repositories ) {
    Collection<Capability> capabilities = repository.findProviders(
          Collections.singleton( requirement ) ).get( requirement);
    for ( Capability c : capabilities )
      if ( match( requirement, c ) )
          result.add( c );
  }
  return result;
}
```

136.2.5   Existing Wiring State

The Resolver service always creates a list of wires that should be added to an existing state. To get the existing state, the ResolveContext interface specifies the getWirings() method. This method must return the existing state as a Map<Resource,Wiring>. A Wiring is an object that reflects the wired state of a resource in the environment. From this object, all declared and hosted capabilities and requirements can be found, including their wires if any. The Resolver needs this existing state to create a consistent resolution. For example, uses constraints require access to the existing state.

The Resolver service API is based on the generic Requirement-Capability model. This API is implemented by the OSGi framework to reflect its internal wiring, see [2] *Framework Wiring API (Core)*. When the Resolver service is used for an OSGi framework then the Resolve Context can provide the existing wiring state based on the Framework Wiring API. The interfaces used in the org.osgi.framework.wiring package all extend their counterpart in the org.osgi.resource package (the generic model). For example, the BundleCapability interface extends the Capability interface.

The framework wiring API models all the power and complexities of the OSGi framework. One of those aspects is *removal pending*. Each installed bundle is represented by one or more bundle revisions. Each bundle revision is a Resource object but only one is the current bundle revision. During a resolve operation a framework can actually wire to the current bundle revision but is not forbidden to also select the pending removal bundle revisions. The Resolve Context must therefore decide if it provides only the current bundle revisions or all. The best policy solution in this case is to always refresh after a (batch) of install operations and only resolve when there are no pending-removal bundle revisions. However, certain management agents attempt to manage a system that is in this half-way state and will then be required to include the pending-removal revisions.

The following example code shows a possible implementation of the getWirings() method. It only uses the current wiring and ignores removal pending bundle revisions:

```
public Map<Resource,Wiring> getWirings(){
    Map<Resource,Wiring> wirings = new HashMap<Resource,Wiring>();

  for ( Bundle b : context.getBundles() ) {
    BundleRevision revision = b.adapt( BundleRevision.class);
    if ( revision != null ) {
```

```
            Wiring wiring = revision.getWiring();
            if ( wiring != null )
                wirings.put( revision, wiring );
        }
    }
    return wirings;
}
```

136.2.6 Effective

The Resolver service is designed to work with OSGi frameworks but the scope is broader; its design allows many alternative usages. The effective directive on the capabilities and requirements is meant to allow requirements and capabilities to be effective in different phases. The default is resolve, the value for an OSGi framework resolving process. Bundles and other OSGi defined artifacts must declare their capabilities and requirements in the resolve effective time to be considered by an OSGi resolver.

However, Resolvers can use the effective directive to run the Resolver at other times than the standard OSGi framework resolve. For example, it could be possible to define an *active time* to resolve the service dependencies.

For this reason, the Resolver is designed to be agnostic for this directive, it will always ask the ResolveContext if a requirement is effective. It does this with the isEffective(Requirement) method. Since the Resolver service never matches requirements to capabilities it is also up to the Resolve Context to decide how to treat the effective directive. For an OSGi resolve operation, capabilities should also have an effective time value of resolve (or not set since resolve is the default).

To make requirements effective during the resolving of bundles it will be necessary to implement the isEffective(Requirement) method similar to:

```
public boolean isEffective( Requirement requirement) {
    String e = requirement.getDirectives().get( "effective");
    return e==null || "resolve".equals( e );
}
```

136.2.7 Insert Hosted Capabilities

One of the complex aspects of resolving for an OSGi framework is handling fragments. For fragments, the declared capabilities are going to be hosted by their hosts. The Requirement and Capability objects have a getResource method that returns the associated resource. For hosted capabilities and requirements this must be the hosting resource and for others the declaring resource.

The HostedCapability interface defines the interface for allowing the hosting resource to be returned instead of the declaring resource. Since the Resolver service creates these Hosted Capabilities the Resolver needs a way to add them to the lists of capabilities returned from findProviders(Requirement). The Resolver service cannot add them itself since this list has a preference order, the Resolver service must therefore ask the Resolve Context to insert this new capability to allow the Resolve Context to maintain the desired order.

The Resolve Context must therefore implement an insertHostedCapability(List,HostedCapability) method. The given list must have been returned earlier from a findProviders(Requirement) method invocation. The Resolve Context must find the appropriate position to insert the HostedCapability object, insert it, and return the index of the inserted object.

It is the responsibility of the Resolve Context to find the proper position. In *Finding Capabilities* on page 695 it was discussed how the findProviders(Requirement) method must return an ordered list. The insertHostedCapability(List,HostedCapability) has that same responsibility.

The following example shows how the Hosted Capability is inserted based on the index of the hosted resource's index in the sorted list of resources the management agent maintained. The example

iterates through the capabilities and compares the index of sorted resources to indicate preference. If it finds a capability from a resource that was later in the list of sorted resources then it inserts it at that position. A real implementation should also take the version of the capability into account.

```
public int insertHostedCapability(
    List<Capability> caps, HostedCapability hc ) {

    List<Resource> resources = getSortedResources();
    int index = resources.indexOf( hc.getResource() );

    for ( int i =0; i < caps.size(); i++ ) {
        Capability c = caps.get( i );
        int otherIndex = resources.indexOf( c.getResource() );
        if ( otherIndex > index ) {
            caps.add( i, hc );
            return i;
        }
    }
    caps.add( hc );
    return caps.size()-1;
}
```

136.2.8 Fragments

Fragments are resources that have an osgi.wiring.host requirement that must match a capability from one or more host bundles. However, for example an Export-Package in a fragment must be merged with its attached hosts. These capabilities and requirements from namespaces that appear as if they come from the host bundle are called *hosted*.

Since fragments are not required by the host bundle, there will be no resource requiring the fragment bundles. The Resolver will therefore never ask the Resolve Context to provide fragments. However, fragments will require their hosts. A Resolver should attach any fragments available in a resolution to suitable hosts.

Since fragments will not be discovered from their hosts, it can be necessary to run the resolver twice in certain scenarios. The first resolve operation finds a resolution that is then used to find fragments that could be attached to the included bundles. These fragments can then be added to set of mandatory or optional resources for an additional resolve operation.

Fragments can of course also be found by the normal finding of capabilities.

136.2.9 Singleton Capabilities

A resource can be marked as a *singleton*. A singleton resource has the singleton directive set to true on the osgi.identity capability. A singleton resource *conflicts* with another singleton resource if:

- They have the same osgi.identity, and
- They have the same type, and
- They have a different or identical version.

This constraint is not enforced by the Resolver service to give more flexibility to management agents. The Resolve Context must ensure that it does not return capabilities from conflicting singleton resources from the findProviders(Requirement) method. When the Resolver is used with a limited set of resources then it is possible to enumerate all singletons ahead of time and use permutations. However, when repositories are used an incremental method works better because the scope is so much larger.

When the findProviders(Requirement) method is called for a requirement that selects a capability declared in a singleton then it is likely that repositories will return multiple versions of this single-

ton including the resource with the highest available version for conflicting resources. It is therefore possible to maintain a white list of singletons incrementally.

Once the findProviders(Requirement) method has created a result list, it is possible to prune this list of conflicting singletons. This can be an incremental process, the first time a singleton is encountered in such a list of capabilities the highest version can be *selected* as the singleton. Other singletons that are in that list or come in other invocations of findProviders(Requirement) can then no longer provide capabilities. For example:

```
Map<String,Resource> whitelist = new HashMap<String,Resource>();

void prune( List<Capability> list ) {
    Map<String,Resource> singletons = new HashMap<String,Resource>();

    for ( Capability c : list) {
        Resource r = c.getResource();
        Version now = getVersion( r );
        String identity = getIdentity( r );

        if ( isSingleton( r ) && !whitelist.containsKey(identity ) ) {
            Resource selected = singletons.get( identity );
            if ( selected == null )
                singletons.put( identity, r );
            else {
                Version old = getVersion( selected );
                if ( now.compareTo( old )> 0 )
                    singletons.put( identity, r );
            }
        }
    }

    this.whitelist.putAll( singletons );

    for ( Iterator<Capability> i=list.iterator(); i.hasNext();) {
        Capability c = i.next();
        Resource r = c.getResource();
        String identity = getIdentity( r );
        Resource selected = this.whitelist.get( identity );
        if ( selected != null && !selected.equals( r ))
            i.remove();
    }
}
```

136.2.10 Diagnostics

The Resolver service throws a ResolutionException when the resolve operation cannot find a solution. This Exception provides the standard human readable message. However, there is also the getUnresolvedRequirements() method. With this method it is possible to find what requirements could not be matched. Though this is very useful in many cases it must be realized that resolving is a complicated process. It is not possible to establish the exact set of missing requirements because any unresolved resolution can have many, different, sets of unresolved requirements. This is an intrinsic part of the resolution problem. There is also no guarantee that providing capabilities that satisfy these requirements will give a successful resolution. There could still be other constraints that cannot be satisfied. In general, the Resolve Context can already detect any unresolved mandatory requirements when it cannot find a matching capability in the findProviders(Requirement) method.

That said, the getUnresolvedRequirements() can often point to a potential solution.

136.2.11   Complexity

Implementing a Resolve Context is a non-trivial task that requires extensive knowledge of the OS-Gi framework, especially the module layer. Though this section contains numerous code examples, they are not sufficient to implement a real Resolve Context since this would require too much code for a specification.

136.3   Resolver Service

The *Resolver* service is an interface to a generic constraint solver based on the *Require-Capability* model defined in [3] *Framework Resource API (Core)*. This model defines a constraint-solving language that is used by the Framework, see [1] *Framework Module Layer*, to create the mesh of class loaders. However, the Resolver service has been designed to be useful in solving other types of constraint problems.

The task of the Resolver is to find a *resolution*. The resolve method returns a *delta* on an *existing wiring state*. The total of existing wiring state and the applied delta is the resolution. The delta is a set of wires between requirements and capabilities.

136.3.1   Variables

The resolve(ResolveContext) method uses a Resolve Context to provide the context and parameters of the resolution. During the resolution process the Resolver service can callback the Resolve Context to retrieve the following information:

R_m	Collection<Resource>	getMandatoryResources()
R_o	Collection<Resource>	getOptionalResources()
C_{env}	Map<Requirement,List<Capability>>	Combined answers from the findProviders(Requirement) method
Q_{eff}	Collection<Requirement>	Set of effective requirements as defined by the isEffective(Requirement) method
X	Map<Resource,Wiring>	An existing Wiring state, getWirings()

The Resolver service returns the following:

D	Map<Resource,List<Wire>>	The resolution, a delta on the existing state

The resolve(ResolveContext) method returns a resolution D that is a delta on the existing Wiring state X. It is up to the Resolve Context to ensure that the delta D is installed. In for example the OSGi framework the framework hooks can be used to guide the framework's resolving process.

136.3.2   Resolving

The goal of the Resolver is to provide a set of wires between a set of *resolved* resources. A resource is resolved when all its *mandatory* and *effective* requirements are satisfied by capabilities from resolved resources. A Resolver must not return wires between resources that have unsatisfied mandatory requirements.

A *mandatory* requirement has a resolution directive that is not set or that is set to mandatory. The *effectiveness* of a requirement is defined by the Resolve Context, a Resolver service must call the isEffective(Requirement) method to establish if a requirement is effective for a given resolve operation. A Resolver must never create a wire from a requirement that is not effective.

To find a resolution, the Resolver must use the Resolve Context to find *candidate* capabilities for the requirements declared in the resources that it needs to resolve. A candidate capability is a capability that *satisfies* the requirement. From the perspective of the Resolver service, a requirement is satis-

fied by a capability when that capability is returned from the findProviders(Requirement) method. A Resolver has no alternative way to find out if a requirement is satisfied by a capability and must therefore not make any assumptions about matching. Any matching rules like for example the osgi.wiring.* mandatory directive must be implemented by the Resolve Context. A Resolve Context must always return the same capabilities during a resolving operation when given the same requirement.

Since the resolver cannot match requirements and capabilities the Resolve Context must return capabilities for all possible resources, this must include:

- The given mandatory resources R_m
- The given optional resources R_o
- The existing Wiring state X

It can include additional resources that were indirectly returned through the findProviders(Requirement) method.

The existing wiring X provides an existing set of constraints that the Resolver service must abide by. It can get this state with the getWirings() method. The purpose of the existing state is to ensure that any wires are compatible with existing wiring. For an OSGi framework it is crucial that the uses constraints are maintained for the resolution, see [1] *Framework Module Layer*.

The Resolver service can wire new requirements to existing capabilities but it can never create wires for an existing requirement from the existing wiring.

If the Resolver service attaches a hosted resource like a fragment, and thereby needs to add new HostedCapability objects to a previously returned list from findProviders(Requirement) then it must call the Resolve Context's insertHostedCapability(List,HostedCapability) method.

Fragments can be attached from resolved resources or from resources that declare the capabilities returned from findProviders(Requirement), that is, $C_{env.}$

This specification does not define the detailed rules and constraints associated with resolving; these rules are defined in their namespaces. An OSGi Resolver service must support at least all namespaces defined in [4] *Framework Namespace (Core)* except for the following directives:

- mandatory - Mandatory attributes on the osgi.wiring.* namespaces must be implemented by the Resolve Context. The Resolve Context should not return capabilities from findProviders(Requirement) unless the rules of the OSGi mandatory directive are followed.
- singleton - Singletons are not implemented by the Resolver, the Resolve Context must not return capabilities from findProviders(Requirement) from conflicting singleton resources.
- effective - The Resolve Context decides what requirements are effective in the isEffective(Requirement) method.

A Resolver service must support the uses constraints and any applicable rule defined in the [1] *Framework Module Layer* for the osgi.wiring.* namespaces.

The Resolver must return a delta wiring to the existing state (potentially empty) or throw an Exception. The resolution:

- Must contain all mandatory resources R_m as provided by getMandatoryResources().
- Must have all resources resolved.
- Must have no wired capabilities that are declared or hosted in resources that are not resolved.
- Should include optional resources R_o as provided by getOptionalResources().

136.3.3 Resolution Exception

If the Resolver cannot find a solution or it runs into problems then it must throw a Resolution Exception, which is a Runtime Exception.

The ResolutionException provides the getUnresolvedRequirements() method. If the resolution failed then it is possible that this was caused because it failed to find matches for certain requirements. The information in this method can be very helpful to find a solution that will work, however, there are a number of caveats.

Resolving is an NP-complete problem. For these problems there exists no algorithm that can infer a solution from the desired outcome. Therefore, the Resolver tries a potential solution and if that solution does not match the constraints it will backtrack and attempt another solution. An unavoidable aspect of such solutions is that it is impossible to pin-point a single failure point if the algorithm fails to find a solution, in general the algorithm gives up after having exhausted its search space. However, during its search it might have been very close to a solution, for example it only missed a single requirement, but its final failure missed many requirements.

The implication is that the reported missing requirements neither give a guarantee for a resolution when satisfied nor indicate that this is the smallest set of missing requirements.

Therefore, getUnresolvedRequirements() is intended for human consumption and not for automated solutions.

136.4 Security

136.4.1 Resolving

The Resolver service is a pure function that has no state. The Resolve Context maintains the state and is therefore the actor that requires most permissions. In general, it will require access to the Wiring API and Repositories.

Since the Resolver requires no external access it does not have to be a trusted service. Resolve Contexts that support security must ensure that the callbacks are executed in a privileged block.

136.4.2 Minimum Implementation Permissions

```
PackagePermission[org.osgi.service.resolver,IMPORT]
ServicePermission[...Resolver, REGISTER ]
```

136.4.3 Minimum Using Permissions

```
PackagePermission[org.osgi.service.repository,IMPORT]
PackagePermission[org.osgi.service.resolver,IMPORT]
PackagePermission[org.osgi.resource,IMPORT]
PackagePermission[org.osgi.framework.wiring,IMPORT]
PackagePermission[org.osgi.framework.namespaces,IMPORT]
ServicePermission[...Resolver, GET ]
... likely needs AdaptPermissions and ServicePermission[...Repository,GET]
```

136.5 org.osgi.service.resolver

Resolver Service Package Version 1.0.

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

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.resolver; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.resolver; version="[1.0,1.1)"

136.5.1 Summary

- HostedCapability - A capability hosted by a resource.
- ResolutionException - Indicates failure to resolve a set of requirements.
- ResolveContext - A resolve context provides resources, options and constraints to the potential solution of a resolve operation.
- Resolver - A resolver service resolves the specified resources in the context supplied by the caller.

136.5.2 public interface HostedCapability extends Capability

A capability hosted by a resource.

A HostedCapability is a Capability where the getResource() method returns a Resource that hosts this Capability instead of declaring it. This is necessary for cases where the declaring Resource of a Capability does not match the runtime state. For example, this is the case for fragments attached to a host. Most fragment declared capabilities and requirements become hosted by the host resource. Since a fragment can attach to multiple hosts, a single capability can actually be hosted multiple times.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

136.5.2.1 public Capability getDeclaredCapability()

□ Return the Capability hosted by the Resource.

Returns The Capability hosted by the Resource.

136.5.2.2 public Resource getResource()

□ Return the Resource that hosts this Capability.

Returns The Resource that hosts this Capability.

136.5.3 public class ResolutionException extends Exception

Indicates failure to resolve a set of requirements.

If a resolution failure is caused by a missing mandatory dependency a resolver may include any requirements it has considered in the resolution exception. Clients may access this set of dependencies via the getUnresolvedRequirements() method.

Resolver implementations may extend this class to provide extra state information about the reason for the resolution failure.

136.5.3.1 public ResolutionException(String message,Throwable cause,Collection<Requirement> unresolvedRequirements)

message The message.

cause The cause of this exception.

unresolve- The unresolved mandatory requirements from mandatory resources or null if no unresolved require-
dRequirements ments information is provided.

□ Create a ResolutionException with the specified message, cause and unresolved requirements.

136.5.3.2   **public ResolutionException(String message)**

message   The message.

   □ Create a ResolutionException with the specified message.

136.5.3.3   **public ResolutionException(Throwable cause)**

cause   The cause of this exception.

   □ Create a ResolutionException with the specified cause.

136.5.3.4   **public Collection<Requirement> getUnresolvedRequirements()**

   □ Return the unresolved requirements, if any, for this exception.

The unresolved requirements are provided for informational purposes and the specific set of unre-
solved requirements that are provided after a resolve failure is not defined.

Returns   A collection of the unresolved requirements for this exception. The returned collection may be emp-
ty if no unresolved requirements information is available.

136.5.4   public abstract class ResolveContext

A resolve context provides resources, options and constraints to the potential solution of a resolve
operation.

Resolve Contexts:

- Specify the mandatory and optional resources to resolve. The mandatory and optional resources
  must be consistent and correct. For example, they must not violate the singleton policy of the im-
  plementer.
- Provide capabilities that the Resolver can use to satisfy requirements via the
  findProviders(Requirement) method
- Constrain solutions via the getWirings() method. A wiring consists of a map of existing re-
  sources to wiring.
- Filter requirements that are part of a resolve operation via the isEffective(Requirement).

A resolver may call the methods on the resolve context any number of times during a resolve opera-
tion using any thread. Implementors should ensure that this class is properly thread safe.

Except for insertHostedCapability(List, HostedCapability), the resolve context methods must be
idempotent. This means that resources must have constant capabilities and requirements and the re-
solve context must return a consistent set of capabilities, wires and effective requirements.

Concurrency   Thread-safe

136.5.4.1   **public ResolveContext()**

136.5.4.2   **public abstract List<Capability> findProviders(Requirement requirement)**

requirement   The requirement that a resolver is attempting to satisfy. Must not be null.

   □ Find Capabilities that match the given Requirement.

The returned list contains Capability objects where the Resource must be the declared Re-
source of the Capability. The Resolver can then add additional HostedCapability objects with the
insertHostedCapability(List, HostedCapability) method when it, for example, attaches fragments.
Those HostedCapability objects will then use the host's Resource which likely differs from the de-
clared Resource of the corresponding Capability.

The returned list is in priority order such that the Capabilities with a lower index have a preference
over those with a higher index. The resolver must use the insertHostedCapability(List, HostedCapa-

bility) method to add additional Capabilities to maintain priority order. In general, this is necessary when the Resolver uses Capabilities declared in a Resource but that must originate from an attached host.

Each returned Capability must match the given Requirement. This means that the filter in the Requirement must match as well as any namespace specific directives. For example, the mandatory attributes for the osgi.wiring.package namespace.

Returns A list of Capability objects that match the specified requirement.

136.5.4.3   **public Collection<Resource> getMandatoryResources()**

☐ Return the resources that must be resolved for this resolve context.

The default implementation returns an empty collection.

Returns A collection of the resources that must be resolved for this resolve context. May be empty if there are no mandatory resources. The returned collection may be unmodifiable.

136.5.4.4   **public Collection<Resource> getOptionalResources()**

☐ Return the resources that the resolver should attempt to resolve for this resolve context. Inability to resolve one of the specified resources will not result in a resolution exception.

The default implementation returns an empty collection.

Returns A collection of the resources that the resolver should attempt to resolve for this resolve context. May be empty if there are no optional resources. The returned collection may be unmodifiable.

136.5.4.5   **public abstract Map<Resource,Wiring> getWirings()**

☐ Returns the wirings for existing resolved resources.

For example, if this resolve context is for an OSGi framework, then the result would contain all the currently resolved bundles with each bundle's current wiring.

Multiple calls to this method for this resolve context must return the same result.

Returns The wirings for existing resolved resources. The returned map is unmodifiable.

136.5.4.6   **public abstract int insertHostedCapability(List<Capability> capabilities,HostedCapability hostedCapability)**

capabilities The list returned from findProviders(Requirement). Must not be null.

hostedCapability The HostedCapability to insert in the specified list. Must not be null.

☐ Add a HostedCapability to the list of capabilities returned from findProviders(Requirement).

This method is used by the Resolver to add Capabilities that are hosted by another Resource to the list of Capabilities returned from findProviders(Requirement). This function is necessary to allow fragments to attach to hosts, thereby changing the origin of a Capability. This method must insert the specified HostedCapability in a place that makes the list maintain the preference order. It must return the index in the list of the inserted HostedCapability.

Returns The index in the list of the inserted HostedCapability.

136.5.4.7   **public abstract boolean isEffective(Requirement requirement)**

requirement The Requirement to test. Must not be null.

☐ Test if a given requirement should be wired in the resolve operation. If this method returns false, then the resolver should ignore this requirement during the resolve operation.

The primary use case for this is to test the effective directive on the requirement, though implementations are free to use any effective test.

Returns true if the requirement should be considered as part of the resolve operation.

136.5.5 public interface Resolver

A resolver service resolves the specified resources in the context supplied by the caller.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

136.5.5.1 public Map<Resource,List<Wire>> resolve(ResolveContext context) throws ResolutionException

context The resolve context for the resolve operation. Must not be null.

☐ Resolve the specified resolve context and return any new resources and wires to the caller.

The resolver considers two groups of resources:

- Mandatory - any resource in the mandatory group must be resolved. A failure to satisfy any mandatory requirement for these resources will result in throwing a ResolutionException
- Optional - any resource in the optional group may be resolved. A failure to satisfy a mandatory requirement for a resource in this group will not fail the overall resolution but no resources or wires will be returned for that resource.

The resolve method returns the delta between the start state defined by ResolveContext.getWirings() and the end resolved state. That is, only new resources and wires are included.

The behavior of the resolver is not defined if the specified resolve context supplies inconsistent information.

Returns The new resources and wires required to satisfy the specified resolve context. The returned map is the property of the caller and can be modified by the caller.

Throws ResolutionException– If the resolution cannot be satisfied.

136.6   References

[1]   *Framework Module Layer*
       OSGi Core, Chapter 3 Module Layer

[2]   *Framework Wiring API (Core)*
       OSGi Core, Chapter 7 Bundle Wiring API Specification

[3]   *Framework Resource API (Core)*
       OSGi Core, Chapter 6 Resource API Specification

[4]   *Framework Namespace (Core)*
       OSGi Core, Chapter 8 Framework Namespaces Specification

137 REST Management Service Specification

Version 1.0

137.1 Introduction

Cloud computing is a continuing trend in the IT industry. Due to its service model which embraces dynamism as opposed to masking it, OSGi appears to be an ideal base for building scalable and dependable applications for the cloud where changes in the deployment, network topology, and service availability are the norm rather than the exception. One of the possible scenarios for OSGi to be successfully applied to cloud computing is using it in a Platform as a Service (PaaS) spirit. Users write their bundles and can deploy them to a provided OSGi instance running in the cloud. This, however, requires the platform provider to expose the OSGi management API to the end user and make them available through a network protocol. One of the popular approaches in cloud computing to remote communication is the use of RESTful web services.

Representational State Transfer (REST) is the architectural style of the world wide web. It can be described as a set of constraints that govern the interactions between the main components of the Internet. Recently, REST style interaction has gained popularity as a architecture for web services (RESTful web services), mainly to overcome the perceived complexity and verbosity of SOAP-based web services. This specification describes a REST interface for framework management, client-side Java and JavaScript APIs, and an extension mechanism through which other bundles can contribute their own RESTful management APIs and make them discoverable by clients.

137.1.1 Essentials

- *Client-Server* - A separation of concern between the entity responsible for the user-interaction (client) and the other entity (server) responsible for data storage. For instance, in the original world wide web the browser is the client rendering and presenting the content delivered by one or more web servers. As a result, web content becomes more portable and content providers more scalable.
- *Stateless* - State is entirely kept at the client side. Therefore, every request must contain all state required for the server to accomplish the transaction and deliver content. The main rationale behind this design constraint is to again improve the scalability since in a pure stateless design the server resources are not burdened with maintaining any client state. Another perceived advantage is that the failure models of stateless interactions is simpler and fault tolerance easier to achieve.
- *Cacheable* - Content marked as cacheable can be temporarily stored and used to immediately answer future equivalent requests and improve efficiency and reduce network utilization and access latencies. Due to the end-to-end principle, caches can be placed where necessary, e.g., at the client (forward-proxy), at the server side (backward-proxy), or somewhere in-between for example in a content delivery network. Content marked as non-cacheable must be freshly retrieved with every request even in the presence of caches.
- *Layered* - Layering introduces natural boundaries to coupling since every layer only accesses the services provided by the lower layer and provides services to the next higher layer.

- *Uniform Interface* - Generality of component interfaces provides a natural decoupling of implementation and interface. REST furthermore encourages the separation of identifiable resources (addressing) and their representation (content delivery).

137.1.2   Entities

- *Resource* - A resource is an abstract piece of information that can be addressed by a resource identifier. The mapping of a resource to a concrete set of entities can vary over time.
- *Representation* - A representation is a sequence of bytes plus associated meta-data that describe the state of a resource. The data format of a representation is called the media-type. Every concrete representation of a resource is just one of arbitrarily many possible representations. The selection of a concrete representation of a resource can be made according to the media types supported by both the client and the server.
- *REST Management Service* - The management service exposes a REST API for remotely managing an OSGi framework through the network in a lightweight and portable fashion.
- *Client* - The client is a machine using the management service by issuing REST requests through the network. It can do so either directly or indirectly, i.e., through client-side libraries using the REST calls internally.

137.1.3   Synopsis

The manageable entities of an OSGi framework are mapped to resources accessible through resource identifiers. These identifiers are relative to the (usually externally accessible) root URL of the management service. Clients can either discover this root URL or receive it through configuration. Subsequently, a client is able to introspect the state of the framework and perform management operations.

The internal state of a framework resource is expressed and transmitted as a representation. The format of the representation is subject to a mutual agreement between client and management service regarding media types commonly supported by both endpoints. This specification describes two representation formats: JSON and XML.

137.2   Interacting with the REST Management Service

The REST Management Service is not a traditional OSGi service and it does not appear in the service registry. Its purpose is to expose a management interface to clients which can perform operations on the framework through a network connection. Therefore, it is ideally suited for situations where the user of an OSGi framework does not have direct access to the machine it is running on, a typical situation in Infrastructure as a Service (IaaS) or Platform as a Service (PaaS). However, even in other domains having a lightweight and easily accessible management solution can be of benefit, e.g., for embedded devices. The advantage of REST is that it uses HTTP and therefore does usually not interfere with firewalls. Furthermore, the REST format is easily embeddable into client-side scripting technologies like JavaScript and can be consumed in web browsers.

Much of the value of the REST Management Service lies in client-side libraries which can use the REST protocol and interact with the OSGi framework through the Management Service. Therefore, this specification contains API for two clients, a Java Client API and a JavaScript Client API.

137.2.1   Resource Identifier Overview

The REST Management Service comprises of a set of resources that can be retrieved and in some cases also modified through REST requests. These resources need to be made available under well-defined paths so that clients can interact with them. As the initial entry point a client receives a URL to the REST Management Service. This can be done, e.g., as part of the creation of a cloud-based OSGi

framework, and the precise mechanism would be proprietary to the cloud platform used. Relative to this URL the client can access the resources through the following resource identifiers:

```
framework
framework/state
framework/startlevel
framework/bundles

framework/bundles/representations

framework/bundle/{bundleid}
framework/bundle/{bundleid}/state
framework/bundle/{bundleid}/startlevel
framework/bundle/{bundleid}/header
framework/services

framework/services/representations

framework/service/{serviceid}
```

`framework/bundle/0/state` is an alias for `framework/state`

Extensions to the REST Management Service can be discovered by visiting the Extensions Resource at:

```
extensions
```

For more details on the extension mechanism see *Extending the REST Management Service* on page 723

137.2.2 Filtering Results

The `bundles`, `bundles/representations`, `services`, and `services/representations` resources allow the use of a query parameter which specifies a filter to restrict the result set. The filter expression follows the Core Specifications *Framework Filter Syntax*; see [1] *Framework Filter Syntax*.

Filters on services are matched against the service attributes. The query parameter is of the form:

`framework/services?filter=ldap-filter`

Filters on bundles are matched against the attributes of capabilities in the respective namespaces. Filters on bundles have the form:

`framework/bundles?namespace1=ldap-filter1&namespace2=ldap-filter2&...`

If multiple capabilities for a given namespace are present, then a filter succeeds when one of these capabilities matches. When multiple filter expressions across namespaces are given, these are combined with the *and* operator.

137.2.3 Content Type Matching

Resources can present themselves through different representation variants. An implementation of this specification must support at least the JSON representation and the XML representation of resources. Clients can support a subset of representations. Matching the clients capabilities to understand certain representation formats with the servers supported formats follows the typical HTTP pattern of content negotiation and requires the client to set corresponding HTTP Accept headers for supported formats in the form of their media types. This specification describes the format and media types for representations in JSON and XML format in *Representations* on page 717.

Implementations of the REST Management Service offering different variants of representations must return the best matching variant based on the HTTP accept header. In addition, they must re-

spect the file extensions defined for the different media types as specified in the respective IETF RFC (e.g., ".xml" as specified in IETF RFC 3032 and ".json" as specified in IETF RFC 4627). If a file extension is appended to the resource, an implementation must return the variant mandated by the file extension provided that it supports this content type.

137.3 Resources

The framework and its state is mapped to a set of different resources. Each resource is accessible through a resource identifier, as summarized in *Resource Identifier Overview* on page 710.

137.3.1 Framework Startlevel Resource

framework/startlevel

The startlevel resource represents the active start level of the framework. It supports the GET and PUT requests.

137.3.1.1 GET

The GET request retrieves a *Framework Startlevel Representation* from the REST management service. The request can return the following status codes:

- 200 (OK): the request has been served successfully and the body of the response is a startlevel representation.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.3.1.2 PUT

The PUT request sets the target framework startlevel. The body of the request needs to be a *Framework Startlevel Representation*. The request can return the following status codes:

- 204 (NO CONTENT): the request was received and valid. The framework will asynchronously start to adjust the framework startlevel until the target startlevel has been reached.
- 415 (UNSUPPORTED MEDIA TYPE): the request had a media type that is not supported by the REST management service.
- 400 (BAD REQUEST): the REST management service received an IllegalArgumentException when trying to adjust the framework startlevel, e.g., because the requested startlevel was zero or negative.

137.3.2 Bundles Resource

framework/bundles

The bundles resource represents the list of all bundles installed on the managed framework. It supports the GET request and two syntactically different forms of POST requests which are used to install new bundles to the framework.

Results for this resource can be filtered as described in *Filtering Results* on page 711.

137.3.2.1 GET

The GET request retrieves a *Bundle List Representation* from the REST management service. The request can return the following status codes:

- 200 (OK): the request has been served successfully and the body of the response is a bundle list representation.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.3.2.2   **POST with Location String**

The POST request installs a new bundle to the managed framework and thereby logically appends it to the bundles resource. The new bundle to be installed is referenced by a location string which is passed as the body of the request. In order to disambiguate the request from the other form of POST, the content type must be set to text/plain. In practice, the location string is usually a URL. Since the framework will use the location retrieving the physical bundle, it needs to be accessible from the remotely managed framework and not necessarily from the managing client.

The management service implementation must check if the result of the install request matches the requested bundle since the OSGi framework will return an existing bundle object as the return value of an install call if there was already one with the same location string installed. One way of doing it is comparing the last modification timestamp. A detected collision is indicated to the requesting clients through an error code 409.

The body of the response is *Bundle Representation* of the newly installed bundle. The following status codes can be returned:

- 200 (OK): the bundle has been successfully installed and the body of the response contains the URI.
- 400 (BAD REQUEST): the REST management service received a BundleException when trying to install. The body of the message is a *Bundle Exception Representation* describing the reason why the installation did not succeed.
- 409 (CONFLICT): there is already a bundle installed with the same location string.

137.3.2.3   **POST with Bundle**

This variant of the POST request uploads the bundle as the body of the request. The media type of the request should be set to application/vnd.osgi.bundle which must be supported by all REST management services. Implementations are free to accept other media types for this request with the exception of text/plain. For instance, they can opt to additionally support application/zip or application/x-jar.

Clients should use the HTTP Content-Location field to set a bundle location. If no content location is given, REST management service implementations must generate a unique location string in order to avoid unintended collisions between unrelated bundles.

The body of the response is *Bundle Representation* of the newly installed bundle. The following status codes can be returned:

- 200 (OK): the bundle has been successfully installed and the body of the response contains the URI.
- 400 (BAD REQUEST): the REST management service received a BundleException when trying to install. The body of the message is a *Bundle Exception Representation* describing the reason why the installation did not succeed.
- 409 (CONFLICT): there is already a bundle installed with the same location string.

137.3.3   Bundles Representations Resource

framework/bundles/representations

137.3.3.1   **GET of the Representations**

The bundles resource returns a list of the URIs of all bundles installed on the framework. For clients interested in the details of multiple bundles there is also the possibility to retrieve the bundle representation of each installed bundle with a single request through the *bundles/representations* resource.

The body of the response is a *Bundle Representations List Representation*. The request can return the following status codes:

Results for this resource can be filtered as described in *Filtering Results* on page 711.

- 200 (OK): the request has been served successfully and the body of the response is a bundle list representation.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.3.4　Bundle Resource

framework/bundle/{bundleid}

The bundle resource represents a single, distinct bundle in the system. Hence, it has to be qualified by a bundle id. The resource supports the GET, two variants of PUT, and the DELETE requests.

137.3.4.1　GET

The GET request retrieves a *Bundle Representation* from the REST management service. The request can return the following status codes:

- 200 (OK): the request has been served successfully and the body of the response is a bundle representation.
- 404 (NOT FOUND): there is not bundle with the given bundle id.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.3.4.2　PUT with Location String

The PUT request updates the bundle with a new version, referenced by a location string which is passed as the body of the request. In order to disambiguate the request from the other form of PUT, the content type must be set to text/plain. The same rationale applies as for *POST with Location String* and *POST with Bundle* on page 713, if a location string is given it must point to a location reachable by the managed framework. If no location string is passed as the body of the request, the framework will perform an update based on the existing bundle's location string.

The body of the response is *Bundle Representation* of the updated bundle. The following status codes can be returned:

- 204 (NO CONTENT): the request was received and valid and the framework has issued the update.
- 400 (BAD REQUEST): the REST management service received a BundleException when trying to update. The body of the message is a *Bundle Exception Representation* describing the reason why the update did not succeed.
- 404 (NOT FOUND): there is not bundle with the given bundle id.

137.3.4.3　PUT with Bundle

The PUT request updates the bundle with a new version, uploaded as the body of the request. The media type of the request should be set to application/vnd.osgi.bundle which must be supported by all REST management services. Implementations are free to accept other media types for this request with the exception of text/plain. For instance, they can opt to additionally support application/zip or application/x-jar.

The body of the response is *Bundle Representation* of the updated bundle. The following status codes can be returned:

- 204 (NO CONTENT): the request was received and valid and the framework has issued the update.
- 400 (BAD REQUEST): the REST management service received a BundleException when trying to update. The body of the message is a *Bundle Exception Representation* describing the reason why the update did not succeed.
- 404 (NOT FOUND): there is not bundle with the given bundle id.

137.3.4.4   **DELETE**

The DELETE request uninstalls the bundle from the framework.

The body of the response is *Bundle Representation* of the uninstalled bundle, where the bundle state will be UNINSTALLED. The following status codes can be returned:

- 204 (NO CONTENT): the request was received and valid and the framework has uninstalled the bundle.
- 400 (BAD REQUEST): the REST management service received a BundleException when trying to uninstall. The body of the message is a *Bundle Exception Representation* describing the reason why the uninstallation did not succeed.
- 404 (NOT FOUND): there is not bundle with the given bundle id.

137.3.5   Bundle State Resource

framework/bundle/{bundleid}/state

The bundle state resource represents the internal state of an installed bundle qualified through its bundle id. It supports the GET and PUT requests.

137.3.5.1   **GET**

The GET request retrieves a *Bundle State Representation* from the REST management service. The request can return the following status codes:

- 200 (OK): the request has been served successfully and the body of the response is a bundle state representation.
- 404 (NOT FOUND): there is not bundle with the given bundle id.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.3.5.2   **PUT**

The PUT request sets the target state for the given bundle. This can, e.g., be state=32 for transitioning the bundle to started, or state=4 for stopping the bundle and transitioning it to resolved. The body of the request needs to be a *Bundle State Representation*. Not all state transitions are valid. The body of the response is the new *Bundle State Representation*. The request can return the following status codes:

- 200 (OK): the request was received and valid. The framework has performed a state change and the new bundle state is contained in the body.
- 400 (BAD REQUEST): the REST management service received a BundleException when trying to perform the state transition. The body of the message is a *Bundle Exception Representation* describing the reason why the operation did not succeed.
- 402 (PRECONDITION FAILED): the requested target state is not reachable from the current bundle state or is not a target state. An example such state is the STOPPING state.
- 404 (NOT FOUND): there is not bundle with the given bundle id.
- 415 (UNSUPPORTED MEDIA TYPE): the request had a media type that is not supported by the REST management service.

137.3.6   Bundle Header Resource

framework/bundle/{bundleid}/header

The bundle header resource represents manifest header of a bundle which is qualified by its bundle id. It can only be read through a GET request.

137.3.6.1   **GET**

The GET request retrieves a *Bundle Header Representation* from the REST management service. The raw header value is used unless an `Accept-Language` header is set on the HTTP request. If multiple accepted languages are set only the first is used to localize the header. The request can return the following status codes:

- 200 (OK): the request has been served successfully and the body of the response is a bundle header representation.
- 404 (NOT FOUND): there is not bundle with the given bundle id.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.3.7   Bundle Startlevel Resource

framework/bundle/{bundleid}/startlevel

The bundle startlevel resource represents the start level of the bundle qualified by its bundle id. It supports the GET and PUT requests.

137.3.7.1   **GET**

The GET request retrieves a *Bundle Startlevel Representation* from the REST management service. The request can return the following status codes:

- 200 (OK): the request has been served successfully and the body of the response is a bundle startlevel representation.
- 404 (NOT FOUND): there is not bundle with the given bundle id.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.3.7.2   **PUT**

The PUT request sets the target bundle startlevel. The body of the request needs to be a *Bundle Startlevel Representation*, however only the `startLevel` property is used. The request can return the following status codes:

- 200 (OK): the request was received and valid. The REST management service has changed the bundle startlevel according to the target value. The body of the response is the new bundle startlevel representation.
- 400 (BAD REQUEST): either the target startlevel state involved invalid values, e.g., a startlevel smaller or equal to zero and the REST management service got an IllegalArgumentException, or the REST management service received a BundleException when trying to perform the startlevel change. In the latter case, the body of the message is a *Bundle Exception Representation* describing the reason why the operation did not succeed.
- 404 (NOT FOUND): there is not bundle with the given bundle id.
- 415 (UNSUPPORTED MEDIA TYPE): the request had a media type that is not supported by the REST management service.

137.3.8   Services Resource

framework/services

The services resource represents the set of all services available on the framework, optionally constrained by a filter expression. It is read-only and therefore only supports the GET request.

Results for this resource can be filtered as described in *Filtering Results* on page 711.

137.3.8.1   **GET**

The GET request retrieves a *Service List Representation* from the REST management service. The request can return the following status codes:

- 200 (OK): the request has been served successfully and the body of the response is a service list representation.
- 400 (BAD REQUEST): the provided filter expression was not valid.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.3.9   Services Representations Resource

framework/services/representations

137.3.9.1   **GET of the Representations**

The services resource returns a list of the URIs of all services registered on the framework. For clients interested in the details of multiple services there is also the possibility to retrieve the service representation of each available service with a single request through the *services/representations* resource. The body of the response is a *Service Representations List Representation* from the REST management service. The request can return the following status codes:

Results for this resource can be filtered as described in *Filtering Results* on page 711.

- 200 (OK): the request has been served successfully and the body of the response is a service list representation.
- 400 (BAD REQUEST): the provided filter expression was not valid.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.3.10   Service Resource

framework/service/{serviceid}

The service resource represents a single, distinct service in the framework. Hence, it has to be qualified by a service id. Services can only be read through the REST Management Service and therefore only support the GET request.

137.3.10.1   **GET**

The GET request retrieves a *Service Representation* . The request can return the following status codes:

- 200 (OK): the request has been served successfully and the body of the response is a service representation.
- 404 (NOT FOUND): there is not service with the given service id.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.4   Representations

137.4.1   Bundle Representation

137.4.1.1   **JSON**

Content-Type: application/org.osgi.bundle+json

```
{
```

```
    "id":0,
    "lastModified":1314999275542,
    "state":32,
    "symbolicName":"org.eclipse.osgi",
    "version":"3.7.0.v20110613"
}
```

137.4.1.2   **XML**

Content-Type: application/org.osgi.bundle+xml

```
<bundle>
    <id>0</id>
    <lastModified>1314999275542</lastModified>
    <state>32</state>
    <symbolicName>org.eclipse.osgi</symbolicName>
    <version>3.7.0.v20110613</version>
</bundle>
```

137.4.2   Bundles Representations

137.4.2.1   **Bundle List Representation**

137.4.2.1.1   *JSON*

Content-Type: application/org.osgi.bundles+json

```
{
    [bundleURI, bundleURI, ..., bundleURI]
}
```

137.4.2.1.2   *XML*

Content-Type: application/org.osgi.bundles+xml

```
<bundles>
    <uri>bundleURI</uri>
    <uri>bundleURI</uri>
    ...
    <uri>bundleURI</uri>
</bundles>
```

137.4.2.2   **Bundle Representations List Representation**

137.4.2.2.1   *JSON*

Content-Type: application/org.osgi.bundles.representations+json

```
{
    [BUNDLE REPRESENTATION, BUNDLE REPRESENTATION, ..., BUNDLE REPRESENTATION]
}
```

137.4.2.2.2   *XML*

Content-Type: application/org.osgi.bundles.representations+xml

```
<bundles>
    BUNDLE REPRESENTATION
    BUNDLE REPRESENTATION
    ...
    BUNDLE REPRESENTATION
```

```
</bundles>
```

137.4.3   Bundle State Representation

137.4.3.1   JSON

Content-Type: application/org.osgi.bundlestate+json

```
{
    "state":32
    "options":1
}
```

The options are used in start or stop calls. Valid options include, e.g., Bundle.START_TRANSIENT and Bundle.START_ACTIVATION_POLICY.

137.4.3.2   XML

Content-Type: application/org.osgi.bundlestate+xml

```
<bundleState>
    <state>32</state>
    <options>1</options>
</bundleState>
```

137.4.4   Bundle Header Representation

137.4.4.1   JSON

Content-Type: application/org.osgi.bundleheader+json

```
{
    key: value,
    key: value,
    ...
    key: value
}
```

137.4.4.2   XML

Content-Type: application/org.osgi.bundleheader+xml

```
<bundleHeader>
    <entry key="key" value="value"/>
    <entry key="key" value="value"/>
    ...
    <entry key="key" value="value"/>
<bundleHeader>
```

137.4.5   Framework Startlevel Representation

137.4.5.1   JSON

Content-Type: application/org.osgi.frameworkstartlevel+json

```
{
    "startLevel":6,
    "initialBundleStartLevel":4
}
```

137.4.5.2   **XML**

Content-Type: application/org.osgi.frameworkstartlevel+xml

```
<frameworkStartLevel>
    <startLevel>6</startLevel>
    <initialBundleStartLevel>4</initialBundleStartLevel>
</frameworkStartLevel>
```

137.4.6   Bundle Startlevel Representation

137.4.6.1   **JSON**

Content-Type: application/org.osgi.bundlestartlevel+json

```
{
    "startLevel":6

    "activationPolicyUsed":true
    "persistentlyStarted":false
}
```

137.4.6.2   **XML**

Content-Type: application/org.osgi.bundlestartlevel+xml

```
<bundleStartLevel>
    <startLevel>6</startLevel>

    <activationPolicyUsed>true</activationPolicyUsed>
    <persistentlyStarted>false</persistentlyStarted>
</bundleStartLevel>
```

137.4.7   Service Representation

137.4.7.1   **JSON**

Content-Type: application/org.osgi.service+json

```
{
    "id":10,
    "properties":
    {
        "prop1":"val1",
        "prop2":2.82,
        ...
        "prop3":true
    },
    "bundle":bundleURI,
    "usingBundles": [bundleURI, bundleURI, ... bundleURI]
}
```

Note: service properties are converted to JSON-supported data types where possible: "string", number or boolean (true|false). If there is no conversion to JSON data types is possible the toString() result is used as a string value.

137.4.7.2   **XML**

Content-Type: application/org.osgi.service+xml

```
<service>
```

```
        <id>10</id>
        <properties>
           <property name="prop1" value="val1"/>
           <property name="prop2" type="Float" value="2.82"/>
           ...
           <property name="prop3" type="Boolean" value="true"/>
        </properties>
        <bundle>bundleURI</bundle>
        <usingBundles>
           <bundle>bundleURI</bundle>
           <bundle>bundleURI</bundle>
           ...
           <bundle>bundleURI</bundle>
        </usingBundles>
     </service>
```

Note: service properties are represented using the same method as used for the property XML element in the Declarative Services specification, see *Property and Properties Elements* on page 195. Service properties that cannot be represented using the supported data types, will be represented as String values obtained via the toString() method.

137.4.8   Services Representations

137.4.8.1   Service List Representation

137.4.8.1.1   *JSON*

Content-Type: application/org.osgi.services+json

```
{
    [serviceURI, serviceURI, ..., serviceURI]
}
```

137.4.8.1.2   *XML*

Content-Type: application/org.osgi.services+xml

```
<services>
    <uri>serviceURI</uri>
    <uri>serviceURI</uri>
    ...
    <uri>serviceURI</uri>
</services>
```

137.4.8.2   Service Representations List Representation

137.4.8.2.1   *JSON*

Content-Type: org.osgi.services.representations+json

```
{
    [SERVICE REPRESENTATION, SERVICE REPRESENTATION, ..., SERVICE REPRESENTATION]
}
```

137.4.8.2.2   *XML*

Content-Type: application/org.osgi.services.representations+xml

```
<services>
    SERVICE REPRESENTATION
```

```
        SERVICE REPRESENTATION
        ...
        SERVICE REPRESENTATION
    </services>
```

137.4.9   Bundle Exception Representation

137.4.9.1   JSON

Content-Type: application/org.osgi.bundleexception+json

```
{
    "typecode": 5,
    "message": "BundleException: Bundle activation error"
}
```

137.4.9.2   XML

Content-Type: application/org.osgi.bundleexception+xml

```
<bundleexception>
    <typecode>5</typecode>
    <message>BundleException: Bundle activation error</message>
</bundleexception>
```

137.5   Clients

The REST service can be used by a variety of clients directly. In addition this specification describes Client APIs built over this REST protocol to facilitate use from Java and JavaScript clients.

137.5.1   Java Client

The Java Client provides a Java API over the REST API providing a convenient and portable way to use this API from a Java application.

To use the Java Client, obtain the RestClientFactory service. Create a client by providing the root URL of the REST service, for example:

```
RestClientFactory restClientFactory = ... // from Service Registry
RestClient restClient = restClientFactory.createRestClient(
    new URI("http://localhost:8080/restendpoint"));

// Now we can start interacting
Collection<String> bundles = restClient.getBundlePaths();
BundleDTO newBundle = restClient.installBundle(bundleLocation, bundleStream);
restClient.startBundle(newBundle.id);
```

The more details on the Java Client can be found in the org.osgi.service.rest.client API documentation section.

137.5.2   JavaScript Client

This specification also describes a JavaScript client to the REST Management service. This client makes it easy to manage an OSGi framework from any JavaScript environment, including Web Browsers.

The JavaScript client follows the *promises* programming style; the request is made asynchronously and a success() or failure() callback is made when the response arrives.

To use the JavaScript client create an instance of `OSGiRestClient` providing the root URL of the REST service.

```
var client = new OSGiRestClient('http://localhost:8080/restendpoint');
client.installBundle({
  success : function(res) {
    // Start the bundle once the install has finished
    client.startBundle(res.id);
  },
  failure : function(httpCode, res) {
    // handle failure
  }
});
```

More details on the JavaScript Client can be found in the JavaScript Client API API documentation section.

137.6   Extending the REST Management Service

This specification describes a REST-based management interface for Core Framework functionality. Other services in the framework might also benefit from management access through REST. This can involve services specified by the OSGi Alliance as part of the Core Framework, Compendium, or Enterprise Specifications but also application-specific functionality provided by the developer. It is desirable to expose such management services as extensions of the REST Management Service.

This REST service can be implemented by using various technologies such as Java Servlets, Restlet, JAX-RS, and others. Therefore, it might not always be possible to integrate extensions at the implementation level because they might use other underlying technologies to implement their REST interface. Defining a format for delegating requests between the REST Management Service and extensions would furthermore necessarily expose implementation details and is therefore not feasible either. As a consequence, this specification only describes how to logically integrate extensions with the REST Management Service. Implementations of this specification might offer mechanisms for tighter integration for the case that extensions are developed using the same underlying technology.

The main purpose of the extension mechanism is to advertise extensions to the core REST implementation, which makes them discoverable for clients. This mechanism can be used to check if a REST interface exists for a specific service. This is done through the *Extensions Resource* which contains a description and a path for every extension currently available. Implementations that want to contribute their extensions to the REST Management Service can do so by registering the RestApiExtension service using the [4] *Whiteboard Pattern*. The extension interface is only a marker and the relevant information is exposed through the NAME, URI_PATH and optionally SERVICE properties. Note that it is the responsibility of the extension to ensure that the endpoint announced via the `RestApiExtension` service is actually present. The Whiteboard service does not realize the extension endpoint; it purely announces it to the main REST implementation for inclusion in the Extensions Resource.

In order to be discoverable REST interface extensions to OSGi Core, Compendium, or Enterprise services must use their canonical package name as advertised name. E.g., the name of the REST interface for the User Admin must be `org.osgi.service.useradmin`. This way, a client is able to check if there is a given extension available on a host. User-defined extensions should use the package name of the service they provide management capabilities for.

137.6.1   Extensions Resource

extensions

The extensions resource enumerates all extensions currently registered through the Whiteboard Pattern. It is read-only and therefore only supports the GET request.

137.6.1.1 GET

The GET request retrieves a *Extensions Representation*. The request can return the following status codes:

- 200 (OK): the request has been served successfully and the body of the response is a extension list representation.
- 406 (NOT ACCEPTABLE): the REST management service does not support any of the requested representations.

137.6.2 Extensions Representation

137.6.2.1 JSON

Content-Type: application/org.osgi.extensions+json

```
{
    [ { "name" : "org.osgi.service.event", "path" : "contributions/eventadmin",
        "service" : 12 }, ... ]
}
```

137.6.2.2 XML

Content-Type: application/org.osgi.extensions+xml

```
<extensions>
    <extension>
        <name>org.osgi.service.event</name>
        <path>contributions/eventadmin</path>
        <service>12</service>
    </extension>
</extensions>
```

137.7 XML Schema

The namespace for XML representations is:

http://www.osgi.org/xmlns/rest/v1.0.0

The recommended prefix for this namespace is rest.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:rest="http://www.osgi.org/xmlns/rest/v1.0.0"
    targetNamespace="http://www.osgi.org/xmlns/rest/v1.0.0"
    elementFormDefault="unqualified"
    attributeFormDefault="unqualified"
    version="1.0.0">

    <annotation>
        <documentation xml:lang="en">
            This is the XML Schema for
            XML representations used by
            the REST Management Service
            Specification.
        </documentation>
    </annotation>

    <element name="bundle" type="rest:Tbundle">
        <annotation>
            <documentation xml:lang="en">
```

```
                    Representation for the
                    application/org.osgi.bundle+xml content type.
                </documentation>
            </annotation>
        </element>
        <complexType name="Tbundle">
            <all>
                <element name="id" type="long" />
                <element name="lastModified" type="long" />
                <element name="state" type="integer" />
                <element name="symbolicName" type="string" />
                <element name="version" type="string" />
            </all>
        </complexType>

        <element name="bundles" type="rest:Tbundles">
            <annotation>
                <documentation xml:lang="en">
                    Representation for the
                    application/org.osgi.bundles+xml and
                    application/org.osgi.bundles.representations+xml content
                    types.
                </documentation>
            </annotation>
        </element>
        <complexType name="Tbundles">
            <choice>
                <element name="uri" type="string" minOccurs="0"
                    maxOccurs="unbounded">
                    <annotation>
                        <documentation xml:lang="en">
                            Representation for the
                            application/org.osgi.bundles+xml content type.
                        </documentation>
                    </annotation>
                </element>
                <element name="bundle" type="rest:Tbundle"
                    minOccurs="0" maxOccurs="unbounded">
                    <annotation>
                        <documentation xml:lang="en">
                            Representation for the
                            application/org.osgi.bundles.representations+xml
                            content type.
                        </documentation>
                    </annotation>
                </element>
            </choice>
        </complexType>

        <element name="bundleState" type="rest:TbundleState">
            <annotation>
                <documentation xml:lang="en">
                    Representation for the
                    application/org.osgi.bundlestate+xml content type.
                </documentation>
            </annotation>
        </element>
        <complexType name="TbundleState">
            <all>
                <element name="state" type="integer" />
                <element name="options" type="integer" />
            </all>
        </complexType>

        <element name="bundleHeader" type="rest:TbundleHeader">
            <annotation>
                <documentation xml:lang="en">
                    Representation for the
                    application/org.osgi.bundleheader+xml content type.
                </documentation>
            </annotation>
        </element>
        <complexType name="TbundleHeader">
            <sequence>
                <element name="entry" minOccurs="0" maxOccurs="unbounded">
```

```
            <complexType>
                <attribute name="key" type="string" use="required" />
                <attribute name="value" type="string" use="required" />
            </complexType>
        </element>
    </sequence>
</complexType>

<element name="frameworkStartLevel" type="rest:TframeworkStartLevel">
    <annotation>
        <documentation xml:lang="en">
            Representation for the
            application/org.osgi.frameworkstartlevel+xml content
            type.
        </documentation>
    </annotation>
</element>
<complexType name="TframeworkStartLevel">
    <all>
        <element name="startLevel" type="integer" />
        <element name="initialBundleStartLevel" type="integer" />
    </all>
</complexType>

<element name="bundleStartLevel" type="rest:TbundleStartLevel">
    <annotation>
        <documentation xml:lang="en">
            Representation for the
            application/org.osgi.bundlestartlevel+xml content type.
        </documentation>
    </annotation>
</element>
<complexType name="TbundleStartLevel">
    <all>
        <element name="startLevel" type="integer" />
        <element name="activationPolicyUsed" type="boolean" />
        <element name="persistentlyStarted" type="boolean" />
    </all>
</complexType>

<element name="service" type="rest:Tservice">
    <annotation>
        <documentation xml:lang="en">
            Representation for the
            application/org.osgi.service+xml content type.
        </documentation>
    </annotation>
</element>
<complexType name="Tservice">
    <all>
        <element name="id" type="long" />
        <element name="properties">
            <complexType>
                <sequence>
                    <element name="property" minOccurs="0"
                        maxOccurs="unbounded">
                        <complexType>
                            <simpleContent>
                                <extension base="string">
                                    <attribute name="name"
                                        type="string" use="required" />
                                    <attribute name="value"
                                        type="string" use="optional" />
                                    <attribute name="type"
                                        default="String" use="optional">
                                        <simpleType>
                                            <restriction
                                                base="string">
                                                <enumeration
                                                    value="String" />
                                                <enumeration
                                                    value="Long" />
                                                <enumeration
                                                    value="Double" />
                                                <enumeration
```

```
                                                       value="Float" />
                                               <enumeration
                                                   value="Integer" />
                                               <enumeration
                                                   value="Byte" />
                                               <enumeration
                                                   value="Character" />
                                               <enumeration
                                                   value="Boolean" />
                                               <enumeration
                                                   value="Short" />
                                           </restriction>
                                       </simpleType>
                                   </attribute>
                               </extension>
                           </simpleContent>
                       </complexType>
                   </element>
               </sequence>
           </complexType>
       </element>
       <element name="bundle" type="string" />
       <element name="usingBundles">
           <complexType>
               <sequence>
                   <element name="bundle" type="string"
                       minOccurs="0" maxOccurs="unbounded" />
               </sequence>
           </complexType>
       </element>
   </all>
</complexType>

<element name="services" type="rest:Tservices">
   <annotation>
       <documentation xml:lang="en">
           Representation for the
           application/org.osgi.services+xml and
           application/org.osgi.services.representations+xml
           content types.
       </documentation>
   </annotation>
</element>
<complexType name="Tservices">
   <choice>
       <element name="uri" type="string" minOccurs="0"
           maxOccurs="unbounded">
           <annotation>
               <documentation xml:lang="en">
                   Representation for the
                   application/org.osgi.services+xml content type.
               </documentation>
           </annotation>
       </element>
       <element name="service" type="rest:Tservice"
           minOccurs="0" maxOccurs="unbounded">
           <annotation>
               <documentation xml:lang="en">
                   Representation for the
                   application/org.osgi.services.representations+xml
                   content type.
               </documentation>
           </annotation>
       </element>
   </choice>
</complexType>

<element name="bundleexception" type="rest:Tbundleexception">
   <annotation>
       <documentation xml:lang="en">
           Representation for the
           application/org.osgi.bundleexception+xml content type.
       </documentation>
   </annotation>
</element>
```

```
<complexType name="Tbundleexception">
    <all>
        <element name="typecode" type="integer" />
        <element name="message" type="string" />
    </all>
</complexType>

<element name="extensions" type="rest:Textensions">
    <annotation>
        <documentation xml:lang="en">
            Representation for the
            application/org.osgi.extensions+xml content type.
        </documentation>
    </annotation>
</element>
<complexType name="Textensions">
    <sequence>
        <element name="extension" minOccurs="0" maxOccurs="unbounded">
            <complexType>
                <all>
                    <element name="name" type="string" />
                    <element name="path" type="string" />
                    <element name="service" type="long" minOccurs="0" />
                </all>
            </complexType>
        </element>
    </sequence>
</complexType>
</schema>
```

The schema is also available in digital form from [3] *OSGi XML Schemas.*

137.8 Capabilities

137.8.1 osgi.implementation Capability

An implementation of this specification must provide the osgi.implementation capability with name osgi.rest. This capability can be used by provisioning tools and during resolution to ensure that a REST Management implementation is present to handle REST requests defined in this specification. The capability must also declare a uses constraint on the org.osgi.service.rest package:

```
Provide-Capability: osgi.implementation;
  osgi.implementation="osgi.rest";
  uses:="org.osgi.service.rest";
  version:Version="1.0"
```

This capability must follow the rules defined for the *osgi.implementation Namespace* on page 685.

137.8.2 osgi.service Capability

A bundle providing the RestClientFactory service as described by this specification must inform tools about this service by providing the osgi.service capability representing this service. This capability must also declare a uses constraint for the org.osgi.service.rest.client package:

```
Provide-Capability: osgi.service;
  objectClass:List<String>="org.osgi.service.rest.client.RestClientFactory";
  uses:="org.osgi.service.rest.client"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 685.

137.9   Security

Like any externally visible management interface, the REST interface exposes privileged operations and hence requires access control. Since REST builds upon the HTTP(s) protocol, authentication mechanisms and encryption can be applied the same way as usually done for web servers: they can be layered below the REST protocol. E.g., confidentiality of the transmitted commands can be ensured by using HTTPS as the underlying transport. Authentication can be added by requiring, e.g., basic authentication prior to accepting a REST command. The REST interface should only be implemented by a trusted bundle. Implementations of this specification require all Admin Permissions and all Service Permissions.

137.10   org.osgi.service.rest

Rest Service Package Version 1.0.

137.10.1   Summary

- `RestApiExtension` - Marker interface for registering extensions to the Rest API service.

137.10.2   public interface RestApiExtension

Marker interface for registering extensions to the Rest API service.

The REST service provides a RESTful interface to clients that need to manage an OSGi framework through a network connection. Other components running on the same framework can contribute their own specific REST interface and make it available and discoverable by registering this marker service using the Whiteboard pattern.

Integration of third-party REST interfaces with the framework REST service on the implementation level might not always be possible since it requires knowledge about the underlying implementation and an extension mechanism on that level. Specific technologies such as servlets might support this but the REST service could as well be implemented without the use of a supporting abstraction layer and not offer extensibility.

Using this marker service, the REST service includes the advertised service in the Extensions Resource, allowing clients to discover it and use the extension's functionality.

137.10.2.1   public static final String NAME = "org.osgi.rest.name"

This service property describes the package name of the technology manageable by this REST API extension. Services specified in OSGi specifications must use their canonical package name as the name. Third-party technologies should also use their package names. The type of this property is `java.lang.String` and the property is mandatory.

137.10.2.2   public static final String SERVICE = "org.osgi.rest.service"

This service property refers to the id of the service the REST API extension provides management capabilities for. This can be useful if more than one service of a given type is present in the framework. For example if more than one Http Service is available this property is used to associate a REST extension managing the Http Service with a specific service instance. The type of the property is `java.lang.Long` and the property is optional; if the REST extension is not directly associated with a service in the service registry, the property should not be set.

137.10.2.3   public static final String URI_PATH = "org.osgi.rest.uri.path"

This service property describes a URI to the REST extension on this local machine. It is either an fully qualified URI with a different port if no integration with the framework REST service is possible

or a relative URI implicitly using the same port if integration is possible. The type of this property is java.lang.String and the property is mandatory.

137.11   org.osgi.service.rest.client

Rest Service Client Package Version 1.0.

137.11.1   Summary

- `RestClient` - A Java client API for a REST service endpoint.
- `RestClientFactory` - Factory to construct new REST client instances.

137.11.2   public interface RestClient

A Java client API for a REST service endpoint.

Provides a Java client API for accessing and managing a remote OSGi framework through the REST API. Implementations of this interface will usually take the URL to the remote REST Management Service instance as an argument in their constructor. Further arguments might be needed, for example, if the cloud provider requires URL signing.

Provider Type Consumers of this API must not implement this type

137.11.2.1   public BundleDTO getBundle(long id) throws Exception

id Addresses the bundle by its identifier.

☐ Retrieve the bundle representation for a given bundle Id.

Returns A BundleDTO for the requested bundle.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.2   public BundleDTO getBundle(String bundlePath) throws Exception

bundlePath Addresses the bundle by its URI path.

☐ Retrieve the bundle representation for a given bundle path.

Returns A BundleDTO for the requested bundle.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.3   public Map<String,String> getBundleHeaders(long id) throws Exception

id Addresses the bundle by its identifier.

☐ Get the header for a bundle given by its bundle Id.

Returns Returns the map of headers entries.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.4   public Map<String,String> getBundleHeaders(String bundlePath) throws Exception

bundlePath Addresses the bundle by its URI path.

☐ Get the header for a bundle given by its URI path.

Returns Returns the map of headers entries.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.5   public Collection<String> getBundlePaths() throws Exception

☐ Get the bundles currently installed on the managed framework.

Returns Returns a collection of the bundle URIs in the form of Strings. The URIs are relative to the REST API root URL and can be used to retrieve bundle representations.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.6 **public Collection<BundleDTO> getBundles() throws Exception**

□ Get the bundle representations for all bundles currently installed in the managed framework.

Returns Returns a collection of BundleDTO objects.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.7 **public BundleStartLevelDTO getBundleStartLevel(long id) throws Exception**

id Addresses the bundle by its identifier.

□ Get the start level for a bundle given by its bundle Id.

Returns Returns a BundleStartLevelDTO describing the current start level of the bundle.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.8 **public BundleStartLevelDTO getBundleStartLevel(String bundlePath) throws Exception**

bundlePath Addresses the bundle by its URI path.

□ Get the start level for a bundle given by its URI path.

Returns Returns a BundleStartLevelDTO describing the current start level of the bundle.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.9 **public int getBundleState(long id) throws Exception**

id Addresses the bundle by its identifier.

□ Get the state for a given bundle Id.

Returns Returns the current bundle state as defined in (@link org.osgi.framework.Bundle}.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.10 **public int getBundleState(String bundlePath) throws Exception**

bundlePath Addresses the bundle by its URI path.

□ Get the state for a given bundle path.

Returns Returns the current bundle state as defined in (@link org.osgi.framework.Bundle}.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.11 **public FrameworkStartLevelDTO getFrameworkStartLevel() throws Exception**

□ Retrieves the current framework start level.

Returns Returns the current framework start level in the form of a FrameworkStartLevelDTO.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.12 **public Collection<String> getServicePaths() throws Exception**

□ Gets a collection of URI paths to all installed services.

Returns Returns a collection of URI paths to the installed services.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.13 **public Collection<String> getServicePaths(String filter) throws Exception**

filter Passes a filter to restrict the result set.

□ Gets a collection of URI paths to all installed services.

Returns Returns a collection of URI paths to the installed services.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.14   **public ServiceReferenceDTO getServiceReference(long id) throws Exception**

id Addresses the service by its identifier.

□ Get the service representation for a service given by its service Id.

Returns The service representation as ServiceReferenceDTO.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.15   **public ServiceReferenceDTO getServiceReference(String servicePath) throws Exception**

servicePath Addresses the service by its URI path.

□ Get the service representation for a service given by its URI path.

Returns The service representation as ServiceReferenceDTO.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.16   **public Collection‹ServiceReferenceDTO› getServiceReferences() throws Exception**

□ Get the service representations for all services.

Returns Returns the service representations in the form of ServiceReferenceDTO objects.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.17   **public Collection‹ServiceReferenceDTO› getServiceReferences(String filter) throws Exception**

filter Passes a filter to restrict the result set.

□ Get the service representations for all services.

Returns Returns the service representations in the form of ServiceReferenceDTO objects.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.18   **public BundleDTO installBundle(String location) throws Exception**

location Passes the location string to retrieve the bundle content from.

□ Install a new bundle given by an externally reachable location string, typically describing a URL.

Returns Returns the BundleDTO of the newly installed bundle.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.19   **public BundleDTO installBundle(String location,InputStream in) throws Exception**

location Passes the location string to be used to install the new bundle.

in Passes the input stream to a bundle.

□ Install a new bundle given by an InputStream to a bundle content.

Returns Returns the BundleDTO of the newly installed bundle.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.20   **public void setBundleStartLevel(long id,int startLevel) throws Exception**

id Addresses the bundle by its identifier.

startLevel The target start level.

□ Set the start level for a bundle given by its bundle Id.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.21   **public void setBundleStartLevel(String bundlePath,int startLevel) throws Exception**

bundlePath Addresses the bundle by its URI path.

startLevel The target start level.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.22   **public void setFrameworkStartLevel(FrameworkStartLevelDTO startLevel) throws Exception**

startLevel set the framework start level to this target.

□ Sets the current framework start level.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.23   **public void startBundle(long id) throws Exception**

id Addresses the bundle by its identifier.

□ Start a bundle given by its bundle Id.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.24   **public void startBundle(String bundlePath) throws Exception**

bundlePath Addresses the bundle by its URI path.

□ Start a bundle given by its URI path.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.25   **public void startBundle(long id,int options) throws Exception**

id Addresses the bundle by its identifier.

options Passes additional options as defined in org.osgi.framework.Bundle.start(int)

□ Start a bundle given by its bundle Id.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.26   **public void startBundle(String bundlePath,int options) throws Exception**

bundlePath Addresses the bundle by its URI path.

options Passes additional options as defined in org.osgi.framework.Bundle.start(int)

□ Start a bundle given by its URI path.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.27   **public void stopBundle(long id) throws Exception**

id Addresses the bundle by its identifier.

□ Stop a bundle given by its bundle Id.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.28   **public void stopBundle(String bundlePath) throws Exception**

bundlePath Addresses the bundle by its URI path.

□ Stop a bundle given by its URI path.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.2.29　　**public void stopBundle(long id,int options) throws Exception**

id　Addresses the bundle by its identifier.

options　Passes additional options as defined in org.osgi.framework.Bundle.stop(int)

□　Stop a bundle given by its bundle Id.

Throws　Exception– An exception representing a failure in the underlying REST call.

137.11.2.30　　**public void stopBundle(String bundlePath,int options) throws Exception**

bundlePath　Addresses the bundle by its URI path.

options　Passes additional options as defined in org.osgi.framework.Bundle.stop(int)

□　Stop a bundle given by its URI path.

Throws　Exception– An exception representing a failure in the underlying REST call.

137.11.2.31　　**public BundleDTO uninstallBundle(long id) throws Exception**

id　Addresses the bundle by its identifier.

□　Uninstall a bundle given by its bundle Id.

Returns　Returns the BundleDTO of the uninstalled bundle.

Throws　Exception– An exception representing a failure in the underlying REST call.

137.11.2.32　　**public BundleDTO uninstallBundle(String bundlePath) throws Exception**

bundlePath　Addresses the bundle by its URI path.

□　Uninstall a bundle given by its URI path.

Returns　Returns the BundleDTO of the uninstalled bundle.

Throws　Exception– An exception representing a failure in the underlying REST call.

137.11.2.33　　**public BundleDTO updateBundle(long id) throws Exception**

id　Addresses the bundle by its identifier.

□　Updates a bundle given by its bundle Id using the bundle-internal update location.

Returns　Returns the BundleDTO of the updated bundle.

Throws　Exception– An exception representing a failure in the underlying REST call.

137.11.2.34　　**public BundleDTO updateBundle(long id,String url) throws Exception**

id　Addresses the bundle by its identifier.

url　The URL whose content is to be used to update the bundle.

□　Updates a bundle given by its URI path using the content at the specified URL.

Returns　Returns the BundleDTO of the updated bundle.

Throws　Exception– An exception representing a failure in the underlying REST call.

137.11.2.35　　**public BundleDTO updateBundle(long id,InputStream in) throws Exception**

id　Addresses the bundle by its identifier.

in　Passes an input stream to the new bundle content.

□　Updates a bundle given by its bundle Id and passing the new bundle content in the form of an In-putStream.

Returns　Returns the BundleDTO of the updated bundle.

Throws Exception– An exception representing a failure in the underlying REST call.

137.11.3 public interface RestClientFactory

Factory to construct new REST client instances. Each instance is specific to a REST service endpoint.

Implementations can choose to extend this interface to add additional creation methods, where additional arguments are needed for request signing, etc.

In OSGi environments, this factory is registered as a service.

Provider Type Consumers of this API must not implement this type

137.11.3.1 public RestClient createRestClient(URI uri)

uri The URI to the REST service endpoint.

☐ Create a new REST client instance.

Returns A new REST client instance for the specified REST service endpoint.

137.12 JavaScript Client API

REST JavaScript Client API Version 1.0

137.12.1 Summary

- `OSGiRestClient` - A JavaScript client API for accessing and managing a remote OSGi framework through the REST API.
- `OSGiRestCallback` - Callback object provided to the `OSGiRestClient` functions. Invoked on completion of the remote invocation.

JavaScript does not support the concept of interfaces and therefore implementations of the JavaScript client specification can provide objects of any type as long as they conform to the to the signatures described in this specification.

To facilitate documenting the JavaScript APIs *Web IDL* is used; see [2] *Web IDL*. This clarifies the accepted arguments and return types for otherwise untyped functions. Web IDL is only used for documentation purposes and has no bearing on the implementation of this API.

Note: some data types in Web IDL have slightly different names than commonly used in languages like Java or JavaScript. For example a `String` is called `DOMString` and the equivalent of a Java `long` is called `long long`. Additionally, when a representation as defined in this specification is passed to one of the JavaScript client APIs this representation is provided as a JavaScript object. Following the recommendations for mapping these to Web IDL, these JavaScript Object parameters are described using the `dictionary` data type. For more information see the Web IDL specification.

137.12.2 interface OSGiRestClient

Provides a JavaScript client API for accessing and managing a remote OSGi framework through the REST API. Implementations will provide a proprietary constructor to create objects of this signature. Once created the object can be used from JavaScript environments to manage the framework.

137.12.2.1 void getBundle((DOMString or long long) bundle, OSGiRestCallback cb)

bundle The bundle, either the numeric bundle ID or the bundle URI path.

cb The callbacks invoked on completion of the remote invocation. On success the `success()` callback is invoked with the Bundle representation as JavaScript object.

☐ Get the Bundle representation of a specific bundle.

137.12.2.2   **void getBundleHeader((DOMString or long long) bundle, OSGiRestCallback cb)**

bundle The bundle, either the numeric bundle ID or the bundle URI path.

cb The callbacks invoked on completion of the remote invocation. On success the success() callback is invoked with the Bundle Header representation as JavaScript object.

☐ Get the Bundle Header representation of a specific bundle.

137.12.2.3   **void getBundleRepresentations(OSGiRestCallback cb)**

cb The callbacks invoked on completion of the remote invocation. On success the success() callback is invoked with the Bundle Representations List representation as JavaScript object.

☐ List the bundles details.

137.12.2.4   **void getBundles(OSGiRestCallback cb)**

cb The callbacks invoked on completion of the remote invocation. On success the success() callback is invoked with the Bundle List representation as JavaScript object.

☐ List the bundles.

137.12.2.5   **void getBundleStartLevel((DOMString or long long) bundle, OSGiRestCallback cb)**

bundle The bundle, either the numeric bundle ID or the bundle URI path.

cb The callbacks invoked on completion of the remote invocation. On success the success() callback is invoked with the Bundle Start Level representation as JavaScript object.

☐ Get the Bundle Start Level representation of a specific bundle.

137.12.2.6   **void getBundleState((DOMString or long long) bundle, OSGiRestCallback cb)**

bundle The bundle, either the numeric bundle ID or the bundle URI path.

cb The callbacks invoked on completion of the remote invocation. On success the success() callback is invoked with the Bundle State representation as JavaScript object.

☐ Get the Bundle State representation of a specific bundle.

137.12.2.7   **void getFrameworkStartLevel(OSGiRestCallback cb)**

cb The callbacks invoked on completion of the remote invocation. On success the success() callback is invoked with the Framework Start Level representation as JavaScript object.

☐ Obtain the Framework Start Level.

137.12.2.8   **void getService((DOMString or long long) service, OSGiRestCallback cb)**

service The service, either the numeric service ID or the service URI path.

cb The callbacks invoked on completion of the remote invocation. On success the success() callback is invoked with the Service representation as JavaScript object.

☐ Get a service representation.

137.12.2.9   **void getServiceRepresentations(OSGiRestCallback cb)**

cb The callbacks invoked on completion of the remote invocation. On success the success() callback is invoked with the Service Representations List representation as JavaScript object.

☐ Get all services representations.

137.12.2.10   **void getServices(OSGiRestCallback cb)**

cb The callbacks invoked on completion of the remote invocation. On success the success() callback is invoked with the Service List representation as JavaScript object.

□ Get all services URIs.

137.12.2.11   **void installBundle((DOMString or ArrayBuffer) bundle, OSGiRestCallback cb)**

bundle The Bundle to install, either represented as a URL or as an ArrayBuffer of

cb The callbacks invoked on completion of the remote invocation. On success the success() callback is invoked with the Bundle representation of the newly installed Bundle. This parameter is optional.

□ Install a bundle from a URI or by value.

137.12.2.12   **void setBundleStartLevel((DOMString or long long) bundle, dictionary bsl, OSGiRestCallback cb)**

bundle The bundle, either the numeric bundle ID or the bundle URI path.

bsl A Bundle Start Level representation dictionary with the desired state.

cb The callbacks invoked on completion of the remote invocation. On success the success() callback is invoked with the resulting Framework Start Level representation as JavaScript object. This parameter is optional.

□ Change the Framework Start Level and/or initial bundle start level.

137.12.2.13   **void setBundleState((DOMString or long long) bundle, dictionary state, OSGiRestCallback cb)**

bundle The bundle, either the numeric bundle ID or the bundle URI path.

state Bundle State representation dictionary with the desired state.

cb The callbacks invoked on completion of the remote invocation. On success the success() callback is invoked with the resulting Bundle Start Level representation as JavaScript object. This parameter is optional.

□ Change the Bundle Start Level and/or other options defined in the Bundle Start Level representation.

137.12.2.14   **void setFrameworkStartLevel(dictionary fwsl, OSGiRestCallback cb)**

fwsl Framework Start Level representation dictionary with the desired state.

cb The callbacks invoked on completion of the remote invocation. On success the success() callback is invoked with the resulting Framework Start Level representation as JavaScript object. This parameter is optional.

□ Change the Framework Start Level and/or initial bundle start level.

137.12.2.15   **void startBundle((DOMString or long long) bundle, OSGiRestCallback cb)**

bundle The bundle, either the numeric bundle ID or the bundle URI path.

cb The callbacks invoked on completion of the remote invocation. On success the success() callback is invoked with the Bundle State representation as JavaScript object. This parameter is optional.

□ Start a bundle.

137.12.2.16   **void startBundle((DOMString or long long) bundle, long options, OSGiRestCallback cb)**

bundle The bundle, either the numeric bundle ID or the bundle URI path.

options The options passed to the bundle's start method as a number.

cb The callbacks invoked on completion of the remote invocation. On success the success() callback is invoked with the Bundle State representation as JavaScript object. This parameter is optional.

□ Start a bundle.

137.12.2.17   **void stopBundle((DOMString or long long) bundle, OSGiRestCallback cb)**

bundle The bundle, either the numeric bundle ID or the bundle URI path.

cb   The callbacks invoked on completion of the remote invocation. On success the success() callback is invoked with the Bundle State representation as JavaScript object. This parameter is optional.

☐   Stop a bundle.

137.12.2.18   **void stopBundle((DOMString or long long) bundle, long options, OSGiRestCallback cb)**

bundle   The bundle, either the numeric bundle ID or the bundle URI path.

options The options passed to the bundle's start method as a number.

cb   The callbacks invoked on completion of the remote invocation. On success the success() callback is invoked with the Bundle State representation as JavaScript object. This parameter is optional.

☐   Stop a bundle.

137.12.2.19   **void uninstallBundle((DOMString or long long) bundle, OSGiRestCallback cb)**

bundle   The bundle, either the numeric bundle ID or the bundle URI path.

cb   The callbacks invoked on completion of the remote invocation. On success the success() callback is invoked with the Bundle representation of the uninstalled Bundle. This parameter is optional.

☐   Uninstall a bundle.

137.12.2.20   **void updateBundle((DOMString or long long) bundle, (DOMString or ArrayBuffer) updated, OSGiRestCallback cb)**

bundle   The bundle, either the numeric bundle ID or the bundle URI path.

updated The Bundle to update, either represented as a URL or as an ArrayBuffer of

cb   The callbacks invoked on completion of the remote invocation. On success the success() callback is invoked with the Bundle representation of the updated Bundle. This parameter is optional.

☐   Update a bundle from a URI or by value.

137.12.3   callback interface OSGiRestCallback

Objects implementing this signature are provided by users of the OSGiRestClient as callbacks. One of the callback functions is invoked on completion of a REST invocation.

137.12.3.1   **void success(object response)**

response The result of the invocation. The type of this parameter is depends on the function being invoked. It can be found in the documentation of the function.

☐   Called when the invocation completes successfully.

137.12.3.2   **void failure(short httpCode, object response)**

httpCode The HTTP code returned. If no HTTP code is associated with the failure this parameter is set to -1.

response The failure response.

☐   Called when the invocation failed.

137.13   References

[1]   *Framework Filter Syntax*
       OSGi Core, Chapter 3.2.7 Filter Syntax

[2]   *Web IDL*
       http://www.w3.org/TR/WebIDL/

[3]   *OSGi XML Schemas*
        http://www.osgi.org/Specifications

[4]   *Whiteboard Pattern*
        http://www.osgi.org/wiki/uploads/Links/whiteboard.pdf

138   Asynchronous Service Specification

Version 1.0

138.1   Introduction

OSGi Bundles collaborate using loosely coupled services registered in the OSGi service registry. This is a powerful and flexible model, and allows for the dynamic replacement of services at runtime. OSGi services are therefore a very common interaction pattern within OSGi.

As with most Java APIs and Objects, OSGi services are primarily synchronous in operation. This has several benefits; synchronous APIs are typically easier to write and to use than asynchronous ones; synchronous APIs provide immediate feedback; synchronous implementations typically have a less complex threading model.

Asynchronous APIs, however, have different advantages. Asynchronous APIs can reduce bottlenecks by encouraging more effective use of parallelism, improving the responsiveness of the application. In many cases high throughput systems can be written more simply and elegantly using asynchronous programming techniques.

The *Promises Specification* on page 805 provides powerful primitives for asynchronous programming, including the ability to compose flows in a functional style. There are, however, many existing services that do not use the Promise API. The purpose of the Asynchronous Service is to bridge the gap between these existing, primarily synchronous, services in the OSGi service registry, and asynchronous programming. The Asynchronous Service therefore provides a way to invoke arbitrary OSGi services asynchronously, providing results and failure notifications through the Promise API.

138.1.1   Essentials

- *Async Invocation* - A single method call that is to be executed without blocking the requesting thread.
- *Client* - Application code that wishes to invoke one or more OSGi services asynchronously.
- *Async Service* - The OSGi service representing the Asynchronous Services implementation. Used by the Client to make one or more Async Invocations.
- *Async Mediator* - A mediator object created by the Async Service which represents the target service. Used by the Client to register Async Invocations.
- *Success Callback* - A callback made when an Async Invocation completes with a normal return value.
- *Failure Callback* - A callback made when an Async Invocation completes with an exception.
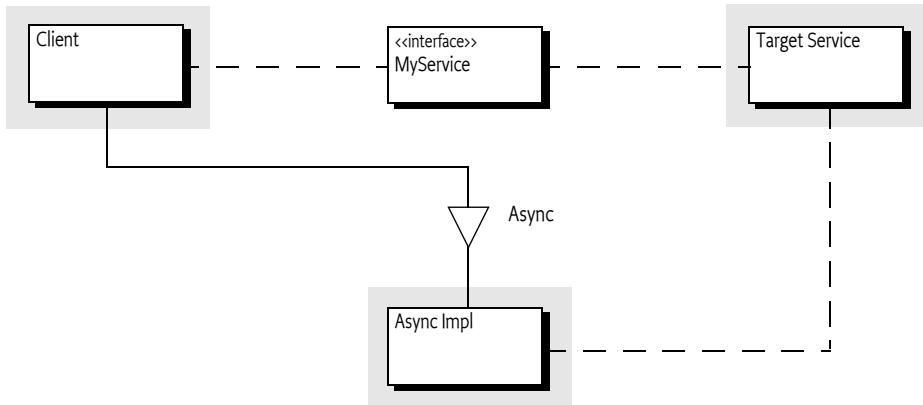
138.1.2   Entities

- *Async Service* - A service that can create Async Mediators and run Async Invocations.
- *Target Service* - A service that is to be called asynchronously by the Client.
- *Client* - The code that makes Async Invocations using the Async Service

• *Promise* - A promise, representing the result of the Async Invocation.

Figure 138.1   *Class and Service overview*

138.2   Usage

This section is an introduction in the usage of the Async Service. It is not the formal specification, the normative part starts at *Async Service* on page 745. This section leaves out some of the details for clarity.

138.2.1   Synopsis

The Async Service provides a mechanism for a client to *asynchronously* invoke methods on a target service. The service may be aware of the asynchronous nature of the call and actively participate in it, or be unaware and execute normally. In either case the client's thread will not block, and will continue executing its next instructions. Clients are notified of the completion of their task, and whether it was successful or not, through the use of the Promise API.

Each async invocation is registered by the client making a method call on an *Async Mediator*, and then started by making a call to the Async Service that created the mediator. This call returns a Promise that will eventually be resolved with the return value from the async invocation.

An Async Mediator can be created by the client, either from an Object, or directly from a Service Reference. Using a service reference has the advantage that the mediator will track the underlying service. This means that if the service is unregistered before the asynchronous call begins then the Promise will resolve with a failure, rather than continuing using an invalid service object.

138.2.2   Making Async Invocations

The general pattern for a client is to obtain the Async Service, and a service reference for the target service. The client then creates an Async Mediator for the target service, invokes a method on the mediator, then starts the asynchronous call. This is demonstrated in the following example:

```
private Async asyncService;
private ServiceReference<Foo> fooRef;
private Foo mediated;

@Reference
void setAsync(Async async) {
```

```
        asyncService = async;
    }

    @Reference(service = Foo.class)
    void setList(ServiceReference<Foo> foo) {
        fooRef = foo;
    }

    @Activate
    void start() {
        mediated = asyncService.mediate(fooRef, Foo.class);
    }

    public synchronized void doStuff() {
        Promise<Boolean> promise = asyncService
                .call(mediated.booleanMethod("aValue"));
        ...
    }
}
```

This example demonstrates how simply clients can make asynchronous calls using the Async Service. The eventual result can be obtained from the promise using one of the relevant callbacks.

One important thing to note is that whilst the call to call() or call(R) causes the async invocation to begin, the actual execution of the underlying task may be queued until a thread is available to run it. If the service has been unregistered before the execution actually begins then the promise will be resolved with a Service Exception. The type of the Service Exception will be ASYNC_ERROR.

138.2.3   Async Invocations of Void Methods

The return value of the mediator method call is used to provide type information to the Async Service. This, however, does not work for void methods that have no return value. In this case the client can either pass an arbitrary object to the call(R) method, or use the zero argument call() method. In either case the returned promise will eventually resolve with a value of null. This is demonstrated in the following example.

```
private Async asyncService;
private ServiceReference<Foo> fooRef;
private Foo mediated;

@Reference
void setAsync(Async async) {
    asyncService = async;
}

@Reference(service = Foo.class)
void setList(ServiceReference<Foo> foo) {
    fooRef = foo;
}

@Activate
void start() {
    mediated = asyncService.mediate(fooRef, Foo.class);
}
```

```
public synchronized void doStuff() {
    mediated.voidMethod();
    Promise<?> promise = asyncService
            .call();
    ...
}
```

138.2.4   Fire and Forget Calls

Sometimes a client does not require any notification that an async invocation has completed. In this case the client could use one of the call() or call(R) methods and simply discard the returned Promise object. This, however, can be wasteful of resources. The act of resolving the Promise object may be expensive, for example it may involve serializing the return value over a network if the remote call was asynchronous.

If the client knows that no Promise object representing the result of the asynchronous task is needed then it can signal this to the Async Service. This allows the Async Service to better optimize the async invocation by not providing a result.

To indicate that the client wants to make a fire-and-forget style call the client invokes the mediator as normal, but then begins the asynchronous invocation using the execute() method as show below.

```
private Async asyncService;
private ServiceReference<Foo> fooRef;

private Foo mediated;

@Reference
void setAsync(Async async) {
    asyncService = async;
}

@Reference(service = Foo.class)
void setList(ServiceReference<Foo> foo) {
    fooRef = foo;
}

@Activate
void start() {
    mediated = asyncService.mediate(fooRef, Foo.class);
}

public synchronized void doStuff() {
    mediated.someMethod();
    asyncService.execute();
    ...
}
```

Note that the execute() method does still return a Promise. This Promise is not the same as the ones returned by call() or call(R), its resolution value does not provide access to the result, but instead indicates whether the fire-and-forget call could be successfully started. If there is a failure which prevents the task from being executed then this is used to fail the returned promise.

138.2.5   Multi Threading

By their very definition asynchronous tasks do not run inline, and typically they will not run on the same thread as the caller. This is not, however, a guarantee. A valid implementation of the Async Service may have only one worker thread, which may be the thread currently running in the client code. Async invocations also have the same threading model as the Promise API. This means that callbacks may run on arbitrary threads, which may, or may not, be the same as the client thread, or the thread which executed the asynchronous work.

It is important for multi-threaded clients to note that calls to the mediator and Async Service must occur on the same thread. For example it is not supported to invoke a mediator using one thread, and then to begin the async invocation by calling the call(), call(R) or execute() method on a different thread.

138.3   Async Service

The Async Service is the primary interaction point between a client and the Async Service implementation. An Async Service implementation must expose a service implementing the Async interface. Clients obtain an instance of the Async Service using the normal OSGi service registry mechanisms, either directly using the OSGi framework API, or using dependency injection.

The Async Service is used to:

- Create async mediators
- Begin async invocations
- Obtain Promise objects representing the result of the async invocation

138.3.1   Using the Async Service

The first action that a client wishing to make an async invocation must take is to create an async mediator using one of the mediate methods. Once created the client invokes the method that should be run asynchronously, supplying the arguments that should be used. This call records the invocation, but does not start the asynchronous task. The asynchronous task begins when the client invokes one of the call or execute methods on the Async Service. The call methods must return a Promise representing the async invocation. The promise must resolve with the value returned by the async invocation, or fail with the failure thrown by the async invocation.

If the client attempts to begin an async invocation without first having called a method on the mediator object then the Async Service must detect this usage error and throw an IllegalStateException to the client. This applies to all methods that begin an async invocation.

138.3.2   Asynchronous Failures

There are a variety of reasons that async invocations may be started correctly by the client, but then fail without running the asynchronous task. In any of these cases the Promise representing the async invocation must fail with a Service Exception. This Service Exception must be initialized with a type of ASYNC_ERROR. If there is no promise representing the async invocation then there is no way to notify the client of the failure, therefore the Service Exception must be logged by the Async Service using all available Log Service implementations.

The following list of scenarios is not exhaustive, but indicates failure scenarios that must result in a Service Exception with a type of async

- If the client is using a service reference backed mediator and the client bundle's bundle context becomes invalid before looking up the target service.
- If the client is using a service reference backed mediator and the service is unregistered before making the async invocation.

- If the client is using a service reference backed mediator and the service lookup returns null
- If the Async Service is unable to accept new work, for example it is in the process of being shut down.
- If the type of the mediator object does not match the type of the service object to be invoked.

138.3.3 Thread Safety and Instance Sharing

Implementations of the Async Service must be thread safe and may be used simultaneously across multiple clients and from multiple threads within the same client. Whilst the Async Service is able to be used across multiple threads, if a client wishes to make an async invocation then the call to the mediator and the call to begin the async invocation must occur on the same thread. The returned Promise may then be shared between threads if required.

It is expected, although not required, that the Async Service implementation will use a Service Factory to create customized implementations for each client bundle. This simplifies the tracking of the relevant client bundle context to use when performing service lookups on the client bundle's behalf. Clients should therefore not share instances of the Async Service with other bundles. Instead both bundles should obtain their own instances from the service registry.

138.3.4 Service Object Lifecycle Management

If the Async Service is being used to call an OSGi service object and the service reference is available then the service object should be looked up immediately before the asynchronous task begins executing. This ensures that the service is still available at the point it is eventually called. Any call to getService must have a corresponding call to ungetService after the mediated method invoked has returned and, if available, the promise is resolved, but before the asynchronous task releases its thread of execution.

138.4 The Async Mediator

Async mediators are dynamically created objects that have the same type or interface as the object being mediated, and are used to record method invocations and arguments. Mediator objects are specific to an Async Service implementation, and must only be used in conjunction with the Async Service object that they were created by.

Mediators may be created either from a ServiceReference or from a service object. The actions and overall result are similar for both the mediate(ServiceReference,Class) and mediate(T,Class) methods, with the primary difference being that mediated objects created from a ServiceReference will validate whether the service object is still available immediately before the asynchronous task is executed.

138.4.1 Building the Mediator Object

The client passes in a Class indicating the type that should be mediated. If the class object represents an interface type then the generated mediator object must implement that interface. If the class object represents a Java class type then the mediator object must either be an instance of that type or extend it.

When building a mediator object the Async Service has the opportunity to detect numerous problems, for example if the referenced service to be mediated has been unregistered. Although fail-fast behavior is usually preferable, in this case it would force the client to handle errors in two places; both when creating the mediator, and for the returned Promise. To simplify client usage, error cases detected when creating a mediator must not prevent the mediator from being created and must not result in an exception being thrown. The only reason that the Async Service may fail to create a mediator is if the class object passed in cannot be mediated.

There are three reasons why the Async Service may not be able to mediate a class type:

- The class object passed in represents a final type.
- The class object passed in represents a type that has no zero-argument constructor.
- The class object passed in represents a type which has one or more public final methods present in its type hierarchy (other than those declared by java.lang.Object).

If any of these constraints are violated and prevent the Async Service from creating a mediator then the Async Service must throw an IllegalArgumentException.

138.4.2   Async Mediator Behaviors

When invoked, the Async mediator must record the method call, and its arguments, and then return rapidly and should avoid performing blocking operations. The values returned by the mediator object are opaque, and the client should not attempt to interpret the returned value. The value may be null (or null-like in the case of primitives) or contain implementation specific information. If the mediated method call has a return type, specifically it is non-void, then this object must be passed to the Async Service's call method when beginning the async invocation

Async mediators should make a best-effort attempt to detect incorrect API usage from the client. If this incorrect usage is detected then the mediator object must throw an IllegalStateException when invoked. An example of incorrect usage that must be detected is when a client makes multiple invocations on a single mediator object from the same thread without making any calls to the Async Service.

After a usage error has been detected and an IllegalStateException has been thrown the mediator object must be reset so that a subsequent invocation from the client thread can proceed normally.

138.4.3   Thread Safety and Instance Sharing

Async mediators, like instances of the Async Service, are required to be thread safe. Clients may therefore share mediator objects across threads, and can safely store them as instance fields. Whilst mediators are thread safe, if a client wishes to make an async invocation then the call to the mediator and the call to call() or call(R) must occur on the same thread. The returned Promise may then be shared between threads if required.

Async mediators created from ServiceReference objects remain directly associated with the service reference and client bundle after creation. Clients should therefore not share mediator objects with other bundles. Instead each bundle should create its own mediator.

138.5   Fire and Forget Invocations

The Async Service provides call() and call(R) methods for clients to use when they wish to receive results from asynchronous tasks. Clients that do not need the result can simply discard the returned Promise object. This, however, can be wasteful of resources. The act of resolving the Promise object may be expensive, for example it may involve serializing the return value over a network.

To address this use case the Async Service provides the execute() method, which behaves similarly to call() and call(R), but does not provide access to the eventual result. Instead the execute() method returns a Promise that indicates whether the fire-and-forget call is able to be successfully started.

The returned Promise must be resolved with null if the asynchronous task begins executing successfully. There is no *happens-before* relationship required, meaning that if the Promise resolves successfully then the task may, or may not, have started or finished. The primary usage of the Promise is actually to detect failures. If the fire-and-forget task cannot be executed for some reason, for example the backing service has been unregistered, then the returned promise must be failed appropriately using the same rules as defined in *Asynchronous Failures* on page 745. If the returned Promise is failed then the fire-and-forget task has not executed and will not execute in the future.

138.6   Delegating to Asynchronous Implementations

Some service APIs are already asynchronous in operation, and others are partly asynchronous, in that some methods run asynchronously and others do not. There are also services which have a synchronous API, but could run asynchronously because they are a proxy to another service. A good example of this kind of service is a remote service. Remote services are local views of a remote endpoint, and depending upon the implementation of the endpoint it may be possible to make the remote call asynchronously, optimizing the thread usage of any local asynchronous call.

Services that already have some level of asynchronous support may advertise this to clients and to the Async Service by having their service object be an instanceof AsyncDelegate. The service object can be cast to AsyncDelegate to be used by the Async Service implementation, or by the client directly, to make an asynchronous call on the service.

Because the Async Delegate behavior is transparently handled by the Async Service, clients of the Async Service do not need to know whether the service object is an instanceof AsyncDelegate or not. Their usage pattern can remain unchanged.

When making an async invocation, the Async Service must check to see whether the service object is an instanceof AsyncDelegate. If the service object is an instanceof AsyncDelegate, then the Async Service must attempt to delegate the asynchronous call. The exact delegation operation depends on whether a Promise result is required.

138.6.1   Obtaining a Promise from an Async Delegate

If the result of the method invocation is needed by the client, then the Async Service must attempt to delegate to the async(Method,Object[]) method. The delegation proceeds as follows:

- If the call to the Async Delegate returns a Promise, then the Promise returned by the Async Service must be resolved with that Promise.
- If the call to the Async Delegate throws an exception, then the Promise returned by the Async Service must be failed with the exception.
- If the Async Delegate is unable to optimize the call and returns null from the async(Method,Object[]) method, the Async Service must continue processing the async invocation, treating the service as a normal service object.

138.6.2   Delegating Fire and Forget Calls to an Async Delegate

If the result of the method invocation is not needed by the client, then the Async Service must attempt to delegate to the execute(Method,Object[]) method. This gives the Async Delegate implementation the opportunity to further optimize its processing. The delegation proceeds as follows:

- If the call to the Async Delegate returns true, then the Promise returned by the Async Service must be resolved with null.
- If the call to the Async Delegate throws an exception, then the Promise returned by the Async Service must be failed with the exception.
- If the Async Delegate is unable to optimize the call and returns false from the execute(Method,Object[]) method, the Async Service must continue processing the async invocation, treating the service as a normal service object.

138.6.3   Lifecycle for Service Objects When Delegating

If an Async Delegate implementation accepts an asynchronous task, via a call to either execute(Method,Object[]) or async(Method,Object[]), then it is responsible for continuing to process the work until completion. This means that if the service implementing Async Delegate is unregistered for some reason, then the task must be properly cleaned up and succeed or fail as appropriate.

If the Async Service implementation used a service reference to obtain the service, then it must release the service object after the task has been accepted. This means that if the service object is provided by a service factory, then the service object should take extra care not to destroy its internal state when released. The service object must remain valid until all executing asynchronous tasks associated with the service object are either completed or failed.

If an Async Delegate implementation rejects an asynchronous task, by returning `false` or `null`, the Async Service implementation must take over the asynchronous invocation of the method. In this case, if the Async Service implementation used a service reference to obtain the service, the Async Service must not release the service object until the asynchronous task is completed.

If an Async Delegate implementation throws an exception and the Async Service implementation used a service reference to obtain the service, then the service object must be released immediately.

138.7 Capabilities

Implementations of the Asynchronous Service specification must provide the following capabilities.

- A capability in the `osgi.implementation` namespace declaring the implemented specification to be `osgi.async`. This capability must also declare a uses constraint for the `org.osgi.service.async` and `org.osgi.service.async.delegate` packages. For example:

```
Provide-Capability: osgi.implementation;
    osgi.implementation="osgi.async";
    version:Version="1.0";
    uses:="org.osgi.service.async,org.osgi.service.async.delegate"
```

This capability must follow the rules defined for the *osgi.implementation Namespace* on page 685.

- A capability in the `osgi.service` namespace representing the Async service. This capability must also declare a uses constraint for the org.osgi.service.async package. For example:

```
Provide-Capability: osgi.service;
    objectClass:List<String>="org.osgi.service.async.Async";
    uses:="org.osgi.service.async"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 685.

138.8 Security

Asynchronous Services implementations must be careful to avoid elevating the privileges of client bundles when calling services asynchronously, and also to avoid restricting the privileges of clients that are permitted to make a call. This means that the implementation must:

- Be granted `AllPermission`. As the Async Service will always be on the stack when invoking a service object asynchronously it must be granted `AllPermission` so that it does not interfere with security any checks made by the service object.
- Establish the caller's `AccessControlContext` in a worker thread before starting to call the service object. This prevents a bundle from being able to call a service asynchronously that it would not normally be able to call. The `AccessControlContext` must be collected during any call to call(), call(R) or execute().
- Use a `doPrivileged` block when mediating a concrete type. A no-args constructor in a concrete type may perform actions that the client may not have permission to perform. This should not

prevent the client from mediating the object, as the client is not directly performing these actions.

- If the mediator object was created using a service reference, then the Async Services implementation must use the client's bundle context when retrieving the target service. If the service lookup occurs on a worker thread, then the lookup must use the `AccessControlContext` collected during the call to call(), call(R) or execute(). This prevents the client bundle from being able to make calls on a service object that they do not have permission to obtain, and ensures that an appropriately customized object is returned if the service is implemented using a service factory.

Further security considerations can be addressed using normal OSGi security rules. For example access to the Async Service can be controlled using `ServicePermission[...Async, GET]`.

138.9   org.osgi.service.async

Asynchronous Services Package Version 1.0.

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

Example import for consumers using the API in this package:

`Import-Package: org.osgi.service.async; version="[1.0,2.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.async; version="[1.0,1.1)"`

138.9.1   Summary

- `Async` - The Asynchronous Execution Service.

138.9.2   public interface Async

The Asynchronous Execution Service. This can be used to make asynchronous invocations on OSGi services and objects through the use of a mediator object.

Typical usage:

```
Async async = ctx.getService(asyncRef);

ServiceReference<MyService> ref = ctx.getServiceReference(MyService.class);

MyService mediator = async.mediate(ref, MyService.class);

Promise<BigInteger> result = async.call(mediator.getSumOverAllValues());
```

The Promise API allows callbacks to be made when asynchronous tasks complete, and can be used to chain Promises.

Multiple asynchronous tasks can be started concurrently, and will run in parallel if the Async Service has threads available.

Provider Type   Consumers of this API must not implement this type

138.9.2.1   public Promise<R> call(R r)

Type Arguments   <R>

r   The return value of the mediated call, used for type information.

☐ Invoke the last method call registered by a mediated object as an asynchronous task. The result of the task can be obtained using the returned Promise.

Typically the parameter for this method will be supplied inline like this:

```
ServiceReference<I> s = ...;
I i = async.mediate(s, I.class);
Promise<String> p = async.call(i.foo());
```

Returns A Promise which can be used to retrieve the result of the asynchronous task.

138.9.2.2   public Promise<?> call()

☐ Invoke the last method call registered by a mediated object as an asynchronous task. The result of the task can be obtained using the returned Promise.

Generally it is preferable to use call(Object) like this:

```
ServiceReference<I> s = ...;
I i = async.mediate(s, I.class);
Promise<String> p = async.call(i.foo());
```

However this pattern does not work for void methods. Void methods can therefore be handled like this:

```
ServiceReference<I> s = ...;
I i = async.mediate(s, I.class);
i.voidMethod()
Promise<?> p = async.call();
```

Returns A Promise which can be used to retrieve the result of the asynchronous task.

138.9.2.3   public Promise<Void> execute()

☐ Invoke the last method call registered by a mediated object as a "fire-and-forget" asynchronous task. This method should be used by clients in preference to call() and call(Object) when no callbacks, or other features of Promise, are needed.

The advantage of this method is that it allows for greater optimization of the underlying asynchronous task. Clients are therefore likely to see better performance when using this method compared to using call(Object) or call() and ignoring the returned Promise. The Promise returned by this method is different from the Promise returned by call(Object) or call(), in that the returned Promise will resolve when the fire-and-forget task is successfully started, or fail if the task cannot be started. Note that there is no *happens-before* relationship and the returned Promise may resolve before or after the fire-and-forget task starts, or completes.

Typically this method is used like call():

```
ServiceReference<I> s = ...;
I i = async.mediate(s, I.class);
i.someMethod()
Promise<Void> p = async.execute();
```

Returns A Promise representing whether the fire-and-forget task was able to start.

138.9.2.4   public T mediate(T target,Class<T> iface)

Type Arguments <T>

target The service object to mediate.

iface The type that the mediated object should provide.

☐ Create a mediator for the specified object. The mediator is a generated object that registers the method calls made against it. The registered method calls can then be run asynchronously using either the call(Object), call(), or execute() method.

The values returned by method calls made on a mediated object are opaque and should not be interpreted.

Normal usage:

```
I s = ...;
I i = async.mediate(s, I.class);
Promise<String> p = async.call(i.foo());
```

Returns A mediator for the service object.

Throws IllegalArgumentException– If the type represented by iface cannot be mediated.

138.9.2.5 **public T mediate(ServiceReference<? extends T> target,Class<T> iface)**

Type Arguments <T>

target The service reference to mediate.

iface The type that the mediated object should provide.

☐ Create a mediator for the specified service. The mediator is a generated object that registers the method calls made against it. The registered method calls can then be run asynchronously using either the call(Object), call(), or execute() method.

The values returned by method calls made on a mediated object are opaque and should not be interpreted.

This method differs from mediate(Object, Class) in that it can track the availability of the specified service. This is recommended as the preferred option for mediating OSGi services as asynchronous tasks may not start executing until some time after they are requested. Tracking the validity of the ServiceReference for the service ensures that these tasks do not proceed with an invalid object.

Normal usage:

```
ServiceReference<I> s = ...;
I i = async.mediate(s, I.class);
Promise<String> p = async.call(i.foo());
```

Returns A mediator for the service object.

Throws IllegalArgumentException– If the type represented by iface cannot be mediated.

138.10   org.osgi.service.async.delegate

Asynchronous Services Delegation Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package contains only interfaces that are implemented by consumers.

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.async.delegate; version="[1.0,2.0)"

138.10.1   Summary

- AsyncDelegate - This interface is used by services to allow them to optimize Asynchronous calls where they are capable of executing more efficiently.

138.10.2   public interface AsyncDelegate

This interface is used by services to allow them to optimize Asynchronous calls where they are capable of executing more efficiently.

This may mean that the service has access to its own thread pool, or that it can delegate work to a remote node, or act in some other way to reduce the load on the Asynchronous Services implementation when making an asynchronous call.

138.10.2.1   public Promise<?> async(Method m,Object[] args) throws Exception

m   The method to be asynchronously invoked.

args   The arguments to be used to invoke the method.

☐   Invoke the specified method as an asynchronous task with the specified arguments.

This method can be used by clients, or the Async Service, to optimize Asynchronous execution of methods.

When called, this method should invoke the supplied method using the supplied arguments asynchronously, returning a Promise that can be used to access the result.

If the method cannot be executed asynchronously by this method then null must be returned.

Returns   A Promise representing the asynchronous result, or null if this method cannot be asynchronously invoked.

Throws   Exception– An exception should be thrown only if there was a serious error that prevented the asynchronous task from starting. For example, the specified method does not exist on this object. Exceptions must not be thrown to indicate that the call does not support asynchronous invocation. Instead this method must return null. Exceptions must also not be thrown to indicate a failure from the execution of the underlying method. This must be handled by failing the returned Promise.

138.10.2.2   public boolean execute(Method m,Object[] args) throws Exception

m   The method to be asynchronously invoked.

args   The arguments to be used to invoke the method.

☐   Invoke the specified method as a "fire-and-forget" asynchronous task with the specified arguments.

This method can be used by clients, or the Async Service, to optimize Asynchronous execution of methods.

When called, this method should invoke the specified method using the specified arguments asynchronously. This method differs from async(Method, Object[]) in that it does not return a Promise. This method therefore allows the implementation to perform more aggressive optimizations because the end result of the invocation does not need to be returned to the caller.

If the method cannot be executed asynchronously by this method then false must be returned.

Returns   true if the asynchronous execution request has been accepted, or false if this method cannot be asynchronously invoked by the AsyncDelegate.

Throws   Exception– An exception should be thrown only if there was a serious error that prevented the asynchronous task from starting. For example, the specified method does not exist on this object. Exceptions must not be thrown to indicate that the call does not support asynchronous invocation. Instead this method must return false. Exceptions must also not be thrown to indicate a failure from the execution of the underlying method.

138.11   References

[1]   *OSGi Core Release 6*

http://www.osgi.org/Specifications/HomePage

140   Http Whiteboard Specification

Version 1.0

140.1   Introduction

Servlets have become a popular and widely supported mechanism for providing dynamic content on the Internet. While servlets are defined in the [4] *Java Servlet 3.1 Specification*, the OSGi Http Whiteboard Specification provides a light and convenient way of using servlets, servlet filters, servlet listeners and web resources in an OSGi environment through the use of the [7] *Whiteboard Pattern*.

The Http Whiteboard specification supports:

- *Registering Servlets* - Registering a servlet in the Service Registry makes it available to be bound to an endpoint to serve content over the network.
- *Registering Servlet Filters* - Servlet filters support pre- and post-processing of servlet requests and responses. Servlet filters can be registered in the Service Registry to include them in the handling pipeline.
- *Registering Resources* - Resources such as HTML files, JavaScript, image files, and other static resources can be made available over the network by registering resource services.
- *Registering Servlet Listeners* - The servlet specification defines a variety of listeners, which receive callbacks when certain events take place.

Implementations of this specification can support the following versions of the HTTP protocol:

- [1] *HTTP 1.0 Specification RFC-1945*
- [2] *HTTP 1.1 Specifications RFCs 7230-7235*
- [3] *HTTP/2 Specifications*

Alternatively, implementations of this service can support other protocols if these protocols can conform to the semantics of the Java Servlet API.

Http Whiteboard implementations must support version 3.1 of the Java Servlet API.

140.1.1   Entities
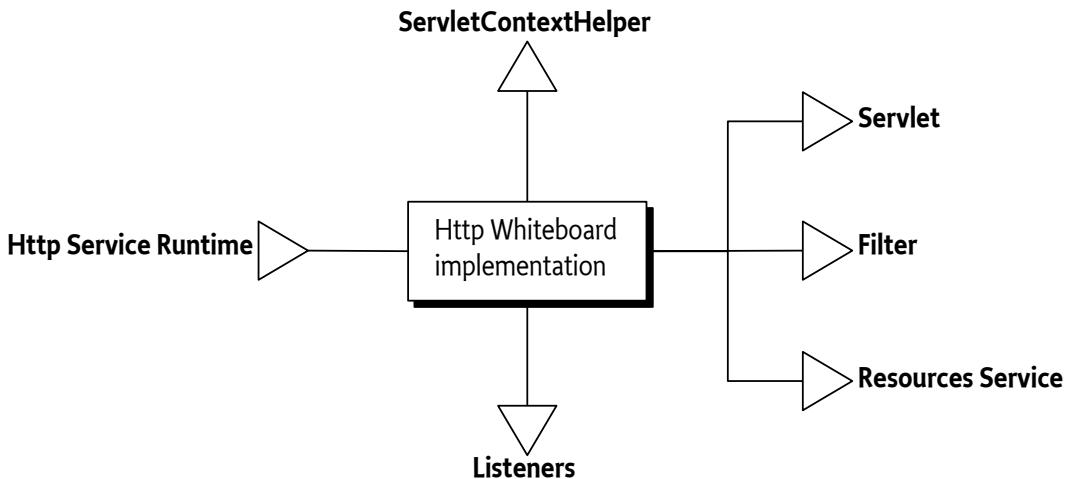
This specification defines the following entities:

- *Http Whiteboard service* - An object registered in the Service Registry under one of the Whiteboard service interfaces defined by this specification.
- *Http Whiteboard implementation* - An implementation that processes Http Whiteboard services.
- *Http Service Runtime service* - Service providing runtime introspection into the Http Whiteboard implementation.
- *Listener* - Various listeners can be registered to receive notifications about servlet or Http Session events.
- *Resource Service* - A service thats binds static resources.
- *Servlet* - Component that dynamically generates web pages or other resources provided over the network.

- *Servlet Context Helper* - A service to control the behavior of the Servlet Context.
- *Servlet Filter* - Can be used to augment or transform web resources or for cross-cutting functionality such as security, common widgets or otherwise.

Figure 140.1   *Http Whiteboard Overview Diagram*

140.2   The Servlet Context

The servlet specification defines the ServletContext which is provided to servlets at runtime by the container. Whiteboard services defined by this specification are also provided with a ServletContext. The behavior of this Servlet Context can be influenced by providing a ServletContextHelper service. A custom ServletContextHelper can provide resources, mime-types, handle security and supports a number of methods from the ServletContext.

The Http Whiteboard implementation must create a separate ServletContext instance for each ServletContextHelper service. Whiteboard services can be associated with the Servlet Context Helper by using the osgi.http.whiteboard.context.select property. If this property is not set, the *default* Servlet Context Helper is used.

To achieve the required behavior for ServletContext.getClassLoader() each bundle must be provided with a separate Servlet Context instance to serve the classloader of the Whiteboard services for that bundle. For more information see getClassLoader() in Table 140.2 on page 759.

Some implementations of the ServletContextHelper may be implemented using a Service Factory, for example to provide resources from the associated bundle, as the *default* implementation does. Therefore the Whiteboard implementation must get the Servlet Context Helper using the Bundle Context of the bundle that registered the Whiteboard service.

Some environments may use [8] *Core Service Hooks* to isolate ServletContextHelper service registrations. For example, *Subsystem Service Specification* on page 613. The Whiteboard implementation must check that the bundle registering the Whiteboard service has the ability to find the ServletContextHelper service before allowing the Whiteboard service to bind to the Servlet Context Helper. This can be done by calling one of the getServiceReferences methods on the Bundle Context of bundle that registered the Whiteboard service.

Table 140.1   *Service registration properties for* ServletContextHelper *services.*

Service Property Name	Type	Description
osgi.http.whiteboard.context.name	String *required*	Name of the Servlet Context Helper. This name can be referred to by Whiteboard services via the osgi.http.whiteboard.context.select property. The syntax of the name is the same as the syntax for a Bundle Symbolic Name. The default Servlet Context Helper is named default. To override the default, register a custom ServletContextHelper service with the name default. If multiple Servlet Context Helper services are registered with the same name, the one with the highest Service Ranking is used. In case of a tie, the service with the lowest service ID wins. In other words, the normal OSGi service ranking applies.
		Registrations with an invalid or unspecified name are not used and reflected in the failure DTOs. See HTTP_WHITEBOARD_CONTEXT_NAME.
osgi.http.whiteboard.context.path	String *required*	Additional prefix to the context path for servlets. This property is mandatory. Valid characters are specified in IETF RFC 3986, section 3.3. The context path of the default Servlet Context Helper is /. A custom default Servlet Context Helper may use an alternative path. If the path is invalid or unspecified, the service is not used and reflected in the failure DTOs. See HTTP_WHITEBOARD_CONTEXT_PATH.
context.init.*	String *optional*	Properties starting with this prefix are provided as init parameters through the ServletContext.getInitParameter() and ServletContext.getInitParameterNames() methods. The context.init. prefix is removed from the parameter name. See HTTP_WHITEBOARD_CONTEXT_INIT_PARAM_PREFIX.

Multiple ServletContextHelper services can have identical or overlapping osgi.http.whiteboard.context.path values. A matching servlet or resource is located as follows:

1. The Servlet Context Helper service with the longest matching path is matched first.
2. In the case of two Servlet Context Helpers with the same path, the service with the highest ranking is searched first for a match. In the case of a tie, the lowest service ID is searched first.

For example, if two ServletContextHelper services are registered as follows

```
osgi.http.whiteboard.context.path = /foo
osgi.http.whiteboard.context.path = /foo/bar
```

Then a request for http://localhost/foo/bar/someServlet is looked up in the following order:

1. /foo/bar context looking for a pattern to match /someServlet
2. /foo context looking for a pattern to match /bar/someServlet

Note that whole path segments must match. Therefore the following request can only be handled by the Servlet Context Helper registered under the /foo path: http://localhost/foo/bars/someOtherServlet.

For details on the association process between servlet, servlet filter, resource and listener services and the ServletContextHelper see *Common Whiteboard Properties* on page 761.

If a Servlet Context Helper can not be used, for example because it is shadowed by another Servlet Context Helper service with the same name, but with a higher ranking, this is reflected in the

FailedServletContextDTO. Similarly, if an alternative default Servlet Context Helper is provided, the default Servlet Context Helper provided by the Http Whiteboard implementation is not used and represented in a failure DTO.

An example Servlet Context Helper defined using Declarative Services annotations can be found below, it prefixes the path with /myapp for any associated whiteboard service. Additionally, it serves static resources from a non-standard location, a content delivery network. Other methods use the default ServletContextHelper implementation.

```
@Component(service = ServletContextHelper.class,
    scope = ServiceScope.BUNDLE,
    property = {
        "osgi.http.whiteboard.context.name=my-context",
        "osgi.http.whiteboard.context.path=/myapp"})
public class CDNServletContextHelper extends ServletContextHelper {
    public URL getResource(String name) {
        try {
            return new URL("http://acmecdn.com/myapp/" + name);
        } catch (MalformedURLException e) {
            return null;
        }
    }
}
```

The following sections outline the methods a custom ServletContextHelper can override and the behavior of the *default* implementation.

140.2.1   String getMimeType(String)

Called to provide the MIME type for a resource.

Default Behavior - Always returns null.

140.2.2   String getRealPath(String)

Called to support the ServletContext.getRealPath() method.

Default Behavior - Always returns null.

140.2.3   URL getResource(String)

Obtain a URL for a given resource request.

Default Behavior - Assumes the resources are in the bundle registering the Whiteboard service. Its Bundle.getEntry() method is called to obtain a URL to the resource. The default Servlet Context Helper implementation assumes the path to be relative to the bundle's root.

140.2.4   Set‹String› getResourcePaths(String)

Called to support the ServletContext.getResourcePaths() method. Returns all the matching resources for the path.

Default Behavior - Assumes the resources are in the bundle registering the Whiteboard service. Its Bundle.findEntries() method is called to obtain the listing.

140.2.5   boolean handleSecurity(HttpServletRequest, HttpServletResponse)

This method is invoked to handle implementation-defined security on the request. It is invoked before the request is sent to the filter-servlet pipeline.

Default Behavior - Always returns true.

140.2.6 Behavior of the Servlet Context

The ServletContext provided to Whiteboard services is based on the associated ServletContextHelper, Whiteboard service registration properties and the underlying servlet container.

Methods to programmatically add servlets, servlet filters and listeners are not supported on the ServletContext. Such functionality is available by registering these entities as Whiteboard services.

Table 140.2   *Behavior of ServletContext methods.*

ServletContext method	Since	Description
addFilter(...)	3.0	Throws UnsupportedOperationException.
addListener(...)	3.0	Throws UnsupportedOperationException.
addServlet(...)	3.0	Throws UnsupportedOperationException.
createFilter(Class)	3.0	Throws UnsupportedOperationException.
createListener(Class)	3.0	Throws UnsupportedOperationException.
createServlet(Class)	3.0	Throws UnsupportedOperationException.
declareRoles(String ...)	3.0	Throws UnsupportedOperationException.
getAttribute(String)	2.0	Stored per ServletContextHelper. The Servlet Context keeps a set of attributes per Servlet Context Helper.
getAttributeNames()	2.1	Stored per ServletContextHelper. The Servlet Context keeps a set of attributes per Servlet Context Helper.
getClassLoader()	3.0	Returns the class loader of the bundle that registered the Whiteboard service. An implementation of this specification can achieve this by returning separate façades of the ServletContext to each Whiteboard service. Each façade accesses the Whiteboard service's Bundle Wiring to obtain its classloader.
getContext(String)	2.1	Backed by the Servlet Container.
getContextPath()	2.5	Return the web context path of the Servlet Context. This takes into account the osgi.http.whiteboard.context.path of the Servlet Context Helper and the path of the Http runtime.
getDefaultSessionTrackingModes()	3.0	Backed by the Servlet Container.
getEffectiveMajorVersion()	3.0	Backed by the Servlet Container.
getEffectiveMinorVersion()	3.0	Backed by the Servlet Container.
getEffectiveSessionTracking-Modes()	3.0	Backed by the Servlet Container.
getFilterRegistration(String)	3.0	Backed by the Servlet Container.
getFilterRegistrations()	3.0	Backed by the Servlet Container.
getInitParameter(String)	2.2	From context.init.* service registration properties.
getInitParameterNames()	2.2	From context.init.* service registration properties.
getJspConfigDescriptor()	3.0	Returns null.
getMajorVersion()	2.1	Backed by the Servlet Container.
getMimeType(String)	2.1	Backed by the ServletContextHelper.
getMinorVersion()	2.1	Backed by the Servlet Container.

| ServletContext method	Since	Description
getNamedDispatcher(String)	2.2	Provides the Whiteboard servlet with the specified name, provided through the osgi.http.whiteboard.servlet.name property, if associated with this Servlet Context Helper. If multiple servlets have the same name and are associated with this Servlet Context Helper then the highest ranked servlet is used. In the case of a tie, the one with the lowest service ID is used.
getRealPath(String)	2.0	Backed by the ServletContextHelper.
getResource(String)	2.1	Backed by the ServletContextHelper.
getRequestDispatcher(String)	2.1	If the argument matches a servlet associated with this Servlet Context Helper, this will be returned.
getResourceAsStream(String)	2.1	Backed by the ServletContextHelper.
getResourcePaths(String)	2.3	Backed by the ServletContextHelper.
getServlet(String)	2.0	Deprecated. Backed by the Servlet Container.
getServletContextName()	2.2	The name of the ServletContextHelper provided via the osgi.http.whiteboard.context.name service property.
getServletNames()	2.0	Deprecated. Backed by the Servlet Container.
getServletRegistration(String)	3.0	Backed by the Servlet Container.
getServletRegistrations()	3.0	Backed by the Servlet Container.
getServlets()	2.0	Deprecated. Backed by the Servlet Container.
getServerInfo()	2.0	Backed by the Servlet Container.
getSessionCookieConfig()	3.0	Returns a SessionCookieConfig object. This object is read-only and all setters throw a IllegalStateException.
getVirtualServerName()	3.1	Backed by the Servlet Container.
log(String)	2.0	Backed by the Servlet Container.
log(Exception, String)	2.0	Deprecated. Backed by the Servlet Container.
log(String, Throwable)	2.1	Backed by the Servlet Container.
removeAttribute(String)	2.1	Stored per ServletContextHelper. The Servlet Context keeps a set of attributes per Servlet Context Helper.
setAttribute(String, Object)	2.1	Stored per ServletContextHelper. The Servlet Context keeps a set of attributes per Servlet Context Helper.
setInitParameter(String, String)	3.0	Throws IllegalStateException. The ServletContext has already been initialized.
setSessionTrackingModes(Set)	3.0	Throws IllegalStateException. The ServletContext has already been initialized.
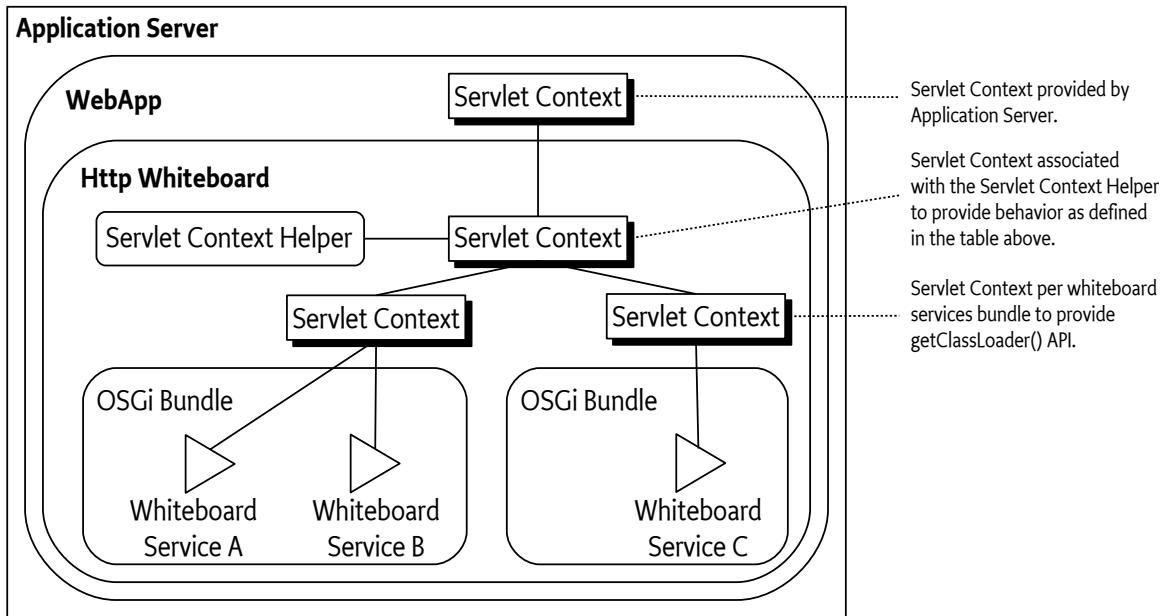
140.2.7 Relation to the Servlet Container

Implementations of this specification will often be backed by existing servlet containers or a Java EE application server. There may also exist implementations which bridge into a servlet container into which the OSGi Framework has been deployed as a Web Application.

In bridged situations the Http Whiteboard implementation will live in one servlet context and all Whiteboard services registered by this implementation will be backed by the same underlying Servlet Context. However, to exhibit the behavior described in Table 140.2 on page 759 different Servlet Context objects may be required. Therefore an implementation of this specification may

need to create additional ServletContext objects which delegate certain functionality to the Servlet-ContextHelper and other functionality to the Servlet Context of the Web Application, yet further functionality can be obtained otherwise. In such cases the relationship may look like the below figure.

Figure 140.2   Servlet Context entities and their relation

Where Table 140.2 on page 759 states *Backed by the Servlet Container* and the Http Whiteboard implementation is deployed in bridged mode, the API call can be forwarded to the top-level Servlet Context. If the Http Whiteboard implementation is not deployed in bridged mode, it must provide another means to handle these APIs.

In bridged deployments, the implementation needs to ensure the following:

1.   That Whiteboard services are provided with the correct ServletContext keeping in mind that each distinct ServletContextHelper should be associated with a separate ServletContext object, which in turn may delegate certain requests to the underlying shared ServletContext as described in the table above.

2.   That Http Sessions are not shared amongst servlets registered with different ServletContextHelpers. That is, HttpRequest.getSession() calls must provide different sessions per associated ServletContextHelper. Http Sessions are defined in chapter 7 of the [4] *Java Servlet 3.1 Specification.*

140.3   Common Whiteboard Properties

Whiteboard servlet, servlet filter, resource and listener services support common service registration properties to associate them with a ServletContextHelper and/or a Http Whiteboard implementation.

Table 140.3   *Common properties*

Service Property Name	Type	Description

osgi.http.whiteboard.context.select	String
	
	optional
		An LDAP-style filter to select the associated ServletContextHelper service to use. Any service property of the Servlet Context Helper can be filtered on. If this property is missing the default Servlet Context Helper is used.
		
		For example, to select a Servlet Context Helper with name myCTX provide the following value:
		
		`(osgi.http.whiteboard. context.name=myCTX)`
		
		To select all Servlet Context Helpers provide the following value:
		
		`(osgi.http.whiteboard. context.name=*)`
		
		If no matching context exists this is reflected in the failure DTOs. See HTTP_WHITEBOARD_CONTEXT_SELECT.

osgi.http.whiteboard.target	String *optional*	The value of this service property is an LDAP-style filter expression to select the Http Whiteboard implementation(s) to handle this Whiteboard service. The LDAP filter is used to match HttpServiceRuntime services. Each Http Whiteboard *implementation* exposes exactly one HttpServiceRuntime service. This property is used to associate the Whiteboard service with the Http Whiteboard implementation that registered the HttpServiceRuntime service. If this property is not specified, all Http Whiteboard implementations can handle the service. See HTTP_WHITEBOARD_TARGET.

If multiple Servlet Context Helper services match the osgi.http.whiteboard.context.select property the servlet, filter, resource or listener will be registered with all these Servlet Context Helpers. To avoid multiple init() and destroy() calls on the same instance, servlets and filters should be registered as Prototype Service Factory.

140.4   Registering Servlets

Servlets can be registered with the Http Whiteboard implementation by registering them as Whiteboard services. This means that Servlet implementations are registered in the Service Registry under the javax.servlet.Servlet interface.

Servlets are registered with one or more pattern through the osgi.http.whiteboard.servlet.pattern service property. Each pattern defines the URL context that will trigger the servlet to handle the request. They are defined by the [4] *Java Servlet 3.1 Specification* in section 12.2, *Specification of Mappings*. Note that these mapping rules are slightly different than those defined in the *Http Service Specification* on page 45. The mapping rules are:

- A string beginning with a '/' character and ending with a "/∗" suffix is used for path mapping.
- A string beginning with a "∗." prefix is used as an extension mapping.

- The empty string ("") is a special URL pattern that exactly maps to the application's context root. That is, requests of the form http://host:port/‹context-root›/. In this case the path info is "/" and the servlet path and context path are the empty string ("").
- A string containing only the '/' character indicates the "default" servlet of the application. In this case, the servlet path is the request URI minus the context path and the path info is null.
- All other strings are used for exact matches only.

Servlet and resource service registrations associated with a single Servlet Context share the same namespace. In case of identical registration patterns, service ranking rules are used to select the service handling a request. That is, Whiteboard servlets that have patterns shadowed by other Whiteboard services associated with the *same* Servlet Context are represented in the failure DTOs.

The above rules can cause servlets that are already bound becoming unbound if a better match arrives. This ensures a predictable end result regardless of the order in which services are registered.

A servlet may be registered with the property osgi.http.whiteboard.servlet.name which can be used by servlet filters to address this servlet. If the servlet service does not have this property, the servlet name defaults to the fully qualified class name of the service object.

With implementations that both implement this specification as well as the *Http Service Specification* on page 45, situations can arise where a servlet is registered for the same pattern with the Http Service as well as with the Http Whiteboard. The Servlet Context of the Http Service is treated in the same way as all contexts managed by the Whiteboard implementation. The highest ranking is associated with the context of the Http Service. For a request, contexts are processed in the order as described in section *The Servlet Context* on page 756.

For example, if the Http Whiteboard implementation is listening on port 80 on the machine www.acme.com and the Servlet object is registered with the pattern "/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

The following table describes the properties that can be used by Servlets registered as Whiteboard services. Additionally, the common properties listed in Table 140.3 on page 762 are supported.

Table 140.4   *Service properties for Servlet Whiteboard services.*

Service Property Name	Type	Description
osgi.http.whiteboard.servlet.« asyncSupported	String *optional*	Declares whether the servlet supports the asynchronous operation mode. Allowed values are true and false independent of case. Defaults to false. See HTTP_WHITEBOARD_SERVLET_ASYNC_SUPPORTED.
osgi.http.whiteboard.servlet.« errorPage	String+ *optional*	Register the servlet as an error page for the error code and/or exception specified; the value may be a fully qualified exception type name or a three-digit HTTP status code in the range 400-599. Special values 4xx and 5xx can be used to match value ranges. Any value not being a three-digit number is assumed to be a fully qualified exception class name. See HTTP_WHITEBOARD_SERVLET_ERROR_PAGE.
osgi.http.whiteboard.servlet.« name	String *optional*	The name of the servlet. This name is used as the value of the javax.servlet.ServletConfig.getServletName() method and defaults to the fully qualified class name of the service object. See HTTP_WHITEBOARD_SERVLET_NAME.

Service Property Name	Type	Description
osgi.http.whiteboard.servlet.« pattern	String+ *required*	Registration pattern(s) for the servlet. See HTTP_WHITEBOARD_SERVLET_PATTERN.
servlet.init.*	String *optional*	Properties starting with this prefix are provided as init parameters to the javax.servlet.Servlet.init() method. The servlet.init. prefix is removed from the parameter name. See HTTP_WHITEBOARD_SERVLET_INIT_PARAM_PREFIX.

Servlet objects are initialized by a Http Whiteboard implementation before they start serving requests. 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. Once the servlet is no longer used by the Http Whiteboard implementation the destroy() method is called. Failure during Servlet.init() will prevent the servlet from being used, which is reflected using a failure DTO. In such a case the system does *not* attempt to find an alternative servlet which might be present in the system with a lower ranking.

If the service properties of the servlet Whiteboard service are modified, the destroy() method is called. Subsequently the servlet is re-initialized. If a Prototype Service Factory is used for the servlet this re-initialization is done on a new service object.

When multiple Http Whiteboard implementations are present all of them can potentially process the Servlet. In such situations it can be useful to associate the servlet with a specific implementation by specifying the osgi.http.whiteboard.target property on the Servlet service to match its HttpServiceRuntime service.

If more than one Http Service Runtime matches the osgi.http.whiteboard.target property or the property is not set, the Servlet will be processed by all the matching implementations. A Servlet service that is processed by more than one Http Whiteboard implementation will have its init() method called for each implementation that processes this Servlet. Similarly, the destroy() method is called once when the Servlet is shut down once for each implementation that processed it. As multiple init() and destroy() calls on the same Servlet instance are generally not desirable, Servlet implementations should be registered as Prototype Service Factories as defined in the *OSGi Core Release 6*. This will ensure that each Http Whiteboard implementation processing the Servlet will use a separate instance, ensuring that only one init() and destroy() call is made per Servlet object. Servlets not registered as a Prototype Service Factory may received init() and destroy() calls multiple times on the same service object.

The following example code uses Declarative Services annotations to register a servlet whiteboard service.

```
@Component(service = Servlet.class,
    scope = ServiceScope.PROTOTYPE,
    property = {
        "osgi.http.whiteboard.servlet.pattern=/myservlet",
        "servlet.init.myname=value"})
public class MyServlet extends HttpServlet {
    private String name = "<not set>";

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

    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws IOException {
        resp.setContentType("text/plain");
```

```
                    resp.getWriter().println("Servlet name: " + name);
        }
    }
```

This example registers the servlet at: /myservlet. Requests for http://www.acme.com/myservlet map to the servlet, whose `service` method is called to process the request.

To associate the above example servlet with the example ServletContextHelper in *The Servlet Context* on page 756, add the following service property:

```
osgi.http.whiteboard.context.select=(osgi.http.whiteboard.context.name=my-context)
```

This will cause the servlet to move to http://www.acme.com/myapp/myservlet as configured by the custom Servlet Context Handler.

140.4.1 Error Pages

Servlets can be used to serve Error Pages. These are invoked when an exception is thrown during processing or if a servlet uses the `sendError()` method with a status code between 400 and 599.

For a servlet service to handle error situations the service property `osgi.http.whiteboard.servlet.errorPage` must be set. This property can have multiple values, allowing a single servlet to handle a variety of error situations. Possible values are 3-digit HTTP error codes and fully qualified exception names.

Two special error code values are recognized. The special value `4xx` means every error code in the 400-499 range. The special value `5xx` means every error code in the 500-599 range. To override such wildcard error page for a specific error, register an error page with the specific error code and a higher service ranking. Error pages shadowed by other error pages are reported via the failure DTOs. A `4xx`/`5xx` wildcard error page is only reported in the failure DTOs if it is shadowed by another wildcard page.

Matching exceptions follows the exception hierarchy. First the most specific exception class - the actual class of the exception - is looked up. If no matching error page for the most specific exception is found, the error page for the super class of the exception is looked up and so on. The process ends by looking up an error page for the `java.lang.Throwable` class.

While not being common practice, it is possible to combine the `osgi.http.whiteboard.servlet.errorPage` and `osgi.http.whiteboard.servlet.pattern` properties. If a single servlet registration has both these registration properties it is considered both an ordinary servlet as well as an error page.

If an error or exception occurs for which an error page servlet can be matched, it is invoked to render the error page. If the error page servlet causes an error or exception while handling the request, an implementation built-in error page is returned.

For example:

```
@Component(service = Servlet.class,
    scope = ServiceScope.PROTOTYPE,
    property = {
        "osgi.http.whiteboard.servlet.errorPage=java.io.IOException",
        "osgi.http.whiteboard.servlet.errorPage=500"})
public class MyErrorServlet extends HttpServlet {
    ...
}
```

The example servlet is invoked in case of a 500 error code, or if an IOException (or subclass) occurs. If there is more than one error page registered for the same exception or error code, service ranking rules are used to select the handling servlet.

140.4.2   Asynchronous request handling

Servlets can use the asynchronous request handling feature, as defined by the servlet specification.

A servlet or servlet filter supporting the asynchronous mode must declare this with the appropriate service property osgi.http.whiteboard.servlet.asyncSupported or osgi.http.whiteboard.filter.asyncSupported.

An example simple asynchronous servlet that handles the servlet requests in a thread from a custom thread pool rather than in the thread provided by the servlet container:

```
@Component(service = Servlet.class,
    scope = ServiceScope.PROTOTYPE,
    property = {
        "osgi.http.whiteboard.servlet.pattern=/as",
        "osgi.http.whiteboard.servlet.asyncSupported=true"})
public class AsyncServlet extends HttpServlet {
    ExecutorService executor = Executors.newCachedThreadPool(
        r -> new Thread(r, "Pooled Thread"));

    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws IOException {
        doGetAsync(req.startAsync());
    }

    private void doGetAsync(AsyncContext asyncContext) {
        executor.submit(() -> {
            try {
                PrintWriter writer = asyncContext.getResponse().getWriter();
                writer.print("Servlet executed async in: " +
                    Thread.currentThread()); // writes 'Pooled Thread'
            } finally {
                asyncContext.complete();
            }
            return null;
        });
    }
}
```

140.4.3   Annotations

Annotations defined in the Servlet API Specifications are ignored by an implementation of the Http Whiteboard Specification. The OSGi service model is used instead by this specification.

Implementations of this specification *may* support these annotations through a proprietary opt-in mechanism.

140.5   Registering Servlet Filters

Servlet filters provide a mechanism to intercept servlet invocations. They support modifying the ServletRequest and ServletResponse objects and are often used to augment web pages generated by servlets, for example with a common header or footer. Servlet filters can also be used to handle security, do logging or transform the content produced by a servlet to a certain format.

Similar to servlets, servlet filters are registered as Whiteboard services, by registering a javax.servlet.Filter instance in the Service Registry. The following table describes the supported ser-

vice properties. In addition the common properties as described in Table 140.3 on page 762 are supported.

Table 140.5   *Service properties for* Filter *Whiteboard services.*

Service Property Name	Type	Description
osgi.http.whiteboard.filter.« asyncSupported	String *optional*	Declares whether the servlet filter supports asynchronous operation mode. Allowed values are true and false independent of case. Defaults to false. See HTTP_WHITEBOARD_FILTER_ASYNC_SUPPORTED.
osgi.http.whiteboard.filter.« dispatcher	String+ *optional*	Select the dispatcher configuration when the servlet filter should be called. Allowed string values are REQUEST, ASYNC, ERROR, INCLUDE, and FORWARD. The default for a filter is REQUEST. See HTTP_WHITEBOARD_FILTER_DISPATCHER.
osgi.http.whiteboard.filter.name	String *optional*	The name of a servlet filter. This name is used as the value of the FilterConfig.getFilterName() method and defaults to the fully qualified class name of the service object. See HTTP_WHITEBOARD_FILTER_NAME.
osgi.http.whiteboard.filter.pattern	String+ *optional†*	Apply this servlet filter to the specified URL path patterns. The format of the patterns is specified in the servlet specification. See HTTP_WHITEBOARD_FILTER_PATTERN.
osgi.http.whiteboard.filter.regex	String+ *optional†*	Apply this servlet filter to the specified URL paths. The paths are specified as regular expressions following the syntax defined in the java.util.regex.Pattern class. See HTTP_WHITEBOARD_FILTER_REGEX.
osgi.http.whiteboard.filter.servlet	String+ *optional†*	Apply this servlet filter to the referenced servlet(s) by name. See HTTP_WHITEBOARD_FILTER_SERVLET.
filter.init.*	String+ *optional*	Properties starting with this prefix are passed as init parameters to the Filter.init() method. The filter.init. prefix is removed from the parameter name. See HTTP_WHITEBOARD_FILTER_INIT_PARAM_PREFIX.

† Note that at least one of the following properties *must* be specified on Filter Whiteboard services:

```
osgi.http.whiteboard.filter.pattern
osgi.http.whiteboard.filter.regex
osgi.http.whiteboard.filter.servlet
```

Similar to servlets, Filter objects are initialized by a Http Whiteboard implementation before they start filtering requests. The initialization is done by calling the Filter.init(FilterConfig) method. The FilterConfig parameter provides access to filter.init.* properties on the servlet filter service registration. Once the Filter is no longer used by the Http Whiteboard implementation, the destroy() method is called. When the service properties on the servlet filter are modified, the destroy() method is called and the servlet filter is subsequently re-initialized, if it can still be associated with a Http Whiteboard implementation after the modification. By default, a servlet filter can be used with any Servlet Context Helper or Http Whiteboard implementation. To restrict a servlet filter to a single implementation or a specific Servlet Context Helper, the *Common Whiteboard Properties* on page 761 can be used.

To deal with the dynamicity of the Whiteboard service lifecycle, it is recommended to implement a servlet filter as Prototype Service Factory service. This will ensure that one single servlet filter instance only receives one init() and one destroy() call. Otherwise a single servlet filter instance can receive multiple such calls. This is similar to the behavior recommended for Servlet Whiteboard services.

Multiple servlet filters can process the same servlet request/response. If more than one Filter matches, the order in which they are processed is governed by their service ranking. The servlet filter with the highest ranking is processed first in the filter chain, while the servlet filter with the lowest ranking is processed last, before the Servlet.service() method is called. In the case of a service ranking tie, the servlet filter with the lowest service.id is processed first. After the servlet completes its service() method the filter chain is unwound in reverse order.

Servlet filters are only applied to servlet requests if they are bound to the same Servlet Context Helper and the same Http Whiteboard implementation.

The example Filter below adds some text before and after the content generated by a servlet:

```
@Component(scope = ServiceScope.PROTOTYPE,
    property = "osgi.http.whiteboard.filter.pattern=/*")
public class MyFilter implements Filter {
    public void init(FilterConfig filterConfig) throws ServletException {}

    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        response.getWriter().write("before");
        chain.doFilter(request, response);
        response.getWriter().write("after");
    }

    public void destroy() {}
}
```

140.6 Registering Resources

A resource is a file containing images, static HTML pages, JavaScript, CSS, sounds, movies, 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 requester.

Resources can be served by registering a service of any type with a service registration property that marks it as a resource service: osgi.http.whiteboard.resource.pattern. The actual service object registered is not used to serve resources, it is merely used to inform the Http Whiteboard implementation to serve resources from a certain source.

The following table describes the supported service properties. In addition the common properties as described in Table 140.3 on page 762 are supported.

Table 140.6   *Service properties for resource services.*

Service Property Name	Type	Description
osgi.http.whiteboard.resource.« pattern	String+ *required*	The pattern(s) to be used to serve resources. As defined by the [4] *Java Servlet 3.1 Specification* in section 12.2, *Specification of Mappings*. This property marks the service as a resource service. See HTTP_WHITEBOARD_RESOURCE_PATTERN.

Service Property Name	Type	Description
osgi.http.whiteboard.resource.« prefix	String *required*	The prefix used to map a requested resource to the bundle's entries. If the request's path info is not null, it is appended to this prefix. The resulting string is passed to the getResource(String) method of the associated Servlet Context Helper. See HTTP_WHITEBOARD_RESOURCE_PREFIX.

The examples below use Declarative Services annotations to register a resources service. Note that this service is purely used to convey information to the Http Whiteboard implementation and is never invoked.

```
@Component(service = MyResourceService.class,
    property = {
        "osgi.http.whiteboard. resource.pattern=/files/*",
        "osgi.http.whiteboard. resource.prefix=/www"})
public class MyResourceService {}
```

A Http Whiteboard implementation configured on port 80 will serve a request for http://local-host/files/cheese.html from the location /www/cheese.html.

The following example maps requests for /favicon.ico to serve the /logo.png resource. Note that the pattern is not appended to the prefix as the path info in this case is null.

```
@Component(service = SomeResourceService.class,
    property = {
        "osgi.http.whiteboard. resource.pattern=/favicon.ico",
        "osgi.http.whiteboard. resource.prefix=/logo.png"})
public class SomeResourceService {}
```

The above examples use the default ServletContextHelper implementation, which loads these resources from the bundle that registered the resource service. For more control around serving resources, a resources service can be associated to a custom ServletContextHelper. For example, a custom Servlet Context Helper can serve resources from locations other than the current bundle.

140.6.1   Overlapping Resource and Servlet registrations

Resources and servlets registered with the same Servlet Context share a single URI namespace. This means that the value specified in osgi.http.whiteboard.resource.pattern competes with the osgi.http.whiteboard.servlet.pattern property specified on servlets. If these values overlap, the rules as outlined in *Registering Servlets* on page 762 are used to resolve conflicts, where resource services are treated just like servlets. Shadowed resource patterns are reported as FailedResourceD-TO.

140.7   Registering Listeners

The servlet specification defines listener interfaces that can be implemented to receive a variety of servlet-related events. When using the Http Whiteboard implementation these listeners can be registered as Whiteboard services.

- ServletContextListener - Receive notifications when Servlet Contexts are initialized and destroyed.
- ServletContextAttributeListener - Receive notifications for Servlet Context attribute changes.
- ServletRequestListener - Receive notifications for servlet requests coming in and being destroyed.

- ServletRequestAttributeListener - Receive notifications when servlet Request attributes change.
- HttpSessionListener - Receive notifications when Http Sessions are created or destroyed.
- HttpSessionAttributeListener - Receive notifications when Http Session attributes change.
- HttpSessionIdListener - Receive notifications when Http Session ID changes.

Events are sent to listeners registered in the Service Registry with the osgi.http.whiteboard.listener service property set to true, independent of case. Listeners can be associated with a ServletContext-tHelper as described in *Common Whiteboard Properties* on page 761. Listeners not specifically associated with a Servlet Context Helper will receive events relating to the *default* Servlet Context Helper.

Multiple listeners of the same type registered with a given Servlet Context Helper are invoked in sequence, service ranking rules are used to determine the order.

Table 140.7   *Service properties for listener services.*

Service Property Name	Type	Description
osgi.http.whiteboard.listener	String *required*	When set to true this listener service is handled by the Http Whiteboard implementation. When not set or set to false the service is ignored. Any other value is invalid and will be reflected in a FailedListenerDTO. Note the property value is case independent. See HTTP_WHITEBOARD_LISTENER.

An example listener that reports on client requests being initialized and destroyed is listed below:

```
@Component(property = "osgi.http.whiteboard.listener=true")
public class MyServletRequestListener implements ServletRequestListener {
    public void requestInitialized(ServletRequestEvent sre) {
        System.out.println("Request initialized for client: " +
            sre.getServletRequest().getRemoteAddr());
    }

    public void requestDestroyed(ServletRequestEvent sre) {
        System.out.println("Request destroyed for client: " +
            sre.getServletRequest().getRemoteAddr());
    }
}
```

For more details on the behavior of the listeners see the [4] *Java Servlet 3.1 Specification.*

140.8 Lifecycle

If a Whiteboard service is used by a Http Whiteboard implementation, the following order of actions are performed:

1. The service is obtained from the service registry.
2. For servlets and servlet filters, init() is called.

When the service is not used anymore, these actions are performed:

3. For servlets and servlet filters, destroy() is called.
4. The service is released.

Note that some of the above actions may not be performed immediately, allowing an implementation to utilize lazy or asynchronous behavior.

As servlets and servlet filters services might come and go as well as ServletContextHelper services might come and go, use of the Whiteboard services can be very dynamic. Therefore servlet and servlet filter services might transition between bound to a Http Whiteboard implementation to being unbound and back to be bound. For example, when a matching Servlet Context Helper with the same name arrives with a higher ranking than the currently bound Servlet Context Helper, the servlet will be destroyed and re-initialized, bound to this better matching Servlet Context Helper. This is to ensure that timing issues cannot dictate the topology of the system.

As init() and destroy() are called each time the service life cycle changes, the recommended way to register services is to use the Prototype Service scope as defined in the *OSGi Core Release 6*. This ensures a new instance is created for each time such service is re-initialized. If the prototype scope is not used, the service should be prepared that after a call to destroy() a new initialization through init() might follow.

140.8.1 Whiteboard Service Dynamics and Active Requests

When the Http Whiteboard implementation receives a network request it establishes the processing pipeline based on the available Whiteboard services (servlets, servlet filters and resource services) and executes this pipeline. Between establishing the pipeline and finishing the processing, services used in this pipeline might become unregistered. It is up to the Http Whiteboard implementation whether it completes the active request or throws a Servlet Exception in this case.

140.9 The Http Service Runtime Service

The HttpServiceRuntime service represents the runtime state information of a Http Whiteboard implementation. This information is provided through Data Transfer Objects (DTOs). The architecture of OSGi DTOs is described in *OSGi Core Release 6*.

Each Http Whiteboard implementation registers exactly one HttpServiceRuntime service which can be used to target Whiteboard services defined in this specification to a specific Http Whiteboard implementation.

Implementations of this specification that also implement the *Http Service Specification* on page 45 can provide runtime information for servlets registered using the HttpService via the HttpServiceRuntime as well. In this case the osgi.http.service.id service property must be set to associate the HttpServiceRuntime service with the HttpService.

The HttpServiceRuntime provides service registration properties to declare its underlying Http Whiteboard implementation. These service properties can include implementation-specific key-value pairs. They also include the following:

Table 140.8   *Service properties for the HttpServiceRuntime service*

Service Property Name	Type	Description
osgi.http.endpoint	String+	Endpoint(s) where this Http Whiteboard implementation is listening. Registered Whiteboard services are made available here. Values could be provided as URLs e.g. http://192.168.1.10:8080/ or relative paths, e.g. /myapp/. Relative paths may be used if the scheme and authority parts of the URLs are not known such as in a bridged Http Service implementation. If the Http Service is serving the root context and scheme and authority are not known, the value of the property is /. Each entry must end with a slash. See HTTP_SERVICE_ENDPOINT.
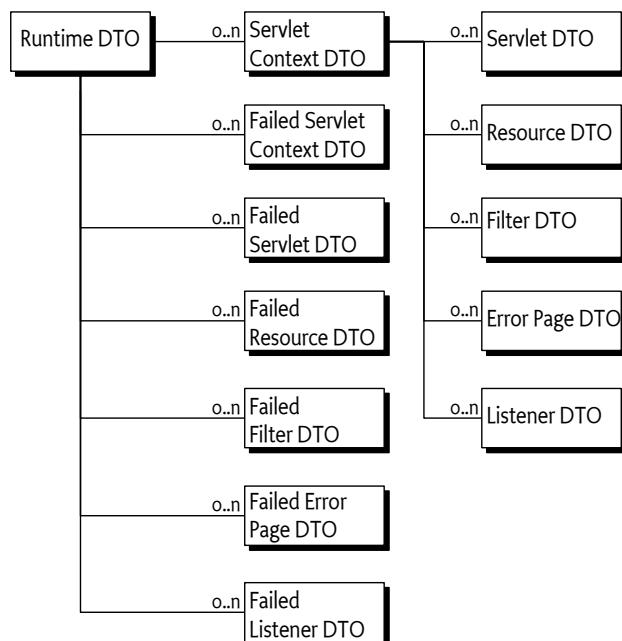
Service Property Name	Type	Description
osgi.http.service.id	Collection‹Long›	If this Http Whiteboard implementation also implements the *Http Service Specification* on page 45, this property is set to hold the service.id values of all the HttpService services provided by this implementation. See HTTP_SERVICE_ID.

The Http Service Runtime service provides information on registered Whiteboard services through the RuntimeDTO and RequestInfoDTO. The RuntimeDTO provides information on services that have been successfully registered as well as information about the Whiteboard services that were not successfully registered. Whiteboard services that have the required properties set but cannot be processed, are reflected in the failure DTOs. Whiteboard services of interfaces described in this specification that do not have the required properties set are ignored and not reflected in the failure DTOs.
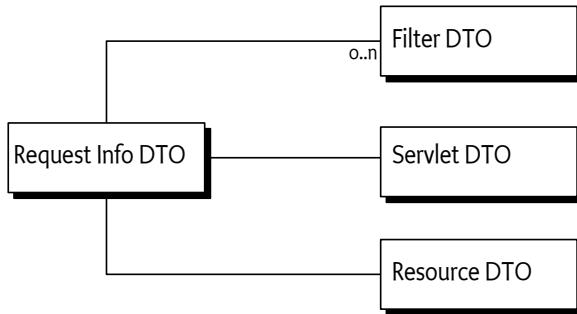
The Runtime DTO can be obtained using the getRuntimeDTO() method. The Runtime DTO provided has the following structure:

Figure 140.3   *Runtime DTO Overview Diagram*

Handlers for a given request path can be found with the calculateRequestInfoDTO(String) method. This method returns a RequestInfoDTO with the following structure:

Figure 140.4   Request Info DTO Overview Diagram

Where servlets registered via the *Http Service Specification* on page 45 are returned via this service, the Servlet DTO will report negative service IDs for these servlets to distinguish them from Servlet Whiteboard services.

140.10   Configuration Properties

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

- 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.

140.11   Capabilities

140.11.1   osgi.implementation Capability

The Http Whiteboard implementation bundle must provide the osgi.implementation capability with name osgi.http. This capability can be used by provisioning tools and during resolution to ensure that a Http Whiteboard implementation is present to process the Whiteboard services defined in this specification. The capability must also declare a uses constraint for the Servlet and OSGi Http Whiteboard packages and provide the version of this specification:

```
Provide-Capability: osgi.implementation;
      osgi.implementation="osgi.http";
      uses:="javax.servlet, javax.servlet.http,
            org.osgi.service.http.context, org.osgi.service.http.whiteboard";
      version:Version="1.0"
```

This capability must follow the rules defined for the *osgi.implementation Namespace* on page 685.

140.11.2   osgi.contract Capability

The Http Whiteboard implementation must provide a capability in the osgi.contract namespace with name JavaServlet if it exports the javax.servlet and javax.servlet.http packages. See [5] *Portable Java Contract Definitions*.

Providing the osgi.contract capability enables developer to build portable bundles for packages that are not versioned under OSGi Semantic Versioning rules. For more details see *osgi.contract Namespace* on page 683.

If the Servlet API is provided by another bundle, the Http Whiteboard implementation should be a consumer of the API and require the contract.

140.11.3 osgi.service Capability

The bundle providing the HttpServiceRuntime service must provide a capability in the osgi.service namespace representing this service. This capability must also declare a uses constraint for the org.osgi.service.http.runtime and org.osgi.service.http.runtime.dto packages:

```
Provide-Capability: osgi.service;
  objectClass:List<String>="org.osgi.service.http.runtime.HttpServiceRuntime";
  uses:="org.osgi.service.http.runtime,org.osgi.service.http.runtime.dto"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 685.

140.12 Security

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

140.12.1 Service Permissions

Bundles that need to register Http Whiteboard services must be granted ServicePermission[interfaceName, REGISTER] where interface name is the Http Whiteboard service interface name.

The Http Whiteboard implementation must be granted ServicePermission[interfaceName, GET] to retrieve the Http Whiteboard services from the service registry.

140.12.2 Introspection

Bundles that need to introspect the state of the Http runtime will need ServicePermission[org.osgi.service.http.runtime.HttpServiceRuntime, GET] to obtain the HttpServiceRuntime service and access the DTO types.

140.12.3 Accessing Resources with the Default Servlet Context Helper Implementation

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

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

140.12.4 Accessing Other Types of Resources

In order to access resources that were not returned from the default ServletContextHelper implementation, the Http Whiteboard implementation must be granted sufficient privileges to access

these resources. For example, if the `getResource` method of a ServletContextHelper service returns a file URL, the Http Whiteboard implementation requires the corresponding FilePermission to read the file. Similarly, if the `getResource` method of a ServletContextHelper service returns an HTTP URL, the Http Whiteboard implementation requires the corresponding SocketPermission to connect to the resource.

Therefore, in most cases, the Http Whiteboard implementation 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 Whiteboard implementation must capture the AccessControlContext object of the bundle registering a ServletContextHelper service, and then use the captured AccessControlContext object when accessing resources returned by the ServletContextHelper service. This situation prevents a bundle from supplying resources that it does not have permission to access.

Therefore, the Http Whiteboard implementation should follow a scheme like the following example. When using a ServletContextHelper service, it should capture the context.

```
ServiceReference<ServletContextHelper> servletContextHelperReference = ...
AccessControlContext acc = servletContextHelperReference.getBundle()
    .adapt(AccessControlContext.class);
```

When a URL returned by the `getResource` method of a ServletContextHelper service is used by the Http Whiteboard implementation, the implementation must use the URL in a doPrivileged construct using the AccessControlContext object of the registering bundle:

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

This ensures the Http Whiteboard implementation can only use the URL if the bundle registering the ServletContextHelper service that returned the URL also has permission to use the URL. The use of a captured AccessControlContext only applies when accessing URL objects returned by the `getResource` method of the ServletContextHelper service.

140.12.5 Calling Http Whiteboard Services

This specification does not require that the Http Whiteboard implementation is granted All Permission or wraps calls to the Http Whiteboard services in a doPrivileged block. Therefore, it is the responsibility of the Http Whiteboard service implementations to use a doPrivileged block when performing privileged operations.

140.13 org.osgi.service.http.context

Http Whiteboard Context Package Version 1.0.

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

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.http.context; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.http.context; version="[1.0,1.1)"

140.13.1   Summary

- ServletContextHelper - Helper service for a servlet context used by a Http Whiteboard implementation to serve HTTP requests.

140.13.2   public abstract class ServletContextHelper

Helper service for a servlet context used by a Http Whiteboard implementation to serve HTTP requests.

This service defines methods that the Http Whiteboard implementation may call to get information for a request when dealing with whiteboard services.

Each ServletContextHelper is registered with a "osgi.http.whiteboard.context.name" service property containing a name to reference by servlets, servlet filters, resources, and listeners. If there is more than one ServletContextHelper registered with the same context name, the one with the highest service ranking is active, the others are inactive.

A context is registered with the "osgi.http.whiteboard.context.path" service property to define a path under which all services registered with this context are reachable. If there is more than one ServletContextHelper registered with the same path, each duplicate context path is searched by service ranking order according to org.osgi.framework.ServiceReference.compareTo(Object) until a matching servlet or resource is found.

Servlets, servlet filters, resources, and listeners services may be associated with a ServletContextHelper service with the "osgi.http.whiteboard.context.select" service property. If the referenced ServletContextHelper service does not exist or is currently not active, the whiteboard services for that ServletContextHelper are not active either.

If no ServletContextHelper service is associated, that is no "osgi.http.whiteboard.context.select" service property is configured for a whiteboard service, a default ServletContextHelper is used.

Those whiteboard services that are associated with the same ServletContextHelper object will share the same ServletContext object.

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

- getMimeType - Always returns null.
- handleSecurity - Always returns true.
- getResource - Assumes the named resource is in the bundle of the whiteboard service, addressed from the root. This method calls the whiteboard service bundle's Bundle.getEntry method, and returns the appropriate URL to access the resource. On a Java runtime environment that supports permissions, the Http Whiteboard implementation needs to be granted org.osgi.framework.AdminPermission[*,RESOURCE].
- getResourcePaths - Assumes that the resources are in the bundle of the whiteboard service. This method calls Bundle.findEntries method, and returns the found entries. On a Java runtime environment that supports permissions, the Http Whiteboard implementation needs to be granted org.osgi.framework.AdminPermission[*,RESOURCE].
- getRealPath - Always returns null.

See Also   HttpWhiteboardConstants.HTTP_WHITEBOARD_CONTEXT_NAME,
HttpWhiteboardConstants.HTTP_WHITEBOARD_CONTEXT_PATH

Concurrency   Thread-safe

140.13.2.1   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.

140.13.2.2 **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(ServletContextHelper.AUTHORIZATION).

140.13.2.3 **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.

140.13.2.4 **public ServletContextHelper()**

□ Construct a new context helper.

If needed, the subclass will have to handle the association with a specific bundle.

140.13.2.5 **public ServletContextHelper(Bundle bundle)**

bundle The bundle to be associated with this context helper.

□ Construct a new context helper associated with the specified bundle.

140.13.2.6 **public String getMimeType(String name)**

name The name for which to determine the MIME type.

□ Maps a name to a MIME type.

Called by the Http Whiteboard implementation to determine the MIME type for the specified name. For whiteboard services, the Http Whiteboard implementation will call this method to support the ServletContext method getMimeType. For resource servlets, the Http Whiteboard implementation will call this method to determine the MIME type for the Content-Type header in the response.

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

140.13.2.7 **public String getRealPath(String path)**

path The virtual path to be translated to a real path.

□ Gets the real path corresponding to the given virtual path.

Called by the Http Whiteboard implementation to support the ServletContext method getRealPath for whiteboard services.

Returns The real path, or null if the translation cannot be performed.

140.13.2.8 **public URL getResource(String name)**

name The name of the requested resource.

□ Maps a resource name to a URL.

Called by the Http Whiteboard implementation to map the specified resource name to a URL. For servlets, the Http Whiteboard implementation will call this method to support the ServletContext methods getResource and getResourceAsStream. For resources, the Http Whiteboard implementation 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).toURI().toURL() or to a resource in the context's bundle via getClass().getResource(name)

Returns A URL that a Http Whiteboard implementation can use to read the resource or null if the resource does not exist.

140.13.2.9   **public Set<String> getResourcePaths(String path)**

path   The partial path used to match the resources, which must start with a /.

☐   Returns a directory-like listing of all the paths to resources within the web application whose longest sub-path matches the supplied path argument.

Called by the Http Whiteboard implementation to support the ServletContext method getResourcePaths for whiteboard services.

Returns   A Set containing the directory listing, or null if there are no resources in the web application whose path begins with the supplied path.

140.13.2.10   **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 Whiteboard implementation 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 [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 Whiteboard implementation will send the response back to the client, thereby completing the request. When this method returns true, the Http Whiteboard implementation will proceed with servicing the request.

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

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

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

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

Throws   IOException– May be thrown by this method. If this occurs, the Http Whiteboard implementation will terminate the request and close the socket.

140.14   **org.osgi.service.http.runtime**

Http Runtime Package Version 1.0.

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

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.http.runtime; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.http.runtime; version="[1.0,1.1)"

140.14.1 Summary

- HttpServiceRuntime - The HttpServiceRuntime service represents the runtime information of an Http Whiteboard implementation.
- HttpServiceRuntimeConstants - Defines standard names for Http Runtime Service constants.

140.14.2 public interface HttpServiceRuntime

The HttpServiceRuntime service represents the runtime information of an Http Whiteboard implementation.

It provides access to DTOs representing the current state of the service.

The HttpServiceRuntime service must be registered with the HttpServiceRuntimeConstants.HTTP_SERVICE_ENDPOINT service property.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

140.14.2.1 public RequestInfoDTO calculateRequestInfoDTO(String path)

path The request path, relative to the root of the Http Whiteboard implementation.

☐ Return a request info DTO containing the services involved with processing a request for the specified path.

Returns The request info DTO for the specified path.

140.14.2.2 public RuntimeDTO getRuntimeDTO()

☐ Return the runtime DTO representing the current state.

Returns The runtime DTO.

140.14.3 public final class HttpServiceRuntimeConstants

Defines standard names for Http Runtime Service constants.

140.14.3.1 public static final String HTTP_SERVICE_ENDPOINT = "osgi.http.endpoint"

Http Runtime Service service property specifying the endpoints upon which the Http Whiteboard implementation is listening.

An endpoint value is a URL or a relative path, to which the Http Whiteboard implementation is listening. For example, http://192.168.1.10:8080/ or /myapp/. A relative path may be used if the scheme and authority parts of the URL are not known, e.g. in a bridged Http Whiteboard implementation. If the Http Whiteboard implementation is serving the root context and neither scheme nor authority is known, the value of the property is "/". Both, a URL and a relative path, must end with a slash.

An Http Whiteboard implementation can be listening on multiple endpoints.

The value of this service property must be of type String, String[], or Collection<String>.

140.14.3.2   **public static final String HTTP_SERVICE_ID = "osgi.http.service.id"**

Http Runtime Service service property to associate the Http Runtime Service with one or more HttpService services.

If this Http Whiteboard implementation also implements the Http Service Specification, this service property is set to a collection of service.id for the HttpService services registered by this implementation.

The value of this service property must be of type Collection<Long>.

140.15   org.osgi.service.http.runtime.dto

Http Runtime DTO Package Version 1.0.

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

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.http.runtime.dto; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.http.runtime.dto; version="[1.0,1.1)"

140.15.1   Summary

- BaseServletDTO - Represents common information about a javax.servlet.Servlet service.
- DTOConstants - Defines standard constants for the DTOs.
- ErrorPageDTO - Represents a javax.servlet.Servlet for handling errors and currently being used by a servlet context.
- FailedErrorPageDTO - Represents a javax.servlet.Servlet service registered as an error page but currently not being used by a servlet context due to a problem.
- FailedFilterDTO - Represents a servlet Filter service which is currently not being used by a servlet context due to a problem.
- FailedListenerDTO - Represents a listener service which is currently not being used by a servlet context due to a problem.
- FailedResourceDTO - Represents a resource definition which is currently not being used by a servlet context due to a problem.
- FailedServletContextDTO - Represents a servlet context that is currently not used due to some problem.
- FailedServletDTO - Represents a javax.servlet.Servlet service which is currently not being used by a servlet context due to a problem.
- FilterDTO - Represents a servlet javax.servlet.Filter service currently being used for by a servlet context.
- ListenerDTO - Represents a listener currently being used by a servlet context.
- RequestInfoDTO - Represents the services used to process a specific request.
- ResourceDTO - Represents a resource definition currently being used by a servlet context.
- RuntimeDTO - Represents the state of a Http Service Runtime.
- ServletContextDTO - Represents a javax.servlet.ServletContext created for servlets, resources, servlet Filters, and listeners associated with that servlet context.
- ServletDTO - Represents a javax.servlet.Servlet currently being used by a servlet context.

140.15.2   public abstract class BaseServletDTO
extends DTO

Represents common information about a javax.servlet.Servlet service.

Concurrency   Not Thread-safe

140.15.2.1   public boolean asyncSupported

Specifies whether the servlet supports asynchronous processing.

140.15.2.2   public Map<String,String> initParams

The servlet initialization parameters as provided during registration of the servlet. Additional parameters like the Http Service Runtime attributes are not included. If the service has no initialization parameters, the map is empty.

140.15.2.3   public String name

The name of the servlet. This value is never null.

140.15.2.4   public long serviceId

Service property identifying the servlet. In the case of a servlet registered in the service registry and picked up by a Http Whiteboard Implementation, this value is not negative and corresponds to the service id in the registry. If the servlet has not been registered in the service registry, the value is negative and a unique negative value is generated by the Http Service Runtime in this case.

140.15.2.5   public long servletContextId

The service id of the servlet context for the servlet represented by this DTO.

140.15.2.6   public String servletInfo

The information string from the servlet.

This is the value returned by the Servlet.getServletInfo() method.

140.15.2.7   public BaseServletDTO()

140.15.3   public final class DTOConstants

Defines standard constants for the DTOs.

140.15.3.1   public static final int FAILURE_REASON_EXCEPTION_ON_INIT = 4

An exception occurred during initializing of the service.

This reason can only happen for servlets and servlet filters.

140.15.3.2   public static final int FAILURE_REASON_NO_SERVLET_CONTEXT_MATCHING = 1

No matching ServletContextHelper.

140.15.3.3   public static final int FAILURE_REASON_SERVICE_IN_USE = 7

The service is not registered as a prototype scoped service and is already in use with a servlet context and therefore can't be used with another servlet context.

140.15.3.4   public static final int FAILURE_REASON_SERVICE_NOT_GETTABLE = 5

The service is registered in the service registry but getting the service fails as it returns null.

140.15.3.5   public static final int FAILURE_REASON_SERVLET_CONTEXT_FAILURE = 2

Matching ServletContextHelper, but the context is not used due to a problem with the context.

140.15.3.6   **public static final int FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE = 3**

Service is shadowed by another service.

For example, a service with the same service properties but a higher service ranking.

140.15.3.7   **public static final int FAILURE_REASON_UNKNOWN = 0**

Failure reason is unknown.

140.15.3.8   **public static final int FAILURE_REASON_VALIDATION_FAILED = 6**

The service is registered in the service registry but the service properties are invalid.

140.15.4   public class ErrorPageDTO
extends BaseServletDTO

Represents a javax.servlet.Servlet for handling errors and currently being used by a servlet context.

Concurrency Not Thread-safe

140.15.4.1   **public long[] errorCodes**

The error codes the error page is used for. This array might be empty.

140.15.4.2   **public String[] exceptions**

The exceptions the error page is used for. This array might be empty.

140.15.4.3   **public ErrorPageDTO()**

140.15.5   public class FailedErrorPageDTO
extends ErrorPageDTO

Represents a javax.servlet.Servlet service registered as an error page but currently not being used by a servlet context due to a problem.

As the servlet represented by this DTO is not used due to a failure, the field FailedErrorPageDTO.servletContextId always returns 0 and does not point to an existing Servlet-ContextHelper.

Concurrency Not Thread-safe

140.15.5.1   **public int failureReason**

The reason why the servlet represented by this DTO is not used.

See Also DTOConstants.FAILURE_REASON_UNKNOWN,
DTOConstants.FAILURE_REASON_EXCEPTION_ON_INIT,
DTOConstants.FAILURE_REASON_NO_SERVLET_CONTEXT_MATCHING,
DTOConstants.FAILURE_REASON_SERVICE_NOT_GETTABLE,
DTOConstants.FAILURE_REASON_SERVLET_CONTEXT_FAILURE,
DTOConstants.FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE

140.15.5.2   **public FailedErrorPageDTO()**

140.15.6   public class FailedFilterDTO
extends FilterDTO

Represents a servlet Filter service which is currently not being used by a servlet context due to a problem.

As the service represented by this DTO is not used due to a failure, the field
FailedFilterDTO.servletContextId always returns 0 and does not point to an existing servlet context.

Concurrency Not Thread-safe

140.15.6.1   **public int failureReason**

The reason why the servlet filter represented by this DTO is not used.

See Also DTOConstants.FAILURE_REASON_UNKNOWN,
DTOConstants.FAILURE_REASON_EXCEPTION_ON_INIT,
DTOConstants.FAILURE_REASON_NO_SERVLET_CONTEXT_MATCHING,
DTOConstants.FAILURE_REASON_SERVICE_NOT_GETTABLE,
DTOConstants.FAILURE_REASON_SERVLET_CONTEXT_FAILURE,
DTOConstants.FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE

140.15.6.2   **public FailedFilterDTO()**

140.15.7   public class FailedListenerDTO
extends ListenerDTO

Represents a listener service which is currently not being used by a servlet context due to a problem.

As the listener represented by this DTO is not used due to a failure, the field
FailedErrorPageDTO.servletContextId always returns 0 and does not point to an existing servlet context.

Concurrency Not Thread-safe

140.15.7.1   **public int failureReason**

The reason why the listener represented by this DTO is not used.

See Also DTOConstants.FAILURE_REASON_UNKNOWN,
DTOConstants.FAILURE_REASON_EXCEPTION_ON_INIT,
DTOConstants.FAILURE_REASON_NO_SERVLET_CONTEXT_MATCHING,
DTOConstants.FAILURE_REASON_SERVICE_NOT_GETTABLE,
DTOConstants.FAILURE_REASON_SERVLET_CONTEXT_FAILURE,
DTOConstants.FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE

140.15.7.2   **public FailedListenerDTO()**

140.15.8   public class FailedResourceDTO
extends ResourceDTO

Represents a resource definition which is currently not being used by a servlet context due to a
problem.

As the resource represented by this DTO is not used due to a failure, the field
FailedResourceDTO.servletContextId always returns 0 and does not point to an existing servlet context.

Concurrency Not Thread-safe

140.15.8.1   **public int failureReason**

The reason why the resource represented by this DTO is not used.

See Also DTOConstants.FAILURE_REASON_UNKNOWN,
DTOConstants.FAILURE_REASON_EXCEPTION_ON_INIT,
DTOConstants.FAILURE_REASON_NO_SERVLET_CONTEXT_MATCHING,
DTOConstants.FAILURE_REASON_SERVICE_NOT_GETTABLE,

DTOConstants.FAILURE_REASON_SERVLET_CONTEXT_FAILURE,
DTOConstants.FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE

140.15.8.2   **public FailedResourceDTO()**

140.15.9   public class FailedServletContextDTO
extends ServletContextDTO

Represents a servlet context that is currently not used due to some problem. The following fields return an empty array for a FailedServletContextDTO:

- ServletContextDTO.servletDTOs
- ServletContextDTO.resourceDTOs
- ServletContextDTO.filterDTOs
- ServletContextDTO.errorPageDTOs
- ServletContextDTO.listenerDTOs

The method ServletContextDTO.attributes returns an empty map for a FailedServletContextDTO.

Concurrency Not Thread-safe

140.15.9.1   **public int failureReason**

The reason why the servlet context represented by this DTO is not used.

See Also DTOConstants.FAILURE_REASON_UNKNOWN,
DTOConstants.FAILURE_REASON_EXCEPTION_ON_INIT,
DTOConstants.FAILURE_REASON_NO_SERVLET_CONTEXT_MATCHING,
DTOConstants.FAILURE_REASON_SERVICE_NOT_GETTABLE,
DTOConstants.FAILURE_REASON_SERVLET_CONTEXT_FAILURE,
DTOConstants.FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE

140.15.9.2   **public FailedServletContextDTO()**

140.15.10   public class FailedServletDTO
extends ServletDTO

Represents a javax.servlet.Servlet service which is currently not being used by a servlet context due to a problem.

As the servlet represented by this DTO is not used due to a failure, the field FailedErrorPageDTO.servletContextId always returns 0 and does not point to an existing servlet context.

Concurrency Not Thread-safe

140.15.10.1   **public int failureReason**

The reason why the servlet represented by this DTO is not used.

See Also DTOConstants.FAILURE_REASON_UNKNOWN,
DTOConstants.FAILURE_REASON_EXCEPTION_ON_INIT,
DTOConstants.FAILURE_REASON_NO_SERVLET_CONTEXT_MATCHING,
DTOConstants.FAILURE_REASON_SERVICE_NOT_GETTABLE,
DTOConstants.FAILURE_REASON_SERVLET_CONTEXT_FAILURE,
DTOConstants.FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE

140.15.10.2   **public FailedServletDTO()**

140.15.11 **public class FilterDTO**
extends DTO

Represents a servlet javax.servlet.Filter service currently being used for by a servlet context.

Concurrency Not Thread-safe

140.15.11.1 **public boolean asyncSupported**

Specifies whether the servlet filter supports asynchronous processing.

140.15.11.2 **public String[] dispatcher**

The dispatcher associations for the servlet filter.

The specified names are used to determine in what occasions the servlet filter is called. This array is never null.

140.15.11.3 **public Map<String,String> initParams**

The servlet filter initialization parameters as provided during registration of the servlet filter. Additional parameters like the Http Service Runtime attributes are not included. If the servlet filter has not initialization parameters, this map is empty.

140.15.11.4 **public String name**

The name of the servlet filter. This field is never null.

140.15.11.5 **public String[] patterns**

The request mappings for the servlet filter.

The specified patterns are used to determine whether a request is mapped to the servlet filter. This array might be empty.

140.15.11.6 **public String[] regexs**

The request mappings for the servlet filter.

The specified regular expressions are used to determine whether a request is mapped to the servlet filter. This array might be empty.

140.15.11.7 **public long serviceId**

Service property identifying the servlet filter. In the case of a servlet filter registered in the service registry and picked up by a Http Whiteboard Implementation, this value is not negative and corresponds to the service id in the registry. If the servlet filter has not been registered in the service registry, the value is negative and a unique negative value is generated by the Http Service Runtime in this case.

140.15.11.8 **public long servletContextId**

The service id of the servlet context for the servlet filter represented by this DTO.

140.15.11.9 **public String[] servletNames**

The servlet names for the servlet filter.

The specified names are used to determine the servlets whose requests are mapped to the servlet filter. This array might be empty.

140.15.11.10 **public FilterDTO()**

140.15.12   **public class ListenerDTO**
                  extends DTO

Represents a listener currently being used by a servlet context.

Concurrency Not Thread-safe

140.15.12.1   **public long serviceId**

Service property identifying the listener. In the case of a Listener registered in the service registry and picked up by a Http Whiteboard Implementation, this value not negative and corresponds to the service id in the registry. If the listener has not been registered in the service registry, the value is negative and a unique negative value is generated by the Http Service Runtime in this case.

140.15.12.2   **public long servletContextId**

The service id of the servlet context for the listener represented by this DTO.

140.15.12.3   **public String[] types**

The fully qualified type names the listener. This array is never empty.

140.15.12.4   **public ListenerDTO()**

140.15.13   **public class RequestInfoDTO**
                  extends DTO

Represents the services used to process a specific request.

Concurrency Not Thread-safe

140.15.13.1   **public FilterDTO[] filterDTOs**

The servlet filters processing this request. If no servlet filters are called for processing this request, an empty array is returned.

140.15.13.2   **public String path**

The path of the request relative to the root.

140.15.13.3   **public ResourceDTO resourceDTO**

The resource processing this request. If the request is processed by a resource, this field points to the DTO of the resource. If the request is processed by another type of component like a servlet, this field is null.

140.15.13.4   **public long servletContextId**

The service id of the servlet context processing the request represented by this DTO.

140.15.13.5   **public ServletDTO servletDTO**

The servlet processing this request. If the request is processed by a servlet, this field points to the DTO of the servlet. If the request is processed by another type of component like a resource, this field is null.

140.15.13.6   **public RequestInfoDTO()**

140.15.14   **public class ResourceDTO**
                  extends DTO

Represents a resource definition currently being used by a servlet context.

Concurrency Not Thread-safe

140.15.14.1   **public String[] patterns**

The request mappings for the resource.

The specified patterns are used to determine whether a request is mapped to the resource. This value is never null.

140.15.14.2   **public String prefix**

The prefix of the resource.

140.15.14.3   **public long serviceId**

Service property identifying the resource. In the case of a resource registered in the service registry and picked up by a Http Whiteboard Implementation, this value is not negative and corresponds to the service id in the registry. If the resource has not been registered in the service registry, the value is negative and a unique negative value is generated by the Http Service Runtime in this case.

140.15.14.4   **public long servletContextId**

The service id of the servlet context for the resource represented by this DTO.

140.15.14.5   **public ResourceDTO()**

140.15.15   public class RuntimeDTO
extends DTO

Represents the state of a Http Service Runtime.

Concurrency Not Thread-safe

140.15.15.1   **public FailedErrorPageDTO[] failedErrorPageDTOs**

Returns the representations of the error page javax.servlet.Servlet services associated with this runtime but currently not used due to some problem. The returned array may be empty.

140.15.15.2   **public FailedFilterDTO[] failedFilterDTOs**

Returns the representations of the servlet javax.servlet.Filter services associated with this runtime but currently not used due to some problem. The returned array may be empty.

140.15.15.3   **public FailedListenerDTO[] failedListenerDTOs**

Returns the representations of the listeners associated with this runtime but currently not used due to some problem. The returned array may be empty.

140.15.15.4   **public FailedResourceDTO[] failedResourceDTOs**

Returns the representations of the resources associated with this runtime but currently not used due to some problem. The returned array may be empty.

140.15.15.5   **public FailedServletContextDTO[] failedServletContextDTOs**

Returns the representations of the javax.servlet.ServletContext objects currently not used by the Http service runtime due to some problem. The returned array may be empty.

140.15.15.6   **public FailedServletDTO[] failedServletDTOs**

Returns the representations of the javax.servlet.Servlet services associated with this runtime but currently not used due to some problem. The returned array may be empty.

140.15.15.7   **public ServiceReferenceDTO serviceDTO**

The DTO for the corresponding org.osgi.service.http.runtime.HttpServiceRuntime. This value is never null.

140.15.15.8   **public ServletContextDTO[] servletContextDTOs**

Returns the representations of the javax.servlet.ServletContext objects used by the Http Service Runtime. The returned array may be empty if the Http Service Runtime is currently not using any javax.servlet.ServletContext objects.

140.15.15.9   **public RuntimeDTO()**

140.15.16   public class ServletContextDTO
extends DTO

Represents a javax.servlet.ServletContext created for servlets, resources, servlet Filters, and listeners associated with that servlet context. The Servlet Context is usually backed by a org.osgi.service.http.context.ServletContextHelper service.

Concurrency   Not Thread-safe

140.15.16.1   **public Map<String,Object> attributes**

The servlet context attributes.

The value type must be a numerical type, Boolean, String, DTO or an array of any of the former. Therefore this method will only return the attributes of the servlet context conforming to this constraint. Other attributes are omitted. If there are no attributes conforming to the constraint, an empty map is returned.

140.15.16.2   **public String contextPath**

The servlet context path. This is the value returned by the ServletContext.getContextPath() method.

140.15.16.3   **public ErrorPageDTO[] errorPageDTOs**

Returns the representations of the error page Servlet services associated with this context. The representations of the error page Servlet services associated with this context. The returned array may be empty if this context is currently not associated with any error pages.

140.15.16.4   **public FilterDTO[] filterDTOs**

Returns the representations of the servlet Filter services associated with this context. The representations of the servlet Filter services associated with this context. The returned array may be empty if this context is currently not associated with any servlet Filter services.

140.15.16.5   **public Map<String,String> initParams**

The servlet context initialization parameters. This is the set of parameters provided when registering this context. Additional parameters like the Http Service Runtime attributes are not included. If the context has no initialization parameters, this map is empty.

140.15.16.6   **public ListenerDTO[] listenerDTOs**

Returns the representations of the listener services associated with this context. The representations of the listener services associated with this context. The returned array may be empty if this context is currently not associated with any listener services.

140.15.16.7   **public String name**

The name of the servlet context. The name of the corresponding org.osgi.service.http.context.ServletContextHelper.

This is the value returned by the ServletContext.getServletContextName() method.

140.15.16.8   **public ResourceDTO[] resourceDTOs**

Returns the representations of the resource services associated with this context. The representations of the resource services associated with this context. The returned array may be empty if this context is currently not associated with any resource services.

140.15.16.9   **public long serviceId**

Service property identifying the servlet context. In the case of a servlet context backed by a Servlet-ContextHelper registered in the service registry and picked up by a Http Whiteboard Implementation, this value is not negative and corresponds to the service id in the registry. If the servlet context is not backed by a service registered in the service registry, the value is negative and a unique negative value is generated by the Http Service Runtime in this case.

140.15.16.10   **public ServletDTO[] servletDTOs**

Returns the representations of the Servlet services associated with this context. The representations of the Servlet services associated with this context. The returned array may be empty if this context is currently not associated with any Servlet services.

140.15.16.11   **public ServletContextDTO()**

140.15.17   public class ServletDTO
extends BaseServletDTO

Represents a javax.servlet.Servlet currently being used by a servlet context.

Concurrency   Not Thread-safe

140.15.17.1   **public String[] patterns**

The request mappings for the servlet.

The specified patterns are used to determine whether a request is mapped to the servlet. This array is never empty.

140.15.17.2   **public ServletDTO()**

140.16   org.osgi.service.http.whiteboard

Http Whiteboard Package Version 1.0.

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

Example import for consumers using the API in this package:

Import-Package: org.osgi.service.http.whiteboard; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.http.whiteboard; version="[1.0,1.1)"

140.16.1   Summary

- HttpWhiteboardConstants - Defines standard constants for the Http Whiteboard services.

140.16.2   public final class HttpWhiteboardConstants

Defines standard constants for the Http Whiteboard services.

140.16.2.1   **public static final String DISPATCHER_ASYNC = "ASYNC"**

Possible value for the HTTP_WHITEBOARD_FILTER_DISPATCHER property indicating the servlet filter is applied in the asynchronous context.

See Also   Java Servlet Specification Version 3.0, Section 6.2.5 Filters and the RequestDispatcher

140.16.2.2   **public static final String DISPATCHER_ERROR = "ERROR"**

Possible value for the HTTP_WHITEBOARD_FILTER_DISPATCHER property indicating the servlet filter is applied when an error page is called.

See Also   Java Servlet Specification Version 3.0, Section 6.2.5 Filters and the RequestDispatcher

140.16.2.3   **public static final String DISPATCHER_FORWARD = "FORWARD"**

Possible value for the HTTP_WHITEBOARD_FILTER_DISPATCHER property indicating the servlet filter is applied to forward calls to the dispatcher.

See Also   Java Servlet Specification Version 3.0, Section 6.2.5 Filters and the RequestDispatcher

140.16.2.4   **public static final String DISPATCHER_INCLUDE = "INCLUDE"**

Possible value for the HTTP_WHITEBOARD_FILTER_DISPATCHER property indicating the servlet filter is applied to include calls to the dispatcher.

See Also   Java Servlet Specification Version 3.0, Section 6.2.5 Filters and the RequestDispatcher

140.16.2.5   **public static final String DISPATCHER_REQUEST = "REQUEST"**

Possible value for the HTTP_WHITEBOARD_FILTER_DISPATCHER property indicating the servlet filter is applied to client requests.

See Also   Java Servlet Specification Version 3.0, Section 6.2.5 Filters and the RequestDispatcher

140.16.2.6   **public static final String HTTP_WHITEBOARD_CONTEXT_INIT_PARAM_PREFIX = "context.init."**

Service property prefix referencing a ServletContextHelper service.

For ServletContextHelper services this prefix can be used for service properties to mark them as initialization parameters which can be retrieved from the associated servlet context. The prefix is removed from the service property name to build the initialization parameter name.

For ServletContextHelper services, the value of each initialization parameter service property must be of type String.

140.16.2.7   **public static final String HTTP_WHITEBOARD_CONTEXT_NAME = "osgi.http.whiteboard.context.name"**

Service property specifying the name of an ServletContextHelper service.

For ServletContextHelper services, this service property must be specified. Context services without this service property are ignored.

Servlet, listener, servlet filter, and resource services might refer to a specific ServletContextHelper service referencing the name with the HTTP_WHITEBOARD_CONTEXT_SELECT property.

For ServletContextHelper services, the value of this service property must be of type String. The value must follow the "symbolic-name" specification from Section 1.3.2 of the OSGi Core Specification.

See Also   HTTP_WHITEBOARD_CONTEXT_PATH, HTTP_WHITEBOARD_CONTEXT_SELECT, HTTP_WHITEBOARD_DEFAULT_CONTEXT_NAME

140.16.2.8   **public static final String HTTP_WHITEBOARD_CONTEXT_PATH = "osgi.http.whiteboard.context.path"**

Service property specifying the path of an ServletContextHelper service.

For ServletContextHelper services this service property is required. Context services without this service property are ignored.

This property defines a context path under which all whiteboard services associated with this context are registered. Having different contexts with different paths allows to separate the URL space.

For ServletContextHelper services, the value of this service property must be of type String. The value is either a slash for the root or it must start with a slash but not end with a slash. Valid characters are defined in rfc3986#section-3.3. Contexts with an invalid path are ignored.

See Also   HTTP_WHITEBOARD_CONTEXT_NAME, HTTP_WHITEBOARD_CONTEXT_SELECT

140.16.2.9   **public static final String HTTP_WHITEBOARD_CONTEXT_SELECT = "osgi.http.whiteboard.context.select"**

Service property referencing a ServletContextHelper service.

For servlet, listener, servlet filter, or resource services, this service property refers to the associated ServletContextHelper service. The value of this property is a filter expression which is matched against the service registration properties of the ServletContextHelper service. If this service property is not specified, the default context is used. If there is no context service matching, the servlet, listener, servlet filter, or resource service is ignored.

For example, if a whiteboard service wants to select a servlet context helper with the name "Admin" the expression would be "(osgi.http.whiteboard.context.name=Admin)". Selecting all contexts could be done with "(osgi.http.whiteboard.context.name=∗)".

For servlet, listener, servlet filter, or resource services, the value of this service property must be of type String.

See Also   HTTP_WHITEBOARD_CONTEXT_NAME, HTTP_WHITEBOARD_CONTEXT_PATH

140.16.2.10   **public static final String HTTP_WHITEBOARD_DEFAULT_CONTEXT_NAME = "default"**

The name of the default ServletContextHelper. If a service is registered with this property, it is overriding the default context with a custom provided context.

See Also   HTTP_WHITEBOARD_CONTEXT_NAME

140.16.2.11   **public static final String HTTP_WHITEBOARD_FILTER_ASYNC_SUPPORTED = "osgi.http.whiteboard.filter.asyncSupported"**

Service property specifying whether a servlet Filter service supports asynchronous processing.

By default servlet filters services do not support asynchronous processing.

The value of this service property must be of type Boolean.

See Also   Java Servlet Specification Version 3.0, Section 2.3.3.3 Asynchronous Processing

140.16.2.12   **public static final String HTTP_WHITEBOARD_FILTER_DISPATCHER = "osgi.http.whiteboard.filter.dispatcher"**

Service property specifying the dispatcher handling of a servlet Filter.

By default servlet filter services are associated with client requests only (see value DISPATCHER_REQUEST).

The value of this service property must be of type String, String[], or Collection<String>. Allowed values are DISPATCHER_ASYNC, DISPATCHER_ERROR, DISPATCHER_FORWARD, DISPATCHER_INCLUDE, DISPATCHER_REQUEST.

See Also   Java Servlet Specification Version 3.0, Section 6.2.5 Filters and the RequestDispatcher

140.16.2.13   **public static final String HTTP_WHITEBOARD_FILTER_INIT_PARAM_PREFIX = "filter.init."**

Service property prefix referencing a Filter service.

For Filter services this prefix can be used for service properties to mark them as initialization parameters which can be retrieved from the associated filter config. The prefix is removed from the service property name to build the initialization parameter name.

For Filter services, the value of each initialization parameter service property must be of type String.

140.16.2.14   **public static final String HTTP_WHITEBOARD_FILTER_NAME = "osgi.http.whiteboard.filter.name"**

Service property specifying the servlet filter name of a Filter service.

This name is used as the value for the FilterConfig.getFilterName() method. If this service property is not specified, the fully qualified name of the service object's class is used as the servlet filter name.

Servlet filter names should be unique among all servlet filter services associated with a single ServletContextHelper.

The value of this service property must be of type String.

140.16.2.15   **public static final String HTTP_WHITEBOARD_FILTER_PATTERN = "osgi.http.whiteboard.filter.pattern"**

Service property specifying the request mappings for a Filter service.

The specified patterns are used to determine whether a request should be mapped to the servlet filter. Filter services without this service property or the HTTP_WHITEBOARD_FILTER_SERVLET or the HTTP_WHITEBOARD_FILTER_REGEX service property are ignored.

The value of this service property must be of type String, String[], or Collection<String>.

See Also   Java Servlet Specification Version 3.0, Section 12.2 Specification of Mappings

140.16.2.16   **public static final String HTTP_WHITEBOARD_FILTER_REGEX = "osgi.http.whiteboard.filter.regex"**

Service property specifying the request mappings for a servlet Filter service.

The specified regular expressions are used to determine whether a request should be mapped to the servlet filter. The regular expressions must follow the syntax defined in java.util.regex.Pattern. Servlet filter services without this service property or the HTTP_WHITEBOARD_FILTER_SERVLET or the HTTP_WHITEBOARD_FILTER_PATTERN service property are ignored.

The value of this service property must be of type String, String[], or Collection<String>.

See Also   java.util.regex.Pattern

140.16.2.17   **public static final String HTTP_WHITEBOARD_FILTER_SERVLET = "osgi.http.whiteboard.filter.servlet"**

Service property specifying the servlet names for a servlet Filter service.

The specified names are used to determine the servlets whose requests should be mapped to the servlet filter. Servlet filter services without this service property or the HTTP_WHITEBOARD_FILTER_PATTERN or the HTTP_WHITEBOARD_FILTER_REGEX service property are ignored.

The value of this service property must be of type String, String[], or Collection<String>.

140.16.2.18   **public static final String HTTP_WHITEBOARD_LISTENER = "osgi.http.whiteboard.listener"**

Service property to mark a Listener service as a Whiteboard service. Listener services with this property set to the string value "true" will be treated as Whiteboard services opting in to being handled by the Http Whiteboard implementation. If the value "false" is specified, the service is opting out and this case is treated exactly the same as if this property is missing. If an invalid value is specified this is treated as a failure.

The value of this service property must be of type String. Valid values are "true" and "false" ignoring case.

140.16.2.19   **public static final String HTTP_WHITEBOARD_RESOURCE_PATTERN = "osgi.http.whiteboard.resource.pattern"**

Service property specifying the request mappings for resources.

The specified patterns are used to determine whether a request should be mapped to resources. Resource services without this service property are ignored.

The value of this service property must be of type String, String[], or Collection<String>.

See Also   Java Servlet Specification Version 3.0, Section 12.2 Specification of Mappings,
HTTP_WHITEBOARD_RESOURCE_PREFIX

140.16.2.20   **public static final String HTTP_WHITEBOARD_RESOURCE_PREFIX = "osgi.http.whiteboard.resource.prefix"**

Service property specifying the resource entry prefix for a resource service.

If a resource service is registered with this property, requests are served with bundle resources.

This prefix is used to map a requested resource to the bundle's entries. The value must not end with slash ("/") with the exception that a name of the form "/" is used to denote the root of the bundle. See the specification text for details on how HTTP requests are mapped.

The value of this service property must be of type String.

See Also   HTTP_WHITEBOARD_RESOURCE_PATTERN

140.16.2.21   **public static final String HTTP_WHITEBOARD_SERVLET_ASYNC_SUPPORTED =**
"osgi.http.whiteboard.servlet.asyncSupported"

Service property specifying whether a Servlet service supports asynchronous processing.

By default servlet services do not support asynchronous processing.

The value of this service property must be of type Boolean.

See Also   Java Servlet Specification Version 3.0, Section 2.3.3.3 Asynchronous Processing

140.16.2.22   **public static final String HTTP_WHITEBOARD_SERVLET_ERROR_PAGE =**
"osgi.http.whiteboard.servlet.errorPage"

Service property specifying whether a Servlet service acts as an error page.

The service property values may be the name of a fully qualified exception class, a three digit HTTP status code, the value "4xx" for all error codes in the 400 range, or the value "5xx" for all error codes in the 500 range. Any value that is not a three digit number, or one of the two special values is considered to be the name of a fully qualified exception class.

The value of this service property must be of type String, String[], or Collection<String>.

140.16.2.23   **public static final String HTTP_WHITEBOARD_SERVLET_INIT_PARAM_PREFIX = "servlet.init."**

Service property prefix referencing a Servlet service.

For Servlet services this prefix can be used for service properties to mark them as initialization parameters which can be retrieved from the associated servlet config. The prefix is removed from the service property name to build the initialization parameter name.

For Servlet services, the value of each initialization parameter service property must be of type String.

140.16.2.24   **public static final String HTTP_WHITEBOARD_SERVLET_NAME = "osgi.http.whiteboard.servlet.name"**

Service property specifying the servlet name of a Servlet service.

This name is used as the value for the ServletConfig.getServletName() method. If this service property is not specified, the fully qualified name of the service object's class is used as the servlet name. Filter services may refer to servlets by this name in their HTTP_WHITEBOARD_FILTER_SERVLET service property to apply the filter to the servlet.

Servlet names should be unique among all servlet services associated with a single ServletContextHelper.

The value of this service property must be of type String.

140.16.2.25   **public static final String HTTP_WHITEBOARD_SERVLET_PATTERN = "osgi.http.whiteboard.servlet.pattern"**

Service property specifying the request mappings for a Servlet service.

The specified patterns are used to determine whether a request should be mapped to the servlet. Servlet services without this service property or HTTP_WHITEBOARD_SERVLET_ERROR_PAGE are ignored.

The value of this service property must be of type String, String[], or Collection<String>.

See Also   Java Servlet Specification Version 3.0, Section 12.2 Specification of Mappings

140.16.2.26   **public static final String HTTP_WHITEBOARD_TARGET = "osgi.http.whiteboard.target"**

Service property specifying the target filter to select the Http Whiteboard implementation to process the service.

An Http Whiteboard implementation can define any number of service properties which can be referenced by the target filter. The service properties should always include the osgi.http.endpoint service property if the endpoint information is known.

If this service property is not specified, then all Http Whiteboard implementations can process the service.

The value of this service property must be of type String and be a valid filter string.

140.17   References

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

[2]   *HTTP 1.1 Specifications RFCs 7230-7235*
      https://tools.ietf.org/html/rfc7230, https://tools.ietf.org/html/rfc7231, https://tools.ietf.org/html/rfc7232, https://tools.ietf.org/html/rfc7233, https://tools.ietf.org/html/rfc7234, https://tools.ietf.org/html/rfc7235

[3]   *HTTP/2 Specifications*
      https://http2.github.io

[4]   *Java Servlet 3.1 Specification*
      https://jcp.org/aboutJava/communityprocess/final/jsr340/

[5]   *Portable Java Contract Definitions*
      http://www.osgi.org/Specifications/ReferenceContract

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

[7]   *Whiteboard Pattern*
      http://www.osgi.org/wiki/uploads/Links/whiteboard.pdf

[8]   *Core Service Hooks*
      OSGi Core, Chapter 55 Service Hook Service Specification

702   XML Parser Service Specification

Version 1.0

702.1   Introduction

The Extensible Markup Language (XML) has become a popular method of describing data. As more bundles use XML to describe their data, a common XML Parser becomes necessary in an embedded environment in order to reduce the need for space. Not all XML Parsers are equivalent in function, however, and not all bundles have the same requirements on an XML parser.

This problem was addressed in the Java API for XML Processing, see [4] *JAXP* for Java 2 Standard Edition and Enterprise Edition. This specification addresses how the classes defined in JAXP can be used in an OSGi framework. It defines how:

- Implementations of XML parsers can become available to other bundles
- Bundles can find a suitable parser
- A standard parser in a JAR can be transformed to a bundle

702.1.1   Essentials
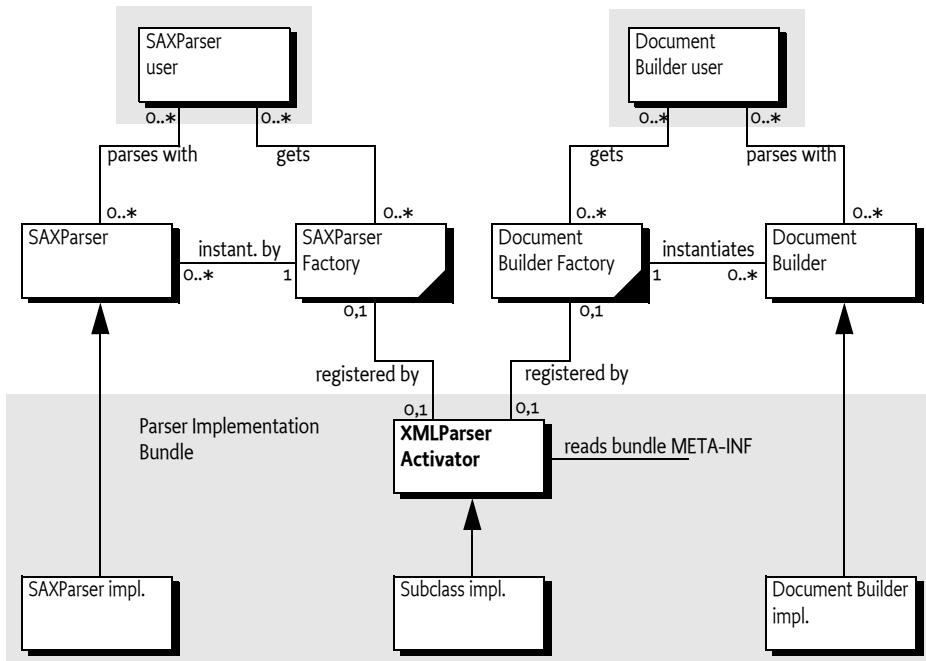
- *Standards* - Leverage existing standards in Java based XML parsing: JAXP, SAX and DOM
- *Unmodified JAXP code* - Run unmodified JAXP code
- *Simple* - It should be easy to provide a SAX or DOM parser as well as easy to find a matching parser
- *Multiple* - It should be possible to have multiple implementations of parsers available
- *Extendable* - It is likely that parsers will be extended in the future with more functionality

702.1.2   Entities

- *XMLParserActivator* - A utility class that registers a parser factory from declarative information in the Manifest file.
- *SAXParserFactory* - A class that can create an instance of a SAXParser class.
- *DocumentBuilderFactory* - A class that can create an instance of a DocumentBuilder class.
- *SAXParser* - A parser, instantiated by a SaxParserFactory object, that parses according to the SAX specifications.
- *DocumentBuilder* - A parser, instantiated by a DocumentBuilderFactory, that parses according to the DOM specifications.

Figure 702.1   *XML Parsing diagram*

702.1.3   **Operations**

A bundle containing a SAX or DOM parser is started. This bundle registers a `SAXParserFactory` and/ or a `DocumentBuilderFactory` service object with the Framework. Service registration properties describe the features of the parsers to other bundles. A bundle that needs an XML parser will get a `SAXParserFactory` or `DocumentBuilderFactory` service object from the Framework service registry. This object is then used to instantiate the requested parsers according to their specifications.

702.2   JAXP

XML has become very popular in the last few years because it allows the interchange of complex information between different parties. Though only a single XML standard exists, there are multiple APIs to XML parsers, primarily of two types:

- The Simple API for XML (SAX1 and SAX2)
- Based on the Document Object Model (DOM 1 and 2)

Both standards, however, define an abstract API that can be implemented by different vendors.

A given XML Parser implementation may support either or both of these parser types by implementing the `org.w3c.dom` and/or `org.xml.sax` packages. In addition, parsers have characteristics such as whether they are validating or non-validating parsers and whether or not they are namespace aware.

An application which uses a specific XML Parser must code to that specific parser and become coupled to that specific implementation. If the parser has implemented [4] *JAXP*, however, the application developer can code against SAX or DOM and let the runtime environment decide which parser implementation is used.

JAXP uses the concept of a *factory*. A factory object is an object that abstracts the creation of another object. JAXP defines a `DocumentBuilderFactory` and a `SAXParserFactory` class for this purpose.

JAXP is implemented in the `javax.xml.parsers` package and provides an abstraction layer between an application and a specific XML Parser implementation. Using JAXP, applications can choose to use any JAXP compliant parser without changing any code, simply by changing a System property which specifies the SAX- and DOM factory class names.

In JAXP, the default factory is obtained with a static method in the `SAXParserFactory` or `DocumentBuilderFactory` class. This method will inspect the associated System property and create a new instance of that class.

702.3   XML Parser service

The current specification of JAXP has the limitation that only one of each type of parser factories can be registered. This specification specifies how multiple `SAXParserFactory` objects and `DocumentBuilderFactory` objects can be made available to bundles simultaneously.

Providers of parsers should register a JAXP factory object with the OSGi service registry under the factory class name. Service properties are used to describe whether the parser:

- Is validating
- Is name-space aware
- Has additional features

With this functionality, bundles can query the OSGi service registry for parsers supporting the specific functionality that they require.

702.4   Properties

Parsers must be registered with a number of properties that qualify the service. In this specification, the following properties are specified:

- PARSER_NAMESPACEAWARE - The registered parser is aware of name-spaces. Name-spaces allow an XML document to consist of independently developed DTDs. In an XML document, they are recognized by the `xmlns` attribute and names prefixed with an abbreviated name-space identifier, like: ‹xsl:if ...›. The type is a `Boolean` object that must be `true` when the parser supports name-spaces. All other values, or the absence of the property, indicate that the parser does not implement name-spaces.
- PARSER_VALIDATING - The registered parser can read the DTD and can validate the XML accordingly. The type is a `Boolean` object that must `true` when the parser is validating. All other values, or the absence of the property, indicate that the parser does not validate.

702.5   Getting a Parser Factory

Getting a parser factory requires a bundle to get the appropriate factory from the service registry. In a simple case in which a non-validating, non-name-space aware parser would suffice, it is best to use `getServiceReference(String)`.

```
DocumentBuilder getParser(BundleContext context)
    throws Exception {
    ServiceReference ref = context.getServiceReference(
        DocumentBuilderFactory.class.getName() );
    if ( ref == null )
        return null;
```

```
    DocumentBuilderFactory factory =
        (DocumentBuilderFactory) context.getService(ref);
    return factory.newDocumentBuilder();
}
```

In a more demanding case, the filtered version allows the bundle to select a parser that is validating and name-space aware:

```
SAXParser getParser(BundleContext context)
    throws Exception {
    ServiceReference refs[] = context.getServiceReferences(
        SAXParserFactory.class.getName(),
            "(&(parser.namespaceAware=true)"
        +   "(parser.validating=true))" );
    if ( refs == null )
        return null;
    SAXParserFactory factory =
        (SAXParserFactory) context.getService(refs[0]);
    return factory.newSAXParser();
}
```

702.6   Adapting a JAXP Parser to OSGi

If an XML Parser supports JAXP, then it can be converted to an OSGi aware bundle by adding a BundleActivator class which registers an XML Parser Service. The utility org.osgi.util.xml.XMLParserActivator class provides this function and can be added (copied, not referenced) to any XML Parser bundle, or it can be extended and customized if desired.

702.6.1   JAR Based Services

Its functionality is based on the definition of the [5] *JAR File specification, services directory*. This specification defines a concept for service providers. A JAR file can contain an implementation of an abstractly defined service. The class (or classes) implementing the service are designated from a file in the META-INF/services directory. The name of this file is the same as the abstract service class.

The content of the UTF-8 encoded file is a list of class names separated by new lines. White space is ignored and the number sign ('#' \u0023) is the comment character.

JAXP uses this service provider mechanism. It is therefore likely that vendors will place these service files in the META-INF/services directory.

702.6.2   XMLParserActivator

To support this mechanism, the XML Parser service provides a utility class that should be normally delivered with the OSGi framework implementation. This class is a Bundle Activator and must start when the bundle is started. This class is copied into the parser bundle, and *not* imported.

The start method of the utility BundleActivator class will look in the META-INF/services service provider directory for the files javax.xml.parsers.SAXParserFactory (SAXFACTORYNAME) or javax.xml.parsers.DocumentBuilderFactory (DOMFACTORYNAME). The full path name is specified in the constants SAXCLASSFILE and DOMCLASSFILE respectively.

If either of these files exist, the utility BundleActivator class will parse the contents according to the specification. A service provider file can contain multiple class names. Each name is read and a new instance is created. The following example shows the possible content of such a file:

```
# ACME example SAXParserFactory file
```

```
com.acme.saxparser.SAXParserFast          # Fast
com.acme.saxparser.SAXParserValidating    # Validates
```

Both the javax.xml.parsers.SAXParserFactory and the javax.xml.parsers.DocumentBuilderFactory provide methods that describe the features of the parsers they can create. The XMLParserActivator activator will use these methods to set the values of the properties, as defined in *Properties* on page 797, that describe the instances.

702.6.3 Adapting an Existing JAXP Compatible Parser

To incorporate this bundle activator into a XML Parser Bundle, do the following:

- If SAX parsing is supported, create a /META-INF/services/javax.xml.parsers.SAXParserFactory resource file containing the class names of the SAXParserFactory classes.
- If DOM parsing is supported, create a /META-INF/services/javax.xml.parsers.DocumentBuilderFactory file containing the fully qualified class names of the DocumentBuilderFactory classes.
- Create manifest file which imports the packages org.w3c.dom, org.xml.sax, and javax.xml.parsers.
- Add a Bundle-Activator header to the manifest pointing to the XMLParserActivator, the sub-class that was created, or a fully custom one.
- If the parsers support attributes, properties, or features that should be registered as properties so they can be searched, extend the XMLParserActivator class and override setSAXProperties(javax.xml.parsers.SAXParserFactory,Hashtable) and setDOMProperties(javax.xml.parsers.DocumentBuilderFactory,Hashtable).
- Ensure that custom properties are put into the Hashtable object. JAXP does not provide a way for XMLParserActivator to query the parser to find out what properties were added.
- Bundles that extend the XMLParserActivator class must call the original methods via super to correctly initialize the XML Parser Service properties.
- Compile this class into the bundle.
- Install the new XML Parser Service bundle.
- Ensure that the org.osgi.util.xml.XMLParserActivator class is contained in the bundle.

702.7 Usage of JAXP

A single bundle should export the JAXP, SAX, and DOM APIs. The version of contained packages must be appropriately labeled. JAXP 1.1 or later is required which references SAX 2 and DOM 2. See [4] *JAXP* for the exact version dependencies.

This specification is related to related packages as defined in the JAXP 1.1 document. The following table contains the expected minimum versions.

Table 702.1   *JAXP 1.1 minimum package versions*

Package	Minimum Version
javax.xml.parsers	1.1
org.xml.sax	2.0
org.xml.sax.helpers	2.0
org.xsml.sax.ext	1.0
org.w3c.dom	2.0

The Xerces project from the Apache group, [6] *Xerces 2 Java Parser*, contains a number libraries that implement the necessary APIs. These libraries can be wrapped in a bundle to provide the relevant packages.

702.8   Security

A centralized XML parser is likely to see sensitive information from other bundles. Provisioning an XML parser should therefore be limited to trusted bundles. This security can be achieved by providing ServicePermission[javax.xml.parsers.DocumentBuilderFactory| javax.xml.parsers.SAXFactory,REGISTER] to only trusted bundles.

Using an XML parser is a common function, and ServicePermission[javax.xml.parsers.DOMParserFactory|javax.xml.parsers.SAXFactory, GET] should not be restricted.

The XML parser bundle will need FilePermission[<<ALL FILES>>,READ] for parsing of files because it is not known beforehand where those files will be located. This requirement further implies that the XML parser is a system bundle that must be fully trusted.

702.9   org.osgi.util.xml

XML Parser Package Version 1.0.

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

Example import for consumers using the API in this package:

Import-Package: org.osgi.util.xml; version="[1.0,2.0)"

702.9.1   Summary

- XMLParserActivator - A BundleActivator class that allows any JAXP compliant XML Parser to register itself as an OSGi parser service.

702.9.2   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 Bundle-Activator class. Bundles who wish to use an XML parser can then use the framework's service registry to locate available XML Parsers with the desired characteristics such as validating and namespace-aware.

The services that this bundle activator enables a bundle to provide are:

- javax.xml.parsers.SAXParserFactory(SAXFACTORYNAME)
- javax.xml.parsers.DocumentBuilderFactory(DOMFACTORYNAME)

The algorithm to find the implementations of the abstract parsers is derived from the JAR file specifications, specifically the Services API.

An XMLParserActivator assumes that it can find the class file names of the factory classes in the following files:

- /META-INF/services/javax.xml.parsers.SAXParserFactory is a file contained in a jar available to the runtime which contains the implementation class name(s) of the SAXParserFactory.
- /META-INF/services/javax.xml.parsers.DocumentBuilderFactory is a file contained in a jar available to the runtime which contains the implementation class name(s) of the DocumentBuilder-Factory

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 DocumentBuilder-Factory. It registers each factory with the framework along with service properties:

- PARSER_VALIDATING- indicates if this factory supports validating parsers. It's value is a Boolean.
- PARSER_NAMESPACEAWARE- indicates if this factory supports namespace aware parsers It's value is a Boolean.

Individual parser implementations may have additional features, properties, or attributes which could be used to select a parser with a filter. These can be added by extending this class and overriding the setSAXProperties and setDOMProperties methods.

Concurrency   Thread-safe

702.9.2.1   **public static final String DOMCLASSFILE = "/META-INF/services/javax.xml.parsers.DocumentBuilderFactory"**

Fully qualified path name of DOM Parser Factory Class Name file

702.9.2.2   **public static final String DOMFACTORYNAME = "javax.xml.parsers.DocumentBuilderFactory"**

Filename containing the DOM Parser Factory Class name. Also used as the basis for the SERVICE_PID registration property.

702.9.2.3   **public static final String PARSER_NAMESPACEAWARE = "parser.namespaceAware"**

Service property specifying if factory is configured to support namespace aware parsers. The value is of type Boolean.

702.9.2.4   **public static final String PARSER_VALIDATING = "parser.validating"**

Service property specifying if factory is configured to support validating parsers. The value is of type Boolean.

702.9.2.5   **public static final String SAXCLASSFILE = "/META-INF/services/javax.xml.parsers.SAXParserFactory"**

Fully qualified path name of SAX Parser Factory Class Name file

702.9.2.6   **public static final String SAXFACTORYNAME = "javax.xml.parsers.SAXParserFactory"**

Filename containing the SAX Parser Factory Class name. Also used as the basis for the SERVICE_PID registration property.

702.9.2.7   **public XMLParserActivator()**

702.9.2.8   **public Object getService(Bundle bundle,ServiceRegistration registration)**

bundle   The bundle using the service.

registration   The ServiceRegistration object for the service.

□   Creates a new XML Parser Factory object.

A unique XML Parser Factory object is returned for each call to this method.

The returned XML Parser Factory object will be configured for validating and namespace aware support as specified in the service properties of the specified ServiceRegistration object. This method can be overridden to configure additional features in the returned XML Parser Factory object.

Returns   A new, configured XML Parser Factory object or null if a configuration error was encountered

702.9.2.9   **public void setDOMProperties(DocumentBuilderFactory factory,Hashtable props)**

factory   - the DocumentBuilderFactory object

props - Hashtable of service properties.

Set the customizable DOM Parser Service Properties.

This method attempts to instantiate a validating parser and a namespace aware parser to determine if the parser can support those features. The appropriate properties are then set in the specified props object.

This method can be overridden to add additional DOM2 features and properties. If you want to be able to filter searches of the OSGi service registry, this method must put a key, value pair into the properties object for each feature or property. For example, properties.put("http://www.acme.com/features/foo", Boolean.TRUE);

702.9.2.10   **public void setSAXProperties(SAXParserFactory factory,Hashtable properties)**

factory - the SAXParserFactory object

properties - the properties object for the service

Set the customizable SAX Parser Service Properties.

This method attempts to instantiate a validating parser and a namespace aware parser to determine if the parser can support those features. The appropriate properties are then set in the specified properties object.

This method can be overridden to add additional SAX2 features and properties. If you want to be able to filter searches of the OSGi service registry, this method must put a key, value pair into the properties object for each feature or property. For example, properties.put("http://www.acme.com/features/foo", Boolean.TRUE);

702.9.2.11   **public void start(BundleContext context) throws Exception**

context The execution context of the bundle being started.

☐ Called when this bundle is started so the Framework can perform the bundle-specific activities necessary to start this bundle. This method can be used to register services or to allocate any resources that this bundle needs.

This method must complete and return to its caller in a timely manner.

This method attempts to register a SAX and DOM parser with the Framework's service registry.

Throws Exception– If this method throws an exception, this bundle is marked as stopped and the Framework will remove this bundle's listeners, unregister all services registered by this bundle, and release all services used by this bundle.

702.9.2.12   **public void stop(BundleContext context) throws Exception**

context The execution context of the bundle being stopped.

☐ This method has nothing to do as all active service registrations will automatically get unregistered when the bundle stops.

Throws Exception– If this method throws an exception, the bundle is still marked as stopped, and the Framework will remove the bundle's listeners, unregister all services registered by the bundle, and release all services used by the bundle.

702.9.2.13   **public void ungetService(Bundle bundle,ServiceRegistration registration,Object service)**

bundle The bundle releasing the service.

registration The ServiceRegistration object for the service.

service The XML Parser Factory object returned by a previous call to the getService method.

☐ Releases a XML Parser Factory object.

702.10   References

[1]   *XML*
       http://www.w3.org/XML

[2]   *SAX*
       http://www.saxproject.org/

[3]   *DOM Java Language Binding*
       http://www.w3.org/TR/REC-DOM-Level-1/java-language-binding.html

[4]   *JAXP*
       http://jaxp.java.net/

[5]   *JAR File specification, services directory*
       http://download.oracle.com/javase/1.4.2/docs/guide/jar/jar.html

[6]   *Xerces 2 Java Parser*
       http://xerces.apache.org/xerces2-j/

705   Promises Specification

Version 1.0

705.1   Introduction

One of the fundamental pieces of an asynchronous programming model is the mechanism by which clients retrieve the result of the asynchronous task. Since Java 5, there has been a java.util.concurrent.Future interface available in the Java class libraries, which means that it is the *de facto* API in Java for handling the result of an asynchronous task. Futures have some limitations however in that they have no mechanism for registering callbacks. Java 8 introduces the class java.util.concurrent.CompletableFuture which addresses this but the requirement of Java 8 is unsuitable for many OSGi users at this time.

This specification defines a Promises API which can be used on many versions of Java including Java 5 and Java ME CDC/Foundation. The Promises API defined by this specification is independent of all other OSGi specifications including the OSGi Framework and thus can be easily used outside of the OSGi environment.

A Promise object holds the result of a potentially asynchronous task. The receiver of a Promise object can register callbacks on the Promise to be notified when the result is available or can block on the result becoming available. Promises can be chained together in powerful ways to handle asynchronous work flows and recovery.

Promises capture the effects of latency and errors by making these explicit in the API signatures. Latency is represented by callbacks which will eventually be called. Errors are represented by the failure member. In essence, this is what sets Promises apart from things such as RPC calls where such effects are not explicitly captured but rather attempted to be transparently handled.

705.1.1   Essentials
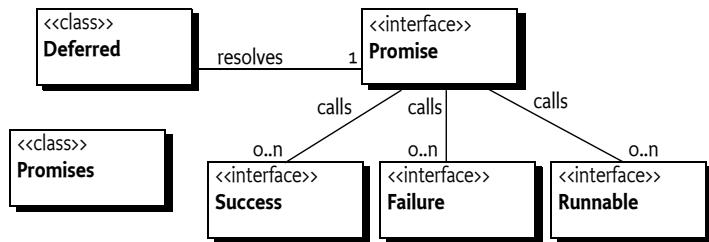
- *Common concepts* - The API is inspired by the Promises work in JavaScript and uses the same basic concepts. See [2] *JavaScript Promises*.
- *Independent* - The design is independent of all other OSGi specifications and can be used outside of an OSGi environment.
- *Asynchronous* - The design supports asynchronous tasks.
- *Small* - The API and implementation are very compact.
- *Complete* - The design provides a very complete set of operations for Promise which are primitives that can be used to address most use cases.
- *Monad* - The design supports monadic programming. See [4] *Monad*.
- *Resolution* - A Promise can be resolved successfully with a value or unsuccessfully with an exception.
- *Generified* - Generics are used to promote type safety.

705.1.2   Entities

- *Promise* - A Promise object holds the eventual result of a potentially asynchronous task.
- *Callback* - The receiver of a Promise can register callbacks on the Promise to be notified when the task is completed.

- *Deferred* - A Deferred object represents the potentially asynchronous task and is used to resolve the Promise.

Figure 705.1   *Class diagram of org.osgi.util.promise*

705.2   Promise

A Promise object holds the eventual result of a potentially asynchronous task. A Promise is either unresolved or resolved. An *unresolved* Promise does not have the result of the associated task available while a *resolved* Promise has the result of the associated task available. The isDone() method must return true if the Promise is resolved and false if the Promise is unresolved. A Promise must only be resolved once.

A resolved Promise can be either resolved with a value, which means the associated task *completed successfully* and supplied a result, or resolved with a failure, which means the associated task *completed unsuccessfully* and supplied an exception. The getFailure() method can be called to determine if the resolved Promise completed successfully with a value or unsuccessfully with a failure. If the getFailure() method returns a Throwable, the Promise resolved unsuccessfully with a failure. If the getFailure() method returns null, the Promise resolved successfully with a value that can be obtained from getValue().

If the Promise is unresolved, then calling getFailure() or getValue() must block until the Promise is resolved. In general, these two methods should not be used outside of a callback. Use callbacks to be notified when the Promise is resolved. See *Callbacks* on page 807.

705.3   Deferred

Promise is an interface which can allow for many Promise implementations. This API contains the Deferred class which provides a standard Promise implementation. A Deferred object can be created with the Deferred() constructor and the Promise associated with the new Deferred object can be obtained using getPromise(). This Promise can then be supplied to other parties who can use it to be notified of and obtain the eventual result.

```
public Promise<String> getTimeConsumingAnswer() {
  Deferred<String> deferred = new Deferred<String>();
  asynchronously(() -> doTask(deferred));
  return deferred.getPromise();
}
```

A Deferred object can later be used to resolve the associated Promise successfully by calling resolve(T) or unsuccessfully by calling fail(Throwable).

```
private void doTask(Deferred<String> deferred) {
  try {
    String answer = computeTimeConsumingAnswer();
```

```
      deferred.resolve(answer); // successfully resolve with value
    } catch (Exception e) {
      deferred.fail(e); // unsuccessfully resolve with exception
    }
  }
```

A Deferred object can also be used to resolve the associated Promise with the eventual result of another Promise by calling resolveWith(Promise).

```
private void doTask(Deferred<String> deferred) {
  try {
    Promise<String> answer = getPromiseWithTheAnswer();
    deferred.resolveWith(answer); // resolve with another Promise
  } catch (Exception e) {
    deferred.fail(e); // unsuccessfully resolve with exception
  }
}
```

If resolve(T) or fail(Throwable) is called when the Promise associated with the Deferred is already resolved, then an Illegal State Exception must be thrown.

Care must be taken in sharing a Deferred object with other parties since the other parties can resolve the associated Promise. A Deferred object should be made available only to the party that will responsible for resolving the associated Promise.

705.4   Callbacks

To be notified when a Promise has been resolved, callbacks are used. The Promise API provides two forms of callbacks: the basic Runnable callback and the more specialized Success and Failure callbacks.

A callback may be called on a different thread than the thread which registered the callback. So the callback must be thread safe but can rely upon that the registration of the callback *happens-before* the callback is called.

Resolving a Promise *happens-before* any registered callback is called. That is, for the resolved Promise, in a registered callback isDone() must return true and getValue() and getFailure() must not block.

Callbacks may be registered at any time including before and after a Promise has been resolved. If a callback is registered before the Promise is resolved, it will be called later when the Promise is resolved. If a callback is registered on an already resolved Promise, it will be called right away.

705.4.1   Runnable

The onResolve(Runnable) method is used to register a Runnable with the Promise which must be called when the Promise is resolved either successfully with a value or unsuccessfully with a failure. The resolved Promise is not passed to the Runnable, so if the Runnable implementation needs access to the resolved Promise, it must take care to ensure it has access.

```
final Promise<String> answer = getTimeConsumingAnswer();
answer.onResolve(new Runnable() {
  public void run() {
    doSomethingWithAnswer(answer);
  }
});
```

The onResolve(Runnable) method returns the Promise object upon which it is called.

705.4.2   Success and Failure

The then(Success) and then(Success,Failure) methods can be used to register the more specialized Success and Failure callbacks. The Success callback is only called if the Promise is successfully resolved with a value. The Failure callback is only called if the Promise is unsuccessfully resolved with a failure.

```
Promise<String> answer = getTimeConsumingAnswer();
answer.then(p -> processResult(p.getValue()), p -> handleFailure(p.getFailure()));
```

The then methods return a new Promise which can be used to chain Promises together.

705.5   Chaining Promises

The then(Success) and then(Success,Failure) methods also provide a means to chain Promises together. The then methods return a new Promise which is chained to the original Promise upon which the then method was called. The returned Promise must be resolved when the original Promise is resolved after the specified Success or Failure callback is executed. The result of the executed callback must be used to resolve the returned Promise. A sequence of calls to the then methods can be used to create a chain of promises which are resolved in sequence.

If the original Promise is successfully resolved, the Success callback is executed and the Promise returned by the Success callback, if any, or thrown exception is used to resolve the Promise returned from the then method. If the original Promise is resolved with a failure, the Failure callback is executed and the Promise returned from the then method is resolved with a failure.

In the following example, a Promise which will supply the name of the file to download is chained to a Promise which will return a mirror URL to use to download the file which is then further chained to a Promise which will return an Input Stream from which to read the download file.

```
Promise<String> name = getDownloadName();
Promise<URL> mirror = name.then(p -> getMirror(p.getValue()));
Promise<InputStream> in = mirror.then(p -> getStream(p.getValue()));
```

Since we probably do not need the intermediate Promises, we can collapse the chain into a single statement.

```
Promise<InputStream> in = getDownloadName().then(p -> getMirror(p.getValue()))
                                .then(p -> getStream(p.getValue()));
```

The chain of Promises will also propagate any exceptions that occur to resolve the last Promise in the chain which means we do not need to do any exception handling in the intermediate tasks.

705.6   Monad

The Promise API supports monadic programming. See [4] *Monad*. The Promise interface defines a number of interesting methods including map, flatMap and filter.

- filter(Predicate) - Filter the value of the Promise.

  If the Promise is successfully resolved, the predicate argument is called with the value of the Promise. If the predicate accepts the value, then the value is used to successfully resolve the Promise returned by the filter method. If the predicate does not accept the value, the Promise returned by the filter method is unsuccessfully resolved with a No Such Element Exception. If the predicate throws an exception, the Promise returned by the filter method is unsuccessfully resolved with that exception.

If the Promise is unsuccessfully resolved, the predicate argument is not called and the Promise returned by the filter method is unsuccessfully resolved with the failure of the Promise.

- `map(Function)` - Map the value of the Promise.

  If the Promise is successfully resolved, the function argument is called with the value of the Promise. The value returned by the function is used to successfully resolve the Promise returned by the map method. If the function throws an exception, the Promise returned by the map method is unsuccessfully resolved with that exception.

  If the Promise is unsuccessfully resolved, the function argument is not called and the Promise returned by the map method is unsuccessfully resolved with the failure of the Promise.

- `flatMap(Function)` - FlatMap the value of the Promise.

  If the Promise is successfully resolved, the function argument is called with the value of the Promise. The Promise returned by the function is used to resolve the Promise returned by the flatMap method. If the function throws an exception, the Promise returned by the flatMap method is unsuccessfully resolved with that exception.

  If the Promise is unsuccessfully resolved, the function argument is not called and the Promise returned by the flatMap method is unsuccessfully resolved with the failure of the Promise.

- `recover(Function)` - Recover from the unsuccessful resolution of the Promise with a recovery value.

  If the Promise is successfully resolved, the function argument is not called and the Promise returned by the recover method is resolved with the value of the Promise.

  If the Promise is unsuccessfully resolved, the function argument is called with the Promise to supply a recovery value. If the recovery value is not `null`, the Promise returned by the recover method is successfully resolved with the recovery value. If the recovery value is `null`, the Promise returned by the recover method is unsuccessfully resolved with the failure of the Promise. If the function throws an exception, the Promise returned by the recover method is unsuccessfully resolved with that exception.

- `recoverWith(Function)` - Recover from the unsuccessful resolution of the Promise with a recovery Promise.

  If the Promise is successfully resolved, the function argument is not called and the Promise returned by the recover method is resolved with the value of the Promise.

  If the Promise is unsuccessfully resolved, the function argument is called with the Promise to supply a recovery Promise. If the recovery Promise is not `null`, the Promise returned by the recover method is resolved with the recovery Promise. If the recovery Promise is `null`, the Promise returned by the recover method is unsuccessfully resolved with the failure of the Promise. If the function throws an exception, the Promise returned by the recover method is unsuccessfully resolved with that exception.

- `fallbackTo(Promise)` - Fall back to the value of the Promise argument if the Promise unsuccessfully resolves.

  If the Promise is successfully resolved, the Promise argument is not used and the Promise returned by the fallbackTo method is resolved with the value of the Promise.

  If the Promise is unsuccessfully resolved, the Promise argument is used to provide a fallback value when it becomes resolved. If the Promise argument is successfully resolved, the Promise returned by the fallbackTo method is resolved with the value of the Promise argument. If the Promise argument is unsuccessfully resolved, the Promise returned by the fallbackTo method is unsuccessfully resolved with the failure of the Promise.

These functions can be used to build pipelines of chained Promises that are processed in sequence. For example, in the following chain, the value of the original promise, once resolved, is filtered for

acceptable values. If the filter says the value is not acceptable, the recover method will be used to replace it with a default value.

```
return promise.filter(v -> isValueOk(v)).recover(p -> getDefaultValue())
```

With these chains, one can write powerful programs without the need to resort to complex if/else and try/catch logic.

705.7 Functional Interfaces

In Java 8, the concept of Functional Interfaces is introduced. See [5] *Function Interfaces*. Functional interfaces are interfaces with a single abstract method. Instances of functional interfaces can be created with lambda expressions, method references, or constructor references. Many methods on Promise take functional interface arguments and so are suitable for use with lambda expressions and method references in Java 8.

Two of these functional interfaces are Function and Predicate. These are equivalent to functional interfaces which are part of the java.util.function package introduced in Java 8. Since OSGi intends the Promise API to be usable on versions of Java prior to Java 8, we define our own interfaces. In the future, if Java 8 or later, becomes the base supported Java level for this specification, OSGi can update the Promise interface to add default methods which accept the java.util.function versions of these functional interfaces.

705.8 Promises Class

The Promises class provides several useful static methods when working with Promises.

Often, you may need to create an already resolved Promise to return or chain with another Promise. The resolved(T) method can be used to create a new Promise already successfully resolved with the specified value. The failed(Throwable) method can be used to create a new Promise already unsuccessfully resolved with the specified exception.

```
return getTimeConsumingAnswer().fallbackTo(Promises.resolved("Fallback Value"));
```

The Promises class also provides the all(Promise...) and all(Collection) methods which return a new Promise that is a latch on the specified Promises. The returned Promise must resolve only when all of the specified Promises have resolved.

705.9 Security

The Promise API does not define any OSGi services nor does the API perform any privileged actions. Therefore, it has no security considerations.

705.10 org.osgi.util.promise

Promise Package Version 1.0.

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

Example import for consumers using the API in this package:

Import-Package: org.osgi.util.promise; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.util.promise; version="[1.0,1.1)"

705.10.1   Summary

- `Deferred` - A Deferred Promise resolution.
- `FailedPromisesException` - Promise failure exception for a collection of failed Promises.
- `Failure` - Failure callback for a Promise.
- `Promise` - A Promise of a value.
- `Promises` - Static helper methods for Promises.
- `Success` - Success callback for a Promise.

705.10.2   public class Deferred<T>

⟨T⟩ The value type associated with the created Promise.

A Deferred Promise resolution.

Instances of this class can be used to create a Promise that can be resolved in the future. The associated Promise can be successfully resolved with resolve(Object) or resolved with a failure with fail(Throwable). It can also be resolved with the resolution of another promise using resolveWith(Promise).

The associated Promise can be provided to any one, but the Deferred object should be made available only to the party that will responsible for resolving the Promise.

Concurrency Immutable

705.10.2.1   public Deferred()

☐ Create a new Deferred with an associated Promise.

705.10.2.2   public void fail(Throwable failure)

failure The failure of the resolved Promise. Must not be null.

☐ Fail the Promise associated with this Deferred.

After the associated Promise is resolved with the specified failure, all registered callbacks are called and any chained Promises are resolved.

Resolving the associated Promise *happens-before* any registered callback is called. That is, in a registered callback, Promise.isDone() must return true and Promise.getValue() and Promise.getFailure() must not block.

Throws IllegalStateException– If the associated Promise was already resolved.

705.10.2.3   public Promise<T> getPromise()

☐ Returns the Promise associated with this Deferred.

Returns The Promise associated with this Deferred.

705.10.2.4   public void resolve(T value)

value The value of the resolved Promise.

☐ Successfully resolve the Promise associated with this Deferred.

After the associated Promise is resolved with the specified value, all registered callbacks are called and any chained Promises are resolved.

Resolving the associated Promise *happens-before* any registered callback is called. That is, in a registered callback, Promise.isDone() must return true and Promise.getValue() and Promise.getFailure() must not block.

Throws IllegalStateException– If the associated Promise was already resolved.

705.10.2.5 **public Promise<Void> resolveWith(Promise<? extends T> with)**

with A Promise whose value or failure must be used to resolve the associated Promise. Must not be null.

☐ Resolve the Promise associated with this Deferred with the specified Promise.

If the specified Promise is successfully resolved, the associated Promise is resolved with the value of the specified Promise. If the specified Promise is resolved with a failure, the associated Promise is resolved with the failure of the specified Promise.

After the associated Promise is resolved with the specified Promise, all registered callbacks are called and any chained Promises are resolved.

Resolving the associated Promise *happens-before* any registered callback is called. That is, in a registered callback, Promise.isDone() must return true and Promise.getValue() and Promise.getFailure() must not block.

Returns A Promise that is resolved only when the associated Promise is resolved by the specified Promise. The returned Promise must be successfully resolved with the value null, if the associated Promise was resolved by the specified Promise. The returned Promise must be resolved with a failure of IllegalStateException, if the associated Promise was already resolved when the specified Promise was resolved.

705.10.3 **public class FailedPromisesException**
extends RuntimeException

Promise failure exception for a collection of failed Promises.

705.10.3.1 **public FailedPromisesException(Collection<Promise<?>> failed,Throwable cause)**

failed A collection of Promises that have been resolved with a failure. Must not be null, must not be empty and all of the elements in the collection must not be null.

cause The cause of this exception. This is typically the failure of the first Promise in the specified collection.

☐ Create a new FailedPromisesException with the specified Promises.

705.10.3.2 **public Collection<Promise<?>> getFailedPromises()**

☐ Returns the collection of Promises that have been resolved with a failure.

Returns The collection of Promises that have been resolved with a failure. The returned collection is unmodifiable.

705.10.4 **public interface Failure**

Failure callback for a Promise.

A Failure callback is registered with a Promise using the Promise.then(Success, Failure) method and is called if the Promise is resolved with a failure.

This is a functional interface and can be used as the assignment target for a lambda expression or method reference.

Concurrency Thread-safe

705.10.4.1 **public void fail(Promise<?> resolved) throws Exception**

resolved The failed resolved Promise.

□ Failure callback for a Promise.

This method is called if the Promise with which it is registered resolves with a failure.

In the remainder of this description we will refer to the Promise returned by Promise.then(Success, Failure) when this Failure callback was registered as the chained Promise.

If this methods completes normally, the chained Promise must be failed with the same exception which failed the resolved Promise. If this method throws an exception, the chained Promise must be failed with the thrown exception.

Throws Exception– The chained Promise must be failed with the thrown exception.

705.10.5 public interface Promise‹T›

‹T› The value type associated with this Promise.

A Promise of a value.

A Promise represents a future value. It handles the interactions for asynchronous processing. A Deferred object can be used to create a Promise and later resolve the Promise. A Promise is used by the caller of an asynchronous function to get the result or handle the error. The caller can either get a callback when the Promise is resolved with a value or an error, or the Promise can be used in chaining. In chaining, callbacks are provided that receive the resolved Promise, and a new Promise is generated that resolves based upon the result of a callback.

Both callbacks and chaining can be repeated any number of times, even after the Promise has been resolved.

Example callback usage:

```
final Promise<String> foo = foo();
foo.onResolve(new Runnable() {
  public void run() {
    System.out.println(foo.getValue());
  }
});
```

Example chaining usage;

```
Success<String,String> doubler = new Success<String,String>() {
  public Promise<String> call(Promise<String> p) throws Exception {
    return Promises.resolved(p.getValue()+p.getValue());
  }
};
final Promise<String> foo = foo().then(doubler).then(doubler);
foo.onResolve(new Runnable() {
  public void run() {
    System.out.println(foo.getValue());
  }
});
```

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

705.10.5.1 public Promise‹T› fallbackTo(Promise‹? extends T› fallback)

fallback The Promise whose value must be used to resolve the returned Promise if this Promise resolves with a failure. Must not be null.

□ Fall back to the value of the specified Promise if this Promise fails.

If this Promise is successfully resolved, the returned Promise must be resolved with the value of this Promise.

If this Promise is resolved with a failure, the successful result of the specified Promise is used to resolve the returned Promise. If the specified Promise is resolved with a failure, the returned Promise must be failed with the failure of this Promise rather than the failure of the specified Promise.

This method may be called at any time including before and after this Promise has been resolved.

Returns A Promise that returns the value of this Promise or falls back to the value of the specified Promise.

705.10.5.2   **public Promise<T> filter(Predicate<?> predicate)**

predicate The Predicate to evaluate the value of this Promise. Must not be null.

☐ Filter the value of this Promise.

If this Promise is successfully resolved, the returned Promise must either be resolved with the value of this Promise, if the specified Predicate accepts that value, or failed with a NoSuchElementException, if the specified Predicate does not accept that value. If the specified Predicate throws an exception, the returned Promise must be failed with the exception.

If this Promise is resolved with a failure, the returned Promise must be failed with that failure.

This method may be called at any time including before and after this Promise has been resolved.

Returns A Promise that filters the value of this Promise.

705.10.5.3   **public Promise<R> flatMap(Function<?,Promise<? extends R>> mapper)**

Type Arguments <R>

⟨R⟩ The value type associated with the returned Promise.

mapper The Function that must flatMap the value of this Promise to a Promise that must be used to resolve the returned Promise. Must not be null.

☐ FlatMap the value of this Promise.

If this Promise is successfully resolved, the returned Promise must be resolved with the Promise from the specified Function as applied to the value of this Promise. If the specified Function throws an exception, the returned Promise must be failed with the exception.

If this Promise is resolved with a failure, the returned Promise must be failed with that failure.

This method may be called at any time including before and after this Promise has been resolved.

Returns A Promise that returns the value of this Promise as mapped by the specified Function.

705.10.5.4   **public Throwable getFailure() throws InterruptedException**

☐ Returns the failure of this Promise.

If this Promise is not resolved, this method must block and wait for this Promise to be resolved before completing.

If this Promise was resolved with a failure, this method returns with the failure of this Promise. If this Promise was successfully resolved, this method must return null.

Returns The failure of this resolved Promise or null if this Promise was successfully resolved.

Throws InterruptedException– If the current thread was interrupted while waiting.

705.10.5.5   **public T getValue() throws InvocationTargetException, InterruptedException**

☐ Returns the value of this Promise.

If this Promise is not resolved, this method must block and wait for this Promise to be resolved before completing.

If this Promise was successfully resolved, this method returns with the value of this Promise. If this Promise was resolved with a failure, this method must throw an InvocationTargetException with the failure exception as the cause.

Returns The value of this resolved Promise.

Throws InvocationTargetException– If this Promise was resolved with a failure. The cause of the InvocationTargetException is the failure exception.

InterruptedException– If the current thread was interrupted while waiting.

705.10.5.6   **public boolean isDone()**

☐ Returns whether this Promise has been resolved.

This Promise may be successfully resolved or resolved with a failure.

Returns true if this Promise was resolved either successfully or with a failure; false if this Promise is unresolved.

705.10.5.7   **public Promise<R> map(Function<?,? extends R> mapper)**

Type Arguments <R>

⟨R⟩ The value type associated with the returned Promise.

mapper The Function that must map the value of this Promise to the value that must be used to resolve the returned Promise. Must not be null.

☐ Map the value of this Promise.

If this Promise is successfully resolved, the returned Promise must be resolved with the value of specified Function as applied to the value of this Promise. If the specified Function throws an exception, the returned Promise must be failed with the exception.

If this Promise is resolved with a failure, the returned Promise must be failed with that failure.

This method may be called at any time including before and after this Promise has been resolved.

Returns A Promise that returns the value of this Promise as mapped by the specified Function.

705.10.5.8   **public Promise<T> onResolve(Runnable callback)**

callback A callback to be called when this Promise is resolved. Must not be null.

☐ Register a callback to be called when this Promise is resolved.

The specified callback is called when this Promise is resolved either successfully or with a failure.

This method may be called at any time including before and after this Promise has been resolved.

Resolving this Promise *happens-before* any registered callback is called. That is, in a registered callback, isDone() must return true and getValue() and getFailure() must not block.

A callback may be called on a different thread than the thread which registered the callback. So the callback must be thread safe but can rely upon that the registration of the callback *happens-before* the registered callback is called.

Returns This Promise.

705.10.5.9   **public Promise<T> recover(Function<Promise<?>,? extends T> recovery)**

recovery If this Promise resolves with a failure, the specified Function is called to produce a recovery value to be used to resolve the returned Promise. Must not be null.

☐ Recover from a failure of this Promise with a recovery value.

If this Promise is successfully resolved, the returned Promise must be resolved with the value of this Promise.

If this Promise is resolved with a failure, the specified Function is applied to this Promise to produce a recovery value.

- If the recovery value is not null, the returned Promise must be resolved with the recovery value.
- If the recovery value is null, the returned Promise must be failed with the failure of this Promise.
- If the specified Function throws an exception, the returned Promise must be failed with that exception.

To recover from a failure of this Promise with a recovery value of null, the recoverWith(Function) method must be used. The specified Function for recoverWith(Function) can return Promises.resolved(null) to supply the desired null value.

This method may be called at any time including before and after this Promise has been resolved.

Returns A Promise that resolves with the value of this Promise or recovers from the failure of this Promise.

705.10.5.10   **public Promise<T> recoverWith(Function<Promise<?>,Promise<? extends T>> recovery)**

recovery If this Promise resolves with a failure, the specified Function is called to produce a recovery Promise to be used to resolve the returned Promise. Must not be null.

□ Recover from a failure of this Promise with a recovery Promise.

If this Promise is successfully resolved, the returned Promise must be resolved with the value of this Promise.

If this Promise is resolved with a failure, the specified Function is applied to this Promise to produce a recovery Promise.

- If the recovery Promise is not null, the returned Promise must be resolved with the recovery Promise.
- If the recovery Promise is null, the returned Promise must be failed with the failure of this Promise.
- If the specified Function throws an exception, the returned Promise must be failed with that exception.

This method may be called at any time including before and after this Promise has been resolved.

Returns A Promise that resolves with the value of this Promise or recovers from the failure of this Promise.

705.10.5.11   **public Promise<R> then(Success<?,? extends R> success,Failure failure)**

Type Arguments <R>

<R> The value type associated with the returned Promise.

success A Success callback to be called when this Promise is successfully resolved. May be null if no Success callback is required. In this case, the returned Promise must be resolved with the value null when this Promise is successfully resolved.

failure A Failure callback to be called when this Promise is resolved with a failure. May be null if no Failure callback is required.

□ Chain a new Promise to this Promise with Success and Failure callbacks.

The specified Success callback is called when this Promise is successfully resolved and the specified Failure callback is called when this Promise is resolved with a failure.

This method returns a new Promise which is chained to this Promise. The returned Promise must be resolved when this Promise is resolved after the specified Success or Failure callback is executed. The result of the executed callback must be used to resolve the returned Promise. Multiple calls to this method can be used to create a chain of promises which are resolved in sequence.

If this Promise is successfully resolved, the Success callback is executed and the result Promise, if any, or thrown exception is used to resolve the returned Promise from this method. If this Promise is resolved with a failure, the Failure callback is executed and the returned Promise from this method is failed.

This method may be called at any time including before and after this Promise has been resolved.

Resolving this Promise *happens-before* any registered callback is called. That is, in a registered callback, isDone() must return true and getValue() and getFailure() must not block.

A callback may be called on a different thread than the thread which registered the callback. So the callback must be thread safe but can rely upon that the registration of the callback *happens-before* the registered callback is called.

Returns A new Promise which is chained to this Promise. The returned Promise must be resolved when this Promise is resolved after the specified Success or Failure callback, if any, is executed.

705.10.5.12 **public Promise<R> then(Success<?,? extends R> success)**

Type Arguments `<R>`

⟨*R*⟩ The value type associated with the returned Promise.

success A Success callback to be called when this Promise is successfully resolved. May be null if no Success callback is required. In this case, the returned Promise must be resolved with the value null when this Promise is successfully resolved.

☐ Chain a new Promise to this Promise with a Success callback.

This method performs the same function as calling then(Success, Failure) with the specified Success callback and null for the Failure callback.

Returns A new Promise which is chained to this Promise. The returned Promise must be resolved when this Promise is resolved after the specified Success, if any, is executed.

See Also then(Success, Failure)

705.10.6 public class Promises

Static helper methods for Promises.

Concurrency Thread-safe

705.10.6.1 **public static Promise<List<T>> all(Collection<Promise<S>> promises)**

Type Arguments `<T,S extends T>`

⟨*T*⟩ The value type of the List value associated with the returned Promise.

⟨*S*⟩ A subtype of the value type of the List value associated with the returned Promise.

promises The Promises which must be resolved before the returned Promise must be resolved. Must not be null and all of the elements in the collection must not be null.

☐ Create a new Promise that is a latch on the resolution of the specified Promises.

The new Promise acts as a gate and must be resolved after all of the specified Promises are resolved.

Returns A Promise that is resolved only when all the specified Promises are resolved. The returned Promise must be successfully resolved with a List of the values in the order of the specified Promises if all the specified Promises are successfully resolved. The List in the returned Promise is the property of the caller and is modifiable. The returned Promise must be resolved with a failure of FailedPromisesException if any of the specified Promises are resolved with a failure. The failure FailedPromisesException must contain all of the specified Promises which resolved with a failure.

705.10.6.2 **public static Promise<List<T>> all(Promise<? extends T> ... promises)**

Type Arguments <T>

〈T〉 The value type associated with the specified Promises.

promises The Promises which must be resolved before the returned Promise must be resolved. Must not be null and all of the arguments must not be null.

□ Create a new Promise that is a latch on the resolution of the specified Promises.

The new Promise acts as a gate and must be resolved after all of the specified Promises are resolved.

Returns A Promise that is resolved only when all the specified Promises are resolved. The returned Promise must be successfully resolved with a List of the values in the order of the specified Promises if all the specified Promises are successfully resolved. The List in the returned Promise is the property of the caller and is modifiable. The returned Promise must be resolved with a failure of FailedPromisesException if any of the specified Promises are resolved with a failure. The failure FailedPromisesException must contain all of the specified Promises which resolved with a failure.

705.10.6.3 **public static Promise<T> failed(Throwable failure)**

Type Arguments <T>

〈T〉 The value type associated with the returned Promise.

failure The failure of the resolved Promise. Must not be null.

□ Create a new Promise that has been resolved with the specified failure.

Returns A new Promise that has been resolved with the specified failure.

705.10.6.4 **public static Promise<T> resolved(T value)**

Type Arguments <T>

〈T〉 The value type associated with the returned Promise.

value The value of the resolved Promise.

□ Create a new Promise that has been resolved with the specified value.

Returns A new Promise that has been resolved with the specified value.

705.10.7 **public interface Success<T,R>**

〈T〉 The value type of the resolved Promise passed as input to this callback.

〈R〉 The value type of the returned Promise from this callback.

Success callback for a Promise.

A Success callback is registered with a Promise using the Promise.then(Success) method and is called if the Promise is resolved successfully.

This is a functional interface and can be used as the assignment target for a lambda expression or method reference.

Concurrency Thread-safe

705.10.7.1 **public Promise<R> call(Promise<T> resolved) throws Exception**

resolved The successfully resolved Promise.

□ Success callback for a Promise.

This method is called if the Promise with which it is registered resolves successfully.

In the remainder of this description we will refer to the Promise returned by this method as the returned Promise and the Promise returned by Promise.then(Success) when this Success callback was registered as the chained Promise.

If the returned Promise is null then the chained Promise must resolve immediately with a successful value of null. If the returned Promise is not null then the chained Promise must be resolved when the returned Promise is resolved.

Returns   The Promise to use to resolve the chained Promise, or null if the chained Promise is to be resolved immediately with the value null.

Throws   Exception– The chained Promise must be failed with the thrown exception.

705.11   org.osgi.util.function

Function Package Version 1.0.

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

Example import for consumers using the API in this package:

Import-Package: org.osgi.util.function; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.util.function; version="[1.0,1.1)"

705.11.1   Summary

- Function - A function that accepts a single argument and produces a result.
- Predicate - A predicate that accepts a single argument and produces a boolean result.

705.11.2   public interface Function<T,R>

‹T›   The type of the function input.

‹R›   The type of the function output.

A function that accepts a single argument and produces a result.

This is a functional interface and can be used as the assignment target for a lambda expression or method reference.

Concurrency   Thread-safe

705.11.2.1   public R apply(T t)

t   The input to this function.

□   Applies this function to the specified argument.

Returns   The output of this function.

705.11.3   public interface Predicate<T>

‹T›   The type of the predicate input.

A predicate that accepts a single argument and produces a boolean result.

This is a functional interface and can be used as the assignment target for a lambda expression or method reference.

Concurrency   Thread-safe

705.11.3.1   **public boolean test(T t)**

t   The input to this predicate.

□   Evaluates this predicate on the specified argument.

Returns   true if the specified argument is accepted by this predicate; false otherwise.

705.12   References

[1]   *JavaScript Promises*
      https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

[2]   *JavaScript Promises*
      http://www.html5rocks.com/en/tutorials/es6/promises/

[3]   *ECMAScript 6 drafts*
      http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts

[4]   *Monad*
      https://en.wikipedia.org/wiki/Monad_%28functional_programming%29

[5]   *Function Interfaces*
      http://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.8

End Of Document