OSGi Working Group

OSGi Compendium

Release 8.1
December 2022

Preface

Implementation Requirements

An implementation of a Specification: (i) must fully implement the Specification including all its required interfaces and functionality; (ii) must not modify, subset, superset or otherwise extend the OSGi Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the OSGi Name Space other than those required and authorized by the Specification. An implementation that does not satisfy limitations (i)-(ii) is not considered an implementation of the Specification and must not be described as an implementation of the Specification. "OSGi Name Space" shall mean the public class or interface declarations whose names begin with "org.osgi" or any recognized successors or replacements thereof. An implementation of a Specification must not claim to be a compatible implementation of the Specification unless it passes the Technology Compatibility Kit ("TCK") for the Specification.

Feedback

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

`https://docs.osgi.org/specification/`

Comments about this specification can be raised at:

`https://github.com/osgi/osgi/issues`

Table of Contents

1 Introduction 5

1.1 Reader Level. 5
1.2 Version Information. 5
1.3 References. 8
1.4 Changes. 8

104 Configuration Admin Service Specification 11

104.1 Introduction. 11
104.2 Configuration Targets. 13
104.3 The Persistent Identity. 14
104.4 The Configuration Object. 17
104.5 Managed Service. 20
104.6 Managed Service Factory. 23
104.7 Configuration Admin Service. 27
104.8 Configuration Events. 32
104.9 Configuration Plugin. 33
104.10 Meta Typing. 35
104.11 Coordinator Support. 36
104.12 Capabilities. 36
104.13 Security. 37
104.14 org.osgi.service.cm. 39
104.15 org.osgi.service.cm.annotations. 59

135 Common Namespaces Specification 61

135.1 Introduction. 61
135.2 osgi.extender Namespace. 61
135.3 osgi.contract Namespace. 63
135.4 osgi.service Namespace. 65
135.5 osgi.implementation Namespace. 65
135.6 osgi.unresolvable Namespace. 66
135.7 org.osgi.namespace.contract. 66
135.8 org.osgi.namespace.extender. 67
135.9 org.osgi.namespace.service. 68
135.10 org.osgi.namespace.implementation. 68
135.11 org.osgi.namespace.unresolvable. 69
135.12 References. 70

159 Feature Service Specification 71

159.1 Introduction. 71
159.2 Feature. 72

159.3 Comments. 74
159.4 Bundles. 74
159.5 Configurations. 76
159.6 Variables. 76
159.7 Extensions. 77
159.8 Framework Launching Properties. 79
159.9 Resource Versioning. 80
159.10 Capabilities. 80
159.11 org.osgi.service.feature. 80
159.12 org.osgi.service.feature.annotation. 92
159.13 References. 92

160 Feature Launcher Service Specification 93

160.1 Introduction. 93
160.2 Features and Artifact Repositories. 94
160.3 Common themes. 96
160.4 The Feature Launcher. 98
160.5 The Feature Runtime Service. 103
160.6 Capabilities. 114
160.7 Security. 115
160.8 org.osgi.service.featurelauncher. 115
160.9 org.osgi.service.featurelauncher.annotation. 119
160.10 org.osgi.service.featurelauncher.decorator. 120
160.11 org.osgi.service.featurelauncher.repository. 123
160.12 org.osgi.service.featurelauncher.runtime. 125
160.13 References. 135

1 Introduction

This compendium contains the specifications of all current OSGi services.

1.1 Reader Level

This specification is written for the following audiences:

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

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

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

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

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

1.2 Version Information

This document is the Compendium Specification for the OSGi Compendium Release 8.1.

1.2.1 OSGi Core Release 8

This specification is based on OSGi Core Release 8. This specification can be downloaded from:

```
https://docs.osgi.org/specification/
```

1.2.2 Component Versions

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

Table 1.1 *Packages and versions*

Item	Package	Version
???	-	???
???	???	???
104 *Configuration Admin Service Specification*	org.osgi.service.cm	Version 1.6
	org.osgi.service.cm.annotations	

Item	Package	Version
???	???	???
	???	
???	???	???
???	???	???
???	???	???
???	???	???
???	???	???
	???	
	???	
	???	
	???	
???	???	???
	???	
	???	
???	???	???
	???	
	???	
	???	
	???	
???	???	???
	???	
???	-	???
???	???	???
???	???	???
???	???	???
	???	
???	-	???
???	???	???
???	???	???
???	???	???
???	???	???
135 *Common Namespaces Specification*	org.osgi.namespace.contract	Version 1.2
	org.osgi.namespace.extender	
	org.osgi.namespace.implementation	
	org.osgi.namespace.service	
	org.osgi.namespace.unresolvable	
???	???	???
	???	
???	???	???
	???	

Item	Package	Version
???	???	???
	???	
???	???	???
	???	
	???	
	???	
	???	
	???	
???	???	???
???	???	???
	???	
???	???	???
???	???	???
	???	
???	???	???
???	???	???
???	???	???
	???	
	???	
	???	
???	???	???
	???	
???	???	???
	???	
	???	
	???	
???	???	???
	???	
	???	
???	???	???
	???	
	???	
	???	
	???	
	???	

Item	Package	Version
???	???	???
	???	
	???	
	???	
	???	
	???	
	???	
???	???	???
	???	
???	???[1]	???
???	-	???
???	???	???
	???	
	???	
	???	
???	???	???
159 *Feature Service Specification*	org.osgi.service.feature	Version 1.0
???	???	???
???	???	???
	???	
???	???	???
???	???	???

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.

1.2.3 Notes

1. This is not a Java package but contains DMT Types.

1.3 References

[1] *OSGi Specifications*
https://docs.osgi.org/specification/

1.4 Changes

- Updated ???.
- Updated ???.
- Updated ???.
- ??? updated for Jakarta EE and replaces the Http Whiteboard specification which is based upon the javax-namespace Servlet API. The old Http Service specification was also removed as it is based upon the old version 2.1 of the javax-namespace Servlet API.

- ??? updated for Jakarta EE and replaces the JaxRS Whiteboard specification which is based upon the javax-namespace JAX-RS API.

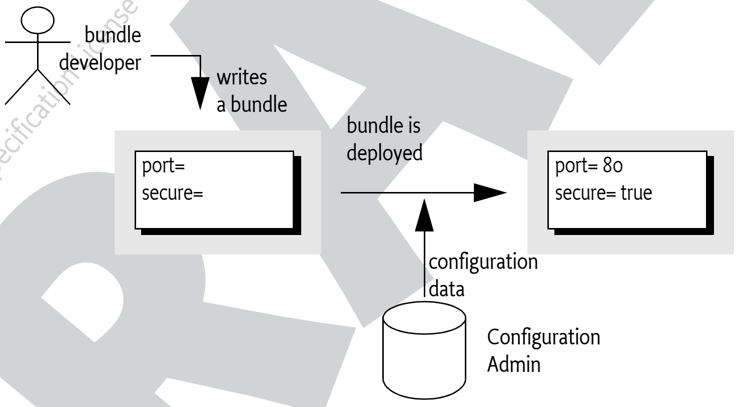
104 Configuration Admin Service Specification

Version 1.6

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

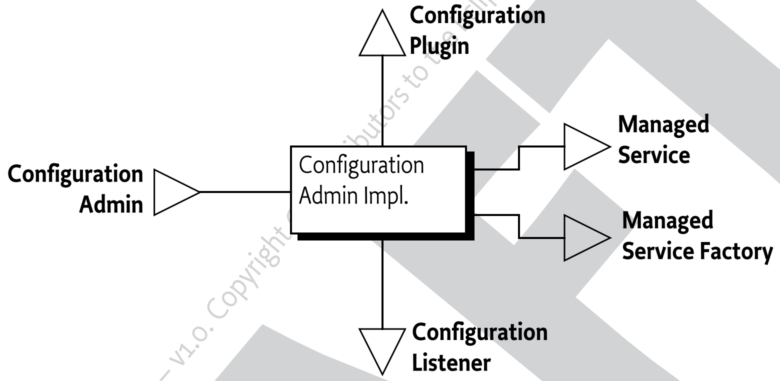
  Trade-offs in simplicity should be made at the expense of the bundle implementing the Configuration Admin service and in favor of bundles that need configuration information. The reason for this choice is that normal bundles will outnumber Configuration Admin bundles.
- *Regions* - It should be possible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

104.1.2 Entities

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

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

Figure 104.2 Overall Service Diagram



104.1.3 Synopsis

This specification is based on the concept of a Configuration Admin service that manages the configuration of an OSGi 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.

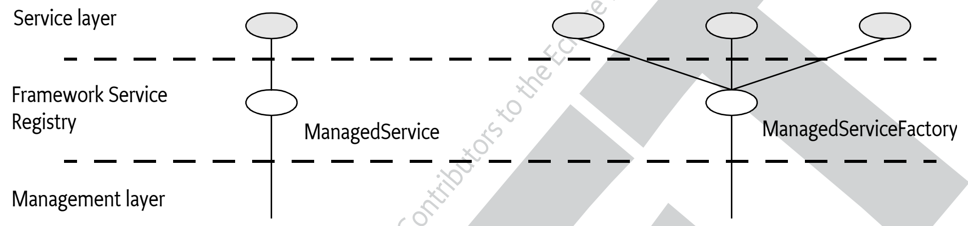
104.2 Configuration Targets

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 17.

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

104.3.2 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 14 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), getFactoryConfiguration(String,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) as well as the three argument version of getFactoryConfiguration(String,String,String) take a location String as their last 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 30 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 8* 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 ???.

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 33. 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 15.
- 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 35.

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

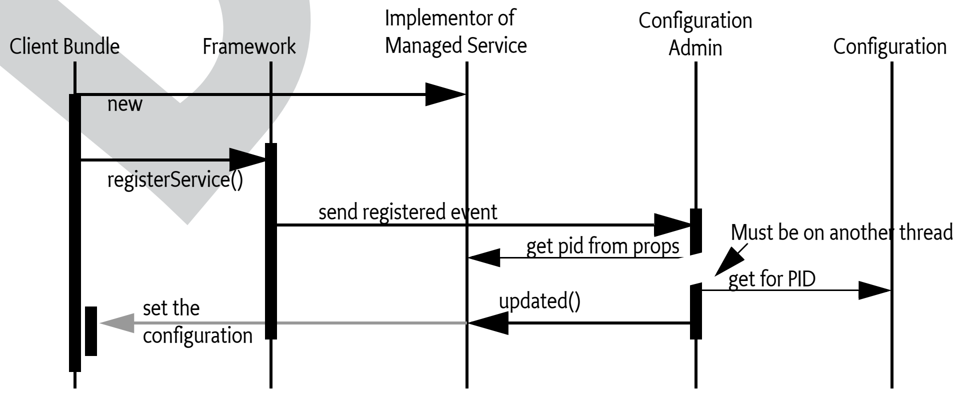
- The configuration stored for the registered PID must be retrieved. If there is a Configuration object for this PID and the configuration is visible for the associated bundle then it is sent to the Managed Service with updated(Dictionary).
- If a Managed Service is registered and no configuration information 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 17.

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

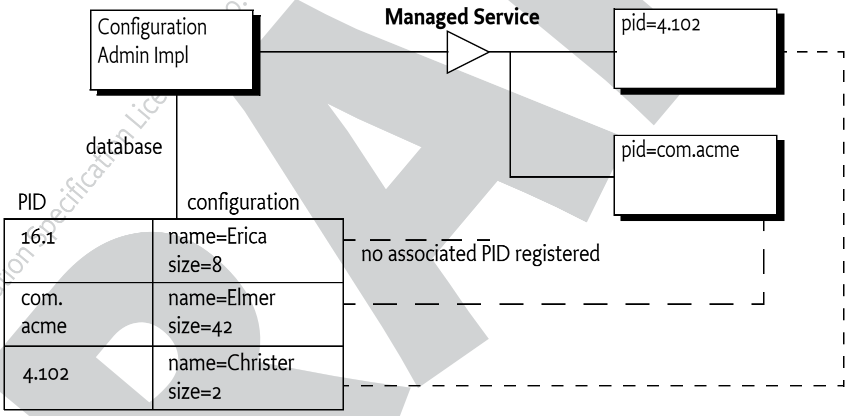
When a Managed Service is registered, the default properties may be visible in the service registry for a short period before they are replaced by the properties of the actual configuration dictionary. Care should be taken that this visibility does not cause race conditions for other bundles.

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.

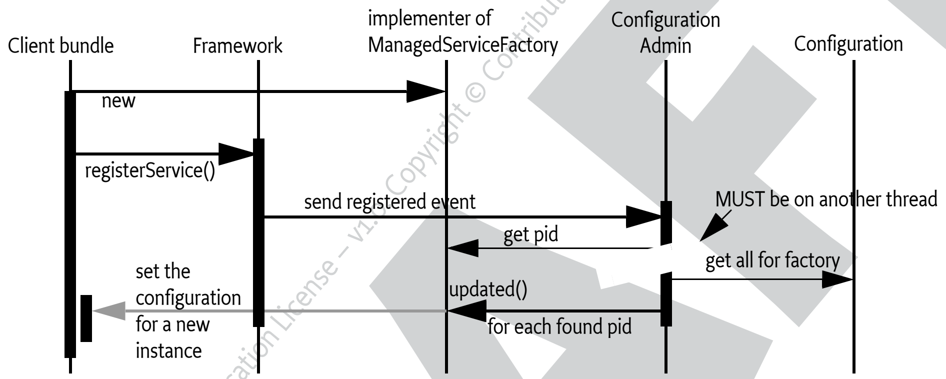
When the Configuration Admin service detects the registration of a Managed Service Factory, it must find all visible configuration dictionaries for this factory and must then sequentially call ManagedServiceFactory.updated(String,Dictionary) for each configuration dictionary. The first argument is the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

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

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

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

Figure 104.6 *Managed Service Factory Action Diagram*



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

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

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

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

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

104.6.3 Deletion

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

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

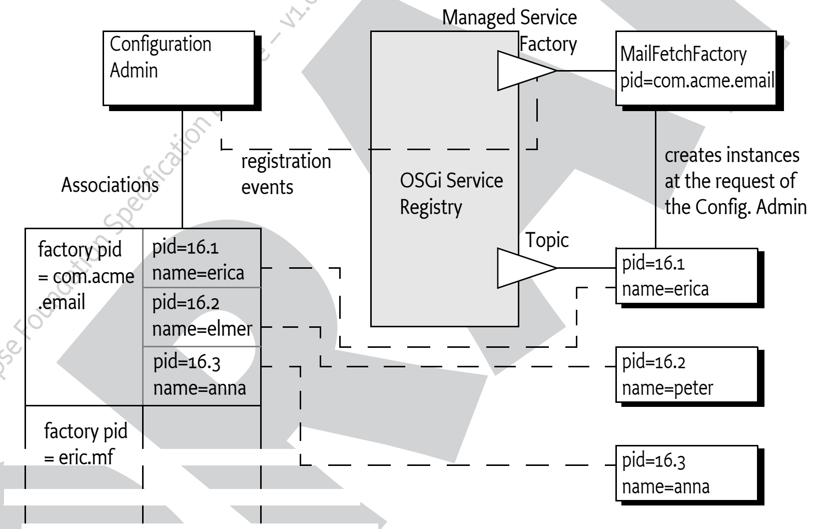
104.6.4 Managed Service Factory Example

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

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

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

Figure 104.7 Managed Service Factory Example



104.6.5 Multiple Consoles Example

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

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

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

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

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

104.7 Configuration Admin Service

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

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

104.7.1 Creating a Managed Service Configuration Object

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

Two variants of this method are:

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

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

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

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

104.7.2 Creating a Managed Service Factory Configuration Object

The ConfigurationAdmin class provides two sets of methods to create a new Configuration for a Managed Service Factory. The first set delegates the creation of the unique PID to the Configuration Admin service. The second set allows the caller to influence the generation of the PID.

The ConfigurationAdmin class provides the following two methods which generate a unique PID when creating a new Configuration for a Managed Service Factory. A new, unique PID is created for the Configuration 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.

- 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.

The ConfigurationAdmin class provides the following two methods allowing the caller to influence the generation of the PID when creating a new Configuration for a Managed Service Factory. The PID for the Configuration object is generated from the provided factory PID and the provided name by starting with the factory PID, appending a tilde ('~' \u007e), and then appending the name. The getFactoryConfiguration methods must atomically create and persistently store a Configuration object if it does not yet exist.

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

Creating a new 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 17.

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 15.

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), getConfiguration(String) or getFactoryConfiguration(String,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 methods in Configuration objects are not executed synchronously with the related target services updated method. The updated method must be called asynchronously. The Configuration Admin service, however, must have updated the persistent storage before the update method returns.

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

Invoking the update(Dictionary) method results in Configuration Admin service blindly updating the Configuration object and performing the above outlined actions. This even happens if the updated set of properties is the same as the already existing properties in the Configuration object.

To optimize configuration updates if the caller does not know whether properties of a Configuration object have changed, the updateIfDifferent(Dictionary) method can be used. The provided dictionary is compared with the existing properties. If there is no change, no action is taken. If there is any change detected, updateIfDifferent(Dictionary) acts exactly as update(Dictionary). Properties are compared as follows:

- Scalars are compared using equals
- Arrays are compared using Arrays.equals
- Collections are compared using equals

The boolean result of updateIfDifferent(Dictionary) is true if the Configuration object has been updated.

If the Configuration object has the READ_ONLY attribute set, calling one of the update methods results in a ReadOnlyConfigurationException and the configuration is not changed.

104.7.5 Using Multi-Locations

Sharing configuration between different bundles can be done using multi-locations, see *Location Binding* on page 17. 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 30. 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 17, 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 delete method must also asynchronously send out a Configuration Event CM_DELETED to all registered Configuration Listeners.

If the Configuration object has the READ_ONLY attribute set, calling the delete method results in a ReadOnlyConfigurationException and the configuration is not deleted.

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.

104.7.9 Configuration Attributes

The Configuration object supports attributes, similar to setting attributes on files in a file system. Currently only the READ_ONLY attribute is supported.

Attributes can be set by calling the addAttributes(ConfigurationAttribute...) method and listing the attributes to be added. In the same way attributes can be removed by calling removeAttributes(ConfigurationAttribute...). Each successful change in attributes is persisted.

A Bundle can only change the attributes if it has Configuration Permission with the ATTRIBUTE action. Otherwise a Security Exception is thrown.

The currently set attributes can be queried using the getAttributes() method.

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 8*.

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 via the Event Admin service, 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.

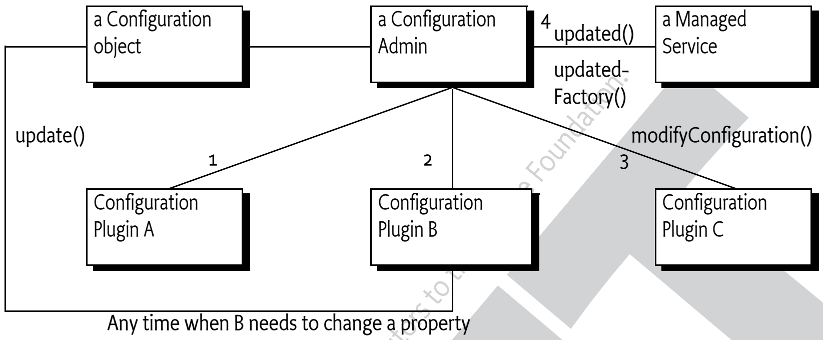
The ConfigurationPlugin interface has only one method: modifyConfiguration(ServiceReference,Dictionary). This method inspects or modifies configuration data.

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 19.

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. In the event of more than one plugin having the same value of service.cmRanking, then the order in which these are called is undefined.

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. Any modification from the Configuration Plugin service is ignored.
>= 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. Any modification from the Configuration Plugin service is ignored.

104.9.6 Manual Invocation

The Configuration Admin service ensures that Configuration Plugin services are automatically called for a Managed Service or a Managed Service Factory as outlined above. If a bundle needs to get the configuration properties processed by the Configuration Plugin services, the getProcessedProperties(ServiceReference) method provides this view.

The service reference passed into the method must either point to a Managed Service or Managed Service Factory registered on behalf of the bundle getting the processed properties. If that service should not be called by the Configuration Admin service, that service must be registered without a PID service property.

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 Coordinator Support

The ??? defines a mechanism for multiple parties to collaborate on a common task without *a priori* knowledge of who will collaborate in that task. The Configuration Admin service must participate in such scenarios to coordinate with provisioning or configuration tasks.

If configurations are created, updated or deleted and an implicit coordination exists, the Configuration Admin service must delay notifications until the coordination terminates. However the configuration changes must be persisted immediately. Updating a Managed Service or Managed Service Factory and informing asynchronous listeners is delayed until the coordination terminates, regardless of whether the coordination fails or terminates regularly. Registered synchronous listeners will be informed immediately when the change happens regardless of a coordination.

104.12 Capabilities

104.12.1 osgi.implementation Capability

The Configuration Admin implementation bundle must provide the osgi.implementation capability with the name osgi.cm. This capability can be used by provisioning tools and during resolution to ensure that a Configuration Admin implementation is present to manage configurations. The capability must also declare a uses constraint for the org.osgi.service.cm package and provide the version of this specification:

```
Provide-Capability: osgi.implementation;
       osgi.implementation="osgi.cm";
       uses:="org.osgi.service.cm";
       version:Version="1.6"
```

This capability must follow the rules defined for the *osgi.implementation Namespace* on page 65.

Bundles relying on the Configuration Admin service should require the osgi.implementation capability from the Configuration Admin Service.

```
Require-Capability: osgi.implementation;
  filter:="(&(osgi.implementation=osgi.cm)(version>=1.6)(!(version>=2.0)))"
```

This requirement can be easily generated using the RequireConfigurationAdmin annotation.

104.12.2 osgi.service Capability

The bundle providing the Configuration Admin 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.cm package:

```
Provide-Capability: osgi.service;
  objectClass:List<String>="org.osgi.service.cm.ConfigurationAdmin";
  uses:="org.osgi.service.cm"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 65.

104.13 Security

104.13.1 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
- ATTRIBUTE - Can manage attributes for 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.

104.13.2 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.

104.13.3 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.

104.14 org.osgi.service.cm

Configuration Admin Package Version 1.6.

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="[1.6,2.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.cm; version="[1.6,1.7)"`

104.14.1 Summary

- `Configuration` - The configuration information for a `ManagedService` or `ManagedServiceFactory` object.
- `Configuration.ConfigurationAttribute` - Configuration Attributes.
- `ConfigurationAdmin` - Service for administering configuration data.
- `ConfigurationConstants` - Defines standard constants for the Configuration Admin service.
- `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.
- `ReadOnlyConfigurationException` - An Exception class to inform the client of a `Configuration` about the read only state of a configuration object.
- `SynchronousConfigurationListener` - Synchronous Listener for Configuration Events.

104.14.2 Permissions

104.14.2.1 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
- `addAttributes(ConfigurationAttribute...)`
  - `ConfigurationPermission[this.location,ATTRIBUTE]` - if this.location is not null
  - `ConfigurationPermission["*",ATTRIBUTE]` - if this.location is null
- `removeAttributes(ConfigurationAttribute...)`

  - ConfigurationPermission[this.location,ATTRIBUTE] - if this.location is not null
  - ConfigurationPermission["*",ATTRIBUTE] - if this.location is null

104.14.2.2 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
- getFactoryConfiguration(String,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
- getFactoryConfiguration(String,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

104.14.2.3 ManagedService

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

104.14.2.4 ManagedServiceFactory

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

104.14.3 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

104.14.3.1 public void addAttributes(Configuration.ConfigurationAttribute... attrs) throws IOException

attrs The attributes to add.

□ Add attributes to the configuration.

Throws IOException– If the new state cannot be persisted.

IllegalStateException– If this configuration has been deleted.

SecurityException– when the required permissions are not available

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

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

Since 1.6

104.14.3.2 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 ReadOnlyConfigurationException– If the configuration is read only.

IOException– If delete fails.

IllegalStateException– If this configuration has been deleted.

104.14.3.3 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.

104.14.3.4 public Set<Configuration.ConfigurationAttribute> getAttributes()

□ Get the attributes of this configuration.

Returns The set of attributes.

Throws IllegalStateException– If this configuration has been deleted.

Since 1.6

104.14.3.5 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

104.14.3.6 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

104.14.3.7 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.

104.14.3.8 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

104.14.3.9 public Dictionary<String, Object> getProcessedProperties(ServiceReference<?> reference)

reference The reference to the Managed Service or Managed Service Factory to pass to the registered ConfigurationPlugins handling this configuration. Must not be null.

□ Return the processed 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.

Before the properties are returned they are processed by all the registered ConfigurationPlugins handling this configuration.

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

Returns A private copy of the processed 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.

Since 1.6

104.14.3.10 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.

104.14.3.11 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

104.14.3.12 public void removeAttributes(Configuration.ConfigurationAttribute... attrs) throws IOException

attrs The attributes to remove.

□ Remove attributes from this configuration.

Throws IOException– If the new state cannot be persisted.

IllegalStateException– If this configuration has been deleted.

SecurityException– when the required permissions are not available

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

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

Since 1.6

104.14.3.13 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

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

104.14.3.14 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 ReadOnlyConfigurationException– If the configuration is read only.

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.

104.14.3.15 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

104.14.3.16 public boolean updateIfDifferent(Dictionary<String, ?> properties) throws IOException

properties The new set of properties for this configuration.

□ Update the properties of this Configuration object if the provided properties are different than the currently stored set. Properties are compared as follows.

- Scalars are compared using equals
- Arrays are compared using Arrays.equals
- Collections are compared using equals

If the new properties are not different than the current properties, no operation is performed. Otherwise, the behavior of this method is identical to the update(Dictionary) method.

Returns If the properties are different and the configuration is updated true is returned. If the properties are the same, false is returned.

Throws ReadOnlyConfigurationException– If the configuration is read only.

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.

Since 1.6

104.14.4 enum Configuration.ConfigurationAttribute

Configuration Attributes.

Since 1.6

104.14.4.1 READ_ONLY

The configuration is read only.

104.14.4.2 public static Configuration.ConfigurationAttribute valueOf(String name)

104.14.4.3 public static Configuration.ConfigurationAttribute[] values()

104.14.5 public interface ConfigurationAdmin

Service for administering configuration data.

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

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

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

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

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

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

When the Configuration Admin service detects a Managed Service Factory registration, it checks its storage for configuration objects whose service.factoryPid property matches the PID service property of the Managed Service Factory. For each such Configuration objects, it calls the

ManagedServiceFactory.updated method asynchronously with the new properties. The calls to the updated method of a ManagedServiceFactory must be executed sequentially and not overlap in time.

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

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

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

If the configuration location starts with ?, that is, the location is a region, then the configuration must be delivered to all targets registered with the given PID. If security is on, the target bundle must have Configuration Permission[location,TARGET], where location matches given the configuration location with wildcards as in the Filter substring match. The security must be verified using the org.osgi.framework.Bundle.hasPermission(Object) method on the target bundle.

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

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

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

104.14.5.1 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

104.14.5.2 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

104.14.5.3 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.

104.14.5.4 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

104.14.5.5 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

104.14.5.6 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

104.14.5.7 public Configuration getFactoryConfiguration(String factoryPid, String name, String location) throws IOException

factoryPid PID of factory (not null).

name A name for Configuration (not null).

location The bundle location string, or null.

□ Get an existing or new Configuration object from the persistent store. The PID for this Configuration object is generated from the provided factory PID and the name by starting with the factory PID appending a tilde ('~' \u007E), and then appending the name.

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

Since 1.6

104.14.5.8 public Configuration getFactoryConfiguration(String factoryPid, String name) throws IOException

factoryPid PID of factory (not null).

name A name for Configuration (not null).

□ Get an existing or new Configuration object from the persistent store. The PID for this Configuration object is generated from the provided factory PID and the name by starting with the factory PID appending a tilde ('~' \u007E), and then appending the name.

If a 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

Since 1.6

104.14.5.9 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

104.14.6 public final class ConfigurationConstants

Defines standard constants for the Configuration Admin service.

104.14.6.1 public static final String CONFIGURATION_ADMIN_IMPLEMENTATION = "osgi.cm"

The name of the implementation capability for the Configuration Admin specification

Since 1.6

104.14.6.2 public static final String CONFIGURATION_ADMIN_SPECIFICATION_VERSION = "1.6"

The version of the implementation capability for the Configuration Admin specification

Since 1.6

104.14.7 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

104.14.7.1 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.

104.14.7.2 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

104.14.7.3 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.

104.14.7.4 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.

104.14.7.5 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.

104.14.7.6 public String getPid()

□ Returns the pid of the associated configuration.

Returns Returns the pid of the associated configuration.

104.14.7.7 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.

104.14.7.8 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.

104.14.8 public class ConfigurationException extends Exception

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

104.14.8.1 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.

104.14.8.2 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

104.14.8.3 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

104.14.8.4 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

104.14.8.5 public String getReason()

□ Return the reason for this exception.

Returns reason of the failure

104.14.8.6 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

104.14.9 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

104.14.9.1 public void configurationEvent(ConfigurationEvent event)

event The ConfigurationEvent.

□ Receives notification of a Configuration that has changed.

104.14.10 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

104.14.10.1 public static final String ATTRIBUTE = "attribute"

Provides permission to set or remove an attribute on the configuration. The action string "attribute".

Since 1.6

104.14.10.2 public static final String CONFIGURE = "configure"

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

104.14.10.3 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

104.14.10.4 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, ATTRIBUTE (case insensitive).

□ Create a new ConfigurationPermission.

104.14.10.5 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.

104.14.10.6 public String getActions()

□ Returns the canonical string representation of the ConfigurationPermission actions.

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

Returns Canonical string representation of the ConfigurationPermission actions.

104.14.10.7 public int hashCode()

□ Returns the hash code value for this object.

Returns Hash code value for this object.

104.14.10.8 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.

104.14.10.9 public PermissionCollection newPermissionCollection()

□ Returns a new PermissionCollection object suitable for storing ConfigurationPermissions.

Returns A new PermissionCollection object.

104.14.11 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. Any modifications made by such plugins must be ignored.

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

104.14.11.1 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

104.14.11.2 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.

104.14.11.3 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. Any modification from this plugin is ignored.

If this method throws any Exception, the Configuration Admin service must catch it and should log it. Any modifications made by the plugin before the exception is thrown are applied.

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.

104.14.12 public interface ManagedService

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

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

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

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

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

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

```
class SerialPort implements ManagedService {

  ServiceRegistration registration;
  Hashtable configuration;
  CommPortIdentifier id;

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

  Hashtable getDefaults() {
    Hashtable defaults = new Hashtable();
    defaults.put( "port", id.getName() );
    defaults.put( "product", "unknown" );
    defaults.put( "baud", "9600" );
    defaults.put( Constants.SERVICE_PID,
      "com.acme.serialport." + id.getName() );
    return defaults;
  }

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

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

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

If a Managed Service is registered without the service.pid property, it will be ignored.

Concurrency Thread-safe

104.14.12.1 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

104.14.13 public interface ManagedServiceFactory

Manage multiple service instances. Bundles registering this interface are giving the Configuration Admin service the ability to create and configure a number of instances of a service that the implementing bundle can provide. For example, a bundle implementing a DHCP server could be instantiated multiple times for different interfaces using a factory.

Each of these *service instances* is represented, in the persistent storage of the Configuration Admin service, by a factory Configuration object that has a PID. When such a Configuration is updated, the Configuration Admin service calls the ManagedServiceFactory updated method with the new properties. When updated is called with a new PID, the Managed Service Factory should create a new factory instance based on these configuration properties. When called with a PID that it has seen before, it should update that existing service instance with the new configuration information.

In general it is expected that the implementation of this interface will maintain a data structure that maps PIDs to the factory instances that it has created. The semantics of a factory instance are defined by the Managed Service Factory. However, if the factory instance is registered as a service object with the service registry, its PID should match the PID of the corresponding Configuration object (but it should **not** be registered as a Managed Service!).

An example that demonstrates the use of a factory. It will create serial ports under command of the Configuration Admin service.

```
class SerialPortFactory
  implements ManagedServiceFactory {
  ServiceRegistration registration;
  Hashtable ports;
  void start(BundleContext context) {
    Hashtable properties = new Hashtable();
    properties.put( Constants.SERVICE_PID,
      "com.acme.serialportfactory" );
    registration = context.registerService(
      ManagedServiceFactory.class.getName(),
      this,
      properties
    );
  }
  public void updated( String pid,
    Dictionary properties  ) {
    String portName = (String) properties.get("port");
    SerialPortService port =
      (SerialPort) ports.get( pid );
    if ( port == null ) {
      port = new SerialPortService();
      ports.put( pid, port );
      port.open();
    }
    if ( port.getPortName().equals(portName) )
      return;
    port.setPortName( portName );
  }
  public void deleted( String pid ) {
    SerialPortService port =
      (SerialPort) ports.get( pid );
    port.close();
    ports.remove( pid );
  }
  ...
}
```

If a ManagedServiceFactory is registered without the service.pid property, it will be ignored.

Concurrency Thread-safe

104.14.13.1 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.

104.14.13.2 public String getName()

□ Return a descriptive name of this factory.

Returns the name for the factory, which might be localized

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

pid The PID for this configuration.

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

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

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

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

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

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

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

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

Throws ConfigurationException– when the configuration properties are invalid.

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

104.14.14 public class ReadOnlyConfigurationException extends RuntimeException

An Exception class to inform the client of a Configuration about the read only state of a configuration object.

Since 1.6

104.14.14.1 public ReadOnlyConfigurationException(String reason)

reason reason for failure

□ Create a ReadOnlyConfigurationException object.

104.14.15 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 SynchronousConfigurationListener service.

Since 1.5

Concurrency Thread-safe

104.15 org.osgi.service.cm.annotations

Configuration Admin Annotations Package Version 1.6.

This package contains annotations that can be used to require the Configuration Admin implementations

Bundles should not normally need to import this package as the annotations are only used at build-time.

104.15.1 Summary

- `RequireConfigurationAdmin` - This annotation can be used to require the Configuration Admin implementation.

104.15.2 @RequireConfigurationAdmin

This annotation can be used to require the Configuration Admin implementation. It can be used directly, or as a meta-annotation.

Since 1.6

Retention `CLASS`

Target `TYPE`, `PACKAGE`

135 Common Namespaces Specification

Version 1.2

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 8* 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 Working Group reserves names that start with "osgi.".
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 Working Group 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 osgi.unresolvable Namespace

The Unresolvable Namespace is intended to be used to mark a bundle as unresolvable:

- Preventing the bundle from resolving since it is intended for compilation use only and is not intended for runtime use.
- Providing a hint to the provisioning agent that the bundle must not be included in a provisioning solution.

For example, a bundle that must be unresolvable at runtime can include the following requirement:

```
Require-Capability: osgi.unresolvable;
  filter:="(&(must.not.resolve=*)(!(must.not.resolve=*)))"
```

The filter expression in the example above always evaluates to false.

See the UnresolvableNamespace class for this Namespace definition.

135.7 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 Working Group.

135.7.1 Summary

- ContractNamespace - Contract Capability and Requirement Namespace.

135.7.2 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

135.7.2.1 public static final String CAPABILITY_VERSION_ATTRIBUTE = "version"

The capability attribute contains the Versions of the specification of the contract. The value of this attribute must be of type Version, Version[], or List<Version>.

135.7.2.2 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.

135.8 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 Working Group.

135.8.1 Summary

- ExtenderNamespace - Extender Capability and Requirement Namespace.

135.8.2 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

135.8.2.1 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.

135.8.2.2 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.

135.9 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 Working Group.

135.9.1 Summary

- `ServiceNamespace` - Service Capability and Requirement Namespace.

135.9.2 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 specified directive and attribute keys are of type `String`, unless otherwise indicated.

Concurrency Immutable

135.9.2.1 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.

135.9.2.2 public static final String SERVICE_NAMESPACE = "osgi.service"

Namespace name for service capabilities and requirements.

135.10 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 Working Group.

135.10.1 Summary

- ImplementationNamespace - Implementation Capability and Requirement Namespace.

135.10.2 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 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

135.10.2.1 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.

135.10.2.2 public static final String IMPLEMENTATION_NAMESPACE = "osgi.implementation"

Namespace name for "implementation" capabilities and requirements. This is also the capability attribute used to specify the name of the specification or contract being implemented.

A ImplementationNamespace capability should express a uses constraint for the appropriate packages defined by the specification/contract the packages mentioned in the value of this attribute.

135.11 org.osgi.namespace.unresolvable

Unresolvable 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 Working Group.

135.11.1 Summary

- UnresolvableNamespace - Unresolvable Capability and Requirement Namespace.

135.11.2 public final class UnresolvableNamespace extends Namespace

Unresolvable 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 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

135.11.2.1 public static final String UNRESOLVABLE_FILTER = "(&(must.not.resolve=*)(!(must.not.resolve=*)))"

An unresolvable filter expression.

This can be used as the filter expression for an UnresolvableNamespace requirement.

```
@Requirement(namespace = UnresolvableNamespace.UNRESOLVABLE_NAMESPACE,
             filter = UnresolvableNamespace.UNRESOLVABLE_FILTER)
```

135.11.2.2 public static final String UNRESOLVABLE_NAMESPACE = "osgi.unresolvable"

Namespace name for "unresolvable" capabilities and requirements.

This is typically used as follows to prevent a bundle from being resolvable.

```
Require-Capability: osgi.unresolvable;
  filter:="(&(must.not.resolve=*)(!(must.not.resolve=*)))"
```

135.12 References

[1] *Specification References*
https://docs.osgi.org/reference/

[2] *Portable Java Contract Definitions*
https://docs.osgi.org/reference/portable-java-contracts.html

[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

159 Feature Service Specification

Version 1.0

159.1 Introduction

OSGi has become a platform capable of running large applications for a variety of purposes, including rich client applications, server-side systems and cloud and container based architectures. As these applications are generally based on many bundles, describing each bundle individually in an application definition becomes unwieldy once the number of bundles reaches a certain level.

When developing large scale applications it is often the case that few people know the role of every single bundle or configuration item in the application. To keep the architecture understandable a grouping mechanism is needed that allows for the representation of parts of the application into larger entities that keep reasoning about the system manageable. In such a domain members of teams spread across an organization will need to be able to both develop new parts for the application as well as make tweaks or enhancements to parts developed by others such as adding configuration and resources or changing one or more bundles relevant to their part of the application.

The higher level constructs that define the application should be reusable in different contexts, for example if one team has developed a component to handle job processing, different applications should be able to use it, and if needed tune its configuration or other aspects so that it works in each setting without having to know each and every detail that the job processing component is built up from.

Applications are often associated with additional resources or metadata, for example database scripts or custom artifacts. By including these with the application definition, all the related entities are encapsulated in a single artifact.

By combining various applications or subsystems together, systems are composed of existing, reusable building blocks, where all these blocks can work together. Architects of these systems need to think about components without having to dive into the individual implementation details of each subcomponent. The Features defined in this specification can be used to model such applications. Features contain the definition of an application or component and may be composed into larger systems.

159.1.1 Essentials

- *Declarative* - Features are declarative and can be mapped to different implementations.
- *Extensible* - Features are extensible with custom content to facilitate all information related to a Feature to be co-located.
- *Human Readable* - No special software is needed to read or author Features.
- *Machine Readable* - Features are easily be processed by tools.

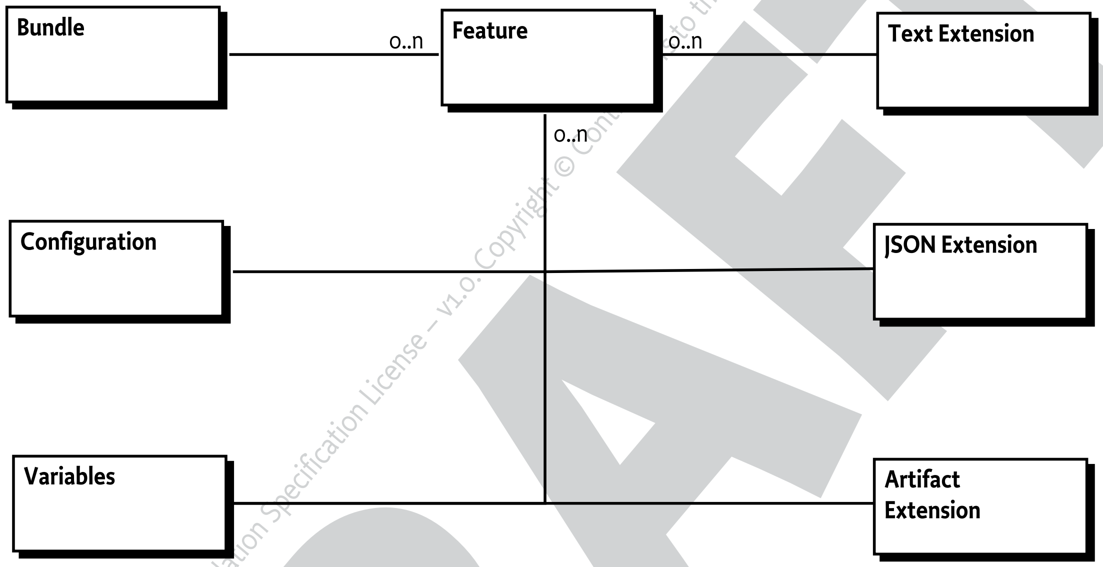
159.1.2 Entities

The following entities are used in this specification:

- *Feature* - A Feature contains a number of entities that, when provided to a launcher can be turned into an executable system. Features are building blocks which may be assembled into larger systems.

- *Bundles* - A Feature can contain one ore more bundles.
- *Configuration* - A Feature can contain configurations for the Configuration Admin service.
- *Extension* - A Feature can contain a number of extensions with custom content.
- *Launcher* - A launcher turns one or more Features into an executable system.
- *Processor* - A Feature processor reads Features and perform a processing operation on them, such as validation, transformation or generation of new entities based on the Features.
- *Properties* - Framework launching properties can be specified in a Feature.

Figure 159.1 Features Entity overview



159.2 Feature

Features are defined by declaring JSON documents or by using the Feature API. Each Feature has a unique ID which includes a version. It holds a number of entities, including a list of bundles, configurations and others. Features are extensible, that is a Feature can also contain any number of custom entities which are related to the Feature.

Features may have dependencies on other Features. Features inherit the capabilities and requirements from all bundles listed in the Feature.

Once created, a Feature is immutable. Its definition cannot be modified. However it is possible to record caching related information in a Feature through transient extensions. This cached content is not significant for the definition of the Feature or part of its identity.

159.2.1 Identifiers

Identifiers used throughout this specification are defined using the Maven Identifier model. They are composed of the following parts:

- Group ID
- Artifact ID
- Version
- Type (optional)

- Classifier (optional)

Note that if Version has the `-SNAPSHOT` suffix, the identifier points at an unreleased artifact that is under development and may still change.

For more information see [3] *Apache Maven Pom Reference*. The format used to specify identifiers is as follows:

```
groupId ':' artifactId ( ':' type ( ':' classifier )? )? ':' version
```

159.2.2 Feature Identifier

Each Feature has a unique identifier. Apart from providing a persistent handle to the Feature, it also provides enough information to find the Feature in an artifact repository. This identifier is defined using the format described in *Identifiers* on page 72.

159.2.2.1 Identifier type

Features use as identifier type the value `osgifeature`.

159.2.3 Attributes

A Feature can have the following attributes:

Table 159.1 Feature Attributes

Attribute	Data Type	Kind	Description
name	String	Optional	The short descriptive name of the Feature.
categories	Array of String	Optional, defaults to an empty array	The categories this Feature belongs to. The values are user-defined.
complete	boolean	Optional, defaults to `false`	Completeness of the Feature. A Feature is complete when it has no external dependencies.
description	String	Optional	A longer description of the Feature.
docURL	String	Optional	A location where documentation can be found for the Feature.
license	String	Optional	The license of the Feature. The license only relates to the Feature itself and not to any artifacts that might be referenced by the Feature. The license follows the `Bundle-License` format as specified in the Core specification.
SCM	String	Optional	SCM information relating to the feature. The syntax of the value follows the Bundle-SCM format. See the 'Bundle Manifest Headers' section in the OSGi Core specification.
vendor	String	Optional	The vendor of the Feature.

An initial Feature without content can be declared as follows:

```
{
  "feature-resource-version": "1.0",
  "id": "org.acme:acmeapp:1.0.0",

  "name": "The ACME app",
  "description":
    "This is the main ACME app, from where all functionality is reached."

  /*
    Additional Feature entities here
    ...
  */
}
```

159.2.4 Using the Feature API

Features can also be created, read and written using the Feature API. The main entry point for this API is the FeatureService. The Feature API uses the builder pattern to create entities used in Features.

A builder instance is used to create a single entity and cannot be re-used to create a second one. Builders are created from the BuilderFactory, which is available from the FeatureService through getBuilderFactory().

```
FeatureService fs = ... // from Service Registry
BuilderFactory factory = fs.getBuilderFactory();

FeatureBuilder builder = factory.newFeatureBuilder(
  fs.getID("org.acme", "acmeapp", "1.0.0"));
builder.setName("The ACME app");
builder.setDescription("This is the main ACME app, "
  + "from where all functionality is reached.");

Feature f = builder.build();
```

The Feature API can also be useful in environments outside of an OSGi Framework where no service registry is available, for example in a build-system environment. In such environments the FeatureService can be obtained by using the java.util.ServiceLoader mechanism.

159.3 Comments

Comments in the form of [2] *JSMin (The JavaScript Minifier)* comments are supported, that is, any text on the same line after // is ignored and any text between /* */ is ignored.

159.4 Bundles

Features list zero or more bundles that implement the functionality provided by the Feature. Bundles are listed by referencing them in the bundles array so that they can be resolved from a repository. Bundles can have metadata associated with them, such as the relative start order of the bundle in the Feature. Custom metadata may also be provided. A single Feature can provide multiple versions of the same bundle, if desired.

Bundles are referenced using the identifier format described in *Identifiers* on page 72. This means that Bundles are referenced using their Maven coordinates. The bundles array contains JSON objects which can contain the bundle IDs and specify optional additional metadata.

159.4.1 Bundle Metadata

Arbitrary key-value pairs can be associated with bundle entries to store custom metadata alongside the bundle references. Reverse DNS naming should be used with the keys to avoid name clashes when metadata is provided by multiple entities. Keys not using the reverse DNS naming scheme are reserved for OSGi use.

Bundle metadata supports string keys and string, number or boolean values.

The following example shows a simple Feature describing a small application with its dependencies:

```
{
  "feature-resource-version": "1.0",
  "id": "org.acme:acmeapp:1.0.1",

  "name": "The Acme Application",
  "license": "https://opensource.org/licenses/Apache-2.0",
  "complete": true,

  "bundles": [
    { "id": "org.osgi:org.osgi.util.function:1.1.0" },
    { "id": "org.osgi:org.osgi.util.promise:1.1.1" },
    {
      "id": "org.apache.commons:commons-email:1.5",

      // This attribute is used by custom tooling to
      // find the associated javadoc
      "org.acme.javadoc.link":
        "https://commons.apache.org/proper/commons-email/javadocs/api-1.5"
    },
    { "id": "com.acme:acmelib:1.7.2" }
  ]

  /*
    Additional Feature entities here
    ...
  */
}
```

159.4.2 Using the Feature API

A Feature with Bundles can be created using the Feature API as follows:

```
FeatureService fs = ... // from Service Registry
BuilderFactory factory = fs.getBuilderFactory();

FeatureBuilder builder = factory.newFeatureBuilder(
  fs.getID("org.acme", "acmeapp", "1.0.1"));
builder.setName("The Acme Application");
builder.setLicense("https://opensource.org/licenses/Apache-2.0");
builder.setComplete(true);

FeatureBundle b1 = factory
  .newBundleBuilder(fs.getIDfromMavenCoordinates(
    "org.osgi:org.osgi.util.function:1.1.0"))
  .build();
FeatureBundle b2 = factory
  .newBundleBuilder(fs.getIDfromMavenCoordinates(
    "org.osgi:org.osgi.util.promise:1.1.1"))
  .build();
```

```
FeatureBundle b3 = factory
  .newBundleBuilder(fs.getIDfromMavenCoordinates(
    "org.apache.commons:commons-email:1.1.5"))
  .addMetadata("org.acme.javadoc.link",
    "https://commons.apache.org/proper/commons-email/javadocs/api-1.5")
  .build();
FeatureBundle b4 = factory
  .newBundleBuilder(fs.getIDfromMavenCoordinates(
    "com.acme:acmelib:1.7.2"))
  .build();

builder.addBundles(b1, b2, b3, b4);
Feature f = builder.build();
```

159.5 Configurations

Features support configuration using the OSGi Configurator syntax, see ???. This is specified with the configurations key in the Feature. A Launcher can apply these configurations to the Configuration Admin service when starting the system.

It is an error to define the same PID twice in a single Feature. An entity processing the feature must fail in this case.

Example:

```
{
    "feature-resource-version": "1.0",
    "id": "org.acme:acmeapp:osgifeature:configs:1.0.0",
    "configurations": {
        "org.apache.felix.http": {
            "org.osgi.service.http.port": 8080,
            "org.osgi.service.http.port.secure": 8443
        }
    }
}
```

159.6 Variables

Configurations and Framework Launching Properties support late binding of values. This enables setting these items through a Launcher, for example to specify a database user name, server port number or other information that may be variable between runtimes.

Variables are declared in the variables section of the Feature and they can have a default value specified. The default must be of type string, number or boolean. Variables can also be declared to *not* have a default, which means that they must be provided with a value through the Launcher. This is done by specifying null as the default in the variable declaration.

Example:

```
{
    "feature-resource-version": "1.0",
    "id": "org.acme:acmeapp:osgifeature:configs:1.1.0",
    "variables": {
        "http.port": 8080,
        "db.username": "scott",
        "db.password": null
    },
    "configurations": {
        "org.acme.server.http": {
            "org.osgi.service.http.port:Integer": "${http.port}"
        },
        "org.acme.db": {
            "username": "${db.username}-user",
            "password": "${db.password}"
        }
    }
}
```

Variables are referenced with the curly brace placeholder syntax: `${` *`variable-name`* `}` in the configuration value or framework launching property value section. To support conversion of variables to non-string types the configurator syntax specifying the datatype with the configuration key is used, as in the above example.

Multiple variables can be referenced for a single configuration or framework launching property value and variables may be combined with text. If no variable exist with the given name, then the `${` *`variable-name`* `}` must be retained in the value.

159.7 Extensions

Features can include custom content. This makes it possible to keep custom entities and information relating to the Feature together with the rest of the Feature.

Custom content is provided through Feature extensions, which are in one of the following formats:

- *Text* - A text extension contains an array of text.
- *JSON* - A JSON extension contains embedded custom JSON content.
- *Artifacts* - A list of custom artifacts associated with the Feature.

Extensions can have a variety of consumers. For example they may be handled by a Feature Launcher or by an external tool which can process the extension at any point of the Feature life cycle.

Extensions are of one of the following three kinds:

- *Mandatory* - The entity processing this Feature *must* know how to handle this extension. If it cannot handle the extension it must fail.
- *Optional* - This extension is optional. If the entity processing the Feature cannot handle it, the extension can be skipped or ignored. This is the default.
- *Transient* - This extension contains transient information which may be used to optimize the processing of the Feature. It is not part of the Feature definition.

Extensions are specified as JSON objects under the `extensions` key in the Feature. A Feature can contain any number of extensions, as long as the extension keys are unique. Extension keys should use reverse domain naming to avoid name clashing of multiple extensions in a single Feature. Extensions names without a reverse domain naming prefix are reserved for OSGi use.

159.7.1 Text Extensions

Text extensions support the addition of custom text content to the Feature. The text is provided as a JSON array of strings.

Example:

```
{
    "feature-resource-version": "1.0",
    "id": "org.acme:acmeapp:2.0.0",

    "name": "The Acme Application",
    "license": "https://opensource.org/licenses/Apache-2.0",

    "extensions": {
        "org.acme.mydoc": {
            "type": "text",
            "text": [
                "This application provides the main acme ",
                "functionality."
            ]
        }
    }
}
```

159.7.2 JSON Extensions

Custom JSON content is added to Features by using a JSON extension. The content can either be a JSON object or a JSON array.

The following example extension declares under which execution environment the Feature is complete, using a custom JSON object.

```
{
    "feature-resource-version": "1.0",
    "id": "org.acme:acmeapp:2.1.0",

    "name": "The Acme Application",
    "license": "https://opensource.org/licenses/Apache-2.0",

    "extensions": {
        "org.acme.execution-environment": {
            "type": "json",
            "json": {
                "environment-capabilities":
                    ["osgi.ee; filter:=\"(&(osgi.ee=JavaSE)(version=11))\""],
                "framework": "org.osgi:core:6.0.0",
                "provided-features": ["org.acme:platform:1.1"]
            }
        }
    }
}
```

159.7.3 Artifact list Extensions

Custom extensions can be used to associate artifacts that are not listed as `bundles` with the Feature.

For example, database definition resources may be listed as artifacts in a Feature. In the following example, the extension `org.acme.ddlfiles` lists Database Definition Resources which *must* be handled by the launcher agent, that is, the database must be configured when the application is run:

```
{
    "feature-resource-version": "1.0",
    "id": "org.acme:acmeapp:2.2.0",
    "name": "The Acme Application",
    "license": "https://opensource.org/licenses/Apache-2.0",
    "complete": true,

    "bundles": [
        "org.osgi:org.osgi.util.function:1.1.0",
        "org.osgi:org.osgi.util.promise:1.1.1",
        "com.acme:acmelib:2.0.0"
    ],

    "extensions": {
        "org.acme.ddlfiles": {
            "kind": "mandatory",
            "type": "artifacts",
            "artifacts": [
              { "id": "org.acme:appddl:1.2.1" },
              {
                "id": "org.acme:appddl-custom:1.0.3",
                "org.acme.target": "custom-db"
              }
            ]
        }
    }
}
```

As with bundle identifiers, custom artifacts are specified in an object in the artifacts list with an explicit id and optional additional metadata. The keys of the metadata should use a reverse domain naming pattern to avoid clashes. Keys that do not use reverse domain name as a prefix are reserved for OSGi use. Supported metadata values must be of type string, number or boolean.

159.8 Framework Launching Properties

When a Feature is launched in an OSGi framework it may be necessary to specify Framework Properties. These are provided in the Framework Launching Properties extension section of the Feature. The Launcher must be able to satisfy the specified properties. If it cannot ensure that these are present in the running Framework the launcher must fail.

Framework Launching Properties can reference Variables as defined in *Variables* on page 76. These variables are substituted before the properties are set.

Example:

```
{
  "feature-resource-version": "1.0",
  "id": "org.acme:acmeapp:osgifeature:fw-props:2.0.0",

  "variables": {
    "fw.storage.dir": "/tmp" // Can be overridden through the launcher
  },

  "extensions": {
    "framework-launching-properties": {
      "type": "json",
      "json": {
          "org.osgi.framework.system.packages.extra":
            "javax.activation;version=\"1.1.1\"",
          "org.osgi.framework.bootdelegation": "javax.activation",
          "org.osgi.framework.storage": "${fw.storage.dir}"
      }
    }
  }
}
```

159.9 Resource Versioning

Feature JSON resources are versioned to support updates to the JSON structure in the future. To declare the document version of the Feature use the feature-resource-version key in the JSON document.

```
{
  "feature-resource-version": "1.0",
  "id": "org.acme:acmeapp:1.0.0"

  /*
    Additional Feature entities here
    ...
  */
}
```

The currently supported version of the Feature JSON documents is 1.0. If no Feature Resource Version is specified 1.0 is used as the default.

159.10 Capabilities

159.10.1 osgi.service Capability

The bundle providing the Feature Service must provide a capability in the osgi.service namespace representing the services it is registering. This capability must also declare uses constraints for the relevant service packages:

```
Provide-Capability: osgi.service;
  objectClass:List<String>="org.osgi.service.feature.FeatureService";
  uses:="org.osgi.service.feature"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 65.

159.11 org.osgi.service.feature

Feature 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.feature; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.feature; version="[1.0,1.1)"

159.11.1 Summary

- BuilderFactory - The Builder Factory can be used to obtain builders for the various entities.
- Feature - The Feature Model Feature.
- FeatureArtifact - An Artifact is an entity with an ID, for use in extensions.
- FeatureArtifactBuilder - A builder for FeatureArtifact objects.
- FeatureBuilder - A builder for Feature Models.
- FeatureBundle - A Bundle which is part of a feature.
- FeatureBundleBuilder - A builder for Feature Model FeatureBundle objects.
- FeatureConfiguration - Represents an OSGi Configuration in the Feature Model.
- FeatureConfigurationBuilder - A builder for Feature Model FeatureConfiguration objects.
- FeatureConstants - Defines standard constants for the Feature specification.
- FeatureExtension - A Feature Model Extension.
- FeatureExtension.Kind - The kind of extension: optional, mandatory or transient.
- FeatureExtension.Type - The type of extension
- FeatureExtensionBuilder - A builder for Feature Model FeatureExtension objects.
- FeatureService - The Feature service is the primary entry point for interacting with the feature model.
- ID - ID used to denote an artifact.

159.11.2 public interface BuilderFactory

The Builder Factory can be used to obtain builders for the various entities.

Provider Type Consumers of this API must not implement this type

159.11.2.1 public FeatureArtifactBuilder newArtifactBuilder(ID id)

id The artifact ID for the artifact object being built.

□ Obtain a new builder for Artifact objects.

Returns The builder.

159.11.2.2 public FeatureBundleBuilder newBundleBuilder(ID id)

id The ID for the bundle object being built. If the ID has no type specified, a default type of @{code jar} is assumed.

□ Obtain a new builder for Bundle objects.

Returns The builder.

159.11.2.3 public FeatureConfigurationBuilder newConfigurationBuilder(String pid)

pid The persistent ID for the Configuration being built.

□ Obtain a new builder for Configuration objects.

Returns The builder.

159.11.2.4 public FeatureConfigurationBuilder newConfigurationBuilder(String factoryPid, String name)

factoryPid The factory persistent ID for the Configuration being built.

name The name of the configuration being built. The PID for the configuration will be the factoryPid + '~' + name

□ Obtain a new builder for Factory Configuration objects.

Returns The builder.

159.11.2.5 public FeatureExtensionBuilder newExtensionBuilder(String name, FeatureExtension.Type type, FeatureExtension.Kind kind)

name The extension name.

type The type of extension: JSON, Text or Artifacts.

kind The kind of extension: Mandatory, Optional or Transient.

□ Obtain a new builder for Feature objects.

Returns The builder.

159.11.2.6 public FeatureBuilder newFeatureBuilder(ID id)

id The ID for the feature object being built. If the ID has no `type` specified, a default type of `osgifeature` is assumed.

□ Obtain a new builder for Feature objects.

Returns The builder.

159.11.3 public interface Feature

The Feature Model Feature.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.3.1 public List<FeatureBundle> getBundles()

□ Get the bundles.

Returns The bundles. The returned list is unmodifiable.

159.11.3.2 public List<String> getCategories()

□ Get the categories.

Returns The categories. The returned list is unmodifiable.

159.11.3.3 public Map<String, FeatureConfiguration> getConfigurations()

□ Get the configurations. The iteration order of the returned map should follow the definition order of the configurations in the feature.

Returns The configurations. The returned map is unmodifiable.

159.11.3.4 public Optional<String> getDescription()

□ Get the description.

Returns The description.

159.11.3.5 public Optional<String> getDocURL()

□ Get the documentation URL.

Returns The documentation URL.

159.11.3.6 public Map<String, FeatureExtension> getExtensions()

□ Get the extensions. The iteration order of the returned map should follow the definition order of the extensions in the feature.

Returns The extensions. The returned map is unmodifiable.

159.11.3.7 public ID getID()

□ Get the Feature's ID.

Returns The ID of this Feature.

159.11.3.8 public Optional<String> getLicense()

□ Get the license of this Feature. The syntax of the value follows the Bundle-License header syntax. See the 'Bundle Manifest Headers' section in the OSGi Core specification.

Returns The license.

159.11.3.9 public Optional<String> getName()

□ Get the name.

Returns The name.

159.11.3.10 public Optional<String> getSCM()

□ Get the SCM information relating to the feature. The syntax of the value follows the Bundle-SCM format. See the 'Bundle Manifest Headers' section in the OSGi Core specification.

Returns The SCM information.

159.11.3.11 public Map<String, Object> getVariables()

□ Get the variables. The iteration order of the returned map should follow the definition order of the variables in the feature. Values are of type: String, Boolean or BigDecimal for numbers. The null JSON value is represented by a null value in the map.

Returns The variables. The returned map is unmodifiable.

159.11.3.12 public Optional<String> getVendor()

□ Get the vendor.

Returns The vendor.

159.11.3.13 public boolean isComplete()

□ Get whether the feature is complete or not.

Returns Completeness value.

159.11.4 public interface FeatureArtifact

An Artifact is an entity with an ID, for use in extensions.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.4.1 public ID getID()

□ Get the artifact's ID.

Returns The ID of this artifact.

159.11.4.2 public Map<String, Object> getMetadata()

□ Get the metadata for this artifact.

Returns The metadata. The returned map is unmodifiable.

159.11.5 public interface FeatureArtifactBuilder

A builder for FeatureArtifact objects.

Concurrency Not Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.5.1 public FeatureArtifactBuilder addMetadata(String key, Object value)

key Metadata key.

value Metadata value.

□ Add metadata for this Artifact.

Returns This builder.

159.11.5.2 public FeatureArtifactBuilder addMetadata(Map<String, Object> metadata)

metadata The map with metadata.

□ Add metadata for this Artifact by providing a map. All metadata in the map is added to any previously provided metadata.

Returns This builder.

159.11.5.3 public FeatureArtifact build()

□ Build the Artifact object. Can only be called once on a builder. After calling this method the current builder instance cannot be used any more.

Returns The Feature Artifact.

159.11.6 public interface FeatureBuilder

A builder for Feature Models.

Concurrency Not Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.6.1 public FeatureBuilder addBundles(FeatureBundle... bundles)

bundles The Bundles to add.

□ Add Bundles to the Feature.

Returns This builder.

159.11.6.2 public FeatureBuilder addCategories(String... categories)

categories The Categories.

□ Adds one or more categories to the Feature.

Returns This builder.

159.11.6.3 public FeatureBuilder addConfigurations(FeatureConfiguration... configs)

configs The Configurations to add.

□ Add Configurations to the Feature.

Returns This builder.

159.11.6.4 public FeatureBuilder addExtensions(FeatureExtension... extensions)

extensions The Extensions to add.

□ Add Extensions to the Feature

Returns This builder.

159.11.6.5 public FeatureBuilder addVariable(String key, Object defaultValue)

key The key.

defaultValue The default value.

□ Add a variable to the Feature. If a variable with the specified key already exists it is replaced with this one. Variable values are of type: String, Boolean or BigDecimal for numbers.

Returns This builder.

Throws IllegalArgumentException– if the value is of an invalid type.

159.11.6.6 public FeatureBuilder addVariables(Map<String, Object> variables)

variables to be added.

□ Add a map of variables to the Feature. Pre-existing variables with the same key in are overwritten if these keys exist in the map. Variable values are of type: String, Boolean or BigDecimal for numbers.

Returns This builder.

Throws IllegalArgumentException– if a value is of an invalid type.

159.11.6.7 public Feature build()

□ Build the Feature. Can only be called once on a builder. After calling this method the current builder instance cannot be used any more.

Returns The Feature.

159.11.6.8 public FeatureBuilder setComplete(boolean complete)

complete If the feature is complete.

□ Set the Feature Complete flag. If this method is not called the complete flag defaults to false.

Returns This builder.

159.11.6.9 public FeatureBuilder setDescription(String description)

description The description.

□ Set the Feature Description.

Returns This builder.

159.11.6.10 public FeatureBuilder setDocURL(String docURL)

docURL The Documentation URL.

□ Set the documentation URL.

Returns This builder.

159.11.6.11 public FeatureBuilder setLicense(String license)

license The License.

□ Set the License.

Returns This builder.

159.11.6.12 public FeatureBuilder setName(String name)

name The Name.

□ Set the Feature Name.

Returns This builder.

159.11.6.13 public FeatureBuilder setSCM(String scm)

scm The SCM information.

□ Set the SCM information.

Returns This builder.

159.11.6.14 public FeatureBuilder setVendor(String vendor)

vendor The Vendor.

□ Set the Vendor.

Returns This builder.

159.11.7 public interface FeatureBundle

A Bundle which is part of a feature.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.7.1 public ID getID()

□ Get the bundle's ID.

Returns The ID of this bundle.

159.11.7.2 public Map<String, Object> getMetadata()

□ Get the metadata for this bundle.

Returns The metadata. The returned map is unmodifiable.

159.11.8 public interface FeatureBundleBuilder

A builder for Feature Model FeatureBundle objects.

Concurrency Not Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.8.1 public FeatureBundleBuilder addMetadata(String key, Object value)

key Metadata key.

value Metadata value.

□ Add metadata for this Bundle.

Returns This builder.

159.11.8.2 public FeatureBundleBuilder addMetadata(Map<String, Object> metadata)

metadata The map with metadata.

□ Add metadata for this Bundle by providing a map. All metadata in the map is added to any previously provided metadata.

Returns This builder.

159.11.8.3 public FeatureBundle build()

□ Build the Bundle object. Can only be called once on a builder. After calling this method the current builder instance cannot be used any more.

Returns The Bundle.

159.11.9 public interface FeatureConfiguration

Represents an OSGi Configuration in the Feature Model.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.9.1 public Optional<String> getFactoryPid()

□ Get the Factory PID from the configuration, if any.

Returns The Factory PID, or null if there is none.

159.11.9.2 public String getPid()

□ Get the PID from the configuration.

Returns The PID.

159.11.9.3 public Map<String, Object> getValues()

□ Get the configuration key-value map.

Returns The key-value map. The returned map is unmodifiable.

159.11.10 public interface FeatureConfigurationBuilder

A builder for Feature Model FeatureConfiguration objects.

Concurrency Not Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.10.1 public FeatureConfigurationBuilder addValue(String key, Object value)

key The configuration key.

value The configuration value. Acceptable data types are the data type supported by the Configuration Admin service, which are the Primary Property Types as defined for the Filter Syntax in the OSGi Core specification.

□ Add a configuration value for this Configuration object. If a value with the same key was previously provided (regardless of case) the previous value is overwritten.

Returns This builder.

Throws IllegalArgumentException– if the value is of an invalid type.

159.11.10.2 public FeatureConfigurationBuilder addValues(Map<String, Object> configValues)

configValues The map of configuration values to add. Acceptable value types are the data type supported by the Configuration Admin service, which are the Primary Property Types as defined for the Filter Syntax in the OSGi Core specification.

□ Add a map of configuration values for this Configuration object. Values will be added to any previously provided configuration values. If a value with the same key was previously provided (regardless of case) the previous value is overwritten.

Returns This builder.

Throws IllegalArgumentException– if a value is of an invalid type or if the same key is provided in different capitalizations (regardless of case).

159.11.10.3 public FeatureConfiguration build()

□ Build the Configuration object. Can only be called once on a builder. After calling this method the current builder instance cannot be used any more.

Returns The Configuration.

159.11.11 public final class FeatureConstants

Defines standard constants for the Feature specification.

159.11.11.1 public static final String FEATURE_IMPLEMENTATION = "osgi.feature"

The name of the implementation capability for the Feature specification.

159.11.11.2 public static final String FEATURE_SPECIFICATION_VERSION = "1.0"

The version of the implementation capability for the Feature specification.

159.11.12 public interface FeatureExtension

A Feature Model Extension. Extensions can contain either Text, JSON or a list of Artifacts.

Extensions are of one of the following kinds:

- Mandatory: this extension must be processed by the runtime
- Optional: this extension does not have to be processed by the runtime
- Transient: this extension contains transient information such as caching data that is for optimization purposes. It may be changed or removed and is not part of the feature's identity.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.12.1 public List<FeatureArtifact> getArtifacts()

□ Get the Artifacts from this extension.

Returns The Artifacts. The returned list is unmodifiable.

Throws `IllegalStateException` – If called on an extension which is not of type ARTIFACTS.

159.11.12.2 public String getJSON()

□ Get the JSON from this extension.

Returns The JSON.

Throws `IllegalStateException` – If called on an extension which is not of type JSON.

159.11.12.3 public FeatureExtension.Kind getKind()

□ Get the extension kind.

Returns The kind.

159.11.12.4 public String getName()

□ Get the extension name.

Returns The name.

159.11.12.5 public List<String> getText()

□ Get the Text from this extension.

Returns The lines of text. The returned list is unmodifiable.

Throws `IllegalStateException` – If called on an extension which is not of type TEXT.

159.11.12.6 public FeatureExtension.Type getType()

□ Get the extension type.

Returns The type.

159.11.13 enum FeatureExtension.Kind

The kind of extension: optional, mandatory or transient.

159.11.13.1 MANDATORY

A mandatory extension must be processed.

159.11.13.2 OPTIONAL

An optional extension can be ignored if no processor is found.

159.11.13.3 TRANSIENT

A transient extension contains computed information which can be used as a cache to speed up operation.

159.11.13.4 public static FeatureExtension.Kind valueOf(String name)

159.11.13.5 public static FeatureExtension.Kind[] values()

159.11.14 enum FeatureExtension.Type

The type of extension

159.11.14.1 JSON

A JSON extension.

159.11.14.2 TEXT

A plain text extension.

159.11.14.3 ARTIFACTS

An extension that is a list of artifact identifiers.

159.11.14.4 public static FeatureExtension.Type valueOf(String name)

159.11.14.5 public static FeatureExtension.Type[] values()

159.11.15 public interface FeatureExtensionBuilder

A builder for Feature Model FeatureExtension objects.

Concurrency Not Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.15.1 public FeatureExtensionBuilder addArtifact(FeatureArtifact artifact)

artifact The artifact to add.

□ Add an Artifact to the extension. Can only be called for extensions of type FeatureExtension.Type.ARTIFACTS.

Returns This builder.

159.11.15.2 public FeatureExtensionBuilder addText(String text)

text The text to be added.

□ Add a line of text to the extension. Can only be called for extensions of type FeatureExtension.Type.TEXT.

Returns This builder.

159.11.15.3 public FeatureExtension build()

□ Build the Extension. Can only be called once on a builder. After calling this method the current builder instance cannot be used any more.

Returns The Extension.

159.11.15.4 public FeatureExtensionBuilder setJSON(String json)

json The JSON to be added.

□ Add JSON in String form to the extension. Can only be called for extensions of type FeatureExtension.Type.JSON.

Returns This builder.

159.11.16 public interface FeatureService

The Feature service is the primary entry point for interacting with the feature model.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.16.1 public BuilderFactory getBuilderFactory()

□ Get a factory which can be used to build feature model entities.

Returns A builder factory.

159.11.16.2 public ID getID(String groupId, String artifactId, String version)

groupId The group ID (not null, not empty).

artifactId The artifact ID (not null, not empty).

version The version (not null, not empty).

□ Obtain an ID.

Returns The ID.

159.11.16.3 public ID getID(String groupId, String artifactId, String version, String type)

groupId The group ID (not null, not empty).

artifactId The artifact ID (not null, not empty).

version The version (not null, not empty).

type The type (not null, not empty).

□ Obtain an ID.

Returns The ID.

159.11.16.4 public ID getID(String groupId, String artifactId, String version, String type, String classifier)

groupId The group ID (not null, not empty).

artifactId The artifact ID (not null, not empty).

version The version (not null, not empty).

type The type (not null, not empty).

classifier The classifier (not `null`, not empty).

□ Obtain an ID.

Returns The ID.

159.11.16.5 public ID getIDfromMavenCoordinates(String coordinates)

coordinates The Maven Coordinates.

□ Obtain an ID from a Maven Coordinates formatted string. The supported syntax is as follows:

```
groupId ':' artifactId ( ':' type ( ':' classifier )? )? ':' version
```

Returns the ID.

159.11.16.6 public Feature readFeature(Reader jsonReader) throws IOException

jsonReader A Reader to the JSON input

□ Read a Feature from JSON

Returns The Feature represented by the JSON

Throws `IOException`– When reading fails

159.11.16.7 public void writeFeature(Feature feature, Writer jsonWriter) throws IOException

feature the Feature to write.

jsonWriter A Writer to which the Feature should be written.

□ Write a Feature Model to JSON

Throws `IOException`– When writing fails.

159.11.17 public interface ID

ID used to denote an artifact. This could be a feature model, a bundle which is part of the feature model or some other artifact.

Artifact IDs follow the Maven convention of having:

- A group ID
- An artifact ID
- A version
- A type identifier (optional)
- A classifier (optional)

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

159.11.17.1 public static final String FEATURE_ID_TYPE = "osgifeature"

ID type for use with Features.

159.11.17.2 public String getArtifactId()

□ Get the artifact ID.

Returns The artifact ID.

159.11.17.3 public Optional<String> getClassifier()

□ Get the classifier.

Returns The classifier.

159.11.17.4 public String getGroupId()

□ Get the group ID.

Returns The group ID.

159.11.17.5 public Optional<String> getType()

□ Get the type identifier.

Returns The type identifier.

159.11.17.6 public String getVersion()

□ Get the version.

Returns The version.

159.11.17.7 public String toString()

□ This method returns the ID using the following syntax:

```
groupId ':' artifactId ( ':' type ( ':' classifier )? )? ':' version
```

Returns The string representation.

159.12 org.osgi.service.feature.annotation

Feature Annotations Package Version 1.0.

This package contains annotations that can be used to require the Feature Service implementation.

Bundles should not normally need to import this package as the annotations are only used at build-time.

159.12.1 Summary

- `RequireFeatureService` - This annotation can be used to require the Feature implementation.

159.12.2 @RequireFeatureService

This annotation can be used to require the Feature implementation. It can be used directly, or as a meta-annotation.

Retention CLASS

Target TYPE, PACKAGE

159.13 References

[1] *JSON (JavaScript Object Notation)*
https://www.json.org

[2] *JSMin (The JavaScript Minifier)*
https://www.crockford.com/javascript/jsmin.html

[3] *Apache Maven Pom Reference*
https://maven.apache.org/pom.html

160 Feature Launcher Service Specification

Version 1.0

160.1 Introduction

The *Feature Service Specification* on page 71 defines a model to design and declare Complex Applications and reusable Sub-Components that are composed of multiple bundles, configurations and other metadata. These models are, however, only descriptive and have no standard mechanism for installing them into an OSGi framework.

This specification focuses on turning these Features into a running system, by introducing the Feature Launcher and Feature Runtime. The Feature Launcher takes a Feature definition, obtains a framework instance for it and then starts the Feature in that environment. The Feature Runtime extends this capability to a running system, enabling one or more Features to be installed, updated, and later removed from a running OSGi framework.

The Launcher and Runtime also interact with the Configuration Admin Service, that is, they provide configuration to the system if it is present in the Feature being launched or installed.

160.1.1 Essentials

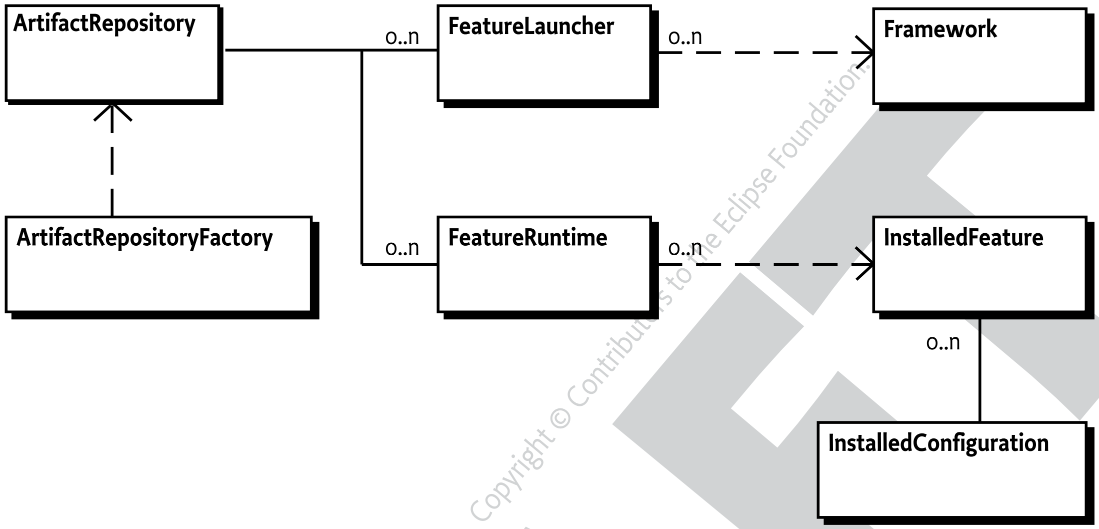
- *Dynamic* - The Feature Runtime dynamically adds, updates and removes Features in a running system.
- *Parameterizable* - Feature installation may be customised using local parameters if the Feature supports it.
- *Zero code* - The Feature Launcher can launch a framework containing an installed Feature in an implementation independent way without a user writing any code .

160.1.2 Entities

The following entities are used in this specification:

- *Feature* - A Feature as defined by the *Feature Service Specification* on page 71
- *Artifact Repository* - A means of accessing the installable bytes for bundles in a Feature
- *Feature Launcher* - A Feature Launcher obtains an OSGi Framework instance and installs a Feature into it.
- *Framework* - A running implementation of the OSGi core specification.
- *Launch Properties* - Framework launching properties defined in a Feature.
- *Feature Parameters* - Key value pairs that can be used to customise the installation of a Feature.
- *Configuration* - A configuration for the Configuration Admin service.
- *Feature Runtime* - A Feature Runtime is an OSGi service capable of installing Features into the running OSGi framework, removing installed Features from the OSGi framework, and updating an installed Feature with a new Feature definition.
- *Installed Feature* - A representation of a Feature installed by the Feature Runtime.
- *Installed Configuration* - A representation of a Configuration installed by the Feature Runtime.

Figure 160.1 Features Entity overview



160.2 Features and Artifact Repositories

OSGi Features exist either as JSON documents, or as runtime objects created by the Feature Service API. The primary purpose of a Feature is to define a list of bundles and configurations that should be installed, however the Feature provides no information about the location of the bundle artifacts. A key challenge with installing a Feature is therefore finding the appropriate artifacts to install.

The ArtifactRepository interface is designed to be implemented by users of the Feature Launcher Service to provide a way for the Feature Launcher Service to find an installable InputStream of bytes for a given bundle artifact using the getArtifact(ID) method. Artifact Repository implementations are free to use any mechanism for locating the bundle artifact data. If no artifact can be found for the supplied ID then the implementation of the Artifact Repository should return null. If the Artifact Repository throws an exception then this must be logged by the Feature Launcher Service and then treated in the same manner as a null return value.

160.2.1 The Artifact Repository Factory

In order to support the *Zero Code* objective of this specification, and to simplify usage for most users, the ArtifactRepositoryFactory provides a factory for commonly used repository types.

160.2.1.1 Obtaining an Artifact Repository Factory

The Artifact Repository Factory is useful both for the Feature Launcher and the Feature Runtime, and as such it must be easy to access both inside and outside an OSGi framework. The Feature Launcher Service implementation must provide an implementation of the Artifact Repository Factory interface. A user of the Artifact Repository Factory service may use the following ways to find an instance.

When outside OSGi:

- Using the Java ServiceLoader API to find instances of org.osgi.service.featurelauncher.repository.ArtifactRepositoryFactory
- From configuration, and then using Class.forName, getConstructor() and newInstance()

- By hard coding the implementation and using the new operator.

When inside an OSGi framework:

- Using the OSGi service registry to find instances of org.osgi.service.featurelauncher.repository.ArtifactRepositoryFactory
- Using the Java ServiceLoader API and the OSGi Service Loader Mediator to find instances of org.osgi.service.featurelauncher.repository.ArtifactRepositoryFactory
- By hard coding the implementation type and using the new operator.

160.2.1.2 Local Repositories

A Local Repository is one that exists on a locally accessible file system. Note that this does not require that the file system is local, and technologies such as NFS or other network file systems would still be considered as Local Repositories. The key aspects of a Local Repository are that:

- The root of the repository can be accessed and resolved as a java.nio.file.Path or file: URI.
- The repository uses [1] *The Maven 2 Repository Layout*

An Artifact Repository representing a Local Repository can be created using the createRepository(Path) method, passing in the path to the root of the repository. A NullPointerException must be thrown if the path is null and an IllegalArgumentException must be thrown if the path does not exist, or represents a file which is not a directory.

An Artifact Repository representing a Local Repository can also be created using the createRepository(URI,Map) method, passing a URI using the file scheme which points to the root of the repository. A NullPointerException must be thrown if the URI is null and an IllegalArgumentException must be thrown if the path does not exist, or represents a file which is not a directory.

Once created this Artifact Repository will search the supplied repository for any requested artifact data. Implementations are free to optimise checks using repository metadata.

160.2.1.3 Remote Repositories

A Remote Repository is one that exists with an accessible http or https endpoint for retrieving artifact data. Note that this does not require that the repository is on a remote machine, only that the means of accessing data is via HTTP requests. The key aspects of a Remote Repository are that:

- The root of the repository can be accessed and resolved as a http or https URI
- The repository uses [1] *The Maven 2 Repository Layout*

An Artifact Repository representing a Remote Repository can be created using the createRepository(URI,Map) method, passing in the uri to the root of the repository. A NullPointerException must be thrown if the uri is null and an IllegalArgumentException must be thrown if the uri does not use the http or https scheme.

In addition to the repository URI the user may pass configuration properties in a Map. Implementations may support custom configuration properties, but those properties should use Reverse Domain Name keys. Keys not using the reverse DNS naming scheme are reserved for OSGi use. Implementations must ignore any configuration property keys that they do not recognise. All implementations must support the following properties:

- REMOTE_ARTIFACT_REPOSITORY_NAME - The name for this repository
- REMOTE_ARTIFACT_REPOSITORY_USER - The user name to use for authenticating with this repository
- REMOTE_ARTIFACT_REPOSITORY_PASSWORD - The password to use for authenticating with this repository
- REMOTE_ARTIFACT_REPOSITORY_BEARER_TOKEN - A bearer token to use when authenticating with this repository

- REMOTE_ARTIFACT_REPOSITORY_SNAPSHOTS_ENABLED - A Boolean indicating that SNAPSHOT versions are supported. Defaults to true
- REMOTE_ARTIFACT_REPOSITORY_RELEASES_ENABLED - A Boolean indicating that release versions are supported. Defaults to true
- REMOTE_ARTIFACT_REPOSITORY_TRUST_STORE - A trust store to use when validating a server certificate. May be a file system path or a data URI as defined by [2] *The Data URI scheme*.
- REMOTE_ARTIFACT_REPOSITORY_TRUST_STORE_FORMAT - The format of the trust store to use when validating a server certificate.
- REMOTE_ARTIFACT_REPOSITORY_TRUST_STORE_PASSWORD - The password to use when validating the trust store integrity.

Once created this Artifact Repository will search the supplied repository for any requested artifact data. Implementations are free to optimise checks using repository metadata.

160.3 Common themes

This specification includes support for bootstrapping an OSGi runtime, for ongoing management of an OSGi runtime, and for merging features. There are many concepts that apply across more than one of these scenarios, and so they are described here.

160.3.1 Overriding Feature variables

Some Feature definitions include variables which can be used to customise their deployment. These variables are intended to be set at the point where a Feature is installed, and may contain default values. To enable these variables to be overridden there are overloaded versions of methods which permit a Map of variables to be provided. The keys in this map must be strings and the values must be one of the types permitted by the *Feature Service Specification* on page 71

If a Feature declares a variable with no default value then this variable *must* be provided. If no value is provided then the method must fail to launch by throwing a LaunchException

160.3.2 Setting the bundle start levels

An OSGi framework contains a number of bundles which collaborate to produce a functioning application. There are times when some bundles require the system to have reached a certain state before they can be started. To address this use case the OSGi framework has the concept of *start levels* as described in the Start Level API Specification chapter of *OSGi Core Release 8*..

Setting the initial start level for the OSGi framework when bootstrapping can easily be achieved using the framework launch property org.osgi.framework.startlevel.beginning as defined by the OSGi core specification.

Controlling the start levels assigned to the bundles in a feature is managed through the use of Feature Bundle metadata. Specifically the Feature Launcher will look for a Feature Bundle metadata property named BUNDLE_START_LEVEL_METADATA which is of type integer and has a value between 1 and 2147483647 inclusive. If the property does not exist then the default start level will be used. If the property does exist and is not a suitable integer then launching must fail with a LaunchException.

Setting the default start level for the bundles, and the minimum start level required for an installed Feature is accomplished by using a Feature Extension named BUNDLE_START_LEVELS with Type JSON. The JSON contained in this extension is used to configure the default start level for the bundles, and the target start level for the framework. The schema of this JSON is as follows: ### Add Schema in build

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "http://www.osgi.org/json.schema/featurelauncher/v1.0.0/bundle-start-levels.json",
  "title": "bundle-start-levels",
  "description": "The definition of the bundle-start-levels feature extension",
   "type": "object",
   "properties": {
      "version": {
        "description": "The version of the Feature Launcher extension",
        "const": "1.0.0"
      },
      "defaultStartLevel": {
         "description": "The default start level for bundles in the feature",
         "type": "integer",
         "minimum": 1,
         "maximum": 2147483647
      },
      "minimumStartLevel": {
         "description": "The minimum required start level for the framework after feature installation",
         "type": "integer",
         "minimum": 1,
         "maximum": 2147483647
      }
   },
   "required": [ "version", "defaultStartLevel", "minimumStartLevel" ]
}
```

Setting the default start level for bundles installed by the framework is achieved using the defaultStartLevel property of the JSON extension. This must be an integer greater than zero and less than Integer.MAX_INT, or the special marker value null. A null value is used to indicate that the default start level for newly installed bundles is the current framework start level, or 1 if the current framework start level is 0. If the value is not valid then a LaunchException must be thrown when attempting to use the feature.

The minimum final start level for the OSGi framework required by the feature can be set using the minimumStartLevel property. of the JSON extension. This must be an integer greater than zero and less than Integer.MAX_INT. If the value is not valid then a LaunchException must be thrown when attempting to use the feature. This property sets the minimum start level that the OSGi framework must use to complete the installation of a Feature.

Finally the version property defines the version of the extension schema being used. This can be used by the implementation to determine whether the Feature is targeting a newer version of the specification. If the version is not understood by the implementation then a LaunchException must be thrown when attempting to use the feature.

160.3.3 Feature Decoration

Feature Decoration is a process by which features can be pre-processed before they are installed or updated. This gives users an opportunity to modify the feature, accept it as is, or block the operation from proceeding. There are two types of decorator:

- *Feature Decorators* - called for all operations. Can re-write the bundles, configurations, variables and extensions present in a feature.
- *Feature Extension Handlers* - called operations where the feature defines the named extension. Can re-write the bundles, configurations and variables present in a feature, but not the extensions.

Both types of decorator may pass through the feature unchanged by returning the feature object passed into them. This will cause the operation to continue as normal. Decorators may also block an operation from proceeding by throwing an AbandonOperationException. This will cause the operation to be immediately halted, and an exception thrown to the caller who requested the operation.

160.3.3.1 Building decorated features

Feature objects are expected to be immutable, and therefore a decorator cannot, and should not, change the feature object that is passed to them. Instead the decorator must create a new feature object which includes the decorated content.

To enable this both types of decorator are passed two builders, the first of which implements BaseFeatureDecorationBuilder and the second of which implements DecoratorBuilderFactory.

The former builder is similar to a FeatureBuilder but with three important differences:

- The builder is pre-populated with the information from the existing feature, such that immediately calling build() would create a feature with identical content to the original.
- Except where explicitly stated the builder configuration methods *replace* content rather than adding to it
- Only a limited subset of the feature content can be changed.

The latter builder is similar to a BuilderFactory but it cannot create FeatureBuilder instances.

By using these two builders a decorated feature can be configured and created. This decorated feature can then be returned from the decorator. Note that the *only* valid way to create a decorated feature is by using the builder. Any attempt to return a feature object which is not either:

- The original feature object.
- The object returned by build()

is an error and will result in the operation being abandoned.

160.3.3.2 Using Decorators

Decorators may be included using one of the relevant builder methods for a launch or runtime operation:

- withDecorator(FeatureDecorator)
- withExtensionHandler(String,FeatureExtensionHandler)
- withDecorator(FeatureDecorator)
- withExtensionHandler(String,FeatureExtensionHandler)

When registering a FeatureExtensionHandler the name of the extension to be handled must be passed, and cannot be null. This defines the name of the extension that the Feature Extension Handler will be used to process.

If multiple FeatureDecorator instances are registered then they will be called in the order that they were added.

If multiple FeatureExtensionHandler instances are registered for the same extension name then the earlier instances will be discarded. It is not possible to register more than one Feature Extension Handler for a single extension.

160.4 The Feature Launcher

The FeatureLauncher is the main entry point for creating a running OSGi framework containing the bundles and configurations defined in a Feature. As such the Feature Launcher is primarily designed for use outside of an OSGi framework.

To support usage in a non-OSGi environment implementations of the Feature Launcher Service must register the following implementation classes with the Java ServiceLoader API, and any necessary module metadata.

- org.osgi.service.featurelauncher.FeatureLauncher
- org.osgi.service.featurelauncher.repository.ArtifactRepositoryFactory

160.4.1 Obtaining and configuring a Feature Launcher

A Feature Launcher Service implementation must provide an implementation of the Feature Launcher interface. A user of the Feature Launcher service may use the following ways to find this class and create an instance:

- Using the Java ServiceLoader API to find instances of org.osgi.service.featurelauncher.FeatureLauncher
- From configuration, and then using Class.forName, getConstructor() and newInstance()
- By hard coding the implementation type and using the new operator.

Once instantiated the Feature Launcher may be supplied with a Feature, either as a Reader providing access to the JSON text of a Feature document or a parsed Feature to create a FeatureLauncher.LaunchBuilder. The Launch Builder can be configured in a fluent manner using the withConfiguration(Map), withVariables(Map), withFrameworkProperties(Map) and withRepository(ArtifactRepository) methods. Configuration properties for the Feature Launcher are implementation specific, and any unrecognised property names should be ignored. Artifact Repository instances may be created by the user using as described in *The Artifact Repository Factory* on page 94, or using custom implementations.

160.4.1.1 Thread Safety

Instances of the Feature Launcher and Launch Builder are not required to be Thread Safe, and should not be shared between threads. Changing the configuration of a Launch Builder instance only affects that instance, and not any other instances that exist.

160.4.2 Using a Feature Launcher

Once a configured Launch Builder instance has been created the launchFramework() method can be used to launch an OSGi framework containing the supplied Feature. The Feature Launcher will then return a running Framework instance representing the launched OSGi framework and the Feature that it contains. If an error occurs creating the framework, or locating and installing any of the feature bundles, then a LaunchException must be thrown.

Once the caller has received their framework instance they may carry on with other work, or they may wait for the OSGi framework to stop using the waitForStop() method.

160.4.2.1 Providing Framework Launch Properties

Framework launch properties are key value pairs which are passed to the OSGi framework as it is created. They can control many behaviours, including operations which happen before the framework starts, meaning that is not always possible to set them *after* startup.

Feature definitions that require particular framework launch properties can define them using a Feature Extension named FRAMEWORK_LAUNCHING_PROPERTIES. The Type of this Feature Extension must be TEXT, where each entry is in the form key=value All implementations of the Feature Launcher must support this extension, and use it to populate the Framework Launch Properties.

In addition to Framework Launch properties defined inside the Feature, users of the Feature Launcher can add and override Framework Launch Properties using one of the withFrameworkProperties method that permits a Map of framework properties to be provided. Any key value pairs defined in this map must take precedence over those defined in the Feature. A key with a null value must result in the removal of a key value pair if it is defined in the Feature.

160.4.2.2 Selecting a framework implementation

When defining a feature it is not always possible to be framework independent. Sometimes specific framework APIs, or licensing restrictions, will require that a particular implementation is used. In this case a Feature Extension named LAUNCH_FRAMEWORK with Type ARTIFACTS can be used to list one or more artifacts representing OSGi framework implementations.

The list of artifacts is treated as a preference order, with the first listed artifact being used if available, and so on, until a framework is found. If a listed artifact is not an OSGi framework implementation then the Feature Launcher must log a warning and continue on to the next artifact in the list. If the Kind of the feature is MANDATORY and none of the listed artifacts are available then launching must fail with a LaunchException.

The Feature Launcher implementation may identify that an artifact is an OSGi framework implementation in any way that it chooses, however it must recognise framework implementations that provide the Framework Launch API using the service loader pattern, as described in the Launching and Controlling a Framework section of *OSGi Core Release 8*.

160.4.2.3 A simple example

The following code snippet demonstrates a simple example of using the Feature Launcher to start an OSGi framework containing one or more bundles.

```
// Load the Feature Launcher
ServiceLoader<FeatureLauncher> sl = ServiceLoader.load(FeatureLauncher.class);
FeatureLauncher launcher = sl.iterator().next();

// Set up a repository
ArtifactRepository localRepo = launcher.createRepository(Paths.get("bundles"));

// Launch the framework
Framework fw = launcher
        .launch(Files.newBufferedReader(Paths.get("myfeature.json")))
        .withRepository(localRepo)
        .launchFramework();

fw.waitForStop(0);
```

160.4.3 The Feature Launching Process

The following section defines the process through which the Feature Launcher must locate, initialize and populate an OSGi framework when launching a feature. Unless explicitly stated implementations may perform one or more parts of this process in a different order to that described in the specification.

160.4.3.1 Feature Decoration

The first stage of launching is to determine the feature that should be launched by running the configured feature decoration handlers.

First the Feature Launcher must execute any registered FeatureDecorator instances in the order that they were registered. The feature returned by each decorator is used as input to the next.

Once the decoration is complete the Feature Launcher must iterate through the Feature Extensions defined by the feature. For each Feature Extension the launcher must:

1. Identify the Feature Extension Handler for the named extension.
2. If no Feature Extension Handler can be found, and the extension name is one of:
    - LAUNCH_FRAMEWORK
    - FRAMEWORK_LAUNCHING_PROPERTIES
    - BUNDLE_START_LEVELS

   then create an empty Feature Extension Handler which may validate the FeatureExtension.Type of the extension and must return the original feature.

3. If no Feature Extension Handler has been found or created then check the FeatureExtension.Kind of the extension. If it is MANDATORY then the launch fails with a LaunchException
4. Otherwise call the Feature Extension Handler, and use its result as input when calling any subsequent Feature Extension Handlers.

If any of the decorators throws an AbandonOperationException then the launch operation must immediately fail.

160.4.3.2 Locating a framework implementation

Before a framework instance can be created the Feature Launcher must identify a suitable implementation using the following search order:

1. If any provider specific configuration has been given to the Feature Launcher implementation then this should be used to identify the framework.
2. If the Feature declares an Extension LAUNCH_FRAMEWORK then the Feature Launcher implementation must use the first listed artifact that can be found in any configured Artifact Repositories, as described in *Selecting a framework implementation* on page 99.
3. If no framework implementation is found in the previous steps then the Feature Launcher implementation must search the classpath using the Thread Context Class Loader, or, if the Thread Context Class Loader is not set, the Class Loader which loaded the caller of the Feature Launcher's launch method. The first suitable framework instance located is the instance that will be used.
4. In the event that no suitable OSGi framework can be found by any of the previous steps then the Feature Launcher implementation may provide a default framework implementation to be used.

If no suitable OSGi framework implementation can be found then the Feature Launcher implementation must throw a LaunchException.

160.4.3.3 Creating a Framework instance

Once a suitable framework implementation has been located the Feature Launcher implementation must create and initialize a framework instance. Implementations are free to use implementation specific mechanisms for framework implementations that they recognise. The result of this initialization must be the same as if the Feature Launcher used the org.osgi.framework.launch.FrameworkFactory registered by the framework implementation to create the framework instance.

When creating the framework any framework launch properties defined in the Feature must be used. These are defined as described in *Providing Framework Launch Properties* on page 99 and must include any necessary variable replacement as defined by *Overriding Feature variables* on page 96.

Once instantiated the framework must be initialised appropriately so that it has a valid BundleContext. Once initialised the framework is ready for the Feature Launcher implementation to begin populating the framework.

160.4.3.4 Installing bundles and configurations

The Feature Launcher must iterate through the list of bundles in the feature, installing them in the same order that they are declared in the feature. If bundle start levels have been defined, as described in *Setting the bundle start levels* on page 96, then the Feature Launcher must ensure that the start level is correctly set for each installed bundle. If no start level metadata or extension is defined then all bundles are installed with the framework default start level.

If the installation of a bundle fails because it is determined by the framework to be a duplicate of an existing bundle then the Feature Launcher must treat the installation as a success. The start level of such a bundle must be set to the lower of its current value and the start level defined for the feature bundle that failed to install.

If a Feature defines one or more Feature Configurations then these cannot be guaranteed to be made available until the ConfigurationAdmin service has been registered. A Feature Launcher implementation may provide an implementation specific way to pre-register configurations, however in general the Feature Launcher should listen for the registration of the ConfigurationAdmin service and immediately create the defined configurations when it becomes available. Configurations must be created in the same order as they are defined in the Feature.

If the CONFIGURATION_TIMEOUT configuration property is set to 0, and one or more Feature Configurations are defined in the Feature being installed, then the implementation must throw a LaunchException unless it is capable of pre-registering those configurations in an implementation specific way.

160.4.3.5 Starting the framework

Once all of the the bundles listed in the feature are installed, and any necessary configuration listener is registered, the implementation must start the OSGi framework. This action will automatically start the installed bundles as defined by the initial start level of the framework, and the start levels of the installed bundles.

The Feature Launcher implementation must delay returning control to the caller until all configurations have been created, subject to the timeout defined by CONFIGURATION_TIMEOUT. The default timeout is 5000 milliseconds, and it determines the maximum length of time that the Feature Launcher implementation should wait to begin creating the configurations. A value of -1 indicates that the Feature Launcher implementation must not wait, and must continue immediately, even if the configurations have not yet been created. If it is not possible to begin before the timeout expires then a LaunchException must be thrown.

Finally, if the minimumStartLevel has been set by the BUNDLE_START_LEVELS extension then the Feature Launcher implementation must check the current start level of the framework. If the current start level is less than the value of minimumStartLevel then the framework's start level must be set to this value.

Once the start process is complete the Framework instance must be returned to the caller.

The following failure modes must all result in a LaunchException being thrown:

- A bundle fails to resolve. If one of the installed bundles fails to resolve this is an error *unless* the Feature is not complete. For Features that are not complete resolution failures must be logged, but not cause a failure.
- A resolved bundle fails to start. If one of the resolved bundles fails to start this is an error *unless* the bundle is a fragment or an extension bundle, which the Feature Launcher should not attempt to start.
- A configuration cannot be created. If a configuration cannot be created then this must result in a start failure

If a launching failure is triggered by an exception, for example a BundleException then this must be set as the cause of the LaunchException that is thrown.

160.4.3.6 Cleanup after failure

If the Feature Launcher implementation fails to launch a feature then any intermediate objects must be properly closed and discarded. For example if an OSGi framework instance has been created then it must be stopped and discarded.

160.5 The Feature Runtime Service

The Feature Runtime Service can be thought of as an equivalent of the Feature Launcher for an existing, running OSGi framework. The Feature Runtime Service therefore does not offer any mechanism for launching a framework, but instead allows one or more features to be installed into the running framework. As an OSGi framework is a dynamic environment the Feature Runtime Service also provides snapshots describing the currently installed Features, allows installed Features to be updated, and allows Features to be removed from the system.

An important difference between the Feature Launcher and Feature Runtime Service is that because the Feature Runtime Service allows multiple Features to be installed it must be able to identify and resolve simple conflicts. For example if two Features include the same bundle at different versions then the resolution may be to install only the higher version, or both versions.

160.5.1 Using the Feature Runtime

The Feature Runtime must be registered as a service in the service registry. Management agents that wish to install, manage or introspect Features in the framework must obtain this service. The Feature Service Runtime service must advertise the FeatureRuntime interface.

160.5.1.1 Thread Safety

Instances of the Feature Runtime are Thread Safe, regardless of whether the service is implemented as a singleton or otherwise. Any FeatureRuntime.OperationBuilder instances created by the Feature Runtime are *not* thread safe and must not be shared between threads.

Despite the Operation Builders not being Thread Safe the underlying Feature Runtime must remain Thread Safe, specifically if two Operation Builders complete at the same time then these calls should be handled sequentially such that there are never partially deployed Features present when installing, updating or removing a Feature.

160.5.1.2 Introspecting the installed Features

An important role for any management agent is being able to introspect the system to discover its current state. The Feature Runtime enables this through the getInstalledFeatures() method, which returns a snapshot of the current state of the system.

The returned list of snapshots contains one InstalledFeature entry for each installed Feature, in the order that they were installed, and may be empty if no Features have been installed. If the framework was started using a Feature Launcher from the same implementation as the Feature Runtime then the Feature Runtime may choose to represent the launched Feature in the snapshot list. If the launched Feature is included in the snapshot list then it must set isInitialLaunch() to true. Launch features cannot be removed or updated by the Feature Runtime, and any attempt to do so must throw a FeatureRuntimeException

Each Installed Feature provides:

- The ID of the Feature from getFeatureId()
- The List of InstalledBundle from getInstalledBundles() listing the bundles installed by the Runtime on behalf of the Feature.
- The List of InstalledConfiguration from getInstalledConfigurations() listing the configurations installed by the feature.

The InstalledBundle snapshots each represent a bundle installed by the Feature Runtime on behalf of the Feature. The Installed Bundle contains the following information:

- getBundleId() - The ID of the bundle that was installed.

- getAliases() - A Collection of one or more IDs that are known to correspond to this bundle. This list will always contain the bundleId and may contain additional IDs if their attempted installation resulted in a collision.
- getBundle() - The actual bundle that was installed into the runtime.
- getStartLevel() - The calculated start level for this bundle. Note that this start level may have been affected by other features.
- getOwningFeatures() - A List of the ids of the features which *own* the installed bundle. Ownership of a bundle is tracked by the Feature Runtime, and it is used to identify when the same bundle forms part of more than one Feature. Bundles that are owned by more than one Feature will not be removed until *all* of the Features that own them are removed.

  In the case where a bundle was not installed by the Feature Runtime, but later became owned by an installed Feature, that bundle will also be owned by the EXTERNAL_FEATURE_ID to indicate that they will not be removed if the other owning Feature is removed.

In addition to bundles Features can contain configurations. The InstalledConfiguration snapshots each represent a configuration created by the Feature Runtime on behalf of the Feature. The Installed Configuration contains the following information:

- getPid() - The configuration pid of this configuration.
- getFactoryPid() - The factory pid of this configuration, or an empty Optional if the configuration is not a factory configuration.
- getProperties() - The merged configuration properties that result from the full set of installed Features contributing to this configuration. Note that there is no dynamic link to Configuration Admin and so any configuration changes made outside the Feature Runtime will not be reflected.
- getOwningFeatures() - A List of the ids of the features which *own* the configuration. Ownership of a configuration is tracked by the Feature Runtime, and it is used to identify when the same configuration, as defined by its pid, forms part of more than one Feature. Configurations that are owned by more than one Feature will not be removed until *all* of the Features that own them are removed.

  In the case where a configuration was not installed by the Feature Runtime, but later became owned by an installed Feature, that configuration will also be owned by the EXTERNAL_FEATURE_ID to indicate that they will not be deleted if the other owning Feature is removed.

160.5.1.3 Installing a feature

Installing a Feature uses one of the install methods present on the Feature Runtime. These methods allow the caller to provide the Feature to be installed and return an FeatureRuntime.InstallOperationBuilder to allow the caller to configure their installation operation. Configuration of operations includes:

- *Setting variable overrides* on page 105.
- *Setting the available Artifact Repositories* on page 105
- *Feature Decoration* on page 97
- Adding *Merging strategies* on page 105

Once the operation is fully configured then the caller uses the install() method to begin the installation. The end result of installing a Feature is that all of the bundles listed in the Feature are installed, all of the Feature Configurations have been created, all bundles have been marked as persistently started, and the framework start level is at least the minimum level required by the Feature.

Start levels for the bundles in the Feature may be controlled as described in *Setting the bundle start levels* on page 96. If any bundles are installed with a start level higher than the current framework start level then they will be marked persistently started but will not start until the framework start level is changed.

In more complex cases, where multiple features are installed with overlapping bundles or configurations then *Merging strategies* on page 105 will be applied to determine which bundles are installed, and what configuration properties will be used when creating or updating a configuration.

If a failure occurs during the installation of a Feature then the Feature Runtime must make every effort to return the system to its pre-existing state. After a failure no new bundles should be installed, any altered configurations returned to their previous states, and the framework start level should be the same as it was prior to the failed installation.

160.5.1.4 Setting the available Artifact Repositories

As with the Feature Launcher, in order to successfully locate the bundles listed in a feature the Feature Runtime must have access to one or more Artifact Repositories capable of locating the bundles. These Artifact Repositories are configured into each Operation Builder by the user.

A configured Feature Runtime will typically include one or more pre-defined Artifact Repositories. These pre-defined repositories are available to view via the `getDefaultRepositories()`. By default all Operation Builders will have access to these repositories when completing. This behaviour can be changed using the `useDefaultRepositories(boolean)` method.

Additional Artifact Repositories can be added to an Operation Builder by calling the `addRepository(String,ArtifactRepository)` method. The supplied `name` is used to identify the repository. If the supplied name is already used for an existing Artifact Repository then it will be replaced or, if the supplied Artifact Repository is `null`, removed. A named Artifact Repository added in this way will override a default Artifact Repository with the same name.

160.5.1.5 Setting variable overrides

As described in *Overriding Feature variables* on page 96 a feature may define zero or more overridable properties which can be used to alter the deployment of the feature. These properties may be configured into each Operation Builder by calling the `withVariables(Map)` method. The supplied `Map` contains the keys and values that will override the variables in the Feature.

160.5.1.6 Merging strategies

Merge operations occur when two or more features reference the same, or similar, items to be installed. The purpose of a merge operation is to avoid unnecessary duplication, and to resolve conflicts.

Merging potentially applies whenever a Feature is installed, updated or removed, and may result in different outcomes depending on the strategy used. All runtime merge functions therefore receive a `MergeOperationType` indicating which type of operation is currently running.

160.5.1.6.1 Merging Bundles

Features may define bundles to be installed by including Feature Bundle entries. If two or more Features include Feature Bundles which have IDs with the same group id and artifact id, but which are not the same, then this situation requires a merge to resolve the possible conflict. Determining whether two IDs are the same is accomplished by checking whether they return equal strings from `toString()`.

When a possible conflict is detected the Feature Runtime must call a `RuntimeBundleMerge` to identify the correct actions to take. These actions include:

- Whether to install the candidate Feature Bundle or not
- Whether to re-designate the ownership of any existing Installed Bundles
- Whether to remove any existing Feature Bundles

Although the obvious time for a bundle merge operation to occur is during an `INSTALL` operation, merges may also occur during `UPDATE` and `REMOVE` operations. During an `UPDATE` the existing bundles from the Feature being updated will remain available so that the updated Feature may be merged into the existing runtime. During a REMOVE a merge will occur to allow Feature ownership to be re-allocated if a shared bundle is being removed.

Merges are resolved by the mergeBundle method which receives:

- The type of the operation, one of INSTALL, UPDATE or REMOVE.
- The Feature being operated on
- The Feature Bundle which requires merging
- A Collection of Installed Bundles representing the currently installed bundles which have an overlapping groupId and artifactId. Note that in the case of an UPDATE or REMOVE operation the Feature being updated or removed will not be present in the collection of owning features for any of the Installed Bundles.
- A List of RuntimeBundleMerge.FeatureBundleDefinition representing the existing Features which form part of the merge operation. Note that in the case of an UPDATE or REMOVE operation the Feature Bundle being updated or removed will not be present in the list. Entries in the list are present in the order that the Features were installed into the runtime.

The result of the merge function is a Stream of RuntimeBundleMerge.BundleMapping. Each Bundle Mapping links a bundle ID to List of feature IDs. The Bundle Mapping's bundle id must only be a bundleId found in the list of Installed Bundles or, in the case of an INSTALL or UPDATE operation, the id of the Feature Bundle being merged. The mapped Feature ids must contain the id of every Feature in the supplied Feature Bundle Definitions, and, in the case of an INSTALL or UPDATE operation, the id of the Feature being merged. If the id of any Installed Bundle is not present in the returned Stream then that bundle will be removed as part of the ongoing operation. If the same bundle id is present more than once the the two mappings will be combined using the union of the mapped Feature ids.

A simple example of a merge strategy which combines configurations by upgrading Features to the highest compatible version could be implemented as follows:

```
public Map<ID,List<ID>> mergeBundle(MergeOperationType operation,
  Feature feature, FeatureBundle toMerge,
  List<InstalledBundle> installedBundles,
  Map<FeatureBundle,Feature> existingFeatureBundles) {

 Map<ID,List<ID>> result;

 if (operation == MergeOperationType.REMOVE) {
  // Just keep everything the same
  result = installedBundles.stream()
    .filter(i -> !i.getOwningFeatures().isEmpty())
    .collect(Collectors.toMap(i -> i.getBundleId(),
      i -> i.getOwningFeatures()));
 } else {
  // Find the Installed bundles we might replace
  Version v = RuntimeMerges.getOSGiVersion(toMerge.getID());

  List<InstalledBundle> sameMajor = new ArrayList<>();
  List<InstalledBundle> differentMajor = new ArrayList<>();

  installedBundles.forEach(i -> {
   if (i.getBundle().getVersion().getMajor() == v.getMajor()) {
    sameMajor.add(i);
   } else {
    differentMajor.add(i);
   }
  });

  // Bundles with a different major version stay the same
  result = differentMajor.stream()
    .filter(i -> !i.getOwningFeatures().isEmpty())
    .collect(Collectors.toMap(i -> i.getBundleId(),
      i -> i.getOwningFeatures()));

  // Find the biggest existing version and see if it's bigger than v
  Optional<InstalledBundle> max = sameMajor.stream()
    .max((a, b) -> a.getBundle().getVersion()
      .compareTo(b.getBundle().getVersion()))
    .filter(m -> m.getBundle().getVersion().compareTo(v) >= 0);

  // Use the old version if it's bigger, or the new if not
  ID key = max.isPresent() ? max.get().getBundleId() : toMerge.getID();

  Stream<ID> featureIds = sameMajor.stream()
    .flatMap(i -> i.getOwningFeatures().stream());

  result.put(key,
    Stream.concat(Stream.of(feature.getID()), featureIds)
      .collect(Collectors.toList()));
 }
 return result;
}
```

160.5.1.6.2 Merging Configurations

Features may define configurations by including Feature Configuration entries. If two or more Features include properties for the same configuration PID then this situation requires a merge to resolve the conflict.

Merges are resolved by a RuntimeConfigurationMerge which receives:

- The type of the operation, one of INSTALL, UPDATE or REMOVE.
- The Feature being operated on
- The Feature Configuration which requires merging
- The Installed Configuration representing the current state of the configuration. Note that in the case of an UPDATE or REMOVE operation the Feature being updated or removed will not be present in the list of owning features.
- A List of RuntimeConfigurationMerge.FeatureConfigurationDefinition representing the existing Features which form part of the merge operation. Note that in the case of an UPDATE or REMOVE operation the Feature Configuration being updated or removed will not be present in the list. Entries in the list are present in the order that the Features were installed into the runtime.

The result of the merge function is a map of configuration properties that should be used to update the configuration. If the map is null then the configuration should be deleted.

A simple example of a merge strategy which combines configurations by overlaying each in turn and ignoring null configurations could be implemented as follows:

```
public Map<String,Object> mergeConfiguration(MergeOperationType operation,
  Feature feature, FeatureConfiguration toMerge, InstalledConfiguration configuration,
  List<FeatureConfigurationDefinition> existingFeatureConfigurations) {
```

```
  boolean addedSomething = false;

  Map<String,Object> result = new HashMap<>();

  for (FeatureConfigurationDefinition fcd : existingFeatureConfigurations) {
   FeatureConfiguration fc = fcd.getFeatureConfiguration();
   if(fc.getValues() != null) {
    result.putAll(fc.getValues());
    addedSomething = true;
   }
  }

  if(operation != MergeOperationType.REMOVE && toMerge.getValues() != null) {
   result.putAll(toMerge.getValues());
   addedSomething = true;
  }

  return addedSomething ? result : null;
 }
```

160.5.1.7 Removing a Feature

Removing a feature from the Feature Runtime Service uses the remove(ID) method to uninstall and remove a feature from the framework. Removing a feature is a comparatively simple operation, and therefore does not require the configuration of an FeatureRuntime.OperationBuilder.

Once the remove method returns the feature will have been removed from the Feature Runtime, and any links to installed bundles and configurations will have been removed. If this leaves any installed bundles or installed configurations with no owners then these will be uninstalled or deleted from the system as appropriate.

If a failure occurs during the removal of a feature then the Feature Runtime must make every effort to fully remove the feature, for example by continuing to remove installed bundles that no longer have any owners. Exceptions that occur must be logged, and upon completion the Feature Runtime must throw a FeatureRuntimeException which indicates the incomplete removal.

It is not an error to remove a feature which does not exist in the Feature Runtime and this must return without error, and without altering the state of the system. It is an error to attempt to remove any feature that returns true for isInitialLaunch(), and any attempt to do so must result in a FeatureRuntimeException.

160.5.1.8 Updating a Feature

Updating a Feature uses one of the update methods present on the Feature Runtime. These methods allow the caller to indicate which feature should be updated, and provider the new Feature definition to replace it with. The methods return an FeatureRuntime.UpdateOperationBuilder to allow the caller to configure their update operation. Configuration of operations includes:

- *Setting variable overrides* on page 105.
- *Setting the available Artifact Repositories* on page 105
- *Feature Decoration* on page 97
- Adding *Merging strategies* on page 105

Once the operation is fully configured then the caller uses the update() method to begin the update. The end result of updating a Feature is that all of the bundles listed in the new Feature are installed, all of the Feature Configurations in the new Feature have been created, all bundles have been marked as persistently started, and the framework start level is at least the minimum level required by the new Feature. In addition, any bundles and configurations from the old Feature that are not present in the new Feature will have been removed, and any configurations present in both the old and new Features will have been updated with new any new content.

At a high level an update operation is therefore superficially similar to performing a remove operation followed by an install operation. The key difference, however, is that any bundles and configurations shared by both features, or identified by a merge strategy, will not be removed, and instead will become owned by the new Feature.

As for installation, start levels for the bundles in the new Feature will be determined as described in *Setting the bundle start levels* on page 96. If any bundles are installed with a start level higher than the current framework start level then they will be marked persistently started but will not start until the framework start level is changed.

Where the feature update includes overlapping bundles or configurations then *Merging strategies* on page 105 will be applied to determine which bundles are installed, and what configuration properties will be used when creating or updating a configuration.

If a failure occurs during the update of a Feature then the Feature Runtime must make every effort to return the system to its pre-existing state. After a failure no new bundles should be installed, any altered configurations returned to their previous states, and the framework start level should be the same as it was prior to the failed installation.

160.5.2 The Feature Runtime Behaviour

The following section provides normative requirements for the behaviour of the Feature Runtime when it is used. This includes the necessary end states after installation, update and removal of Features.

160.5.2.1 The Feature installation process

The Feature Installation process has four main phases:

- The feature decoration phase, where the Feature is decorated and validated
- The bundle installation phase, where Feature bundles are installed
- The configuration creation phase, where Feature Configurations are created
- The Feature Start phase, where Bundles are started.

The feature decoration phase must complete before any other phases can begin. The the bundle installation phase and the configuration creation phase may happen in any order, or even with interleaved steps, however the Feature Start phase must not begin until the bundle installation and configuration creation phases are complete.

160.5.2.1.1 Feature Decoration

The first stage of the operation is to determine the feature that should be used by running the configured feature decoration handlers.

First the Feature Runtime must execute any registered FeatureDecorator instances in the order that they were registered. The feature returned by each decorator is used as input to the next.

Once the decoration is complete the Feature Runtime must iterate through the Feature Extensions defined by the feature. For each Feature Extension the Feature Runtime must:

1. Identify the Feature Extension Handler for the named extension.
2. If no Feature Extension Handler can be found, and the extension name is one of:
   - LAUNCH_FRAMEWORK
   - FRAMEWORK_LAUNCHING_PROPERTIES
   - BUNDLE_START_LEVELS

   then create an empty Feature Extension Handler which may validate the FeatureExtension.Type of the extension and must return the original feature.

3. If no Feature Extension Handler has been found or created then check the FeatureExtension.Kind of the extension. If it is MANDATORY then the operation fails with a FeatureRuntimeException
4. Otherwise call the Feature Extension Handler, and use its result as input when calling any subsequent Feature Extension Handlers.

If any of the decorators throws an AbandonOperationException then the operation must immediately fail.

160.5.2.1.2 Bundle Installation

When a feature is being installed the Feature Runtime identifies the bundles to be installed. The Feature Runtime also gathers the set of bundles that are already installed, and then computes the overlap between these. Bundles are deemed to overlap if they have the same group id, artifact id, type and classifier but they may differ in version.

If the overlap list contains entries which overlap exactly, that is they have the same version in the runtime and the Feature being installed, then those bundles are removed from the list of bundles to be installed and the existing bundles are marked as *owned* by the Feature being installed. If the marked bundles were not previously owned by any other feature then they also marked as owned by the EXTERNAL_FEATURE_ID to indicate that they will not be removed if the Feature being installed is removed.

Any remaining overlap entries are processed according to the merge strategy for the feature, as described in *Merging Bundles* on page 105. The final list of bundles to install, which excludes any already installed bundles, is then installed in the same order as it was defined by the feature. Each bundle in the feature, including bundles that were already installed, is then marked as owned by the installing feature.

If the installation of a bundle fails because it is determined by the framework to be a duplicate of an existing bundle then the Feature Runtime must treat the installation as a success and add the ID as an alias for the existing Installed Bundle. The start level of such a bundle must be set to the lower of its current value and the start level defined for the feature bundle that failed to install.

Once the installation of bundles is complete the Feature Runtime must uninstall any bundles which were identified for removal as part of any merge processes.

160.5.2.1.3 Configuration Creation

As part of the initial Feature installation the Feature Runtime must also process and create any Feature Configurations that are defined in the Feature. Feature Configurations cannot be guaranteed to be made available until a ConfigurationAdmin service has been registered. A Feature Runtime implementation should therefore listen for the registration of a ConfigurationAdmin service and immediately create or update any pending configurations when it becomes available. Configurations must be created or updated in the same order as they are defined in the Feature.

If the same configuration, as identified by its configuration pid, is defined in one or more existing installed Features then the configuration properties to be used are determined by merging the previous configuration properties with the new properties defined in the Feature, as described in *Merging Configurations* on page 107. If at the point where the FeatureRuntime attempts to create or update a Feature Configuration there are already configuration properties defined in ConfigurationAdmin then these must be ignored and replaced using updateIfDifferent(Dictionary) unless the Configuration is marked as READ_ONLY. If a READ_ONLY configuration does exist then the Feature Runtime must log a warning and skip that configuration.

160.5.2.1.4 Feature Start

Once all of the bundles listed by the feature are installed then the bundles' start levels are assigned as described in *Setting the bundle start levels* on page 96. This includes any pre-existing bundles and the results of any merge operations. If no start level configuration is defined in the feature for a particular bundle then the start level for that bundle is set to the current start level of the framework.

The Feature Runtime must then identify the lowest start level referenced in the Feature, and repeatedly run through the list of bundles, in the order that they are defined in the Feature, looking for bundles which match the identified start level. For each bundle the Feature Runtime must:

- If the bundle was installed in the Bundle Installation phase then set the start level for the bundle.
- If the bundle was already installed then update the start level for the bundle if, and only if, the new start level is lower than the existing start level.
- Mark the bundle as persistently started unless it is a *fragment* bundle.

The Feature Runtime must then identify the next lowest start level referenced in the Feature and repeat this process until all bundles have been persistently started. Once this process is complete then the framework start level must be increased to the minimum start level required by the Feature, or returned to the original framework start level if this is higher and was decreased as part of *Merging Bundles* on page 105.

160.5.2.1.5 Failure scenarios

The following is a non-exhaustive list of possible failure scenarios that must be handled.

- The feature being installed is already known to the Feature Runtime. This must be treated as a failure as the configuration of the InstallOperationBuilder may not be the same as the previous installation. The Feature Runtime must make no changes and immediately throw a FeatureRuntimeException.
- A Feature Bundle cannot be found by any configured ArtifactRepository.
- A BundleException is thrown during *Bundle Installation* on page 110.
- A BundleException is thrown during *Feature Start* on page 110.
- A Feature Configuration cannot be created by the ConfigurationAdmin service.
- An Exception is thrown by any configured ArtifactRepository, RuntimeBundleMerge or RuntimeConfigurationMerge.

In all cases the first exception must be treated as a failure, with the installation process halting immediately. The feature must then be removed from the runtime in a similar manner to calling remove for the feature id. Once the feature removal is complete the failure may be used in creating the FeatureRuntimeException that must be thrown by this method.

160.5.2.2 The Feature removal process

The Feature removal process has four main phases:

- The feature removal phase, where the feature is removed from the Feature Runtime.
- The bundle stop phase, where Installed Bundles without owners are stopped.
- The configuration deletion phase, where Installed Configurations without owners are removed
- The bundle removal phase, stopped bundles are uninstalled

The the feature removal and bundle stop phases may happen in any order, or even with interleaved steps. The same is true for the configuration deletion phase and the bundle removal phase, however these phases must not begin until the bundle stop phase is complete.

160.5.2.2.1 Feature Removal

Feature removal is a simple operation which removes any reference to the Installed Feature from the Feature Runtime. This includes the list of installed features, and the ownership lists of any Installed Bundles or Installed configurations in the Feature Runtime. After removal is complete the ID of the removed feature should not appear anywhere in the Feature Runtime.

Installed Bundles and Installed Configurations which have zero owners after the removal of the feature are now considered eligible for removal. Their removal processes are described in the next phases.

160.5.2.2.2 Bundle Stop

The Feature Runtime must identify the highest start level set by the list of Installed Features, excluding the Feature being removed. If no start level is defined by this list of features then no action is taken, otherwise the framework start level is set to the newly identified start level.

The list of bundles eligible to be stopped, as determined in *Feature Removal* on page 111, is used to peristently stop any remaining bundles. Bundles that are eligible for removal are stopped in the reverse order in which they were started by *Feature Start* on page 110. This is accomplished by stopping the bundles with the highest start level first, using the reverse order of declaration in the feature where the start level is the same. If an eligible bundle is already stopped due to its start level then it must still be persistently stopped.

160.5.2.2.3 Configuration Removal

Once the *Bundle Stop* on page 112 phase has completed the Feature Runtime may begin removing configurations that are eligible. As with bundles, configurations become eligible for removal if they are no longer owned by any feature. Eligible configurations must be removed in the reverse order of creation, that is the reverse order that they were listed in the feature being removed.

160.5.2.2.4 Bundle Removal

Once the *Bundle Stop* on page 112 phase has completed the Feature Runtime may begin uninstalling bundles from the OSGi framework. These bundles must only be eligible bundles identified and stopped as part of the previous phase. Bundles are uninstalled in reverse installation order, that is the reverse of the order in which they are listed in the feature.

If one or more bundles have been uninstalled, and once all eligible bundles have been uninstalled, the Feature Runtime must refresh the framework wiring by calling `FrameworkWiring.refreshBundles`, passing the list of uninstalled bundles. This will cause the framework to completely remove the uninstalled bundles, and any wirings that link to them.

160.5.2.2.5 Failure scenarios

The following is a non-exhaustive list of possible failure scenarios that must be handled.

- The feature being removed is not known to the Feature Runtime. This must not be treated as a failure, and should simply return immediately.
- One or more `BundleException`s are thrown during *Bundle Stop* on page 112. These exceptions should be logged when they occur, but then ignored.
- One or more `BundleException`s are thrown during *Bundle Removal* on page 112. These exceptions should be logged when they occur, with the Feature Runtime continuing despite the errors. Once the feature removal is complete the failures may be used in creating the `FeatureRuntimeException` that must be thrown by this method.
- One or more Installed Configurations are missing from the `ConfigurationAdmin` service. These missing configurations should be logged with a warning, but not treated as an error.
- One or more Installed Configurations cannot be deleted missing from the `ConfigurationAdmin` service. These exceptions should be logged when they occur, with the Feature Runtime continuing despite the errors. Once the feature removal is complete the failures may be used in creating the `FeatureRuntimeException` that must be thrown by this method.

The Feature Update Process

The Feature Update Process can be viewed as an interleaved remove and installation operation, following the phases present in both.

- The feature decoration phase, where the new Feature is decorated and validated
- The feature removal phase, where the existing feature is removed from the Feature Runtime.
- The bundle installation phase, where the new Feature bundles are installed
- The bundle stop phase, where Installed Bundles without owners are stopped.
- The configuration creation and update phase, where the new Feature Configurations are created or updated
- The configuration deletion phase, where Installed Configurations without owners are removed
- The Feature Start phase, where Bundles in the new feature are started.
- The bundle removal phase, stopped bundles are uninstalled

160.5.2.3.1 Decorating the new Feature

Decorating the new feature proceeds exactly as if a new feature is being installed, as described in *Feature Decoration* on page 109.

160.5.2.3.2 Removing the existing Feature

Removing the existing feature proceeds exactly as if a new feature is being removed, as described in *Feature Removal* on page 111.

160.5.2.3.3 Installing the new bundles

Installing the bundles from the new feature proceeds as if a new feature is being removed, as described in *Bundle Installation* on page 110, but with two important differences.

The first important difference is that bundles being installed must be prevented from wiring to bundles that are eligible for removal. This may be accomplished through the use of a Resolver Hook. As the resolver may attempt to resolve bundles at any time this restriction must be enforced by the Feature Runtime until after all of the eligible bundles are uninstalled.

The second important difference is that any Installed Bundles that are eligible for removal are *still available* in the runtime. This means that they *must be considered* when determining whether bundles are already installed, or whether they need to be merged. This may lead to one or more Installed Bundles that were eligible for removal becoming *ineligible* for removal as they become owned by the new feature. Any Installed Bundles for which this is the case must be removed from the list of eligible bundles, and immediately become available for wiring by newly installed bundles.

160.5.2.3.4 Stopping the eligible bundles

Stopping the eligible bundles proceeds exactly as described in *Bundle Stop* on page 112. Note that if the existing feature used start levels then this process will likely result in one or more bundles shared between the old and new features being stopped temporarily.

Care must be taken in this phase to persistently stop all eligible bundles. Failing to do so may result in eligible bundles being accidentally restarted in later phases.

160.5.2.3.5 Creating and Updating Configurations

Creating and updating configurations proceeds as described in *Configuration Creation* on page 110, but with one important difference.

Any Installed Configurations that are eligible for removal are *still available* in the runtime. This means that they *must be considered* when determining whether they need to be merged. This may lead to one or more Installed Configurations that were eligible for removal becoming *ineligible* for removal as they become owned by the new feature. Any Installed Configurations for which this is the case must be removed from the list of eligible configurations.

160.5.2.3.6 Removing Configurations

Removing eligible configurations proceeds exactly as described in *Configuration Removal* on page 112.

160.5.2.3.7 Starting the new feature

Starting the new feature proceeds exactly as described in *Feature Start* on page 110. As all bundles eligible for removal were persistently stopped in an earier phase they will remain stopped during this phase, and must not be started again.

160.5.2.3.8 Uninstalling the eligible bundles

Until the Feature Runtime reaches this phase of an update it must fail by attempting to roll back to the previous feature. Once this phase has been reached this failure mode changes, and the Feature Runtime must retain the new Feature, attempting to continue despite failures.

Removing the eligible bundles proceeds exactly as described in *Bundle Removal* on page 112.

160.5.2.3.9 Failure scenarios

The following is a non-exhaustive list of possible failure scenarios that must be handled.

- The feature being updated is not known to the Feature Runtime. This must not make any changes and should immediately throw a `FeatureRuntimeException`.
- A Feature Bundle cannot be found by any configured `ArtifactRepository`.
- A `BundleException` is thrown during *Installing the new bundles* on page 113. This should result in the imediate failure of the operation, rolling back to the pre-update state, with a `FeatureRuntimeException` thrown to the caller.
- A `BundleException` is thrown during *Starting the new feature* on page 114. This should result in the imediate failure of the operation, rolling back to the pre-update state, with a `FeatureRuntimeException` thrown to the caller.
- A Feature Configuration cannot be created by the `ConfigurationAdmin` service. This should result in the imediate failure of the operation, rolling back to the pre-update state, with a `FeatureRuntimeException` thrown to the caller.
- An Exception is thrown by any configured `ArtifactRepository`, `RuntimeBundleMerge` or `RuntimeConfigurationMerge`. This should result in the imediate failure of the operation, rolling back to the pre-update state, with a `FeatureRuntimeException` thrown to the caller.
- One or more `BundleException`s are thrown during *Stopping the eligible bundles* on page 113. These exceptions should be logged when they occur, but then ignored.
- One or more `BundleException`s are thrown during *Uninstalling the eligible bundles* on page 114. These exceptions should be logged when they occur, with the Feature Runtime continuing despite the errors. Once the feature removal is complete the failures may be used in creating the `FeatureRuntimeException` that must be thrown by this method.
- One or more Installed Configurations are missing from the `ConfigurationAdmin` service. These missing configurations should be logged with a warning, but not treated as an error.
- One or more Installed Configurations cannot be deleted missing from the `ConfigurationAdmin` service. These exceptions should be logged when they occur, with the Feature Runtime continuing despite the errors. Once the feature removal is complete the failures may be used in creating the `FeatureRuntimeException` that must be thrown by this method.

160.6 Capabilities

The Feature Launcher must provide the following capabilities.

160.6.1 osgi.service Capability

The bundle providing the Feature Runtime service must provide capabilities in the `osgi.service` namespace representing the services it is required to register. This capability must also declare uses constraints for the relevant service packages:

```
Provide-Capability: osgi.service;
 objectClass:List<String>="org.osgi.service.featurelauncher.runtime.FeatureRuntime";
 uses:="org.osgi.service.featurelauncher.runtime",
 osgi.service;
 objectClass:List<String>="org.osgi.service.featurelauncher.repository.ArtifactRepositoryFac
 uses:="org.osgi.service.featurelauncher.repository"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 65.

160.7 Security

When Java permissions are enabled, the following security procedures apply.

160.7.1 Required Permissions

Bundles that need to make use of the Feature Runtime or Artifact Repository Factory services must be granted permission to get the relevant service, for example ServicePermission[org.osgi.service.featurelauncher.runtime.FeatureRuntime, GET] so that they may retrieve the service and use it.

Only a bundle that provides a Feature Runtime implementation should be granted ServicePermission[org.osgi.service.featurelauncher.runtime.FeatureRuntime, REGISTER] and ServicePermission[org.osgi.service.featurelauncher.repository.ArtifactRepositoryFactory, REGISTER] to register the services defined by this specification.

The Feature Runtime implementation must also be granted ServicePermission[org.osgi.service.cm.ConfigurationAdmin, GET], AdminPermission[*, execute], AdminPermission[*, lifecycle], AdminPermission[*, metadata], AdminPermission[*, resolve], AdminPermission[*, startlevel], AdminPermission[*, context], as these actions are all required to implement the specification.

160.8 org.osgi.service.featurelauncher

Feature Launcher 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.featurelauncher; version="[1.0,2.0)"

Example import for providers implementing the API in this package:

Import-Package: org.osgi.service.featurelauncher; version="[1.0,1.1)"

160.8.1 Summary

- FeatureLauncher - The Feature launcher is the primary entry point for launching an OSGi framework and set of bundles.
- FeatureLauncher.LaunchBuilder - A builder for configuring and triggering the launch of an OSGi framework containing the supplied feature
- FeatureLauncherConstants - Defines standard constants for the Feature Launcher specification.
- LaunchException - A LaunchException is thrown by the FeatureLauncher if it is unable to:
  - Locate or start an OSGi Framework instance

- Locate the installable bytes of any bundle in a Feature
- Install a bundle in the Feature
- Determine a value for a Feature variable that has no default value defined

160.8.2 public interface FeatureLauncher extends ArtifactRepositoryFactory

The Feature launcher is the primary entry point for launching an OSGi framework and set of bundles. As it is a means for launching a framework it is designed to be used from outside OSGi and therefore should be obtained using the ServiceLoader.

Provider Type Consumers of this API must not implement this type

160.8.2.1 public FeatureLauncher.LaunchBuilder launch(Feature feature)

feature the feature to launch

□ Begin launching a framework instance based on the supplied feature

Returns A running framework instance.

Throws `LaunchException`–

160.8.2.2 public FeatureLauncher.LaunchBuilder launch(Reader jsonReader)

jsonReader a Reader for the input Feature JSON

□ Begin launching a framework instance based on the supplied feature JSON

Returns A running framework instance.

Throws `LaunchException`–

160.8.3 public static interface FeatureLauncher.LaunchBuilder

A builder for configuring and triggering the launch of an OSGi framework containing the supplied feature

LaunchBuilder instances are single use. Once they have been used to launch a framework instance they become invalid and all methods will throw IllegalStateException

160.8.3.1 public Framework launchFramework()

□ Launch a framework instance based on the configured builder

Returns A running framework instance.

Throws `LaunchException`–

`IllegalStateException`– if the builder has been launched

160.8.3.2 public FeatureLauncher.LaunchBuilder withConfiguration(Map<String, Object> configuration)

configuration the configuration for this implementation

□ Configure this LaunchBuilder with the supplied properties.

Returns `this`

Throws `IllegalStateException`– if the builder has been launched

160.8.3.3 public FeatureLauncher.LaunchBuilder withDecorator(FeatureDecorator decorator)

decorator the decorator to add

□ Add a FeatureDecorator to this LaunchBuilder that will be used to decorate the feature being launched. If called multiple times then the supplied decorators will be called in the same order that they were added to this builder.

Returns this

Throws NullPointerException– if the decorator is null

IllegalStateException– if the builder has been launched

160.8.3.4 public FeatureLauncher.LaunchBuilder withExtensionHandler(String extensionName, FeatureExtensionHandler extensionHandler)

extensionName the name of the extension to handle

extensionHandler the extensionHandler to add

□ Add a FeatureExtensionHandler to this LaunchBuilder that will be used to process the named FeatureExtension if it is found in the Feature being launched. If called multiple times for the same extensionName then later calls will replace the extensionHandler to be used.

Returns this

Throws NullPointerException– if the extension name or decorator is null

IllegalStateException– if the builder has been launched

160.8.3.5 public FeatureLauncher.LaunchBuilder withFrameworkProperties(Map<String, Object> frameworkProps)

frameworkProps the launch properties to use when starting the framework

□ Configure this LaunchBuilder with the supplied Framework Launch Properties.

Returns this

Throws IllegalStateException– if the builder has been launched

160.8.3.6 public FeatureLauncher.LaunchBuilder withRepository(ArtifactRepository repository)

repository the repository to add

□ Add a repository to this LaunchBuilder that will be used to locate installable artifact data.

Returns this

Throws NullPointerException– if the repository is null

IllegalStateException– if the builder has been launched

160.8.3.7 public FeatureLauncher.LaunchBuilder withVariables(Map<String, Object> variables)

variables the variable placeholder overrides for this launch

□ Configure this LaunchBuilder with the supplied variables.

Returns this

Throws IllegalStateException– if the builder has been launched

160.8.4 public final class FeatureLauncherConstants

Defines standard constants for the Feature Launcher specification.

160.8.4.1 public static final String BUNDLE_START_LEVEL_METADATA = "bundleStartLevel"

The name of the metadata property used to indicate the start level of the bundle to be installed. The value must be an integer between 0 and Integer.MAX_VALUE.

160.8.4.2 public static final String BUNDLE_START_LEVELS = "bundle-start-levels"

The name for the FeatureExtension of Type.JSON which defines the start level configuration for the bundles in the feature

160.8.4.3 public static final String CONFIGURATION_TIMEOUT = "configuration.timeout"

The configuration property used to set the timeout for creating configurations from FeatureConfiguration definitions.

The value must be a Long indicating the number of milliseconds that the implementation should wait to be able to create configurations for the Feature. The default is 5000.

A value of 0 means that the configurations must be created before the bundles in the feature are started. In general this will require the ConfigurationAdmin service to be available from outside the feature.

A value of -1 means that the implementation must not wait to create configurations and should return control to the user as soon as the bundles are started, even if the configurations have not yet been created.

160.8.4.4 public static final String FEATURE_LAUNCHER_IMPLEMENTATION = "osgi.featurelauncher"

The name of the implementation capability for the Feature specification.

160.8.4.5 public static final String FEATURE_LAUNCHER_SPECIFICATION_VERSION = "1.0"

The version of the implementation capability for the Feature specification.

160.8.4.6 public static final String FRAMEWORK_LAUNCHING_PROPERTIES = "framework-launching-properties"

The name for the FeatureExtension of Type.TEXT which defines the framework properties that should be used when launching the feature.

160.8.4.7 public static final String LAUNCH_FRAMEWORK = "launch-framework"

The name for the FeatureExtension which defines the framework that should be used to launch the feature. The extension must be of Type.ARTIFACTS and contain one or more ID entries corresponding to OSGi framework implementations. This extension must be processed even if it is Kind.OPTIONAL or Kind.TRANSIENT.

If more than one framework entry is provided then the list will be used as a priority order when determining the framework implementation to use. If none of the frameworks are present then an error is raised and launching will be aborted.

160.8.4.8 public static final String REMOTE_ARTIFACT_REPOSITORY_BEARER_TOKEN = "token"

The configuration property key used to set the bearer token when creating an ArtifactRepository using FeatureLauncher.createRepository(URI, Map)

160.8.4.9 public static final String REMOTE_ARTIFACT_REPOSITORY_NAME = "name"

The configuration property key used to set the repository name when creating an ArtifactRepository using FeatureLauncher.createRepository(URI, Map)

160.8.4.10 public static final String REMOTE_ARTIFACT_REPOSITORY_PASSWORD = "password"

The configuration property key used to set the repository password when creating an ArtifactRepository using FeatureLauncher.createRepository(URI, Map)

160.8.4.11 public static final String REMOTE_ARTIFACT_REPOSITORY_RELEASES_ENABLED = "release"

The configuration property key used to set that release versions are enabled for an ArtifactRepository using FeatureLauncher.createRepository(URI, Map)

160.8.4.12 public static final String REMOTE_ARTIFACT_REPOSITORY_SNAPSHOTS_ENABLED = "snapshot"

The configuration property key used to set that SNAPSHOT release versions are enabled for an ArtifactRepository using FeatureLauncher.createRepository(URI, Map)

160.8.4.13 public static final String REMOTE_ARTIFACT_REPOSITORY_TRUST_STORE = "truststore"

The configuration property key used to set the trust store to be used when accessing a remote ArtifactRepository using FeatureLauncher.createRepository(URI, Map)

160.8.4.14 public static final String REMOTE_ARTIFACT_REPOSITORY_TRUST_STORE_FORMAT = "truststoreFormat"

The configuration property key used to set the trust store format to be used when accessing a remote ArtifactRepository using FeatureLauncher.createRepository(URI, Map)

160.8.4.15 public static final String REMOTE_ARTIFACT_REPOSITORY_TRUST_STORE_PASSWORD = "truststorePassword"

The configuration property key used to set the trust store password to be used when accessing a remote ArtifactRepository using FeatureLauncher.createRepository(URI, Map)

160.8.4.16 public static final String REMOTE_ARTIFACT_REPOSITORY_USER = "user"

The configuration property key used to set the repository user when creating an ArtifactRepository using FeatureLauncher.createRepository(URI, Map)

160.8.5 public class LaunchException extends RuntimeException

A LaunchException is thrown by the FeatureLauncher if it is unable to:

- Locate or start an OSGi Framework instance
- Locate the installable bytes of any bundle in a Feature
- Install a bundle in the Feature
- Determine a value for a Feature variable that has no default value defined

160.8.5.1 public LaunchException(String message)

message

□ Create a LaunchException with the supplied error message

160.8.5.2 public LaunchException(String message, Throwable cause)

message

cause

□ Create a LaunchException with the supplied error message and cause

160.9 org.osgi.service.featurelauncher.annotation

Feature Annotations Package Version 1.0.

This package contains annotations that can be used to require the Feature Service implementation.

Bundles should not normally need to import this package as the annotations are only used at build-time.

160.9.1 Summary

- `RequireFeatureLauncherService` - This annotation can be used to require the Feature implementation.

160.9.2 @RequireFeatureLauncherService

This annotation can be used to require the Feature implementation. It can be used directly, or as a meta-annotation.

Retention CLASS

Target TYPE, PACKAGE

160.10 org.osgi.service.featurelauncher.decorator

Feature Launcher Decorator 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.featurelauncher.decorator; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.featurelauncher.decorator; version="[1.0,1.1)"
```

160.10.1 Summary

- AbandonOperationException - An AbandonOperationException is thrown by a FeatureDecorator or FeatureExtensionHandler if it needs to prevent the operation from continuing.
- BaseFeatureDecorationBuilder - The BaseFeatureDecorationBuilder is used to allow a user to customize a Feature.
- DecoratorBuilderFactory - The Builder Factory can be used to obtain builders for the various entities.
- FeatureDecorator - A FeatureDecorator is used to pre-process a Feature before it is installed or updated.
- FeatureDecorator.FeatureDecoratorBuilder - A reified builder which adds the ability to replace the extensions for the decorated feature
- FeatureExtensionHandler - A FeatureExtensionHandler is used to check and pre-process a Feature based on its FeatureExtensions before the feature is installed or updated.
- FeatureExtensionHandler.FeatureExtensionHandlerBuilder - A reified builder which does not permit extensions to be modified

160.10.2 public final class AbandonOperationException extends Exception

An AbandonOperationException is thrown by a FeatureDecorator or FeatureExtensionHandler if it needs to prevent the operation from continuing. This may be because of a problem detected in the feature, or because an extension has determined that the feature cannot be used in the current environment.

160.10.2.1 public AbandonOperationException(String message)

message

□ Create an AbandonOperationException with the supplied error message

160.10.2.2 public AbandonOperationException(String message, Throwable cause)

message

cause

□ Create an AbandonOperationException with the supplied error message and cause

160.10.3 public interface BaseFeatureDecorationBuilder<T extends BaseFeatureDecorationBuilder<T>>

<T> the type of the FeatureDecorator, used to parameterize the builder return values

The BaseFeatureDecorationBuilder is used to allow a user to customize a Feature. It is pre-populated with data from the original Feature, and calling any of the setXXX methods will replace the data in that section.

Provider Type Consumers of this API must not implement this type

160.10.3.1 public Feature build()

□ Build the Feature. Can only be called once. After calling this method the current builder instance cannot be used any more. and all methods will throw IllegalStateException.

Returns The Feature.

160.10.3.2 public T extends BaseFeatureDecorationBuilder<T> setBundles(List<FeatureBundle> bundles)

bundles The Bundles to add.

□ Replace the bundles in the Feature, discarding the current values.

Returns This builder.

160.10.3.3 public T extends BaseFeatureDecorationBuilder<T> setConfigurations(List<FeatureConfiguration> configs)

configs The Configurations to add.

□ Replace the Configurations in the Feature, discarding the current values.

Returns This builder.

160.10.3.4 public T extends BaseFeatureDecorationBuilder<T> setVariable(String key, Object defaultValue)

key The key.

defaultValue The default value.

□ Set or replace a single variable in the Feature. If a variable with the specified key already exists it is replaced with this one. Variable values are of type: String, Boolean or BigDecimal for numbers.

Returns This builder.

Throws IllegalArgumentException– if the value is of an invalid type.

160.10.3.5 public T extends BaseFeatureDecorationBuilder<T> setVariables(Map<String, Object> variables)

variables to be added.

□ Replace all the variables in the Feature, discarding the current values. Variable values are of type: String, Boolean or BigDecimal for numbers.

Returns This builder.

Throws IllegalArgumentException– if a value is of an invalid type.

160.10.4 public interface DecoratorBuilderFactory

The Builder Factory can be used to obtain builders for the various entities.

This is similar to BuilderFactory but does not permit the creation of FeatureBuilder instances.

Provider Type Consumers of this API must not implement this type

160.10.4.1 public FeatureArtifactBuilder newArtifactBuilder(ID id)

id The artifact ID for the artifact object being built.

□ Obtain a new builder for Artifact objects.

Returns The builder.

160.10.4.2 public FeatureBundleBuilder newBundleBuilder(ID id)

id The ID for the bundle object being built. If the ID has no type specified, a default type of @{code jar} is assumed.

□ Obtain a new builder for Bundle objects.

Returns The builder.

160.10.4.3 public FeatureConfigurationBuilder newConfigurationBuilder(String pid)

pid The persistent ID for the Configuration being built.

□ Obtain a new builder for Configuration objects.

Returns The builder.

160.10.4.4 public FeatureConfigurationBuilder newConfigurationBuilder(String factoryPid, String name)

factoryPid The factory persistent ID for the Configuration being built.

name The name of the configuration being built. The PID for the configuration will be the factoryPid + '~' + name

□ Obtain a new builder for Factory Configuration objects.

Returns The builder.

160.10.4.5 public FeatureExtensionBuilder newExtensionBuilder(String name, FeatureExtension.Type type, FeatureExtension.Kind kind)

name The extension name.

type The type of extension: JSON, Text or Artifacts.

kind The kind of extension: Mandatory, Optional or Transient.

□ Obtain a new builder for Feature objects.

Returns The builder.

160.10.5 public interface FeatureDecorator

A FeatureDecorator is used to pre-process a Feature before it is installed or updated. This allows the caller to potentially add or remove extensions, alter feature bundles, or edit configurations before the feature is installed or updated.

Note that a FeatureDecorator is *always called for all features* and may change the feature extensions, as well as bundles, configurations and variables.

160.10.5.1 public Feature decorate(Feature feature, FeatureDecorator.FeatureDecoratorBuilder decoratedFeatureBuilder, DecoratorBuilderFactory factory) throws AbandonOperationException

feature the feature to be installed or updated

decoratedFeatureBuilder a builder that can be used to produce a decorated feature with updated values

factory - a factory allowing users to create values for use with `decoratedFeatureBuilder`

□ Provides an opportunity to transform a feature before it is installed or updated

Returns The Feature to be installed. This must either be the same instance as `feature` or a new object created by calling `decoratedFeatureBuilder.build()`. Returning any other object is an error that will cause the install or update operation to fail

Throws `AbandonOperationException`– if the feature installation or update operation must not continue

160.10.6 public static interface FeatureDecorator.FeatureDecoratorBuilder extends BaseFeatureDecorationBuilder<FeatureDecorator.FeatureDecoratorBuilder>

A reified builder which adds the ability to replace the extensions for the decorated feature

Provider Type Consumers of this API must not implement this type

160.10.6.1 public FeatureDecorator.FeatureDecoratorBuilder setExtensions(List<FeatureExtension> extensions)

extensions The extensions to add.

□ Replace the extensions in the Feature, discarding the current values.

Returns This builder.

160.10.7 public interface FeatureExtensionHandler

A FeatureExtensionHandler is used to check and pre-process a Feature based on its FeatureExtensions before the feature is installed or updated. This allows the caller to potentially alter feature bundles, or edit configurations before the feature is installed or updated.

Note that a FeatureExtensionHandler is *only called for features with a matching extension* called and may only change the feature bundles or feature configurations.

160.10.7.1 public Feature handle(Feature feature, FeatureExtension extension, FeatureExtensionHandler.FeatureExtensionHandlerBuilder decoratedFeatureBuilder, DecoratorBuilderFactory factory) throws AbandonOperationException

feature the feature to be installed or updated

extension the feature extension which caused this handler to be called

decoratedFeatureBuilder a builder that can be used to produce a decorated feature with updated values

factory - a factory allowing users to create values for use with `decoratedFeatureBuilder`

□ Provides an opportunity to transform a feature before it is installed or updated

Returns The Feature to be installed. This must either be the same instance as `feature` or a new object created by calling `decoratedFeatureBuilder.build()`. Returning any other object is an error that will cause the install or update operation to fail

Throws `AbandonOperationException`– if the feature installation or update operation must not continue

160.10.8 public static interface FeatureExtensionHandler.FeatureExtensionHandlerBuilder extends BaseFeatureDecorationBuilder<FeatureExtensionHandler.FeatureExtensionHandlerBuilder

A reified builder which does not permit extensions to be modified

Provider Type Consumers of this API must not implement this type

160.11 org.osgi.service.featurelauncher.repository

Feature Launcher Repository 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.featurelauncher.repository; version="[1.0,2.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.featurelauncher.repository; version="[1.0,1.1)"`

160.11.1 Summary

- `ArtifactRepository` - An ArtifactRepository is used to get hold of the bytes used to install an artifact.
- `ArtifactRepositoryFactory` - A ArtifactRepositoryFactory is used to create implementations of ArtifactRepository for one of the built in repository types:
  - Local File System
  - HTTP repository

160.11.2 public interface ArtifactRepository

An ArtifactRepository is used to get hold of the bytes used to install an artifact. Users of this specification may provide their own implementations for use when installing feature artifacts. Instances must be Thread Safe.

Concurrency Thread-safe

160.11.2.1 public InputStream getArtifact(ID id)

id the id of the artifact

□ Get a stream to the bytes of an artifact

Returns an InputStream containing the bytes of the artifact or `null` if this repository does not have access to the bytes

160.11.3 public interface ArtifactRepositoryFactory

A ArtifactRepositoryFactory is used to create implementations of ArtifactRepository for one of the built in repository types:

- Local File System
- HTTP repository

Provider Type Consumers of this API must not implement this type

160.11.3.1 public ArtifactRepository createRepository(Path path)

path a path to the root of a Maven Repository Layout containing installable artifacts

□ Create an ArtifactRepository using the local file system

Returns an ArtifactRepository using the local file system

Throws `IllegalArgumentException`– if the path does not exist, or exists and is not a directory

`NullPointerException`– if the path is null

160.11.3.2 public ArtifactRepository createRepository(URI uri, Map<String, Object> props)

uri the URI for the repository. The `http`, `https` and `file` schemes must be supported by all implementations.

props the configuration properties for the remote repository. See FeatureLauncherConstants for standard property names

□ Create an ArtifactRepository using a remote Maven repository.

Returns an ArtifactRepository using the local file system

Throws `IllegalArgumentException`– if the uri scheme is not supported by this implementation

`NullPointerException`– if the path is null

160.12 org.osgi.service.featurelauncher.runtime

Feature Launcher 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.featurelauncher.runtime; version="[1.0,2.0)"`

Example import for providers implementing the API in this package:

`Import-Package: org.osgi.service.featurelauncher.runtime; version="[1.0,1.1)"`

160.12.1 Summary

- `FeatureRuntime` - The Feature runtime service allows features to be installed and removed dynamically at runtime.
- `FeatureRuntime.InstallOperationBuilder` - The OperationBuilder for a FeatureRuntime.install(Feature) operation.
- `FeatureRuntime.OperationBuilder` - An OperationBuilder is used to configure the installation or update of a Feature by the FeatureRuntime.
- `FeatureRuntime.UpdateOperationBuilder` - The OperationBuilder for a FeatureRuntime.install(Feature) operation.
- `FeatureRuntimeConstants` - Defines standard constants for the Feature Runtime.
- `FeatureRuntimeException` - A FeatureRuntimeException is thrown by the FeatureRuntime if it is unable to:
  - Locate the installable bytes of any bundle in a Feature
  - Install a bundle in the Feature
  - Determine a value for a Feature variable that has no default value defined
  - Successfully merge a feature with the existing environment
- `InstalledBundle` - An InstalledBundle represents a configuration that has been installed as a result of one or more feature installations.
- `InstalledConfiguration` - An InstalledConfiguration represents a configuration that has been installed as a result of one or more feature installations.
- `InstalledFeature` - An InstalledFeature represents the current state of a feature installed by the FeatureRuntime.
- `MergeOperationType` - An MergeOperationType represents the type of operation that is in flight

- `RuntimeBundleMerge` - Merge operations occur when two or more features reference the same (or similar) items to be installed.
- `RuntimeBundleMerge.BundleMapping` - A BundleMapping is used to define that a bundle should be (or remain) installed and which Features should own it
- `RuntimeBundleMerge.FeatureBundleDefinition` - A FeatureBundleDefinition is used to show which FeatureBundle(s) are being merged, and the Feature that they relate to.
- `RuntimeConfigurationMerge` - Merge operations occur when two or more features reference the same (or similar) items to be installed.
- `RuntimeConfigurationMerge.FeatureConfigurationDefinition` - A FeatureConfigurationDefinition is used to show which FeatureConfiguration(s) are being merged, and the Feature that they relate to.
- `RuntimeMerges` - Merge operations occur when two or more features reference the same (or similar) items to be installed.

160.12.2 public interface FeatureRuntime extends ArtifactRepositoryFactory

The Feature runtime service allows features to be installed and removed dynamically at runtime.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

160.12.2.1 public Map<String, ArtifactRepository> getDefaultRepositories()

□ Get the default repositories for the FeatureRuntime service. These are the repositories which are used by default when installing or updating features.

This method can be used to select a subset of the default repositories when using an OperationBuilder, or to query for instances manually.

Returns the default repositories

160.12.2.2 public List<InstalledFeature> getInstalledFeatures()

□ Get the features that have been installed by the FeatureRuntime service

Returns a list of installed features

160.12.2.3 public FeatureRuntime.InstallOperationBuilder install(Feature feature)

feature the feature to launch

□ Install a feature into the runtime

Returns An OperationBuilder that can be used to set up the installation of this feature

Throws `LaunchException` – if installation fails

160.12.2.4 public FeatureRuntime.InstallOperationBuilder install(Reader jsonReader)

jsonReader a Reader for the input Feature JSON

□ Install a feature into the runtime based on the supplied feature JSON

Returns An installedFeature representing the results of installing this feature

Throws `LaunchException` – if installation fails

160.12.2.5 public void remove(ID featureId)

featureId the feature id

□ Remove an installed feature

160.12.2.6 public FeatureRuntime.UpdateOperationBuilder update(ID featureId, Feature feature)

featureId the id of the feature to update

feature the feature to launch

□ Update a feature in the runtime

Returns An installedFeature representing the results of updating this feature

160.12.2.7 public FeatureRuntime.UpdateOperationBuilder update(ID featureId, Reader jsonReader)

featureId the id of the feature to update

jsonReader a Reader for the input Feature JSON

□ Update a feature in the runtime based on the supplied feature JSON

Returns An installedFeature representing the results of updating this feature

160.12.3 public static interface FeatureRuntime.InstallOperationBuilder extends FeatureRuntime.OperationBuilder<FeatureRuntime.InstallOperationBuilder>

The OperationBuilder for a FeatureRuntime.install(Feature) operation. Instances are not thread safe and must not be shared.

160.12.3.1 public InstalledFeature install()

□ An alias for the complete() method

Returns the installed feature

160.12.4 public static interface FeatureRuntime.OperationBuilder<T extends FeatureRuntime.OperationBuilder<T>>

<T>

An OperationBuilder is used to configure the installation or update of a Feature by the FeatureRuntime. Instances are not thread safe and must not be shared.

Once the complete() method is called the operation will be run by the feature runtime and the operation builder will be invalidated, with all methods throwing IllegalStateException.

160.12.4.1 public T extends FeatureRuntime.OperationBuilder<T> addRepository(String name, ArtifactRepository repository)

name the name to use for this repository

repository the repository

□ Add an ArtifactRepository for use by this OperationBuilder instance. If an ArtifactRepository is already set for the given name then it will be replaced. Passing a null ArtifactRepository will remove the repository from this operation.

Returns this

Throws IllegalStateException– if the builder has been completed

160.12.4.2 public InstalledFeature complete() throws FeatureRuntimeException

□ Complete the operation by installing or updating the feature

Returns An InstalledFeature representing the result of the operation

Throws FeatureRuntimeException– if an error occurs

IllegalStateException– if the builder has been completed already

160.12.4.3 public T extends FeatureRuntime.OperationBuilder<T> useDefaultRepositories(boolean include)

include

□ Include the default repositories when completing this operation. This value defaults to true. If any ArtifactRepository added using addRepository(String, ArtifactRepository) has the same name as a default repository then the added repository will override the default repository.

Returns this

Throws IllegalStateException– if the builder has been completed

160.12.4.4 public T extends FeatureRuntime.OperationBuilder<T> withBundleMerge(RuntimeBundleMerge merge)

merge

□ Use The supplied RuntimeBundleMerge to resolve any bundle merge operations that are required to complete the operation

Returns this

160.12.4.5 public T extends FeatureRuntime.OperationBuilder<T> withConfigurationMerge(RuntimeConfigurationMerge merge)

merge

□ Use The supplied RuntimeConfigurationMerge to resolve any configuration merge operations that are required to complete the operation

Returns this

160.12.4.6 public T extends FeatureRuntime.OperationBuilder<T> withDecorator(FeatureDecorator decorator)

decorator the decorator to add

□ Add a FeatureDecorator to this OperationBuilder that will be used to decorate the feature. If called multiple times then the supplied decorators will be called in the same order that they were added to this builder.

Returns this

Throws NullPointerException– if the decorator is null

IllegalStateException– if the builder has been launched

160.12.4.7 public T extends FeatureRuntime.OperationBuilder<T> withExtensionHandler(String extensionName, FeatureExtensionHandler extensionHandler)

extensionName the name of the extension to handle

extensionHandler the extensionHandler to add

□ Add a FeatureExtensionHandler to this OperationBuilder that will be used to process the named FeatureExtension if it is found in the Feature. If called multiple times for the same extensionName then later calls will replace the extensionHandler to be used.

Returns this

Throws NullPointerException– if the extension name or decorator is null

IllegalStateException– if the builder has been launched

160.12.4.8 public T extends FeatureRuntime.OperationBuilder<T> withVariables(Map<String, Object> variables)

variables the variable placeholder overrides for this launch

□ Configure this OperationBuilder with the supplied variables.

Returns this

Throws IllegalStateException– if the builder has been completed

160.12.5 public static interface FeatureRuntime.UpdateOperationBuilder extends FeatureRuntime.OperationBuilder<FeatureRuntime.UpdateOperationBuilder>

The OperationBuilder for a FeatureRuntime.install(Feature) operation. Instances are not thread safe and must not be shared.

160.12.5.1 public InstalledFeature update()

□ An alias for the complete() method

Returns the updated feature

160.12.6 public final class FeatureRuntimeConstants

Defines standard constants for the Feature Runtime.

160.12.6.1 public static final String EXTERNAL_FEATURE_ID = "org.osgi.service.featurelauncher:external:1.0.0"

The ID of the virtual *external* feature representing ownership of a bundle or configuration that was deployed by another management agent.

160.12.7 public class FeatureRuntimeException extends RuntimeException

A FeatureRuntimeException is thrown by the FeatureRuntime if it is unable to:

- Locate the installable bytes of any bundle in a Feature
- Install a bundle in the Feature
- Determine a value for a Feature variable that has no default value defined
- Successfully merge a feature with the existing environment

160.12.7.1 public FeatureRuntimeException(String message)

message

□ Create a LaunchException with the supplied error message

160.12.7.2 public FeatureRuntimeException(String message, Throwable cause)

message

cause

□ Create a LaunchException with the supplied error message and cause

160.12.8 public interface InstalledBundle

An InstalledBundle represents a configuration that has been installed as a result of one or more feature installations.

This type is a snapshot and represents the state of the runtime when it was created. It may become out of date if additional features are installed or removed.

Provider Type Consumers of this API must not implement this type

160.12.8.1 public Collection<ID> getAliases()

□ Get any known IDs which correspond to the same bundle

Returns an immutable collection of aliases for this bundle. Always includes the id returned by getBundleId()

160.12.8.2 public Bundle getBundle()

□ The actual bundle installed in the framework

Returns the Bundle installed for this getBundleId()

160.12.8.3 public ID getBundleId()

□ Get the ID of the bundle that has been installed

Returns the id of the bundle that was installed

160.12.8.4 public List<ID> getOwningFeatures()

□ The features responsible for this bundle being installed, in installation order

Returns A list of Feature IDs

160.12.8.5 public int getStartLevel()

□ The start level for this bundle

Returns the start level

160.12.9 public interface InstalledConfiguration

An InstalledConfiguration represents a configuration that has been installed as a result of one or more feature installations.

This type is a snapshot and represents the state of the runtime when it was created. It may become out of date if additional features are installed or removed.

Provider Type Consumers of this API must not implement this type

160.12.9.1 public Optional<String> getFactoryPid()

□ Get the factory PID of the configuration

Returns the factory PID of this configuration, or an empty optional if this is not a factory configuration

160.12.9.2 public List<ID> getOwningFeatures()

□ The features responsible for creating this configuration, in installation order

Returns A list of Feature IDs

160.12.9.3 public String getPid()

□ Get the PID of the configuration

Returns the full PID of this configuration

160.12.9.4 public Map<String, Object> getProperties()

□ Get the merged configuration properties for this configuration

Returns The properties associated with this configuration, may be null if the configuration should not be created

160.12.10 public interface InstalledFeature

An InstalledFeature represents the current state of a feature installed by the FeatureRuntime.

This type is a snapshot and represents the state of the runtime when it was created. It may become out of date if additional features are installed or removed.

Provider Type Consumers of this API must not implement this type

160.12.10.1 public ID getFeatureId()

Returns The ID of the installed feature

160.12.10.2 public List<InstalledBundle> getInstalledBundles()

□ Get the bundles installed by this feature

Returns A List of the bundles installed by this feature, in the order they were declared by the feature

160.12.10.3 public List<InstalledConfiguration> getInstalledConfigurations()

□ Get the configurations installed by this feature

Returns A List of the configurations installed by this feature, in the order they were declared by the feature

160.12.10.4 public boolean isInitialLaunch()

□ Is this a feature installed by FeatureLauncher

Returns `true` If this feature was installed as part of a FeatureLauncher launch operation. `false` if it was installed by the FeatureRuntime

160.12.11 enum MergeOperationType

An MergeOperationType represents the type of operation that is in flight

160.12.11.1 INSTALL

An install operation adds a feature to the runtime

160.12.11.2 UPDATE

An update operation replaces one feature with another

160.12.11.3 REMOVE

A remove operation removes a feature from the runtime

160.12.11.4 public static MergeOperationType valueOf(String name)

160.12.11.5 public static MergeOperationType[] values()

160.12.12 public interface RuntimeBundleMerge

Merge operations occur when two or more features reference the same (or similar) items to be installed.

The purpose of a RuntimeBundleMerge is to resolve possible conflicts between FeatureBundle entries and determine which bundle(s) should be installed as a result.

Merge operations happen in one of three scenarios, indicated by the MergeOperationType:

- INSTALL - a feature is being installed
- UPDATE - a feature is being updated
- REMOVE - a feature is being removed

When any merge operation occurs the merge function will be provided with the Feature being operated upon, the FeatureBundle which needs to be merged, a List of the InstalledBundles representing the currently installed bundles applicable to the merge, and a List of FeatureBundleDefinitions representing the FeatureBundles and installed features participating in the merge. All Installed Bundle and Feature Bundle objects will have the same group id and artifact id.

If an UPDATE or REMOVE operation is underway then the Feature being updated or removed will already have been removed from any Installed Bundles and from the list of Feature Bundle Definitions. For an UPDATE this may result in one or more Installed Bundles having an empty list of owning features, and the list of existing installed Feature Bundle Definitions being empty.

The returned result from the merge function must be a full mapping of installed Bundle IDs to Lists of owning Feature ids. This is returned as a Stream of BundleMappings. The combined BundleMapping.owningFeatures in the stream must contain all of the Feature ids from the list of Feature Bundle Definitions, and in the case of an INSTALL or UPDATE operation also the Feature being operated upon. The entries in the returned stream must only contain BundleMapping.bundleIds from the list of Installed Bundles, and in the case of an INSTALL or UPDATE operation the Feature Bundle being merged

.

It is an error for any value in the returned stream to be null, or to have fields set to null or an empty list. In the case of a REMOVE operation it is an error to include the Feature id being operated upon in the returned Bundle Mappings.

160.12.12.1 public Stream<RuntimeBundleMerge.BundleMapping> mergeBundle(MergeOperationType operation, Feature feature, FeatureBundle toMerge, Collection<InstalledBundle> installedBundles, List<RuntimeBundleMerge.FeatureBundleDefinition> existingFeatureBundles)

operation - the type of the operation triggering the merge.

feature The feature being operated upon

toMerge The FeatureBundle in feature that requires merging

installedBundles A read only list of bundles that have been installed as part of previous installations. This list will always contain at least one entry.

existingFeatureBundles An immutable list of FeatureBundleDefinitions which are part of this merge operation. The entries are in the same order as the Features were installed.

This list may be empty in the case of an UPDATE operation. Note that multiple Feature Bundle Definitions may refer to the same bundle ID, or aliases of a single InstalledBundle.

□ Calculate the bundles that should be installed at the end of a given operation.

Bundle Merge operations occur when two or more features reference a bundle with the same group id and artifact id, and the purpose of this method is to identify which bundles should be/remain installed, and which features they should be owned by.

The returned result from the merge function must be a full mapping of installed Bundle IDs to Lists of owning Features. It is an error for the stream to contain a BundleMapping.bundleId which is not the ID of an entry in in the installedBundle list or, in the case of an INSTALL or UPDATE operation, the ID of the toMerge Feature Bundle.

The combined BundleMapping.owningFeatures in the stream must contain all of the Features from the List of Feature Bundle Definitions, and in the case of an INSTALL or UPDATE operation also the Feature being operated upon. In the case of a REMOVE operation it is an error to include the Feature being operated upon in the returned stream.

It is an error for any entry in the returned stream to be, or contain, null or an empty list.

Returns An unordered Stream of BundleMapping entries linking a bundle id to List of owning Feature ids. Each Bundle Mapping represents a bundle that should be installed as a result of this operation. Note that every Feature id *must* appear in the combined BundleMapping.owningFeatures and that the BundleMapping.bundleId may only contain IDs from toMerge or one of the keys from the installed-

Bundles list. If two BundleMapping entries use the same bundle id, or alias, then this is not an error and these mappings will be combined by the implementation.

160.12.13 public static final class RuntimeBundleMerge.BundleMapping

A BundleMapping is used to define that a bundle should be (or remain) installed and which Features should own it

160.12.13.1 public final ID bundleId

The ID of the bundle to be installed

160.12.13.2 public final List<ID> owningFeatures

The List of features which own the bundle

160.12.13.3 public BundleMapping(ID bundleId, List<ID> owningFeatures)

bundleId

owningFeatures

□ Create a new BundleMapping

160.12.14 public static interface RuntimeBundleMerge.FeatureBundleDefinition

A FeatureBundleDefinition is used to show which FeatureBundle(s) are being merged, and the Feature that they relate to.

160.12.14.1 public Feature getFeature()

Returns The Feature containing the FeatureBundle

160.12.14.2 public FeatureBundle getFeatureBundle()

Returns The FeatureBundle being merged

160.12.15 public interface RuntimeConfigurationMerge

Merge operations occur when two or more features reference the same (or similar) items to be installed.

The purpose of a RuntimeConfigurationMerge is to resolve possible conflicts between FeatureConfiguration entries and determine what configuration should be created as a result.

Merge operations happen in one of three scenarios, indicated by the MergeOperationType:

- INSTALL - a feature is being installed
- UPDATE - a feature is being updated
- REMOVE - a feature is being removed

When any merge operation occurs the merge function will be provided with the Feature being operated upon, the FeatureConfiguration which needs to be merged, the InstalledConfiguration representing the current configuration, and a list of FeatureConfigurationDefinitions representing the installed features participating in the merge. All Feature Configurations will have the same PID.

If an UPDATE or REMOVE operation is underway then the Feature being updated or removed will already have been removed from the Installed Configuration and the list of existing Feature Configuration Definitions. For an UPDATE this may result in the InstalledConfiguration.getOwningFeatures() being an empty list, and the map of existing installed Feature Configurations being empty.

The returned result from the merge function is a map of configuration properties that should be used to complete the operation. This may be null if the configuration should be deleted.

160.12.15.1 public Map<String, Object> mergeConfiguration(MergeOperationType operation, Feature feature, FeatureConfiguration toMerge, InstalledConfiguration configuration, List<RuntimeConfigurationMerge.FeatureConfigurationDefinition> existingFeatureConfigurations)

operation - the type of the operation triggering the merge.

feature The feature being operated upon

toMerge The FeatureConfiguration in feature that requires merging

configuration The existing configuration that has been installed as part of previous installations. This will represent a configuration with the same PID as toMerge.

Note that this value will be null if the configuration does not exist to differentiate it from an empty configuration dictionary

existingFeatureConfigurations An immutable list of FeatureConfigurationDefinitions which are part of this merge operation. The entries are in the same order as the Features were installed.

This list may be empty in the case of an UPDATE operation. Note that all Feature Configuration Definitions will refer to the same PID, and this will match the PID of toMerge. An immutable map of existing Feature Configurations which are part of this merge operation. The keys in the map are the Feature Configurations involved in the merge and the values are the Features which contain the Feature Configuration.

□ Calculate the configuration that should be used at the end of a given operation.

Configuration merge operations occur when two or more features define the same configuration, where configuration identity is determined by the PID of the configuration. The purpose of this function is to determine what configuration properties should be used after the merge has finished.

Returns A map of configuration properties to use. Returning null indicates that the configuration should be deleted.

160.12.16 public static interface RuntimeConfigurationMerge.FeatureConfigurationDefinition

A FeatureConfigurationDefinition is used to show which FeatureConfiguration(s) are being merged, and the Feature that they relate to.

160.12.16.1 public Feature getFeature()

Returns The Feature containing the FeatureConfiguration

160.12.16.2 public FeatureConfiguration getFeatureConfiguration()

Returns The FeatureBundle being merged

160.12.17 public final class RuntimeMerges

Merge operations occur when two or more features reference the same (or similar) items to be installed.

The purpose of a RuntimeMerges is to provide common merge strategies in an easy to construct way.

160.12.17.1 public RuntimeMerges()

160.12.17.2 public static Version getOSGiVersion(ID id)

id

□ Attempts to turn the version String from an ID into an OSGi version

Note that this parsing is more lenient than Version.parseVersion(String). It treats the first three segments separated by . characters as possible integers. If they are integers then they represent the major, minor and micro segments of an OSGi version. If any non-numeric segments are encountered, or the end of the string, then the remaining version segments are 0. Any remaining content from the input version string is used as the qualifier.

Returns An OSGi version which attempts to replicate the version from the ID

160.12.17.3 public static RuntimeBundleMerge preferExistingBundles()

□ The preferExistingBundles() merge strategy tries to reduce the number of new installations by applying semantic versioning rules. The new bundle is only installed if it has:

- A different major version from all installed bundles
- A higher minor version than all other installed bundles with the same major version

Returns the prefer existing merge strategy

160.12.17.4 public static RuntimeConfigurationMerge replaceExistingProperties()

□ The replaceExistingProperties() merge strategy simply replaces any existing configuration values with the new values from the new FeatureConfiguration.

Removal is more complex and relies on the fact that the `existingFeatureConfigurations` are in installation order. This means that we can descend the list looking for the previous configuration properties and apply them

Returns the replace existing merge strategy

160.13 References

[1] *The Maven 2 Repository Layout*
https://maven.apache.org/repository/layout.html#maven2-repository-layout

[2] *The Data URI scheme*
https://en.wikipedia.org/wiki/Data_URI_scheme

End Of Document