OSGi Working Group

OSGi Compendium

Release 8.1
December 2022

Table of Contents

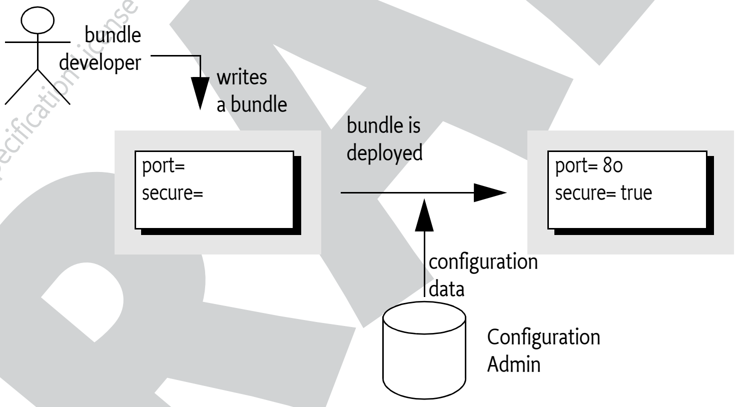
104 Configuration Admin Service Specification

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.

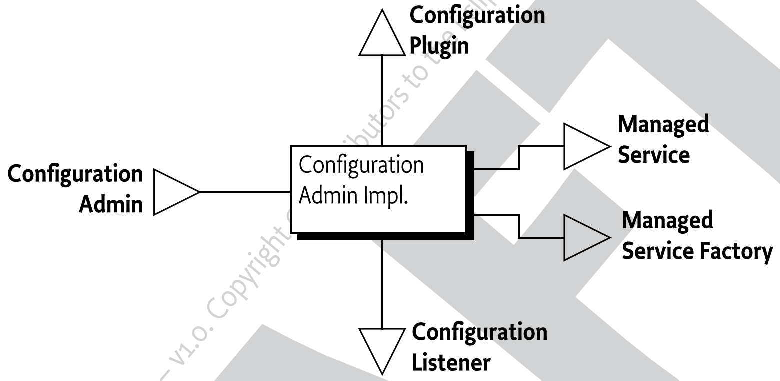
  Trade-offs in simplicity should be made at the expense of the bundle implementing the Configuration Admin service and in favor of bundles that need configuration information. The reason for this choice is that normal bundles will outnumber Configuration Admin bundles.
- *Regions* - It should be possible 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.

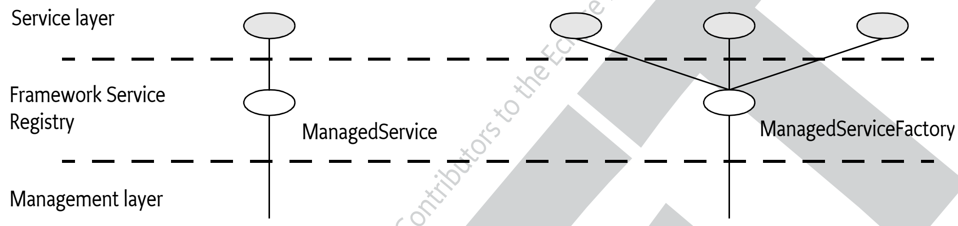
104.2 Configuration Targets

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 11.

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 8 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 24 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 ??? 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 27. 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 9.
- 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 29.

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

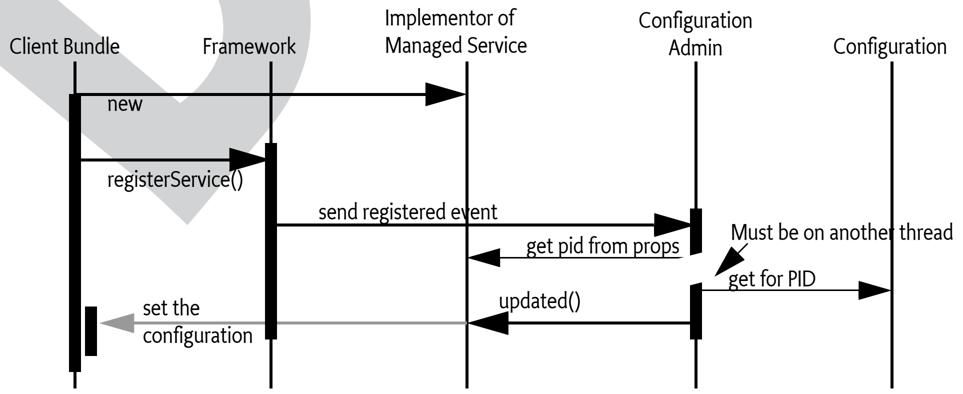
- The configuration stored for the registered PID must be retrieved. If there is a Configuration object for this PID and the configuration is visible for the associated bundle then it is sent to the Managed Service with updated(Dictionary).
- If a Managed Service is registered and no configuration information 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 11.

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

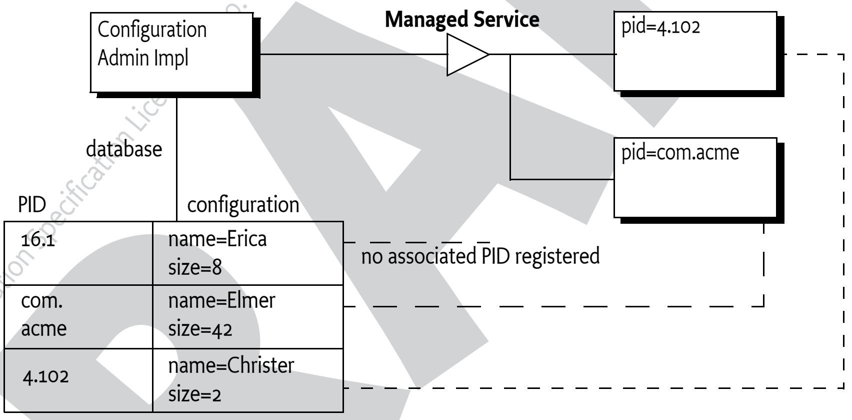
When a Managed Service is registered, the default properties may be visible in the service registry for a short period before they are replaced by the properties of the actual configuration dictionary. Care should be taken that this visibility does not cause race conditions for other bundles.

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.

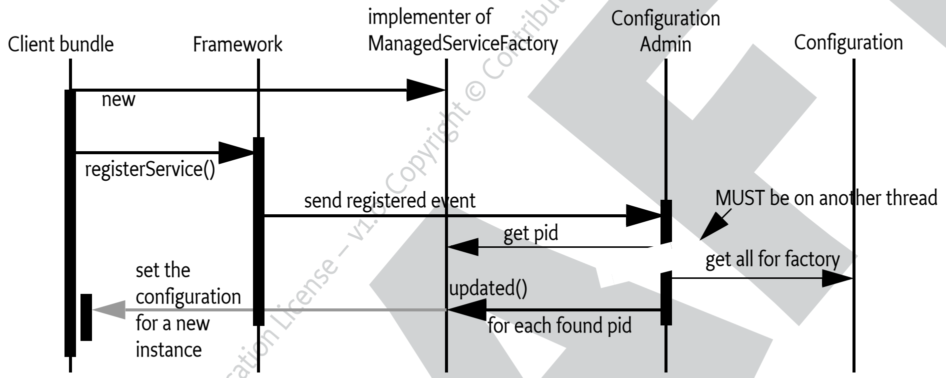
When the Configuration Admin service detects the registration of a Managed Service Factory, it must find all visible configuration dictionaries for this factory and must then sequentially call `ManagedServiceFactory.updated(String,Dictionary)` for each configuration dictionary. The first argument is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

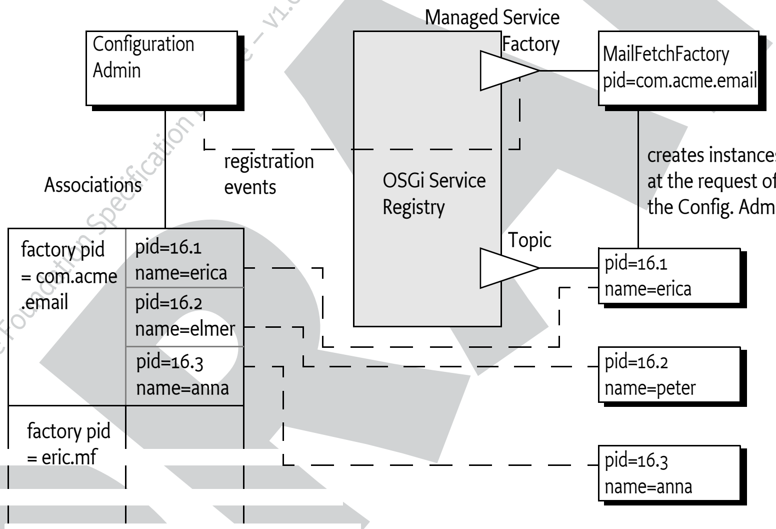
104.6.4 Managed Service Factory Example

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 11.

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 9.

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 11. 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 24. 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 11, 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 ???.

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.

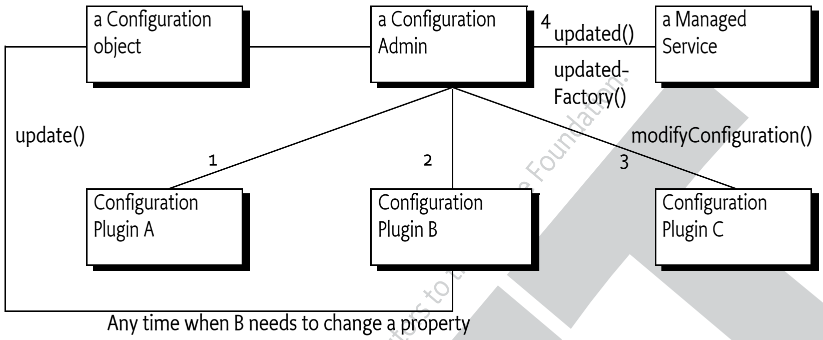
The ConfigurationPlugin interface has only one method: modifyConfiguration(ServiceReference,Dictionary). This method inspects or modifies configuration data.

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 13.

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.

The intend of this integration is that multiple events are collapsed into one as updating a Managed Service or Managed Service Factory might result in services being updated or unregistered what can trigger other service changes what might be a costly operation and therefore result in an unwanted intermediate states until the system has setled again. Collapsing here means, that multiple updates inside a coordination are combined into exactly one update that reflects the final outcome, the following list gives some examples for clarification:

- If a configuration exists and it has currently the values {key: value} and a coordination starts where it is first updated to {key: something else} and later again to {key: value}, then after the coordination terminates one update is delivered with {key: value} even though the configuration has not changed, see *Updating a Configuration* on page 23.

  If no configuration exists and a coordination starts where it is first updated to {key: something else} and later is deleted, then after the coordination terminates one update is delivered with null even though the Managed Service or Managed Service Factory has not changed this is a special case of the blindly update case described in *Updating a Configuration* on page 23.

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 ???.

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 ???.

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

End Of Document