RFC-227 Configuration Admin Updates

Draft

47 Pages

Abstract

10 point Arial Centered.

Updates to Configuration Admin for R7.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

The OSGi Alliance hereby grants you a limited copyright license to copy and display this document (the “Distribution”) in any medium without fee or royalty. This Distribution license is exclusively for the purpose of reviewing and providing feedback to the OSGi Alliance. You agree not to modify the Distribution in any way and further agree to not participate in any way in the making of derivative works thereof, other than as a necessary result of reviewing and providing feedback to the Distribution. You also agree to cause this notice, along with the accompanying consent, to be included on all copies (or portions thereof) of the Distribution. The OSGi Alliance also grants you a perpetual, non-exclusive, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights, to create and/or distribute an implementation of the Distribution that: (i) fully implements the Distribution including all its required interfaces and functionality; (ii) does not modify, subset, superset or otherwise extend the OSGi Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the OSGi Name Space other than those required and authorized by the Distribution. An implementation that does not satisfy limitations (i)-(ii) is not considered an implementation of the Distribution, does not receive the benefits of this license, and must not be described as an implementation of the Distribution. "OSGi Name Space" shall mean the public class or interface declarations whose names begin with "org.osgi" or any recognized successors or replacements thereof. The OSGi Alliance expressly reserves all rights not granted pursuant to these limited copyright licenses including termination of the license at will at any time.

EXCEPT FOR THE LIMITED COPYRIGHT LICENSES GRANTED ABOVE, THE OSGi ALLIANCE DOES NOT GRANT, EITHER EXPRESSLY OR IMPLIEDLY, A LICENSE TO ANY INTELLECTUAL PROPERTY IT, OR ANY THIRD PARTIES, OWN OR CONTROL. Title to the copyright in the Distribution will at all times remain with the OSGi Alliance. The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted therein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

THE DISTRIBUTION IS PROVIDED "AS IS," AND THE OSGi ALLIANCE (INCLUDING ANY THIRD PARTIES THAT HAVE CONTRIBUTED TO THE DISTRIBUTION) MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DISTRIBUTION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.
NEITHER THE OSGi ALLIANCE NOR ANY THIRD PARTY WILL BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE DISTRIBUTION.

Implementation of certain elements of this Distribution may be subject to third party intellectual property rights, including without limitation, patent rights (such a third party may or may not be a member of the OSGi Alliance). The OSGi Alliance is not responsible and shall not be held responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

The Distribution is a draft. As a result, the final product may change substantially by the time of final publication, and you are cautioned against relying on the content of this Distribution. You are encouraged to update any implementation of the Distribution if and when such Distribution becomes a final specification.

The OSGi Alliance is willing to receive input, suggestions and other feedback (“Feedback”) on the Distribution. By providing such Feedback to the OSGi Alliance, you grant to the OSGi Alliance and all its Members a non-exclusive, non-transferable,

worldwide, perpetual, irrevocable, royalty-free copyright license to copy, publish, license, modify, sublicense or otherwise distribute and exploit your Feedback for any purpose. Likewise, if incorporation of your Feedback would cause an implementation of the Distribution, including as it may be modified, amended, or published at any point in the future ("Future Specification"), to necessarily infringe a patent or patent application that you own or control, you hereby commit to grant to all implementers of such Distribution or Future Specification an irrevocable, worldwide, sublicenseable, royalty free license under such patent or patent application to make, have made, use, sell, offer for sale, import and export products or services that implement such Distribution or Future Specification. You warrant that (a) to the best of your knowledge you have the right to provide this Feedback, and if you are providing Feedback on behalf of a company, you have the rights to provide Feedback on behalf of your company; (b) the Feedback is not confidential to you and does not violate the copyright or trade secret interests of another; and (c) to the best of your knowledge, use of the Feedback would not cause an implementation of the Distribution or a Future Specification to necessarily infringe any third-party patent or patent application known to you. You also acknowledge that the OSGi Alliance is not required to incorporate your Feedback into any version of the Distribution or a Future Specification.

I HEREBY ACKNOWLEDGE AND AGREE TO THE TERMS AND CONDITIONS DELINEATED ABOVE.

0.2 Trademarks

OSGi™ is a trademark, registered trademark, or service mark of the OSGi Alliance in the US and other countries. Java is a trademark, registered trademark, or service mark of Oracle Corporation in the US and other countries. All other trademarks, registered trademarks, or service marks used in this document are the property of their respective owners and are hereby recognized.

0.3 Feedback

This document can be downloaded from the OSGi Alliance design repository at https://github.com/osgi/design The public can provide feedback about this document by opening a bug at https://www.osgi.org/bugzilla/.

0.4 Table of Contents

0 Document Information **2**
- 0.1 License 2
- 0.2 Trademarks 3
- 0.3 Feedback 3
- 0.4 Table of Contents 3
- 0.5 Terminology and Document Conventions 4
- 0.6 Revision History 4

1 Introduction **5**

2 Application Domain **5**

3 Problem Description **5**
- 3.1 RFC 218 Configurator 5
- 3.2 Support ConfigurationPlugin-like behavior for non-MS/MSF users (Bug 2908) 5

4 Requirements **6**
- 4.1 Configuration Admin 6

5 Technical Solution **6**
- 5.1 PID Handling of Factory Configurations 6
- 5.2 Locking Configuration Records 7
  - 5.2.1 Security impacts 8
- 5.3 Improving Configuration Updates 8
- 5.4 Capabilities 9

5.5 Improved ConfigurationPlugin Support....9
5.6 Clarifications....9
5.6.1 ConfigurationPlugin Ranking....10
5.6.2 Modifications by ConfigurationPlugin....10
5.6.3 ManagedService(Factory) without PID property....10
5.7 Coordinations....10

6 Data Transfer Objects....10

7 Javadoc....11

8 Considered Alternatives....46

9 Security Considerations....46

10 Document Support....46
10.1 References....46
10.2 Author's Address....46
10.3 Acronyms and Abbreviations....46
10.4 End of Document....47

0.5 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 10.1.

```
Source code is shown in this typeface.
```

0.6 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial		*Initial Version* *Carsten Ziegeler, Adobe <cziegele@adobe.com>*
0.1	*28-JUN-2016*	*Updates from Darmstadt F2F (ConfigurationPlugin)* *Carsten Ziegeler, Adobe <cziegele@adobe.com>*
0.2	*07-SEP-2016*	*Merge in changes from RFC-228 based on results from issue #2911* *Carsten Ziegeler, Adobe <cziegele@adobe.com>*
0.3	*29-NOV-2016*	*Apply suggestions from Hursley F2F* *David Bosschaert, Adobe <bosschae@adobe.com>*

1 Introduction

This RFC collects a numbers of requested enhancements to Configuration Admin Service that were suggested after Release 6 design work was completed. Some of the requirements are extracted from RFC 218 Configurator.

2 Application Domain

The Configuration Admin Specification was last change as part of Release 5. From that specification:

> 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.

3 Problem Description

3.1 RFC 218 Configurator

RFC 218 defines the Configurator, an extender that allows the storage of configuration data in a bundle. Some of the requirements from that RFC can best be realized with new features/requirements for the Configuration Admin Service.

3.2 Support ConfigurationPlugin-like behavior for non-MS/MSF users (Bug 2908)

The ConfigurationPlugin defined in the Configuration Admin Service Specification is invoked before a configuration is delivered to a ManagedService(Factory). The plugin is able to modify the configuration properties. There are several use cases like replacing configuration values with values provided through system properties (or similar mechanism), decode values, or provide additional properties.

While this works with registering a ManagedService(Factory), component containers like Declarative Services or Blueprint are not required to register ManagedServices on behalf of their components. Therefore whether the ConfigurationPlugin mechanism works with such containers is implementation dependent and not specified.

4 Requirements

4.1 Configuration Admin

- C0010 – It must be possible to specify the service.pid value when creating a factory configuration. This implies the need for a get_or_create factory configuration method in ConfigurationAdmin. (RFC 218)
- C0020 – It must be possible to prevent the updating of a configuration by the runtime even the developer forced it. (RFC 218)
- C0030 – It must be possible to avoid any action if a configuration is updated with the exact same properties and values as it already has (RFC 218)
- C0040 - Support ConfigurationPlugin-like behavior for non-MS/MSF users
- C0050 – ManagedService and ManagedServiceFactory instances should be able to be notified of configuration changes part of a unit of work at the end of the unit of work

5 Technical Solution

5.1 PID Handling of Factory Configurations

The current Configuration Admin Service Specification provides no control over the PID of a factory configuration: a new factory configuration gets assigned a PID generated by the *Configuration Admin*. This makes it hard for any (provisioning) tool to manage such a configuration as it needs to store this generated PID in order to later on update or delete the factory configuration.

Configuration Admin specifies that a “PID should follow the symbolic-name syntax”, however in the examples in table 104.1 non symbolic-names are used. For targeted PIDs it's already defined that the pipe character '|' is used to separate the PID part from the target information which in fact means that a PID must not use this character. This should be more precisely specified in section 104.3.1. In the same way as the pipe character has been introduced as a special character the character '#' is introduced as another special character for the PID handling of factory configurations.

By introducing two new methods on the *Configuration Admin* service, a client of the service can specify the PID of the factory configuration by providing the factory PID and a name:

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

public Configuration getFactoryConfiguration(String factoryPid, String name)
throws IOException;
```

These methods require a factoryPID and a name argument. The method still generates a PID however the generated PID has the form: `factoryPid#name`. This ensures that the *Configuration Admin* can still guarantee uniqueness of the PID. If a configuration with the given combination of factoryPID and name already exists, this is returned, otherwise a new factory configuration is returned. The returned configuration has the factoryPID and a generated PID as mentioned above. Location handling, binding and permission checks works as defined for `getConfiguration`.

The name can be used to find a factory configuration using `listConfigurations`:

```
ConfigurationAdmin ca = …;
ca.listConfigurations("(service.pid=my.factory.pid#myname)");
```

5.2 Locking Configuration Records

The `Configuration` interface is enhanced with the following methods:

```
/**
 * Set the read-only state of~~Locks or unlocks~~ the configuration.
 * @param flag If {@code true} the configuration is read-only~~locked~~,
 *             if {@code false} the configuration is read-write~~unlocked~~.
 * @throws IOException If the new ~~lock~~read-only state cannot be persisted.
 * @throws IllegalStateException If this configuration has been deleted.
 */
public void setReadOnly~~Locked~~(boolean flag) throws IOException;

/**
 * Check if the configuration is currently read-only~~locked~~.
 * @return {@code true} if the configuration is read-only~~locked~~, {
 *         {@code false} otherwise.
 * @throws IllegalStateException If this configuration has been deleted.
 */
public boolean isReadOnly~~Locked~~();
```

If the configuration is ~~locked~~marked as read-only using `setReadOnly~~Locked~~(true)` this state is persisted and the configuration remains read-only~~locked~~ until ~~it is explicitly unlocked calling~~ `setReadOnly~~Locked~~(false)` is called. If the configuration is ~~locked~~read-only and Configuration.updateIfDifferent~~setProperties~~(Dictionary),

Configuration.update(Dictionary) or `Configuration.delete()` is called, a ~~Locked~~ReadOnly`ConfigurationException` (a Runtime~~subclass of IO~~Exception) is thrown.

5.2.1 Security impacts

A new action, ~~LOCK~~ATTRIBUTE, is added to the configuration permission. In order to lock or unlock a configuration, the caller needs the permission to do so. The verification of the permission is handled in the same way as for the CONFIGURE action as outline in section 104.11.1.

5.3 Improving Configuration Updates

Currently, any call of the update method on the Configuration object assumes that the configuration actually changed. The change count is increased, listeners and managed service (factories) are informed.

Configuration Admin should actually check whether the updated configuration is the same as the previous configuration, and if so ignore this operation. This allows all (provisioning) clients to simply update a configuration without reimplementing a complicated change check, Doing it once in Configuration Admin is more efficient and improves the handling for every client.

As configurations should only contain a limited set of types, equals can be called on the property values to find out whether the values are the same. For arrays, equals need to be called on each member of the array.

```
    /**
     * Update the properties of this {@code Configuration} object if the
     * provided properties are different than the currently stored set
     *
     * Properties are compared as follows.
     * <ul>
     * <li>scalars are compared using equals()
     * <li>arrays are compared using Arrays.equals()
     * <li>Collections are compared on content not on order
     * </ul>
// TODO define what does it mean to be equal for String+, Long+
     * If the properties ~~are~~compare the same, no operation is performed, otherwise
     * the behaviour is identical to the update(Dictionary) method.
~~it~~
     * ~~stores the properties in persistent storage after adding or overwriting~~
     * ~~the following properties:~~
     * ~~<ul>~~
     * ~~<li>"service.pid" : is set to be the PID of this configuration.</li>~~
     * ~~<li>"service.factoryPid" : if this is a factory configuration it is set~~
     * ~~to the factory PID else it is not set.</li>~~
     * ~~</ul>~~
     * ~~These system properties are all of type {@code String}.~~
     * ~~~~
     * ~~<p>~~
     * ~~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.~~
     * ~~~~
     * ~~<p>~~
     * ~~Also notifies all Configuration Listeners with a~~
     * ~~{@link ConfigurationEvent#CM_UPDATED} event.~~
     *
     * @param properties the new set of properties for this configuration
     * @throws ~~Locked~~ReadOnlyConfigurationException If the configuration is locked
```

```
     * @throws IOException if update cannot be made persistent
     * @throws IllegalArgumentException if the {@code Dictionary} object
     *         contains invalid configuration types or contains case variants of
     *         the same key name.
     * @throws IllegalStateException If this configuration has been deleted.
     */
    public void setPropertiesupdateIfDifferent(Dictionary<String, ?> properties)
throws IOException;
```

5.4 Capabilities

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

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

This capability must follow the rules defined for the osgi.implementation Namespace.

The bundle providing the Configuration Admin service must provide a capability in the osgi.service namespace representing this service. This capability must also declare a uses constraint for the org.osgi.service.cm package:

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

This capability must follow the rules defined for the osgi.service Namespace.

5.5 Improved ConfigurationPlugin Support

This has been discussed as part of RFC 165 and removed from that RFC (see section 6.2 in RFC 165).

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 transform the configuration dictionary just before it reaches the configuration target service. In order to support Configuration Admin clients not registering a ManagedService(Factory), a new method `getModifiedProcessedProperties(ServiceReference<ManagedService>)` is added to the Configuration interface. If this is invoked, Configuration Admin calls all `ConfigurationPlugin` services as already described in the specification and the modified configuration properties are returned as a `Dictionary`.

The service reference passed into the method can point to a placeholder service which is registered on behalf of the bundle using the configuration. For example, DS can register such a service for each bundle it is extending. As that service should not be called by Configuration Admin it will register it without a PID service property, therefore it will be ignored. The advantage of this approach is that the passed in reference is a valid service reference.

5.6 Clarifications

TODO need to add proposed changes...

5.6.1 ConfigurationPlugin Ranking

While it is explained that plugins are ordered by service.cmRanking, it is not explicitly mentioned which order is used when two plugins have the same ranking. It would be nice to clarify this, e.g. ordering by service.id in that case (lowest last).

5.6.2 Modifications by ConfigurationPlugin

The current spec text is a little bit unclear how to handle modifications of plugins with a cmRanking below 0 or above 1000. While it reads that these “should” not modify the properties, it also states that these are called before/after modifications are made. This leads to the assumption that any modifications of such plugins are ignored. This could be made more explicit.

It also states that any exception thrown by a plugin is ignored (and logged), but it’s not stated what happens with modifications already done by this plugin. While this is of course an edge case, the spec should be more precise.

5.6.3 ManagedService(Factory) without PID property

It’s not explicitly mentioned what happens with ManagedService(Factory) services registered without a PID. Such services are obviously ignored.

5.7 Coordinations

If configurations are created, updated or deleted and an implicit coordination exists, Configuration Admin should delay updating the ManagedService(Factory) until the coordination completes.

6 Data Transfer Objects

RFC 185 defines Data Transfer Objects as a generic means for management solutions to interact with runtime entities in an OSGi Framework. DTOs provides a common, easily serializable representation of the technology.

For all new functionality added to the OSGi Framework the question should be asked: would this feature benefit from a DTO? The expectation is that in most cases it would.

The DTOs for the design in this RFC should be described here and if there are no DTOs being defined an explanation should be given explaining why this is not applicable in this case.

This section is optional and could also be provided in a separate RFC.

7 Javadoc

Please include Javadoc of any new APIs here, once the design has matured. Instructions on how to export Javadoc for inclusion in the RFC can be found here: https://www.osgi.org/members/RFC/Javadoc

OSGi Javadoc

08.09.16 09:54

Package Summary		Page
org.osgi.service.cm	Configuration Admin Package Version 1.6.	13

Package org.osgi.service.cm

```
@org.osgi.annotation.versioning.Version(value="1.6")
```

Configuration Admin Package Version 1.6.

See:

Description

Interface Summary		Page
Configuration	The configuration information for a `ManagedService` or `ManagedServiceFactory` object.	14
ConfigurationAdmin	Service for administering configuration data.	21
ConfigurationListener	Listener for Configuration Events.	33
ConfigurationPlugin	A service interface for processing configuration dictionary before the update.	37
ManagedService	A service that can receive configuration data from a Configuration Admin service.	40
ManagedServiceFactory	Manage multiple service instances.	42
SynchronousConfigurationListener	Synchronous Listener for Configuration Events.	45

Class Summary		Page
ConfigurationEvent	A Configuration Event.	27
ConfigurationPermission	Indicates a bundle's authority to configure bundles or be updated by Configuration Admin.	34

Exception Summary		Page
ConfigurationException	An `Exception` class to inform the Configuration Admin service of problems with configuration data.	30
LockedConfigurationException	An `Exception` class to inform the client of a `Configuration` about the locked state of a configuration object.	39

Package org.osgi.service.cm Description

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)"
```

Interface Configuration

org.osgi.service.cm

```
@org.osgi.annotation.versioning.ProviderType
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 `org.osgi.framework.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.

ThreadSafe

Method Summary		Page
void	**delete**() Delete this `Configuration` object.	16
boolean	**equals**(Object other) Equality is defined to have equal PIDs Two Configuration objects are equal when their PIDs are equal.	19
String	**getBundleLocation**() Get the bundle location.	18
long	**getChangeCount**() Get the change count.	18
String	**getFactoryPid**() For a factory configuration return the PID of the corresponding Managed Service Factory, else return `null`.	17
Dictionary<String,Object>	**getModifiedProperties**(org.osgi.framework.ServiceReference<ManagedService> reference) Return the modified properties of this `Configuration` object.	16

String	**getPid**() Get the PID for this `Configuration` object.	15
Dictionary<String,Object>	**getProperties**() Return the properties of this `Configuration` object.	15
int	**hashCode**() Hash code is based on PID.	19
boolean	**isLocked**() Check if the configuration is currently locked.	19
void	**setBundleLocation**(String location) Bind this `Configuration` object to the specified location.	18
void	**setLocked**(boolean flag) Locks or unlocks the configuration.	19
boolean	**setProperties**(Dictionary<String,?> properties) Update the properties of this `Configuration` object if the provided properties are different than the currently stored set If the properties are the same, no operation is performed, otherwise it stores the properties in persistent storage after adding or overwriting the following properties: • "service.pid" : is set to be the PID of this configuration.	17
void	**update**() Update the `Configuration` object with the current properties.	17
void	**update**(Dictionary<String,?> properties) Update the properties of this `Configuration` object.	16

Method Detail

getPid

`String` **`getPid`**`()`

Get the PID for this `Configuration` object.

Returns:
the PID for this `Configuration` object.

Throws:
`IllegalStateException` - if this configuration has been deleted

---

getProperties

`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.

getModifiedProperties

```
Dictionary<String,Object> getModifiedProperties(org.osgi.framework.ServiceReference<ManagedService> reference)
```

Return the modified 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 run through all registered `ConfigurationPlugin`s handling the configuration for this PID.

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.

update

```
void update(Dictionary<String,?> properties)
     throws IOException
```

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.

Parameters:

`properties` - the new set of properties for this configuration

Throws:

`IOException` - if update cannot be made persistent
`LockedConfigurationException` - if the configuration is locked
`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.

delete

```
void delete()
     throws IOException
```

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

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

Throws:
`IOException` - If delete fails.
`LockedConfigurationException` - if the configuration is locked
`IllegalStateException` - If this configuration has been deleted.

---

getFactoryPid

```
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.

---

update

```
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`

---

setProperties

```
boolean setProperties(Dictionary<String,?> properties)
               throws IOException
```

Update the properties of this `Configuration` object if the provided properties are different than the currently stored set If the properties are the same, no operation is performed, otherwise it 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.

Parameters:
`properties` - the new set of properties for this configuration
Returns:
Returns `true` if the configuration was updated.
Throws:
`IOException` - if update cannot be made persistent
`LockedConfigurationException` - If the configuration is locked
`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

---

setBundleLocation

`void` **`setBundleLocation`**`(String location)`

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.

Parameters:
`location` - a location, region, or `null`
Throws:
`IllegalStateException` - If this configuration has been deleted.
`SecurityException` - when the required permissions are not available,
when the required permissions are not available

---

getBundleLocation

`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

---

getChangeCount

`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

---

setLocked

```
void setLocked(boolean flag)
        throws IOException
```

Locks or unlocks the configuration.

Parameters:
`flag` - If `true` the configuration is locked, if `false` the configuration is unlocked.
Throws:
`IOException` - If the new lock state cannot be persisted.
`IllegalStateException` - If this configuration has been deleted.
Since:
1.6

---

isLocked

```
boolean isLocked()
```

Check if the configuration is currently locked.

Returns:
`true` if the configuration is locked, { `false` otherwise.
Throws:
`IllegalStateException` - If this configuration has been deleted.
Since:
1.6

---

equals

```
boolean equals(Object other)
```

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

Overrides:
`equals` in class `Object`
Parameters:
`other` - `Configuration` object to compare against
Returns:
`true` if equal, `false` if not a `Configuration` object or one with a different PID.

---

hashCode

```
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.

Overrides:

`hashCode` in class `Object`

Returns:

hash code for this Configuration object

Interface ConfigurationAdmin

org.osgi.service.cm

```
@org.osgi.annotation.versioning.ProviderType
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 call-backs asynchronously to allow proper synchronization.

When the Configuration Admin service detects a Managed Service Factory registration, it checks its storage for configuration objects whose `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.

ThreadSafe

Field Summary		Page
String	**SERVICE_BUNDLELOCATION** Configuration property naming the location of the bundle that is associated with a `Configuration` object.	22
String	**SERVICE_FACTORYPID** Configuration property naming the Factory PID in the configuration dictionary.	22

Method Summary		Page
Configuration	**createFactoryConfiguration**(String factoryPid) Create a new factory `Configuration` object with a new PID.	23
Configuration	**createFactoryConfiguration**(String factoryPid, String location) Create a new factory `Configuration` object with a new PID.	23
Configuration	**getConfiguration**(String pid) Get an existing or new `Configuration` object from the persistent store.	24
Configuration	**getConfiguration**(String pid, String location) Get an existing `Configuration` object from the persistent store, or create a new `Configuration` object.	24
Configuration	**getFactoryConfiguration**(String factoryPid, String alias) Get an existing or new `Configuration` object from the persistent store.	25
Configuration	**getFactoryConfiguration**(String factoryPid, String alias, String location) Get an existing or new `Configuration` object from the persistent store.	24
Configuration[]	**listConfigurations**(String filter) List the current `Configuration` objects which match the filter.	25

Field Detail

SERVICE_FACTORYPID

```
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

SERVICE_BUNDLELOCATION

```
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

Method Detail

createFactoryConfiguration

`Configuration` **`createFactoryConfiguration`**`(String factoryPid)`
`throws IOException`

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.

Parameters:
`factoryPid` - PID of factory (not `null`).
Returns:
A new `Configuration` object.
Throws:
`IOException` - if access to persistent storage fails.

---

createFactoryConfiguration

`Configuration` **`createFactoryConfiguration`**`(String factoryPid,`
`String location)`
`throws IOException`

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.

Parameters:
`factoryPid` - PID of factory (not `null`).
`location` - A bundle location string, or `null`.
Returns:
a new `Configuration` object.
Throws:
`IOException` - if access to persistent storage fails.
`SecurityException` - when the require permissions are not available

---

getConfiguration

```
Configuration getConfiguration(String pid,
                               String location)
                        throws IOException
```

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.

Parameters:
`pid` - Persistent identifier.
`location` - The bundle location string, or `null`.
Returns:
An existing or new `Configuration` object.
Throws:
`IOException` - if access to persistent storage fails.
`SecurityException` - when the require permissions are not available

---

getConfiguration

```
Configuration getConfiguration(String pid)
                        throws IOException
```

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.

Parameters:
`pid` - persistent identifier.
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

---

getFactoryConfiguration

```
Configuration getFactoryConfiguration(String factoryPid,
                                      String alias,
                                      String location)
                               throws IOException
```

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 alias by starting with the factory PID appending the character # and then appending the alias.

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.

Parameters:
`factoryPid` - PID of factory (not `null`).
`alias` - An alias for `Configuration` (not `null`).
`location` - The bundle location string, or `null`.

Returns:
An existing or new `Configuration` object.

Throws:
`IOException` - if access to persistent storage fails.
`SecurityException` - when the require permissions are not available

Since:
1.6

---

getFactoryConfiguration

```
Configuration getFactoryConfiguration(String factoryPid,
                                      String alias)
                               throws IOException
```

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 alias by starting with the factory PID appending the character # and then appending the alias. 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.

Parameters:
`factoryPid` - PID of factory (not `null`).
`alias` - An alias for `Configuration` (not `null`).

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

Since:
1.6

---

listConfigurations

```
Configuration[] listConfigurations(String filter)
                            throws IOException,
                                   org.osgi.framework.InvalidSyntaxException
```

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 `org.osgi.framework.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.

Parameters:

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

Returns:

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

Throws:

`IOException` - if access to persistent storage fails

`org.osgi.framework.InvalidSyntaxException` - if the filter string is invalid

Class ConfigurationEvent

org.osgi.service.cm

```
java.lang.Object
  └ org.osgi.service.cm.ConfigurationEvent
```

```
public class ConfigurationEvent
extends Object
```

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`.

Since:
1.2
See Also:
`ConfigurationListener`
Immutable

Field Summary	Page
static int **CM_DELETED** A `Configuration` has been deleted.	28
static int **CM_LOCATION_CHANGED** The location of a `Configuration` has been changed.	28
static int **CM_UPDATED** A `Configuration` has been updated.	28

Constructor Summary	Page
ConfigurationEvent(org.osgi.framework.ServiceReference<ConfigurationAdmin> reference, int type, String factoryPid, String pid) Constructs a `ConfigurationEvent` object from the given `ServiceReference` object, event type, and pids.	28

Method Summary	Page
String **getFactoryPid**() Returns the factory pid of the associated configuration.	29
String **getPid**() Returns the pid of the associated configuration.	29

org.osgi.framework.ServiceReference<ConfigurationAdmin>	**getReference**() Return the ServiceReference object of the Configuration Admin service that created this event.	29
int	**getType**() Return the type of this event.	29

Field Detail

CM_UPDATED

```
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.

~~CM_DELETED~~

```
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.

~~CM_LOCATION_CHANGED~~

```
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

Constructor Detail

ConfigurationEvent

```
public ConfigurationEvent(org.osgi.framework.ServiceReference<ConfigurationAdmin> reference,
                          int type,
                          String factoryPid,
                          String pid)
```

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

Parameters:
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.

Method Detail

getFactoryPid

`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.

---

getPid

`public String` **`getPid`**`()`

Returns the pid of the associated configuration.

Returns:

Returns the pid of the associated configuration.

---

getType

`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.

---

getReference

`public org.osgi.framework.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.

Class ConfigurationException

org.osgi.service.cm

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └org.osgi.service.cm.ConfigurationException
```

All Implemented Interfaces:

Serializable

```
public class ConfigurationException
extends Exception
```

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

Constructor Summary	Page
ConfigurationException(String property, String reason) Create a `ConfigurationException` object.	30
ConfigurationException(String property, String reason, Throwable cause) Create a `ConfigurationException` object.	31

Method Summary		Page
Throwable	**getCause**() Returns the cause of this exception or `null` if no cause was set.	31
String	**getProperty**() Return the property name that caused the failure or null.	31
String	**getReason**() Return the reason for this exception.	31
Throwable	**initCause**(Throwable cause) Initializes the cause of this exception to the specified value.	31

Constructor Detail

ConfigurationException

```
public ConfigurationException(String property,
                              String reason)
```

Create a `ConfigurationException` object.

Parameters:

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

`reason` - reason for failure

ConfigurationException

```
public ConfigurationException(String property,
                              String reason,
                              Throwable cause)
```

Create a `ConfigurationException` object.

Parameters:

`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.

Since:

1.2

Method Detail

getProperty

```
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

---

getReason

```
public String getReason()
```

Return the reason for this exception.

Returns:

reason of the failure

---

getCause

```
public Throwable getCause()
```

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

Overrides:

`getCause` in class `Throwable`

Returns:

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

Since:

1.2

---

initCause

```
public Throwable initCause(Throwable cause)
```

Initializes the cause of this exception to the specified value.

Overrides:

`initCause` in class `Throwable`

Parameters:

`cause` - The cause of this exception.

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

Interface ConfigurationListener

org.osgi.service.cm

All Known Subinterfaces:
SynchronousConfigurationListener

```
@org.osgi.annotation.versioning.ConsumerType
public interface ConfigurationListener
```

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

`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
ThreadSafe

Method Summary		Page
void	**configurationEvent**(ConfigurationEvent event) Receives notification of a Configuration that has changed.	33

Method Detail

configurationEvent

```
void configurationEvent(ConfigurationEvent event)
```

Receives notification of a Configuration that has changed.

Parameters:
`event` - The `ConfigurationEvent`.

Class ConfigurationPermission

org.osgi.service.cm

```
java.lang.Object
  └java.security.Permission
      └java.security.BasicPermission
          └org.osgi.service.cm.ConfigurationPermission
```

All Implemented Interfaces:
Guard, Serializable

```
final public class ConfigurationPermission
extends BasicPermission
```

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

Since:
1.2
ThreadSafe

Field Summary		Page
static String	**CONFIGURE** Provides permission to create new configurations for other bundles as well as manipulate them.	35
static String	**LOCK** Provides permission to lock a configurations.	35
static String	**TARGET** The permission to be updated, that is, act as a Managed Service or Managed Service Factory.	35

Constructor Summary	Page
ConfigurationPermission(String name, String actions) Create a new ConfigurationPermission.	35

Method Summary		Page
boolean	**equals**(Object obj) Determines the equality of two ConfigurationPermission objects.	36
String	**getActions**() Returns the canonical string representation of the ConfigurationPermission actions.	36
int	**hashCode**() Returns the hash code value for this object.	36
boolean	**implies**(Permission p) Determines if a ConfigurationPermission object "implies" the specified permission.	35
PermissionCollection	**newPermissionCollection**() Returns a new PermissionCollection object suitable for storing ConfigurationPermissions.	36

Field Detail

CONFIGURE

```
public static final String CONFIGURE = "configure"
```

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

~~TARGET~~

```
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

LOCK

```
public static final String LOCK = "lock"
```

Provides permission to lock a configurations. The action string `"lock"`.

Since:
1.6

Constructor Detail

ConfigurationPermission

```
public ConfigurationPermission(String name,
                               String actions)
```

Create a new ConfigurationPermission.

Parameters:
`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 `org.osgi.framework.Filter`s.
`actions` - Comma separated list of `CONFIGURE`, `TARGET` (case insensitive).

Method Detail

implies

```
public boolean implies(Permission p)
```

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

Overrides:
`implies` in class `BasicPermission`
Parameters:
`p` - The target permission to check.

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

---

equals

```
public boolean equals(Object obj)
```

Determines the equality of two `ConfigurationPermission` objects.

Two `ConfigurationPermission` objects are equal.

Overrides:
`equals` in class `BasicPermission`
Parameters:
`obj` - The object being compared for equality with this object.
Returns:
`true` if `obj` is equivalent to this `ConfigurationPermission`; `false` otherwise.

---

hashCode

```
public int hashCode()
```

Returns the hash code value for this object.

Overrides:
`hashCode` in class `BasicPermission`
Returns:
Hash code value for this object.

---

getActions

```
public String getActions()
```

Returns the canonical string representation of the `ConfigurationPermission` actions.

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

Overrides:
`getActions` in class `BasicPermission`
Returns:
Canonical string representation of the `ConfigurationPermission` actions.

---

newPermissionCollection

```
public PermissionCollection newPermissionCollection()
```

Returns a new `PermissionCollection` object suitable for storing `ConfigurationPermission`s.

Overrides:
`newPermissionCollection` in class `BasicPermission`
Returns:
A new `PermissionCollection` object.

Interface ConfigurationPlugin

org.osgi.service.cm

```
@org.osgi.annotation.versioning.ConsumerType
public interface ConfigurationPlugin
```

A service interface for processing configuration dictionary before the update.

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

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

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

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

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

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

ThreadSafe

Field Summary		Page
String	**CM_RANKING** A service property to specify the order in which plugins are invoked.	38
String	**CM_TARGET** A service property to limit the Managed Service or Managed Service Factory configuration dictionaries a Configuration Plugin service receives.	38

Method Summary		Page
void	**modifyConfiguration**(org.osgi.framework.ServiceReference<?> reference, Dictionary<String,Object> properties) View and possibly modify the a set of configuration properties before they are sent to the Managed Service or the Managed Service Factory.	38

Field Detail

CM_TARGET

```
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.

CM_RANKING

```
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

Method Detail

modifyConfiguration

```
void modifyConfiguration(org.osgi.framework.ServiceReference<?> reference,
                         Dictionary<String,Object> properties)
```

View and possibly modify the a set of configuration properties before they are sent to the Managed Service or the Managed Service Factory. The Configuration Plugin services are called in increasing order of their `service.cmRanking` property. If this property is undefined or is a non- `Integer` type, 0 is used.

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

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

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

Parameters:

`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.

Class LockedConfigurationException

org.osgi.service.cm

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └java.io.IOException
              └org.osgi.service.cm.LockedConfigurationException
```

All Implemented Interfaces:

Serializable

```
public class LockedConfigurationException
extends IOException
```

An `Exception` class to inform the client of a `Configuration` about the locked state of a configuration object.

Since:

1.6

Constructor Summary	Page
`LockedConfigurationException(String reason)` Create a `LockedConfigurationException` object.	39

Constructor Detail

LockedConfigurationException

```
public LockedConfigurationException(String reason)
```

Create a `LockedConfigurationException` object.

Parameters:

`reason` - reason for failure

Interface ManagedService

org.osgi.service.cm

```
@org.osgi.annotation.versioning.ConsumerType
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.

ThreadSafe

Method Summary	Page
void **updated**(Dictionary<String,?> properties) Update the configuration for a Managed Service.	41

Method Detail

updated

```
void updated(Dictionary<String,?> properties)
      throws ConfigurationException
```

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.

Parameters:

`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.

Throws:

`ConfigurationException` - when the update fails

Interface ManagedServiceFactory

org.osgi.service.cm

```
@org.osgi.annotation.versioning.ConsumerType
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 );
  }
  ...
}
```

ThreadSafe

Method Summary	Page
void **deleted**(String pid) Remove a factory instance.	44
String **getName**() Return a descriptive name of this factory.	43
void **updated**(String pid, Dictionary<String,?> properties) Create a new instance, or update the configuration of an existing instance.	43

Method Detail

getName

```
String getName()
```

Return a descriptive name of this factory.

Returns:
the name for the factory, which might be localized

---

updated

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

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.

Parameters:
`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.

Throws:

`ConfigurationException` - when the configuration properties are invalid.

---

deleted

```
void deleted(String pid)
```

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.

Parameters:

`pid` - the PID of the service to be removed

Interface SynchronousConfigurationListener

org.osgi.service.cm

All Superinterfaces:
ConfigurationListener

```
@org.osgi.annotation.versioning.ConsumerType
public interface SynchronousConfigurationListener
extends ConfigurationListener
```

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

`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
ThreadSafe

Methods inherited from interface org.osgi.service.cm.ConfigurationListener
configurationEvent

8 Considered Alternatives

8.1

9 Security Considerations

Description of all known vulnerabilities this may either introduce or address as well as scenarios of how the weaknesses could be circumvented.

10 Document Support

10.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

10.2 Author's Address

Name	Carsten Ziegeler
Company	Adobe Systems Incorporated
Address	Barfüsserplatz 6, 4055 Basel, Switzerland
Voice	+41 61 226 55 0
e-mail	cziegele@adobe.com

10.3 Acronyms and Abbreviations

10.4 End of Document