RFC-218 Configurer

Draft

17 Pages

Abstract

10 point Arial Centered.

OSGi Configuration Admin is a slightly pedantic but highly effective flexible standardized model to configure applications. This RFC seeks a solution to carry configuration information in a bundle.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

0.2 Trademarks

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
2.1 Coordination........5
2.2 Logical PIDs........6
2.3 Delivery........6
2.4 OSGi Application Model........6
2.5 OSGi enRoute Configurer........6

3 Problem Description........7

4 Requirements........7
4.1 General........7
4.2 Configuration Admin........9

5 Technical Solution........9
5.1 Configuration Admin Enhancements........9

5.1.1 PID Handling of Factory Configurations........ 9
5.2 Configurations........ 9
5.2.1 osgi.implementation Capability........ 10
5.2.2 Configurations in a Bundle........ 10
5.2.3 Environments........ 10
5.2.4 Configuration File Formats........ 10
5.2.5 Placeholders........ 12
5.2.6 Processing Configuration Files........ 13
5.2.7 Configuration Ranking........ 13
5.3 Processing Bundles........ 13
5.3.1 Activating a Bundle........ 14
5.3.2 Deactivating a Bundle........ 14
5.3.3 Configurations from the Runtime Environment........ 15

6 Data Transfer Objects........ 15

7 Javadoc........ 15

8 Considered Alternatives........ 16

9 Security Considerations........ 16

10 Document Support........ 16
10.1 References........ 16
10.2 Author's Address........ 16
10.3 Acronyms and Abbreviations........ 17
10.4 End of Document........ 17

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 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	*DEC 04 2015*	*Initial Version* *Carsten Ziegeler, Adobe <cziegele@adobe.com>*

1 Introduction

This RFC originates from the OSGi enRoute work. In this project, a number of services were identified, designed and implemented based on their needs for web based applications. This document analyzes the application domain and defines the problem that needs to be solved.

This RFC discusses the Configurer, an extender that allows the storage of configuration data in a bundle.

2 Application Domain

OSGi provides a standardized model to provide bundles with confgurations. This is specified in the Configuration Admin specification. In this specification, A configuration is identified by a persistent identity (PID). A PID is a unique token, recommended to be conforming to the symbolic name syntax. A configuration consists of a set of properties, where a property consists of a string key and a corresponding value. The type of the value is limited to the primitive types and their wrappers, Strings, or Java Arrays/List/Vector of these.

Sometimes it is necessary to store binary large objects (BLOB) in configuration. For example, a keystore with certificates. Since configuration admin is not suitable for this, these BLOBs are often stored on the files system. The application developers then must manage the life cycles of these files.

Configurations can be grouped with a factory PID. Configurations with a factory PID are called *factory* configurations and without it they are called *singleton* configurations.

The original specification specified that the configurations were sent to a Managed Service for the singletons and Managed Service Factory services for the factory instances. However, over time *component models* became popular and a component can rely on configuration. For example, Declarative Services is tightly integrated with Configuration Admin. For these heavy users of configurations a *Configuration Listener* whiteboard service was added. Configuration update, delete, and bundle location change events are forwarded to this whiteboard service on a background thread.

2.1 Coordination

In OSGi, the *management agent* creates and deletes configurations through the Configuration Admin service. Appropriate Create/Read/Update/Delete (CRUD) methods for singletons and factories are available on this service. If the management agent performs a number of sequential updates then it can group these updates within a Coordination from the Coordinator service. Clients can then register a Synchronous Configuration Listener and delay the application of the properties until the Coordination has ended.

2.2 Logical PIDs

One popular management agent from the early days was *File Install [3].*. File install watched a directory and configurations stored in that directory were configured in the local Configuration Admin. This is straightforward for singletons since the file name of the configuration file can be used to calculate the PID. However, for factories, the PID for the instance is calculated by Configuration Admin and is not predictable. Therefore, File Install needed to manage the number of instances since the operations to create them were not idempotent. To prevent a restart from creating an every growing number of instances it was necessary to create a link between the instance and the file. Therefore, File Install created a *logical (instance) PID* based on the file name. If this instance was detected, File Install first looks in Configuration Admin if there is any instance that has this logical PID as value in a specially designated key. Only if no such instance was found, a new one was created. This made the operation idempotent.

2.3 Delivery

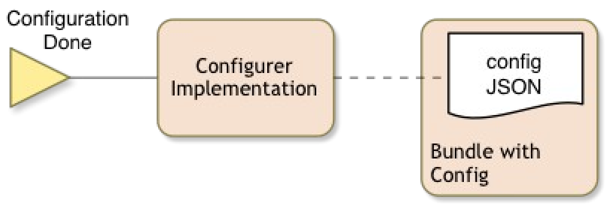
The Deployment Admin specification developed for OSGi Mobile provided the means to carry configuration information inside a JAR via the means of *resource processors*. The Auto Configuration specification defined how an Autoconf resource processor could get an XML file from a Deployment Package. Since the Deployment Package could not define the instance PID for a factory it used an alias in the XML file. When a configuration was found, the alias was looked up to see to what instance PID it was mapped to. If no mapping was found, a new instance was created. Since Deployment Packages had a well defined life cycle, they basically are like bundles, Deployment Admin could therefore also delete stale aliases.

2.4 OSGi Application Model

The Require-Capability model was designed to create applications from one or more *initial requirements*, generally identity requirements, to so called *root bundles*. From these initial requirements a *resolver*, generally with some human help, can then create a *deployable artifact*. For example an executable JAR which includes all dependencies including configuration, a subsystem specification, an Eclipse or subsystem features, etc.

2.5 OSGi enRoute Configurer

The enRoute project defined a *Configurer extender [4].*. Bundles can store configuration data in a “magic” resource at `configuration/configuration.json`. This is a JSON file containing an array of configurations. When a bundle is installed, the Configurer detects the magic resource and installs the corresponding configuration. It does not remove the configuration when the bundle is uninstalled. The Configurer uses logical PIDs to manage factories.



Before the JSON file is interpreted, the Configurer will first run it through a macro pre-processor, this is the bndlib [5]. macro preprocessor. The macro sequence is `@{...}` to not clash with build time processing.

The Configurer also supports BLOBs. A special macro `@{resource:<resource>}` points to a resource in the bundle. The Configurer will then extract this resource to the file system and replace the macro with the actual file path. The Configurer stores this data by default in the bundle's file area.

Additionally, the Configurer also read a standard system property and interpreted the content as a JSON configuration file.

3 Problem Description

The trend in OSGi is to use the Require-Capability model to construct applications out of a set of initial requirements. This means there is no applicable container like Deployment Admin to contain configurations. However, enRout shows that it is actually quite easy to store configuration data in a bundle which can then be part of the resolve process. This configuration bundle can be a root bundle representing the application or a special configuration bundle. Having everything as bundles without artificial grouping will leverage the existing specifications for life cycle management instead of introducing a new layer since interacting life cycles are notoriously hard to make reliable and usually force one to redo the same functionality slightly different.

BLOBs are awkward to handle in configurations. If they are delivered in a bundle then the developer must extract it somewhere on the file system and set the configuration to point to that file.

4 Requirements

4.1 General

- G0010 – A solution that makes it possible to store configurations, either singleton or factory, in a bundle that are installed in Configuration Admin when that bundle is installed.
- G0020 – It must be possible to specify configurations in a system property
- G0030 – Factory configurations must be idempotent.
- G0040 – If the bundle that originated a configuration gets uninstalled then all its configurations must be deleted
- G0050 – If a bundle with configurations is updated then configurations that are already on the system must not be updated.
- G0060 – Developers must be able to indicate that a configuration must be forced updated, even if it already exists.

- G0065 – G0040-G0060 imply the need for configurations to have an update policy. The update policy must define how it interacts with the lifecycle of the bundle containing the configuration.

- G0070 – It must be possible to prevent the updating of a configuration by the runtime even the developer forced it.

- **G0080 – It must be possible to include BLOBs in the configuration that are stored in the bundle but extracted to the file system.**

- **G0085 – BLOBs must always be updated on the file system to a different location than the previous version. This is too avoid open file lock issues on Windows file systems.**

- G0090 – It must be possible to load configuration from the host bundle as well as all its fragments

- G0100 – If configurations are duplicated, the specification must define a proper order

- G0120 – The format used to store the configuration must be a human readable text file. It should be easy to read.

- G0125 – The format must allow configuration keys to be typed. For example: “foo:Integer”:”1”.

- G0130 – The configurations must by default be usable by any bundle. That is the bundle location must be “?”

- ~~G0140 – It must be possible to specify a specific bundle location for a configuration (### yes? Not sure.)~~

- G0150 – It must be possible to coordinate the settings of configurations.

- G0160 – If the Configurer starts up, it must set all configurations that are at installed bundles at that moment in a single coordination.

- G0170 – The solution should group settings inside a coordination as much as possible.

- G0180 – The solution must be able to purge configurations from uninstalled bundles even if it had not been active while they were uninstalled.

- G0190 – It must be possible in runtime to disable the configuration of a complete bundle or specific configurations.

- ~~G0200 – If the Metatype of a configuration can be found then the configuration values must be converted to the indicated types before being set in Configuration Admin.~~

- G0210 – If no metatype is found for a configuration then the specification must prescribe the value types for the chosen text format if this format does not cover all possible Configuration types.

- G0220 – Any errors must be logged

- G0230 – It must be possible to specify the configuration in multiple bundle resources. The specification must define how to resolve conflicting configurations.

- G0240 – It must be possible to specify a profile system property that can select a part of a configuration file. This will allow the same bundle to be used in different settings.

- G0250 – It must be possible to use information from the local system, for example passwords, if permitted.

4.2 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.

5 Technical Solution

5.1 Configuration Admin Enhancements

This chapter contains enhancements to the Configuration Admin Service Specification.

5.1.1 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 service. 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.

By introducing two new methods on the *ConfigurationAdmin* service, a client of the service can specify the PID for a factory configuration:

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

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

These methods require a factoryPID and a PID argument and act in the same way as getConfiguration for singleton configurations: if a configuration with the given combination of factoryPID and PID exists it is returned, otherwise a new factory configuration with the given factoryPID and PID is created. Location handling and binding works as defined for getConfiguration.

5.2 Configurations

The Configurer service is a service defined by this specification to process configurations stored in a bundle. In the following, the term *configuration identity* is used: For factory configurations this is the combination of factory PID and PID, for singleton configurations this is the PID.

This specification refers to the Converter service from the Object Conversion specification (RFC 216). While the text states that the Converter service is used, it is left to the implementation to use such a service or implement it in any other way providing the used functionality as specified in RFC 216.

5.2.1 osgi.implementation Capability

The Configurer implementation bundle must provide the `osgi.implementation` capability with name `osgi.configurer`. This capability can be used by provisioning tools and during resolution to ensure that a Configurer implementation is present to process the configurations. The capability must also provide the version of this specification:

```
Provide-Capability: osgi.implementation;
                    osgi.implementation="osgi.configurer";
                    version: Version="1.0"
```

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

5.2.2 Configurations in a Bundle

If a bundle contains configurations to be managed by the Configurer service, it must require the above mentioned capability. The configurations to be processed must be placed in the `OSGI-INF/configurer` directory. The Configurer service uses `Bundle.findEntries` on the bundle requiring the capability to find the files. All files with the ending ".json" within that directory are handled as configuration files by the Configurer.

5.2.3 Environments

In some cases it is handy to have different configurations based on the environment the instance is running in. Typical scenarios are different configuration during development, for testing and in production. The environment can be specified by setting the framework property `configurer.environment` with a comma separated string value of the active environments. An identifier of an environment must follow the *token* definition from the OSGi core chapter. If the property contains invalid characters, it will be ignored and logged as an error with the log service. Whitespace characters before and after each environment in the property value are ignored.

If the property is set with a valid environment definition, the Configurer uses the environment definition to read additional configurations from the bundle: for each environment from the defined set, it builds a path by appending the name of the environment to the base directory, `OSGI-INF/configurer`, and uses the full path to find entries ending with “*.json”.

For example, if the property is set to the value “dev,c1,west” the following paths are tried in addition to the base path: `OSGI-INF/configurer/c1`, `OSGI-INF/configurer/dev`, and `OSGI-INF/configurer/west`.

5.2.4 Configuration File Formats

Each configuration file processed by the Configurer is in JSON format. Such a configuration file can either contain a single configuration, an array of configurations or an array of configurations in combination with common properties applied to all configurations within the file. If a JSON file contains a syntax error and can't be parsed, the Configurer will ignore the whole bundle and log an error with the log service.

5.2.4.1 Single Configuration

A single configuration is defined through a JSON object. The PID for a configuration is defined by the required `service.pid` property. For factory configurations the factory PID is specified with the `service.factoryPid` property and both properties `service.pid` and `service.factoryPid` are required. All other properties are used as configuration properties for the configuration.

```
{
    "service.pid" : "foo.bar",
    "email" : "something@somewhere.com",
    "port:Integer" : 300
}
```

As JSON only supports a basic set of scalar types (string, double, long, boolean), arrays and maps, the Java type of a property for the Configuration object can be specified as part of the property name by appending a colon followed by the Java type. The Object Converter service (RFC 215) is used to convert the JSON value to the specified type. In addition to the wrapper types like Integer, Long, Double, Float, Boolean, Character, arrays of the scalar types can be used as well as collection types with generics or any other type supported by the Object Converter. The type needs to be visible to the bundle containing the configuration. If it's no visiblet, the configuration is ignored and an error is logged with the Log Service.

```
{
    "an.int.array:int[]" : [2, 3, 4],
    "an.Integer.collection:Collection<Integer>" : [2,3,4],
    "a.Long.vector:Vector" : [4, 5, 6],
    "complex:my.pckage.DTO" : {
                                "a" : 1,
                                "b" : "two"
                            }
}
```

If no type is specified as part of the property key, the mapping between the JSON type and the Java type is as follows:

JSON Type	Java Type
String	String
long	Long
double	Double
boolean	Boolean
Array of String	String[]
Array of long	Long[]
Array of double	Double[]
Array of boolean	Boolean[]
Anything else	JSON String

A comment can be added to the configuration by using the special property ":comment" with any value. This property is for documentation purposes only and is neither passed into the Object Converter service nor added to the Configuration object.

If a required property is missing or a property is using an unavailable type, the configuration will be ignored, and an error is logged with the Log Service.

5.2.4.2 Array of Configurations

If a JSON file contains an array, it is assumed to contain an array of configurations. Each configuration object must follow the specification defined above.

5.2.4.3 Array of Configurations and Common Properties

With the following variant, the file contains a JSON object with two properties: `common` and `configurations`. The common property contains a set of properties which are applied to all configurations within this file. This avoids repeating the same property for each and every configuration if the configurations share the same value for that property. Each property of the common section is added to each configuration in the configurations array unless a configuration has already a property with the same name.

```
{
   "common" : {
       "application" : "myapplication"
   },
   "configurations" : [
      {
        "service.pid" : "foo.bar",
        "email" : "something@somewhere.com",
        "port:Integer" : 300
      }
   ]
}
```

The common property is optional. When reading a configuration file, the Configurer checks whether the root object is an array. If it is an array it assumes an array of configurations. If it is an object, it checks for the service.pid property. If that property exists a single configuration is assumed. Otherwise the format described in this section is assumed.

5.2.5 Placeholders

Placeholders can be used in any of the values for a property. The JSON type of the property needs to be string. A placeholder is identified by the start token “${“ and the end token “}”. A start token without an end token is considered an error, and the configuration will be ignored and the error logged with the Log Service. If a string value should contain the literal “${“ it needs to escape it with a single $, like “$${“. The value between the tokens is the property key. The key must follow the definition of a symbolic name, whitespace characters before and after the key are ignored. A default value can be specified by appending the pipe character after the key followed by the default value. It is allowed to specify the empty value as the default value after the pipe character – in this case the end token is directly following the pipe character.

```
{
    "service.pid" : "web.server",
    "port:Integer" : "${server.port|80}",
    "domain" : "${server.domain}",
    "dto:my.pckage.DTO" : {
                             "a" : "${dto.a}"
                          }
}
```

The key is used to get the corresponding framework property. If a value for this key exists, this value is used as the value for the property. If no value exists for that key but a default value is specified, the default value is used. If no value exists and no default value is specified this is considered an error and this configuration is skipped and the error is logged with the Log Service.

As the value needs to be specified as a JSON string in order to use placeholders, the type of the Java property for the configuration should be provided as part of the property key if the type is different from String.

5.2.6 Processing Configuration Files

The files from the base directory are processed in alphabetical order based on the file names. If the environment property is set with a valid value, all files in an environment are processed based on their alphabetical order. The environment directories are processed in alphabetical order after the base directory. For example a bundle has the following set of files below `OSGI-INF/configurer`

```
configs.json
more-configs.json
dev/config.json
prod/config.json
prod-e1/e1.json
prod-w1/west.json
```

If such a bundle is deployed with the environment property set to “prod,prod-e1”, the configuration files are processed in this order: configs.json, more-configs.json, prod/config.json, prod-e1/e1.json.

The files are read one after the other and a set of configurations is constructed. If a configuration is read whose identity is not already in the set, it is added to the set. If such a configuration is already in the set, the two configurations are merged: all properties of the newly read configuration are added to the existing configuration, potentially overwriting properties.

Bundles are processed sequentially. If more than one bundle provides configurations for the identity, the configuration is processed for each bundle as defined in this RFC. However, it is undefined in which order the bundles are processed by the Configurer. To better control this situation and have a predictable system configuration, configurations have a ranking.

5.2.7 Configuration Ranking

The configuration ranking is modeled after the service ranking: it is a long which defaults to 0. If two bundles have a configuration with the same identity, the configuration with the higher ranking is preferred (see below). If these two configurations have the same ranking, the configuration from the bundle with the lower bundle id is ranked higher..

The configuration ranking can be set through the property `configurer.ranking`, either on a configuration or as a common property for all configurations in a file. The type of the property is converted to a Long (as defined by the Object Converter specification). If the value can't be converted, it is considered invalid and the configuration is ignored and the error is logged.

5.3 Processing Bundles

The Configurer service is active if the implementation bundle is running and the implementation has access to a configuration admin service. When the Configurer service is started, it processes all resolved bundles and “activates” them. The Configurer also checks whether an activated bundle from a previous run has been uninstalled. In that case it “deactivates” this bundle. (The check can be done by simply checking if a bundle with a given bundle id is still present in the framework).

While the service is running, when a bundle transitions from the installed into the resolved state, the Configurer “activates” the contained configurations. This covers both c[illegible]s, installing and updating a bundle. If a bundle gets uninstalled, the Configurer “deactivates” the configurations.

Although the Configurer operates sequentially (no parallel bundle operations, no parallel configuration operations), the Configurer might first compute the actions based on the available set of bundles and perform an optimized set of operations.

5.3.1 Activating a Bundle

When a bundle gets activated by the Configurer, the Configurer does the following steps:

- If the Configurer did previously activate this bundle, it compares the last modified of the bundle with the last modified of the previous run. If it is the same, the bundle is already activated and no further step is performed.
- The set of configurations defined within the bundle is calculated by respecting the current set of environments.
- If this bundle has been activated previously, the Configurer compares the set from the old run with the new calculated set.
- If the old set contains configurations with an identity which is not in the new set, these old configurations are deactivated.
- All configurations from the new set are activated.

The set of configurations is now activated, the configurations are processed in alphabetical order based on their identifier. Each configuration is processed according to its configuration ranking:

- If the Configurer did not yet add a configuration to configuration admin with the given identity, the configuration is added.
- If the Configurer did add a configuration with the same identity but with a lower configuration ranking, the already added configuration is deactivated first and then the new configuration is added to configuration admin..
- Otherwise the configuration is ignored.

If a configuration is added to the configuration admin, it replaces a potentially existing configuration with the same identity.

If a new configuration is added, the bundle location is set to “?”. The bundle location of existing configurations is not altered.

The Configurer keeps the set of activated configurations together with the last modified time of the bundle.

5.3.2 Deactivating a Bundle

When a bundle gets deactivated by the Configurer, the Configurer uses the set of activated configurations (calculated on activation of the bundle). Each configuration is deactivated according to its deactivation policy:

– NOP : No operation is performed, the configuration in the configuration admin is left as is.

– REMOVE: The configuration is removed from configuration admin – even if the configuration stored in configuration admin is not the same as set by this bundle.

The default deactivation policy is NOP. By specifying the `configurer.deactivation.policy` property in the common properties section of a file, the default can be changed for the whole configuration file. Individual configurations can use a different policy by specifying this directly as a configuration property.

The Configurer removes all internal state for that bundle.

After deactivating the bundle, the Configurer checks whether a configuration identity got removed where another bundle provides a configuration for. In this case, the configuration with the highest configuration ranking is activated and the state for the bundle providing the configuration is updated.

5.3.3 Configurations from the Runtime Environment

When the Configurer service starts, it checks for the framework property `configurer.initial`. If such a property is available and contains valid JSON as defined in this specification, this configuration is activated before any bundle configuration is processed. Such a configuration is never deactivated. The ranking of these configurations can be set in the JSON, the bundle id for these configurations is set to -1. The configuration is treated like any other bundle configuration and accordingly activated on startup.

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

8 Considered Alternatives

For posterity, record the design alternatives that were considered but rejected along with the reason for rejection. This is especially important for external/earlier solutions that were deemed not applicable.

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

[3]. http://felix.apache.org/site/apache-felix-file-install.html

[4]. http://enroute.osgi.org/services/osgi.enroute.configurer.api.html

[5]. http://jpm4j.org/#!/p/osgi/biz.aQute.bndlib

[6]. https://angularjs.org/

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