RFP-168-Configurer

Draft

8 Pages

Abstract

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

0 Document Information

0.1 Table of Contents

0 Document Information **2**
0.1 Table of Contents 2
0.2 Terminology and Document Conventions 2
0.3 Revision History 3

1 Introduction **3**

2 Application Domain **3**
2.1 Coordination 4
2.2 Logical PIDs 4
2.3 Delivery 4
2.4 OSGi Application Model 4
2.5 OSGi enRoute Configurer 4
2.6 Terminology + Abbreviations 5

3 Problem Description **5**

4 Use Cases **6**
4.1 OSGi enRoute Model 6
4.2 Update 6

5 Requirements **6**
5.1.1 General 6

6 Document Support **8**
6.1 References 8
6.2 Author's Address 8
6.3 End of Document 8

0.2 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.`

0.3 Revision History

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

Revision	Date	Comments
Initial	*28-11-14*	*Initial* *Peter.Kriens@aQute.biz*

1 Introduction

This RFP 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 RFP 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

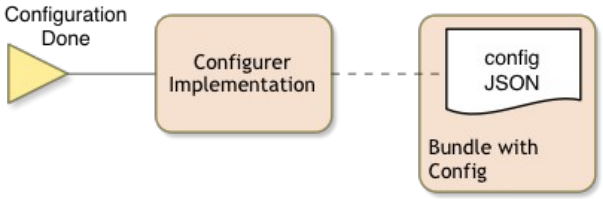
The Deployment Admin specification developed for OSGi Mobile provided the means to carry configuration information inside a JAR via the means of *resource processors*. The Auto Configuration specification defined how an Autoconf resource processor could get an XML file from a Deployment Package. Since 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.

2.6 Terminology + Abbreviations

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.

Therefore, this RFP seeks a solution where an application can provide singleton and factory configurations, with simple or BLOB data, via a bundle and its fragments.

4 Use Cases

4.1 OSGi enRoute Model

Al Bundle has created an OSGi enRoute application to watch his picture library. He created a small application bundle that contains the Angular [8] Javascript code, a JSON RPC facade for the Javascript, static pages, and the configuration.

Since this application uses JSON RPC, it is necessary to configure the JSON RPC component. This component needs an HTTP alias, it has support to protect communications in an Angular JS specific way, and has a debug mode. Al therefore creates the following file in the bundle at `configuration/configuration.json`:

```
[{
    "service.factoryPid":"osgi.web.jsonrpc",
    "service.pid":"jsonrpc",
    "alias":"/jsonrpc/2.0",
    "angular":true,
    "debug":true
}]
```

This configuration is a factory. However, the service.pid is a logical PID. The Configurer will manage the configuration now and will ensure that if the photo app application is installed, there is at least one JSON RPC handler on the `/jsonrpc/2.0` URI.

4.2 Update

Al Bundle uses Web Console to change the JSON RPC endpoint, he does not like the debug mode. A few days later he updates his application bundle. After the update, the debug flag is still false.

5 Requirements

5.1.1 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.
- 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 at the same location as the previous version.
- 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
- G0110 – It must be possible to have a service dependency on the time when all bundles have initially started and all their configurations are processed.
- G0120 – The format used to store the configuration must be a human readable text file. It should be easy to read.
    - (### YAML is quite popular, but JSON would be ok as well, XML …)
- 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 – A Coordination must be used to set the 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.

6 Document Support

6.1 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/

6.2 Author's Address

Name	Peter Kriens
Company	aQute
Address	9c, Avenue St. Drezery
Voice	\+33467542167
e-mail	Peter.kriens@aQute.biz

6.3 End of Document