RFC-218 Configurer

Draft

20 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

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,

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

0.5 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>*

Revision	Date	Comments
Update	*DEC 17 2015*	*Updated based on EEG call* *Rewrote most parts* *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

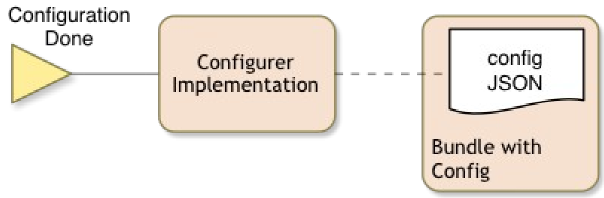
The Deployment Admin specification developed for OSGi Mobile provided the means to carry configuration information inside a JAR via the means of *resource processors*. The Auto Configuration specification defined how an Autoconf resource processor could get an XML file from a Deployment Package. Since 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*. 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 we could introduce a new special character as outlined below. We pick '#' for now.

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

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

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

These methods require a factoryPID and an alias argument. The method still generates a PID however the generated PID has the form: `alias#generatedPart` or even `factoryPid#alias`. This ensures that the *Configuration Admin* can still guarantee uniqueness of the PID. If a configuration with the given combination of factoryPID and alias 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 and binding works as defined for `getConfiguration`.

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

```
    ConfigurationAdmin ca = …;
    ca.listConfigurations("(&(service.factoryPid=FACTORY_PID)
                             (service.pid=ALIAS#*))");
```

5.2 Configurations

The Configurer service is a service defined by this specification to process configurations stored in a bundle.

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 Configurer uses `Bundle.findEntries` on the bundle requiring the above capability to find all files ending in `.json` in the configured paths. Files with other endings in the configured paths are ignored. The default configured path is `OSGI-INF/configurer`. If the bundle wants to store it's configurations at a different location within the bundle it can specify the attribute `configurations` on the require capability clause. The value of that attribute must be of type String or List of Strings. Each value represents a path. The path is always relative to the root of the bundle and may begin with a slash. A path value of slash indicates the root of the bundle. If a path is configured several times, it is only processed once.

```
Require-Capability: osgi.implementation;
                    filter := (&(osgi.implementation="osgi.configurer")
                                (version>=1.0)(!(version>=2.0)));
                    configurations=OSGI-INF/configurations
```

If a configured path does not exist, a warning should be logged. The found JSON files from all configured paths are processed in lexicographical order based on the full path.

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 configurations during development, for testing and in production. The Configurer gets the active environments by calling `BundleContext#getProperty("configurer.environment")` on the bundle context of the Configurer bundle. The value must be a comma separated string 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. Whitespace characters before and after each environment in the property value are ignored. Duplicate entries are removed, building a set of active environments. The environment with the name `always` is always active and added to the set. Configurations which should always be applied independent of the environment the framework is running should associate their configurations with this environment named `always`.

5.2.4 Configuration File Format

Each configuration file processed by the Configurer is in JSON format. In order to allow comments within the file, the file is first run through a preprocessor processing the file as described for JSMin (http://www.crockford.com/javascript/jsmin.html) in order to remove all comments. If the result of the preprocesing is invalid JSON, the file is ignored and an error should be logged.

The JSON object must conform to the following format:

```
{
    "always":
        "PID A" : {
                    ...
                    },
        "PID B" : {

                    ...
                    }
    ],
    ...
    "dev" :
      ...
}
```

The JSON object is a map of environments. The key is the name of the environment, the value is a map of configurations. The key of the configuration map is the PID for singleton configurations and `factoryPID#ALIAS` for factory configurations. This key is referred to as the *configuration identity* which uniquely identifies a configuration within the Configurer. The value is a map with the configuration properties:

```
"my.componnent" : {
                        ...configuration properties
                    },
 "my.factory#foo" : {
                        ...configuration properties
                    },
 "my.factory#bar" : {
                      ...configuration properties
                    }
```

Properties starting with the prefix `":configurer:"` are reserved properties by the Configurer specification and might influence the processing of the configuration. These properties are filtered out by the Configurer before the properties are passed to *Configuration Admin.* All other properties of the configuration map will be used for the dictionary passed to *Configuration Admin*:

```
{
    "email" : "something@somewhere.com",
    "port" : 300
}
```

The value of a configuration property for *Configuration Admin* must be the same type as the set of Primary Property Types specified in the chapter “Filter Syntax” in the OSGi Core. On the other hand, JSON only supports a basic set of scalar types (String, Double, Long, and Boolean), arrays and maps. Without any further type information for the Java type, 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	String containing the JSON

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 allowed types are: the scalar types (String, Integer, Long, Float, Double, Byte, Short, Character, and Boolean), the primitive types (int, long, float, double, byte, short, char, boolean), arrays of a scalar or a primitive type and collections of the scalar types. If any other type is specified, the configuration is ignored and this should be logged as an error. For collections, the Configurer picks a suitable implementation.

```
{
    "port:Integer" : 300,
    "an.int.array:int[]" : [2, 3, 4],
```

```
    "an.Integer.collection:Collection<Integer>" : [2,3,4],
    "complex" : {
                    "a" : 1,
                    "b" : "two"
                }
}
```

For the actual type conversion, the conversion follows the rules as outlined in RFC 215 Object Conversion.

5.2.5 Configuration Ranking

Bundles are processed sequentially by the Configurer. It is undefined in which order the bundles are processed by the Configurer. To better control the situation where more than one bundle provides a configuration for the same identity and ensure a predictable and reproducible system configuration, configurations have a 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 configuration 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`. 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 set to 0. This case should be logged as a warning.

5.2.6 Configurations from the Runtime Environment

When the Configurer service starts, it calls `BundleContext#getProperty("configurer.initial")` on the bundle context of the Configurer implementation to get a configuration from the environment. If a value for this property is available it is interpreted as a URL. The Configurer will try to resolve the URL and read a JSON object from that URL. If the URL can't be resolved or any error occurs during reading from that URL, an error should be logged and this property is ignored. If the url starts with the protocol `json` the part after the colon is directly interpreted as JSON.

If the read JSON is valid as defined in this specification, this configuration is applied by the Configurer. If the jSON is invalid, an error should be logged.

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 applied on startup of the Configurer. If on a follow up start of the framework a different configuration is provided, this is treated like a bundle update with a different configuration.

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. Therefore the Configurer should require the *Configuration Admin* service through a service capability.

When the Configurer service is started, it processes all resolved bundles and applies the configurations. The Configurer also checks whether an activated bundle from a previous run has been uninstalled. In that case it removes the configurations installed from this bundle according to the policy. The check can be done by simply checking if a bundle with a given bundle id is not present in the framework anymore.

While the service is running, when a bundle transitions from the installed into the resolved state, the Configurer applies the contained configurations. This covers both cases, installing and updating a bundle. If a bundle gets uninstalled, the Configurer removes the configurations installed from this bundle according to the policy.

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 Applying Configurations from a Bundle

The Configurer performs the following steps if configurations from that bundle should be applied:

- If the Configurer did previously process 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 processed and no further step is performed.

- The set of configurations defined within the bundle is calculated by respecting the current set of active environments.

- If this bundle has been processed 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 removed according to their policy.

- All configurations from the new set are applied.

5.3.2 Processing Configuration Files

A set of configurations to be applied by the Configurer is calculated for each bundle. This is done in two steps: in the first step, the contents of all files are merged, in the second step the definitions from the environments are condensed into a single set.

As defined above, all JSON files are processed in lexicographical order based on their full path. The files are read one after the other. A map M is built with the active environments and their configurations. For each environment E read from a file, the environment is processed as follows:

- If the environment name of E is malformed, E is ignored. This should be logged as an error.
- If the environment name of E is not in the active set, E is ignored.
- If M does not contain an environment with the name of E yet, E is added as is to M.
- If M contains an environment A with the name of E, these two environments are merged by processing each configuration C from E:
  - If the configuration identity for C is invalid, C is ignored. This should be logged as an error.
  - If A does not contain a configuration with the identity of C, C is added to A
  - If A contains a configuration Y with the configuration identity of C, then all properties from C are added to Y, potentially overwriting existing properties of Y.

In the second step, the set of configurations S is built based on M: the environments of M are now processed in lexicographical order based on their name. For each configuration C within an environment the following actions are performed:

- If S does not contain a configuration with the identity of C, C is added to S
- If S contains a configuration A with the identity of C, all properties from C are added to A, potentially overwriting existing properties.

The Configurer processes the resulting set of configurations S and applies them: the configurations are processed in lexicographical order based on their identifier. Each configuration is processed according to its configuration ranking and policy. The policy can be defined with the property `:configurer:policy`. Allowed values are `default` and `force`. If no property is specified or an invalid value is specified, default is used. If the value is invalid, an error should be logged.

The configuration is applied according to the following table:

	Policy **default**	Policy **force**
No configuration in config admin	Configuration is added	Configuration is added
Configuration added by Configurer: - with higher ranking - lower ranking, unchanged (*) - lower ranking, changed (*)	 No action Configuration is added No action	 No action Configuration is added Configuration is added
Configuration not added by Configurer	No action	Configuration is added

(*) If the Configurer adds a configuration to *Configuration Admin*, it keeps track of the change count. Comparing the current change count with the stored change count allows to find out, whether the configuration has been changed from outside of the Configurer.

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 configurations S together with the last modified time of the bundle and the change count of added configurations.

5.3.3 Removing Configurations

When the Configurer removes configurations for a bundle, the Configurer uses the calculated set of configurations. Each configuration is processed according to its policy:

	Policy **default**	Policy **force**
Configuration added by Configurer - unchanged (*)	 Remove, add configuration with lower ranking if available	 Remove, add configuration with lower ranking if available
- changed (*)	No action	Remove, add configuration with lower ranking if available

The Configurer removes all internal state for that bundle.

5.3.4 Overwriting Configurations vs Merging

The above described approach overwrites configurations instead of merging them. The idea behind this is that a new bundle version contains a new version of a configuration. Example:

Bundle Version 1 contains a configuration for PID A with properties a=1, b=1, and c=2 – configuration ranking is 1

Bundle Version 2 contains a configuration for PID A with properties a=2, c=2, and d=2 – configuration ranking is 2

With overwriting configurations, regardless of the order in which the bundles are installed, the final configuration will be the one from bundle version 2 without the property b.

If a merging approach would be used and version 1 is installed first, and version 2 is merged into version 1, handling the removal of property b is tricky. A pure merge would not remove property b. Therefore, depending on the order, either property b is available in *Configuration Admin* or is not.

Using overwriting makes the processing predictable and cleaner.

5.4 Standalone Configurations

The configuration file format in section 5.2.4 defines a portable representation of configurations in *Configuration Admin*. Whilst the Configurer extender implementation is necessary to process these configurations when they are packaged inside a bundle, these files can also offer significant value to other tools for deployment and management when standalone.

For example:

1. A deployer wishes to install an OSGi Application subsystem, which requires configuration to run. Currently the deployer would have to provide configuration as a separate step, in some provider specific way. With the addition of a standard configuration format the configuration(s) can be included inside the ESA, or referenced from an external repository. The subsystems implementation can then deploy the configuration into the scope of the Application Subsystem at the same time as deploying the application bundles.

2. A runtime assembly tool wishes to package and deploy an OSGi framework, a set of bundles, and one or more configurations into a single executable JAR file. The bundles to be packaged are located in an OSGi repository. In order for the tool to successfully resolve and assemble the runtime it needs to know:

    a) Does a bundle require configuration for a particular PID?

    b) What PIDs are offered by the other resources in the repository?

5.4.1 Configuration capabilities

In order to support the various configuration use cases it is necessary to define requirements and capabilities for the Configurer, the configurations, and the configuration bundles.

For configuration bundles the requirements and capabilities must be provided using the standard Require-Capability and Provide-Capability headers in the manifest file. Configuration files do not have a location to add requirement and capability metadata, therefore the metadata must be automatically generated by the repository if a standalone configuration file is added.

5.4.1.1 Common Requirements and Capabilities

Both configuration files and configuration bundles offer configuration for one or more PIDS. This capability is modeled using the `osgi.configuration` namespace. Configuration files and bundles should define Capabilities for each configuration that they define, and each capability should have resolve time effectiveness.

The osgi.configuration namespace:

Attribute	Required	Default Value	Purpose
service.pid	No	N/A	Defines the PID of the configuration
service.factoryPid	No	N/A	Defines the factory PID if this is a factory

Attribute	Required	Default Value	Purpose
			configuration

Note that at least one of service.pid and service.factorypid must be defined. If the configuration is a standard configuration then only the service.pid is used. If the configuration is a factory configuration with an automatically generated identity then only the service.factoryPid is used. If the configuration is a factory configuration with a specified identity then both the service.pid and service.factoryPid are used.

In order for configurations to be deployed it is necessary for there to be an active ConfigurationAdmin implementation. It is therefore important that the configuration files specify the following requirement:

```
Require-Capability:
osgi.implementation;filter:=”(&(osgi.implementation=osgi.cm)
(version>=1.6)(!(version>=2)))”;effective:=”active”
```

The version range used in this requirement may be expanded downward if the configuration does not require features from a more recent *Configuration Admin*.

This require capability is not necessary if this is a configuration bundle which requires the Configurer as the Configurer in turn requires the *Configuration Admin*.

5.4.1.2 Standalone Configuration Capabilities

In addition to the common requirements and capabilities a standalone configuration needs to provide two other capabilities when in a repository:

1. An osgi.content capability. The OSGi content namespace is easy to automatically generate for a configuration file. The one missing value is the mime type of the configuration file, which will be defined as `application/vnd.osgi.configuration`

2. An osgi.identity capability. The osgi.identity capability requires that each resource define a symbolic name and version. Currently this metadata cannot be automatically derived from the contents of the configuration file, and would have to be based on the file name.

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

8.1.1 Placeholders

The following idea of placeholders for framework properties was dropped as the value gets replaced at the time of processing through the Configurer. For example on a restart with a different value, the configuration is not updated and still points to the value from the previous start. The only way to solve this in a useful way is to add support for placeholders directly into *Configuration Admin*.

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.

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