RFP 102 Modifiable Bundle Metadata

Draft

10 Pages

Abstract

The success of the OSGi Framework has enabled more and more use cases. However, a significant number of these use case require changes or additions to the bundle's metadata. This RFP investigates these use cases and categorizes the problems that need to be solved to make widen the applicability of the OSGi Framework in more areas. It then lists the requirements that any solution should fulfil.

0 Document Information

0.1 Table of Contents

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	AUG 2007	Peter Kriens – Initial draft

1 Introduction

Due the wider adoption of the OSGi Framework the OSGi Alliance sees more and more demands for new use cases. Some of these use cases require changing the OSGi Framework, a process that takes a lot of time and will never be able to cover all requirements because many are too specific for a single use case. During R4 development, it became more and more clear that there was a need to modify, add, or augment the metadata that was specified in the manifest of a bundle, after the bundle was installed. This model would allow *handlers* to provide any needed semantics.

It was therefore decided by CPEG that in preparation for R5 an RFP should be written to create an inventory of the issues that this *manifest rewriting* (the working title) should encompass. This RFP is the result of this decision.

This RFP still has a number of open issues. This RFP describes an area that is highly complicated because it interacts intricately with the OSGi Framework on many levels. These open issues have to be worked out before this RFP is published.

2 Application Domain

The OSGi Framework, as defined in R4, is a comprehensive framework to manage the lifecycle of bundles. It allows the installation of bundles, it resolves their dependencies, it allows bundles to be started and stopped and finally to be uninstalled. The framework derives the metadata to perform this lifecycle management from a manifest that is stored as the first file in the bundle's JAR/ZIP file.

The manifest consists of headers. A manifest contains a *main section*, which contains the metadata for the bundle. It also contains a *name section* that has a set of headers for each file in the JAR file. The headers that the framework recognizes are defined in the specification, however, other headers are allowed as long as they do not conflict. These extraneous headers can be interpreted by *extenders*, management systems, tools, or the bundle itself. The OSGi Framework API provides full access to the main section headers.

Extenders are bundles that use the life cycle events from the framework as well as its introspection API to extend the functionality of a bundle. They do this by inspecting any bundle that is installed; reading the manifest headers or additional metadata in the bundle's JAR. For example, Spring-OSGi is an extender that creates a Spring Application context based on files in the META-INF/spring directory. This powerful model is enabled because the OSGi Framework has a very well defined API and event model for installing and uninstalling applications. That is, bundles can be guaranteed to find out about the installed state of other bundles.

The OSGi Framework specification allows the installation of any JAR file, even if the manifest is absent or if it contains a manifest that has no OSGi specific headers. Such JARs are not recognized by the Framework as bundles and do not interact on the module or service level. These JARs can be used by extenders but the Framework will not perform any class loading or dependency resolution due to the lack of metadata.

2.1 Terminology + Abbreviations

Metadata	Information about the bundle. This is normally the information in the main section of the bundle's manifest
Metadata provider	A bundle that provides metadata to another bundle
Extender	A bundle that extends the functionality an other bundle by inspecting the metadata or contents

3 Problem Description

3.1 Problem Areas

The OSGi framework has become a very general class loading framework that is extremely useful for non-OSGi applications. However, all OSGi semantics are provided through manifest headers which can not be modified on the Framework itself. This binds the meta-data very statically to the manifest headers.

There is a growing need to modify the metadata of a bundle during- or after it is installed. This need is driven by a large number of very different use cases. A key issue that drives these use cases is to use a bundle as-is, without opening up the ZIP file and rewriting the manifest. This section will look at the different use cases that drive the requirements for changing the bundle's metadata dynamically.

3.1.1 Invalid or non-optimal manifest

In most cases it is strongly desirable to use artifacts like JAR files as they are. However, when the metadata of the bundle is invalid or non-optimal the only way to correct is, is to recreate the bundle. This can be quite cumbersome, especially when the bundles are signed by another party. For example, a bundle could specify its dependency on another bundle by version and a package by version. Sometimes it may be desirable or necessary to override the specified versions because a certain combination was found faulty or was tested and shown to be correct. The framework should have some mechanism to allow the requested version information to be overridden.

3.1.2 Foreign Applications

The Java world has spun off many different application models: WAR, EAR, Midlet, Xlet, Applet, and likely others. Each of these application models have their own variations and versions. All these formats use the JAR format as their delivery artifact. It is therefore possible to install a Midlet or EAR file in an OSGi Framework. These JARs will not be able to share code (neither import nor export) because they lack the appropriate import and export headers.

This problem is opportune in MEG. MEG supports the installation of Midlets as bundle but requires the OSGi headers on top of the Midlet headers. Unfortunately, several headers overlap between the Midlet specification and the OSGi specification which makes the manifest unnecessary messy and verbose.

3.1.3 Aspect Oriented Programming

Aspect Oriented Programming (AOP) can modify and add to the byte codes of a bundle by weaving in *aspects*. Obviously, these changes can have their own package dependencies, which are not mentioned in the bundle's dependencies. The weaver must therefore be able to add new dependencies to the bundle in runtime.

A generic AOP language like AspectJ will have the requirement to add additional imports and potentially exports. However, the problem is wider than this. There are many libraries that create proxies to use some crude form of aspect oriented programming. A good example is the Hibernate Object-Relational mapper. The way it creates a proxy causes the bundle using Hibernate to become dependent on classes that the bundle does not use itself. The bundle programmer could add those imports but these imports could change with a newer version. The situation would even be worse when JPA was used and Hibernate was the persistence provider. In such a case the bundle programmer would have no chance to do it right. There is therefore a need to add additional imports based on the actual wiring that is being made during resolving.

Also Spring AOP uses dynamically generated proxy classes to wrap objects (such an OSGi service). These proxy classes can be advised to also implement additional interfaces (beyond those implemented by the wrapped object). So the class loader which loads the dynamically generated proxy class needs access to the classes of the bundle the wrapped object comes from as well as the bundle providing the advice. Spring-OSGi created a chained class loader which chains the bundle class loader and the advice's class loader together. Since the bundle's class loader is not available via an API, they do something similar to http://wiki.eclipse.org/index.php/BundleProxyClassLoader_recipe to chain the class loaders. Dynamic-ImportPackage is not appropriate here since the bundles do not know a priori that this AOP proxying will be occurring and Dynamic-ImportPackage is too promiscuous.

3.2 Manifest rewriting

The key function that allows many of the previously described problems to be solved is if an extender can rewrite any of the manifest headers. This encompasses adding headers, adding clauses to headers, deleting headers, deleting clauses from headers, modifying headers, and modifying clauses or attributes in headers. The following section discusses the main OSGi defined headers and take a look at how they could be rewritten.

3.2.1 Simple Headers

Simple headers only exist for informational purposes. These headers do not alter the behavior of the Framework, they only modify the result of the BundleContext.getHeaders call. These headers are: Bundle-Category, Bundle-Description, Bundle-Name, Export-Service, Import-Service, Bundle-DocURL, Bundle-ContactAddress, Bundle-Vendor.

3.2.2 Content Descriptors

Content descriptors tell something about the files/directories in the JAR by designating a number of files and decorating them with a number of directives and/or attributes. They are usually lists of files/directories with a set of properties and or filters. Changes could add files, remove files, or change the ordering. Adding files/directories to the header is likely to require not only changes in the manifest but also the possibility to add files to the JAR as well.

Headers in this category are Bundle-NativeCode, Bundle-Classpath, Bundle-Localization

3.2.3 Modularization

There are many headers that influence the imports and exports of classes like Import-Package, Export-Package, and DynamicImport-Package, Require-Bundle, Fragment-Host. These headers are likely to be targets for rewriting. The AOP and hibernate case clearly require adding packages the Import-Package header.

3.2.4 Identity

The Bundle-Version and Bundle-SymbolicName define the identity of the bundle. It is likely that they can be set only once, and only if not set.

3.2.5 Class Names

Headers like Bundle-Activator designate a class in or outside the bundle. Theses types of headers might require extra imports or the addition of packages.

3.2.6 Other Framework Headers

- Bundle-RequiredExecutionEnvironment
- Bundle-UpdateLocation
- Bundle-Localization
- Bundle-ManifestVersion
- Bundle-ActivationPolicy

3.2.7 Non-Framework Headers

Several OSGi service specifications use headers that are part of the manifest.

3.2.8 Named Section

Each manifest also contains a named section. The name section names a resource in the JAR file and associates it with a set of headers. This feature is mainly used for signing the bundle, however, it can be used for other purposes as well.

3.3 Non Functional Issues

3.3.1 Multiple Metadata Providers

In the current model there is a single source of the metadata: the bundle. When multiple parties are allowed to modify the metadata it will be necessary to create some ordering and conflict detection scheme with possible

resolution. For example, one metadata supplier could provide local package versions override while another metadata provider provides extra imports for hibernate.

Ordering of metadata will allow one party to rewrite the metadata which is then given to the next in line. This requires that ordering is deterministic. Metadata providers must be able to indicate that they have priority over other metadata providers. Later metadata providers must be able to see the original and changes made before they had access so that changes can be logged for debugging.

Metadata providers must be able to indicate that a change is an addition, deletion, or change. The RFC must indicate what modifications are compatible and what modifications are invalid.

3.3.2 API versus String based

There are a number of distinct ways to solve the problem of rewriting the metadata.

1. String/Stream based. A metadata provider can read the manifest as a String or stream, parse it, and return a String. However, this is error prone, requires a lot of parsing code in the meta providers, and it is inefficient.

2. Manifest object. The metadata providers could get a manifest object and modify this object accordingly. This is more efficient as 1, however, it requires still significant parsing of the headers themselves. This solution also makes it hard to detect conflicts.

3. Map based API. Almost all headers can be represented as a Map. The advantage is that parsing is done only once and it will be easy to detect conflicts. The disadvantage is that not all OSGi headers and many other headers (native code) do not map comfortably to a Map.

4. Semantic API. Each OSGi header could be represented as an API. That is, there would be an API for imports, exports, required bundles, etc. Unknown headers could be treated as strings. This would have the advantage that it would be easy to use, detect many errors during compile time, and it would allow full conflict detection. Disadvantage is that this is a lot of specification work and will never be as flexible as full header rewrites.

It is not clear at this moment what the strategy provides the best solution. A combination of the above might work best. That is, a specific API for the module layer might be combined with a more generic facility for the other headers.

3.3.3 Timing

There are several possible phases when the manifest can be rewritten. During install, update, or before resolve. Rewriting before the bundle is visible in the install state has the advantage that no bundle can see the stale metadata and falsely trigger. It will also be straightforward to persist the changes which will improve performance. However, this means that metadata providers must be present during the install of the to be rewritten bundle. This will make it hard to add metadata providers after a number of bundles have been installed. However, it is of course possible to update a bundle so that it goes through the install phase, allowing its metadata to be rewritten.

Rewriting just before the resolve phase has the advantage that changes can be deferred until more is known. For example for AOP, this is likely to be a requirement because many of the imports depend on the combination of the aspects and the target.

It is not clear at this moment what the strategy provides the best solution.

3.3.4 Lifecycle Dependencies

Metadata modifications depend on the bundle that has set them. If this bundle gets uninstalled, these modifications must be undone.

3.3.5 Security

Changing metadata of a bundle has the potential to make far reaching changes in the security of the system. The following issues are foreseen:

1. Need for additional permissions by the rewritten bundle. For example, adding a package import requires the appropriate Package Permissions.

2. Security Identity. The OSGi Conditional Permission model uses signing or location to authenticate a bundle. Changes to its manifest should not fall under the permissions scoped by the identity of the bundle but must be attributed to the metadata provider. However, this opens a large number of security threats because the interaction between the metadata provider and the bundle will be hard to oversee.

3. Restrictions. Certain headers should not be modified. For example, Bundle-SymbolicName could be set but no bundle should be able to modify it. Other headers could have other restrictions.

4 Use Cases

4.1 Aspect Oriented Weaver

The AOP Inc. company sells a weaver for the AspectJ language. Their product looks for installs of bundles and checks if these bundles are applicable for weaving. If they are, it applies the appropriate aspects and checks for any required new imports that are caused by the aspects. These imports are added to the metadata together with any standard libraries required by the weaver.

A key issue in this use case is the timing. It is not clear where weaving takes place, but it is likely to be deferred to class loading time to minimize the overhead. However, at that time, rewriting the manifest is way to late.

4.2 Midlets

NoMo provides a Java OSGi environment implementing JSR 232. They are running standard Midlets as normal bundles. When a Midlet is installed, their Midlet extender is rewriting the manifest to add the required headers. Because Midlets have a defined Java environment (CLDC/MIDP) their Midlet container just adds the appropriate packages to the manifest. This will allow the Midlet to run in the standard OSGi environment.

4.3 Bnd

Bnd is a tool to add the OSGi manifest headers based on an analysis of the class files. Any references are translated to imports. Bnd can be directed in an OSGi Framework to detect the installation of non-OSGi JARs. It

will analyze the JAR from its location and calculate the appropriate headers. It will then rewrite the manifest for this bundle, making it a first class citizen in the OSGi Framework.

5 Requirements

5.1 Functional

- F001 – Provide a means to change the metadata for a bundle by modifying the metadata during install.
- F002 – Provide a means to order the changes when multiple providers want to change the same metadata.
- F003 – Detect conflicts if the same metadata is modified by multiple providers
- F004 – Metadata changes must be persistent between framework invocations
- F005 – If a metadata provider is uninstalled (stopped?) any changes to the metadata made by the bundle must not be effectuated
- F006 – A provider must be able to remove metadata changes it made earlier
- F007 – Metadata providers must be able to add/delete/replace resources to bundles.
- F008 – Metadata providers must be able to initiate the metadata rewrite of a bundle

5.2 Non Functional

- N001 – Any solution must be able to scale to over 5000 bundles. This implies that no work can be done during startup/resolving.
- N002 – Rewriting metadata should not incur any overhead for normal execution.

5.3 Security

- S001 – Bundles must be able to provide extra permissions to the bundle whose metadata is rewritten. However, these permissions must stay inside the security scope of the metadata provider. That is, a metadata provider may not grant more permissions that it has itself
- S002 – If a bundle is signed, its signature must be ignored if the metadata is modified
- S003 – Metadata should be protected with appropriate permissions
- S004 – Bundles must not be able to rewrite their own metadata

6 Open Issues

- How do we handle signed bundles? Changing their Manifest is opposite to the intent of a signed bundle?
- How to handle multiple metadata providers. Ordering enough? Conflicts?
- What is done when a metadata provider is uninstalled or stopped? This can have disastrous effects!
- When is the metadata rewriting happening? During install seems best, but some dependency knowledge is not available until class loading

7 Document Support

7.1 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

7.2 Author's Address

Name	Peter Kriens
Company	aQute
Address	9c, Avenue St. Drézery, Beaulieu, FRANCE
Voice	+33467542167
e-mail	Peter.Kriens@aQute.biz

7.3 End of Document