RFP 118 - Multiple Extender Bundles

Draft 0.3

9 Pages

Abstract

The extender pattern is increasingly being used to enhance deployment of bundles. For example, RFC 124 (Blueprint Service) defines an extender, and extenders are commonly used to recognize bundle artifact patterns such as the presence of a web.xml file or a persistence.xml file and take some action. When multiple versions of an extender bundle are present, and/or multiple extenders of different types all want to react to the same bundle event, there is no mechanism to determine how conflicts between multiple extenders can be resolved, or cooperation amongst multiple extenders achieved.. Given the increasing use of the extender pattern, we need a common solution to this problem to avoid proliferation of ad-hoc and incompatible point solutions.

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 6.1.

`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	*NOV 28 2008*	*Adrian Colyer, SpringSource, adrian.colyer@springsource.com*
0.1	*Jan 7th 2009*	*Incorporated feedback from Peter Kriens*
0.2	*October 7th 2009*	*Incorporated feedback from Dublin F2F*
0.3	*November 2nd 2009*	*Added "possible" to requirement E0009*

1 Introduction

During the development of RFC 124 it was noted that conflicts could arise if multiple Blueprint providers would be present at the same time. During the discussion of this problem, it became clear that this problem was wider than just Blueprint. OSGi tries to prevent singletons but extenders could easily carry some of the problems of singletons. This RFP is an inventory of those problems and lays down a number of requirements for a possible solution.

2 Application Domain

The OSGi extender pattern [1] defines a patten in which an extender bundle listens to bundle lifecycle events in order to take some action on behalf of other bundles. For example, the Blueprint Service extender bundle defined by RFC 124 creates a module context on behalf of starting bundles that meet its criteria, and destroys that context when the bundles are stopped again.

There are two parties with the extender pattern. The bundle that is extended is called the *target* bundle and the bundle providing the extender code is called the *extender*. An extender can be present in a certain *version* and there can be many different *extender types*. A target can be extended by multiple extender types. An extender that is processing a target is said to be *extending* this target. For a given extender type, normally only one version should be extending a target.

Extender interaction with a target bundle can be based solely upon resources, but quite often there is also a class-based interaction. Normally shared classes are properly specified in specification packages and they are shared using the standard OSGi modularity rules.

The advantages of the extender pattern are:

- Maximum decoupling between the extender and target. Often, the only interface defined between the two parties is some metadata in the bundle and the OSGi life cycle events. A target bundle can be installed without an extender and an extender can be installed without any targets.
- Simplification. Bundles can usually provide complex functionality by adding a small amount of metadata at an agreed place.
- Reduction. The extender pattern can often be used to make the target as small and concise as possible.
- Reliability. Normally, the extender bundle is heavily tested and widely used, making it highly reliable. Because the extender bundle provides so much of the (boiler plate) functionality of a target bundle, and especially the problematic life cycle areas, systems with the extender pattern seem to be more reliable. For example, in many operating systems adding and removing applications is not an embedded operation but implemented on top of the OS. This makes it sometimes very hard to uninstall erroneous installations.
- No installation/uninstallation phase. Traditionally, applications perform scripts at install and uninstall. These scripts copy files all over the file system and add entries to registries to use certain features of available subsystems. Because the life cycle events are a guarantee of the operating environment, the extender pattern makes these scripts unnecessary and their associated problems.

The extender pattern currently has no visible support in the OSGi specifications, and hence each extender is free to make its own interpretation of the pattern. The behavior when multiple extenders are present is unspecified and so each extender implementer must make unilateral decisions. Normally, this case is ignored and left as a problem for the deployer to solve by installing the right set of bundles.

Extenders are increasingly popular and in the 4.2 release there is a Bundle Tracker class that makes the implementation of extenders very simple.

An extender bundle typically reacts to bundle life cycle events for bundles that meet some criteria. Each extender defines its own criteria (for example, the presence of certain resources within the bundle). As an example, the Blueprint Service extender reacts to the presence of XML files in the OSGI-INF/blueprint folder and creates application objects on behalf of the bundle based on the declarations found there. The Pax Web Extender [2] reacts to the deployment of bundles in WAR file format, parses their web.xml file and registered application components with the HttpService. Other extender models are likely within the OSGi Compendium services to support web applications (RFC 66), persistence configuration (RFP 115), and some features of Distributed Services (RFP 119).

Current popular extenders are:

- Spring DM, now translated into the OSGi 4.2 Blueprint Service. The XML used to configure the Module context is placed in a well-known location in the jar.

- Metatype Repository. The XML definitions of the attributes and classes are picked up from a well-known place in the JAR.
- Conditional Permission Admin. Reads permissions.perm from the bundle
- Declarative Services. The XML to configure the components is read from a location specified in the manifest.
- PAX Web Extender. Detects a WAR and installs this in the proper way
- Eclipse Extension Points. Read plugin.xml from bundles. Although there is one caveat, Eclipse does not handle the start/stop life cycle. It extends any resolved bundle.

2.1 Terminology + Abbreviations

- WAR file : Java, Enterprise Edition archive format for web applications

3 Problem Description

The basic extender model, in which a single extender bundle is deployed and reacts to bundle lifecycle events is well understood. However, many challenges remain unanswered in the presence of zero or more extenders for the same target:

Conflict resolution. In a given OSGi framework it is possible that there are multiple extenders (versions and types) that can extend a given target. Currently, there is no conflict resolution and there are (so far as we know) no extender types that have set clear rules about this conflict resolution, even though each extender type could specify such rules to be used within one type. For different types, there is no possibility to resolve these conflicts currently. Issues around conflict resolution are selection (which extenders should run on the target) and ordering (in what order should they run).

Life cycle interaction. The life cycle interaction has a number of cases that can result in a system not running for no obvious reasons. For example, a target is started but the extender not. This results in a system where the target seems ok but does not provide its intended functionality.

Class space inconsistencies. It is possible that the target and extender exchange objects of certain types but their class loaders are wired to different exporters. I.e., they live in different class spaces. This means that objects created from the target are not compatible with the extender and vice versa, causing class cast exceptions.

Implicit dependencies. Normally a target should not concern itself about the extender. There are many legitimate cases where a bundle could have multiple functions and only a part of that functionality is provided through an extender. It is an often made mistake that bundles want to handle their own life cycle, making the overall application (= set of installed bundles) less robust because they make assumptions that are violated by the deployer. However, at the same time deployers are often confused that a target bundle seems to do nothing because their dependency on the extender is not expressed nor verified. There are also cases where a bundle must be able to change its course when an extender is available or not. Overall, there is a need to be able to diagnose the system by having more explicit dependencies (mandatory and optional) from the target to the extender. These dependencies should handle both extender types as well as version ranges.

The OSGi dependency metadata is very suitable for managing a system, specifically finding the right set of bundles that provide the required application functionality. Unfortunately, the extender model is so decoupled that there is no direct dependency. That is, a bundle can be deployed without the dependency to its extender being visible.

Formalizing the extender model to give common answers to these issues so that all extenders can then follow them will benefit the OSGi user community by making the behavior of on OSGi Service Platform more predictable, consistent, and administrable. Failure to do so will mean that as ad-hoc extender mechanisms proliferate, deployment of multiple extenders may cause unexpected behavior and lead to incompatibilities that cannot easily be resolved.

4 Use Cases

The following use cases should be considered as important representative cases of the problem space, rather than as an exhaustive list.

4.1 Bundle requires an extender in order to function correctly

A bundle is installed into an OSGi Service Platform and started by a user. In order to function as intended by the user, it requires an extender to process its contents, but no such extender is deployed because the user was not aware of this dependency. The system informs the user that the target bundle can not operate without a specific type of extender bundle.

4.2 Bundle requires at least a certain version of an extender in order to function correctly

A target bundle contains within it a configuration file that is to be processed by an extender. The configuration file uses features only supported by version 2 of the extender. The system has both version 1 and version 2 of the extender bundle installed. The extender resolution mechanism resolves this conflict and only the version 2 extender is allowed to process the target.

4.3 Several implementations of the same extender are deployed concurrently

The ACME implementation of version 1.0.0 of an extender type is deployed concurrently with the BDNF implementation of version 1.0.0 of the same extender type. The bundle has a hard dependency on the BDNF implementation (discouraged!). The extender resolution mechanism ensures that BDNF is selected as the extender for this target. .

4.4 Multiple different extender types need to react to the same bundle lifecycle events

A bundle containing both Blueprint Service configuration and a web.xml file is deployed, and needs to be processed by both the Blueprint Service extender and a web container extender. The Blueprint Service extender should ideally have finished creating the application objects before the web extender registers the web application.

4.5 Extender requires type compatibility with Bundle

A target is deployed and bound to version 1 of the service package. However, its only matching extender has been wired to version 2 of the service package. This means that the target and extender live in different class spaces. The extender resolution mechanism detects this separation and makes sure the extender does not extend the target and notifies the user that the target can not be extended because of class space separation.

5 Requirements

- E0001 – It SHOULD be possible for a target to be processed by an extender without requiring any special manifest headers or general purpose extender configuration files. That is, the simple (and common) case should “just work”.

5.1 Extender visibility and explicit dependencies

- E0002 – It MUST be possible to discover the set of extenders that are active in a given service platform
- E0021 - It MUST be possible to determine the extension type(s) and version(s) supported by an installed extender
- E0005 – It MUST be possible for a target bundle to explicitly declare a mandatory dependency on an extender for a given extender type being present. Note – this is distinct to declaring a dependency on a particular extender implementation; a bundle should be able to depend on e.g. a Blueprint Service being present, without specifying any details about the provider of the Blueprint Service.
- E0006 – It MUST be possible for a target bundle to declare a version range for extender type versions with which it is compatible.
- E0015 – It SHOULD be possible to determine the set of target bundles currently managed by any given extender bundle
- E0022 - It SHOULD be possible for a provisioning system to determine that a given bundle is an extender, and the extension type(s) and version(s) it supports, without installing and starting the bundle.

- E0010 – It MAY be possible for a bundle to require a particular implementation of a given extender capability, or otherwise specify matching attributes that an extender implementation must support in order to be compatible.

5.2 Lifecycle interactions & conflict resolution

- E0007 - An extender MUST only extend a bundle when any constraints expressed by that bundle in terms of version (E0006) and matching attributes (E0010) are satisfied by the extender.
- E0009 – It MUST be the case that in the presence of multiple compatible extenders for the same extender type, all capable of extending a given target bundle, it is possible to specify that only one of the compatible extenders should extend the target.
- E0023 - The selection of extender in the case of E0009 must be deterministic.
- E0014 - There SHOULD be guidance given to extender implementors on the expected interactions between extender lifecycle and target bundle lifecycle.
- E0024 - There MAY be a convenience class which implements the extender life cycle conventions to assist extender implementors in following the desired model
- E0012 - There MAY be a mechanism by which extender ordering can be influenced in the case that multiple extenders all need to extend the same bundle.

5.3 Class space consistency

- E0008 – Extenders MUST only extend targets when the target and extender use the same class space for the classes that are exchanged between them.

6 Document Support

6.1 References

[1]. Kriens, P. The OSGi Extender Model, http://www.osgi.org/blog/2007/02/osgi-extender-model.html

[2]. Dreghiciu, A. The Pax Web Extender, http://wiki.ops4j.org/confluence/display/ops4j/Pax+Web+Extender

[3]. The OSGi Extender Model http://www.osgi.org/blog/2007/02/osgi-extender-model.html

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

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

6.2 Author's Address

Name	Adrian Colyer
Company	SpringSource
Address	Kenneth Dibben House, Enterprise Road, Chilworth, Southampton SO16 7NS, England.
Voice	+442380111500
e-mail	adrian.colyer@springsource.com

Name	Peter Kriens
Company	aQute
Address	
Voice	
e-mail	peter.kriens@aQute.biz

6.3 End of Document