RFP 128 META-INF/services support

Accepted

9 Pages

Abstract

The Sun SPI plug-in model, also known as the META-INF/services model, is widely used in the Java Platform. When used in an OSGi framework there are issues with this model. This RFP identifies these issues.

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	01/12/09	David Bosschaert, Progress Software. Initial version
0.1	March 2010	David Bosschaert, Progress Software. Updates following the Southampton F2F discussion.
0.2	August 2010	David Bosschaert, Red Hat. Updates wrt the experimentation done in Apache Aries and feedback from the Ottawa F2F discussion.
0.3	September 2010	David Bosschaert, Red Hat. Small updates, preparing for requirements vote.
1	29 October 2010	David Bosschaert, Red Hat. Moved to accepted state.

1 Introduction

The Sun SPI model is a plug-in model widely used in the Java platform. In fact, many implementations of JSR standards are plugged into the JRE using the Sun SPI model. However, there are issues with this model when used in an OSGi Framework.

This RFP identifies the issues and lists the requirements that a solution needs to support.

2 Application Domain

The application domain is any OSGi-based Java application that uses a third-partly library which relies on the Sun SPI model.

There are two sides to the problem, the SPI provider side and the SPI consumer side.

2.1 SPI Providers

SPI providers are typically Jar files that contain an implementation of a standardized interface. An SPI provider could be OSGi-aware or non-OSGi-aware.

2.2 SPI Implementation Consumers

SPI consumers are clients of a standardized API that use it to obtain an implementation which is provided through the Sun SPI mechanism. These clients could be OSGi-aware or non-OSGi-aware.

2.3 Terminology + Abbreviations

SPI – Service Provider Interface.

3 Problem Description

In Java the most widely used pluggability model is the Sun SPI model. This model allows plugging in an implementation of typically a standardized interface such as a JAXP compliant XML parser, A JSR 311 compliant REST reader or writer or a SOAP implementation.

3.1 FactoryFinder

The SPI model is generally based on the following mechanics, although the implementations vary slightly across the board.

It generally uses a 'FactoryFinder' class which tries to obtain a Factory class which binds it to a certain implementation.

The Factory Finder typically does two things:

1. First it tries to work out the name of the factory class implementation for factory with id x.y.Z, typically using the following algorithm:

a) It checks for the existence of a system property x.y.Z that might hold the class name.

b) It looks for a well known properties file in $java.home/lib which might hold a value for the x.y.Z key.

c) It tries to load the resource META-INF/services/x.y.Z using the Thread Context Classloader, the System Classloader or the classloader that loaded the Factory Finder class. If found it reads the implementation class name from the resource.

d) If all of the above fails it uses a fall-back class name which is typically hard coded in the JRE implementation.

2. Once a class name has been obtained, the Factory Finder will try to load the class.

a) In some cases an actual classloader is passed in to the Factory Finder in which case that classloader is used.

b) Otherwise the Thread Context Classloader is tried or if that isn't set the System Classloader is used.

c) If that fails the Factory Finder calls Class.forName() with its own classloader as the classloader argument.

This pattern has a number of disadvantages when used in an OSGi framework:

- A typical 3rd party library implementation that uses the SPI model relies on step 1c, where a resource is loaded from the META-INF/services directory through a classloader. In OSGi frameworks this resource is normally not visible outside of the library as only exported packages are made visible to classloaders outside of the bundle. Exporting the META-INF/services directory as a package is not an option as many libraries might have this directory but the OSGi framework can only resolve a package to a single bundle.

- Even if the class name can be obtained somehow in the OSGi framework, the loading of the implementation class will end up being challenging as it does not take the classloader of the bundle into account.

  This can often be worked around by setting the Thread Context Classloader to the bundle's classloader, but this requires modifications to the user's bundle code.

- To be loadable by another bundle the implementation class needs to be in a exported package of the providing bundle. This is generally undesirable as it exposes packages internal to an implementation outside its bundle, something highly discouraged in OSGi modularity.

- Besides the above issues, there is also a lifecycle issue in general with the SPI pattern as the Factory Finder often uses static variables and hence the Factory can only be set once in the life time of the VM in certain scenarios.

Since the Sun SPI model is highly prevalent in the Java library ecosystem and in fact the standard mechanism used within the JDK itself, an OSGi developer should not have to worry about getting around the problems with this mechanism. It should *just work*.

Other OSGi specifications have addressed this issue on a case-by-case basis, but a general solution to this problem is not available.

The OSGi Alliance should create a generic mechanism to deal with the Sun SPI model so that libraries utilizing this model can be used in an OSGi framework.

3.2 ServiceLoader

The JRE also contains a class called java.util.ServiceLoader which provides a general algorithm for this finding SPI implemetors. The algorithm is different than the one described above in that it only visits the META-INF/services. It doesn't involve system properties, nor files in the java home directory. It also doesn't have a default class name built in.

Additionally, the ServiceLoader class allows the client to obtain all the providers, where the FactoryFinder only selects a single one using the rules above.

3.3 JavaMail and non-empty constructors

The JavaMail API uses a SPI mechanism as well, but in a slightly different way. Classes are specified in a META-INF/javamail.properties file with a number of attributes, like this:

```
# JavaMail IMAP provider Sun Microsystems, Inc
protocol=imap; type=store; class=com.sun.mail.imap.IMAPStore; vendor=Sun;
protocol=imaps; type=store; class=com.sun.mail.imap.IMAPSSLStore; vendor=Sun;
```

The JavaMail framework expects the class specified to have a Constructor that has the following signature:

```
IMAPStore(Session session, URLName url)
```

When a consumer calls one of the Session.getStore() APIs this will result in the JavaMail framework to instantiate the correct class with the constructor as specified above.

Open question at this stage: how should this be supported in OSGi?

4 Use Cases

4.1 Using a library that relies on the Sun SPI model

Paul has obtained a library from a vendor that implements a standard API. The library is hooked into the JRE through a file in its META-INF/services directory. The library does not have any OSGi metadata associated with it, but Paul uses one of the freely available tools to wrap the library in an OSGi bundle with added OSGi metadata such as Bundle Symbolic Name, Import Package and Export Package directives.

However, when the users tries to obtain an implementation through its standard Factory API from within a bundle this fails as the Finder class is not able to find the provided implementation class name.

This is caused by the fact that the resource that specifies the implementation class in the META-INF/services directory isn't available through a classloader outside of the library.

4.2 Using an alternative implementation to the default provided by the JRE

The JRE ships with a number of default implementations that rely on the SPI model, however Linda would like to use an alternative one. In this case she wants to use a JAX-WS provider from Apache CXF rather than the one built in to the JRE. She obtains the implementation by calling `javax.xml.ws.spi.Provider.provider()`.

Outside of OSGi, Linda can achieve this by simply putting the CXF jar on the classpath, but inside an OSGi framework this doesn't work and even if the bundelized version of CXF is installed, the `Provider.provider()` API still returns the implementation provided by the JRE.

4.3 Using the OSGi Service Registry to find an implementation

Jim is writing an OSGi bundle which uses JAX-RS to invoke remote REST-based services. For this he uses JAX-RS implementations that advertise themselves through the Sun SPI model by implementing the `javax.ws.rs.ext.RuntimeDelegate` class and by providing a META-INF/services/javax.ws.rs.ext.RuntimeDelegate file. Jim's implementation is capable of using more than one provider implementation and as a matter of fact Jim wants to choose at runtime which implementation he is using based on certain parameters known to his application. Jim wants to obtain the implementation from the OSGi Service Registry as it provides him with the full list of available implementations plus additional metadata that allows him to make his selection.

4.4 Updating a Service Provider

Ronald is using JAXB in his OSGi bundles. He has come across a bug in the JAXB implementation which has now been fixed by his JAXB provider. He needs to keep his application running, so he wants to update his JAXB implementation without stopping his own bundles.

4.5 Obtaining a JDBC 4 DriverManager

JDBC 4 uses META-INF/services to register JDBC drivers. It is problematic to obtain these from within OSGi as is outlined in the following blog: http://hwellmann.blogspot.com/2009/04/jdbc-drivers-in-osgi.html .

Obviously, it would be better to use a OSGi-JDBC compliant Database Driver which is addressed in the current OSGi 4.2 specification chapter 125. This use-case addresses the use of non OSGi-compliant JDBC drivers.

5 Requirements

5.1 Provider side Requirements

- SP01 – SPI service providers should be automatically registered in the service registry.
- SP02 – The solution should work with providers that are not OSGi-aware. These providers will need to be bundelized, however.
- SP03 – Providers need to opt-in using a Manifest Header.

- SP04 – Bundelized providers should support the full bundle lifecycle and should therefore be able to be uninstalled.
-

5.2 Client side Requirements

- SP10 – Non-OSGi-aware clients should work without modifications to the source code. It is not expected that these clients will get OSGi lifecycle benefits.
- SP11 – OSGi-aware Clients should be able to use OSGi services to obtain SPI implementations. These clients must get OSGi lifecycle benefits wrt to the SPI implementations.
- SP12 – The semantics for non-OSGi-aware clients must not change. For example clients must receive new SPI instances for every `ServiceLoader.load()` invocation.

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

6.2 Author's Address

Name	David Bosschaert
Company	Progress Software
Address	158 Shelbourne Road Ballsbridge Dublin 4 Ireland
Voice	+353 1 637 2000
e-mail	dbosscha@progress.com

Name	David Bosschaert
Company	Red Hat
Address	6700 Cork Airport Business Park Kinsale Road Cork Ireland
Voice	+353 21 230 3400
e-mail	david@redhat.com

6.3 End of Document