Multiplexing JVM Singletons

Draft

13 Pages

Abstract

This RFC describes a proposed specification for multiplexing JVM singletons in support of running more than one OSGi Framework concurrently in a single JVM process.

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 9.1.

```
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
Additional revisions	JAN 12 2009	From RFC 138: bug 909: singleton interface for multiplexing (section 5.4.1 + javadoc) Erin Schnabel, IBM, schnabel@us.ibm.com Tom Watson, IBM, tjwatson@us.ibm.com
Initial	SEP 18, 2009	Split sections related to singleton multiplexing from RFC 138. Erin Schnabel, IBM, schnabel@us.ibm.com

1 Introduction

The OSGi platform is increasingly being used as a basis for composing complex runtime environments. Middleware providers are using OSGi implementations as the foundation of their runtimes, while application developers are looking to use OSGi standards and artifacts to build enterprise applications. As these two uses converge, questions arise over how to isolate applications from the runtime, or even certain applications from other applications.

RFC 138 was originally targeted at addressing the broad range of issues that arise when running multiple OSGi frameworks within a single JVM process. That RFC has been narrowed to focus on the concept of a composite: an abstraction of a nested framework used to provide isolation.

2 Application Domain

A framework instance is a discrete unit: it has its own class loading policy, and is responsible for resolving class loading constraints for all bundles that it "owns" (all bundles installed for that framework).

If there are multiple frameworks in a JVM:

- There may be more than one instance of a singleton bundle active in the JVM: each framework manages its installed bundles independently from the other frameworks-- one framework should not have to know that a singleton bundle has been instantiated by another framework in the same JVM.

- The Java mechanisms for extending the URL class allow handler factories to be specified once per JVM process. As bundles can have a much shorter life cycle than the JVM process as a whole, t he URL Handlers Service Specification requires frameworks to register **URLStreamHandlerFactory** and **ContentHandlerFactory** objects to act as proxies for handlers that are registered with the framework as services. If multiple frameworks are present in a JVM, some mediation would have to occur to ensure that the single URLStreamHandlerFactory and ContentHandlerFactory can multiplex between available frameworks.

- There is only one **Security Manager** reachable via the `java.lang.System` class. The Conditional Permission Admin (CPA) service interacts directly with the configured Security Manager to perform permission checks. The impact of any mechanism to adapt the Security Manager to a multi-framework environment on CPA condition evaluation would have to be well understood. At a base level, the discrete nature of each framework should be maintained: each framework should be able to establish, maintain, and evaluate its conditions, permissions and bundle protection domains independently and without knowledge of any other framework. Postponed conditions cannot be supported without multiplexing the Security Manager.

- Issues with class loader parenting and use of the context class loader are not new, nor are they particular to a multi-framework environment.

2.1 Terminology + Abbreviations

- Embedded framework: An OSGi framework started from within an existing Java process.

- Nested framework: An embedded OSGi framework started by an OSGi bundle. The lifecycle of a nested framework is bounded by the lifecycle of the launching bundle (in other words, it is the launching bundle's responsibility to stop and clean up the framework when it is stopped).

- Hosting framework: An OSGi framework that contains other framework instances. Since nested frameworks (in general) are started by arbitrary bundles, the hosting framework may not be aware of nested instances.

- Peer framework: An embedded framework at the same level of nesting as another embedded framework in a multi-framework environment: e.g. two nested frameworks started by the same bundle, or two child frameworks started by the same parent.

- Multi-framework environment: a JVM containing more than one framework, regardless of how those frameworks are related to each other.

3 Problem Description

Multiple frameworks cannot easily be present in the same JVM due to some of the constraints mentioned above. The focus of this RFC is the absence of a uniform, framework-agnostic way to deal with JVM Singletons like the registered URLStream and ContentHandler factories for URLs.

Some framework implementations (Eclipse Equinox, Apache Felix) have invented multiplexing handler factories that a) only handle frameworks from the same vendor, b) use reflection to ensure the right handler factories are being used, and c) rely on their custom handler factories to mediate calls to factory methods between frameworks. There are synchronization and resource management issues with this approach (what happens when a framework is restarted, etc).

Standardizing a set of rules and expected behavior will allow multiple frameworks, potentially provided by different vendors, to behave and interact consistently when run within the same JVM.

4 Requirements

1. The solution MUST define a framework-neutral way to extend the URL Handlers Service Specification to address a multi-framework environment.

2. The solution MUST define how framework services, like the Conditional Permission Admin service, interact with the JVM singleton Security Manager.

3. The solution SHOULD specify a common multiplexing mechanism that can be shared by other singleton services.

4. The solution MUST enable mixed-vendor, multi-framework environments: a framework must be able to host, embed, or be a peer of another vendor's framework in the same JVM.

5. The solution MUST allow frameworks in the same JVM to be of different OSGi versions.

5 Technical Solution

The proposed technical solution preserves the following assumption: each framework manages its own class space, and is responsible for managing the lifecycle and resolution of its bundles. There is an implicit restriction of visibility based on a framework's classloading structure: class reachability inside the framework is managed by bundle resolution and classloading rules; class reachability outside the framework is managed by standard classloading rules and hierarchies, and doesn't have visibility into a framework's class space without the assistance of special bridging classloaders.

5.1 Multi-framework environments

5.1.1 Peer embedded frameworks

Use case: A non-OSGi runtime hosts two or more applications. Each application uses an OSGi framework as a localized (isolated) mini-runtime: function is delivered, managed, and updated using bundles hosted in that framework. The frameworks (A and B) are isolated from each other, and could be from different vendors.

Launching and using peer frameworks within a single JVM is covered by the mechanisms introduced in RFC 132, however, JVM singletons will require cross-framework multiplexing: framework services like the Thread IO service won't help, as the frameworks are disjoint.

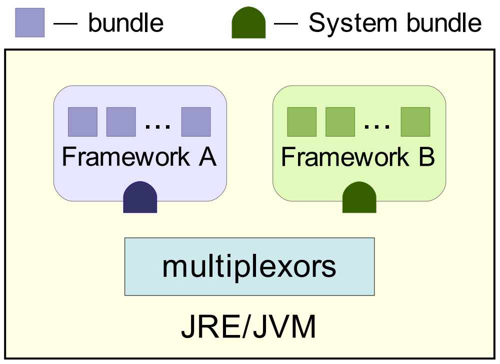

Figure 1: Peer embedded frameworks

- The lifecycle of each framework should be managed by the launching code.
- The parent classloader of each framework is the classloader used to instantiate the framework.

5.1.2 Nested frameworks

Use case: A bundle running within an OSGi framework uses its own private framework instance to provide some function. This is essentially a variant of the peer embedded case.

Launching the nested framework is also covered by the mechanisms introduced in RFC 132. Due to the disjoint/isolated nature of the nested framework, cross-framework multiplexing is also necessary to address JVM singletons.

The nested framework (Framework C) is completely independent of (and hidden from) the hosting framework (Framework A): it is visible only to the launching bundle. As with peer embedded frameworks, the hosting and nested frameworks could be from different vendors.

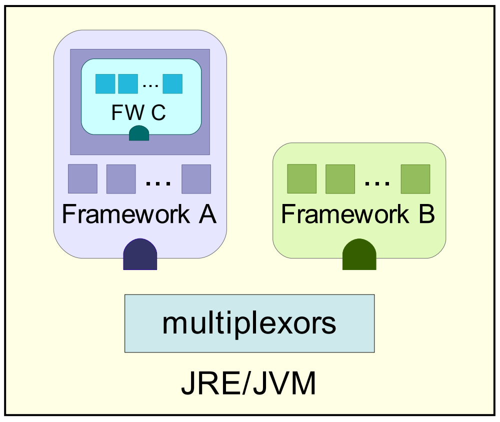

Figure 2: Nested frameworks

- The lifecycle of the nested framework should be managed by the launching code (as with any embedded framework). In this case, it is the launcher's responsibility to ensure that the nested framework is stopped when the launching bundle is stopped.

- The parent classloader of the nested framework is the classloader of the launching bundle. Additional packages may have to be imported by the launching bundle (via Import-Package or DynamicImport-Package) in order to ensure that all packages from the VM execution environment are available to the nested framework classloader.

5.2 Singleton Multiplexing

5.2.1 Multiplexing factories - *Provisional*

An interface defined as part of the OSGi specification will allow multi-vendor, multi-framework environments to correctly support the URL Handlers Service Specification (and hopefully others) without conflicting. The following is based on an algorithm outlined by Simon Kaegi and the implementation used by Equinox.

- `org.osgi.multiplexer.MultiplexSupport`: a simple API that framework providers can use (via reflection) to discover the presence of other multiplexers.
  - `boolean supportsOSGiMultiplexing()` – this method allows a framework to test a pre-registered JVM singleton to see if a) the singleton is aware of the OSGi multiplexing algorithm (the method is present), and if so, b) if it can support multiplexing in the current environment. Singleton multiplexing is heavily dependent on potentially JVM-specific reflection, so it is possible that a framework could find itself in an environment in which it can't support multiplexing, in which case, the implementation of this method (when invoked via reflection) would return false.
  - `Object getLegacySingleton()` – this method preserves the value of a pre-existing non-multiplexing JVM singleton. As new multiplexers are added, `getLegacySingleton()` should be called on the current (multiplexing) singleton, and the returned value (if not null), saved for later use.
  - `Object getNextMultiplexer()`
    `Object getPrevMultiplexer()`
    `void` `setNextMultiplexer(Object multiplexer)`
    `void` `setPrevMultiplexer(Object multiplexer)` – these methods are used to create and maintain a double-linked list of registered multiplexers. This list is used both for insertion and removal of multiplexers as the associated frameworks startup and shutdown, but is also used for delegation between multiplexers to handle object operations.
- Registration of multiplexers: A framework will create an object (x) that satisfies the MultiplexSupport interface, and will first check to see if a JVM singleton is already registered.
  - If a non-multiplexer singleton exists (s), x will store that singleton, and will then use reflection to establish itself as the JVM singleton in place of s. Note that there may be other cached data associated with the previous singleton that must also be cleared when the singleton is reset.
  - If a multiplexer already exists (y), x will call `y.getLegacySingleton()` to preserve delegation to the non-multiplexing singleton s. x must then use `y.getNextMultiplexer()` to add itself to the chain of multiplexers. Frameworks may search the chain for an implementation of “their” multiplexer (such that all multiplexers for a certain frameworks share one link in the chain), or may simply append additional multiplexer instances to the chain. In either case, it is important to maintain the double-linked nature of the list when it is necessary to add themselves to the list (`n.setNextMultiplexer(x)`, and `x.setPrevMultiplexer(n)`).
- De-registration of multiplexers: a framework will find its multiplexer in the chain (how that happens may vary, depending on how the framework adds multiplexers to the chain):

   - If the multiplexer is the set as the JVM singleton, it will use reflection to re-set the JVM singleton to the next multiplexer in the chain, or to the legacy singleton. If a multiplexer is selected, it's "previous" link should be cleared, to indicate that it is the new head of the chain.

   - If the multiplexer is in the middle or end of the chain, it should make the appropriate method calls to reset the previous/next links of neighboring multiplexers (`n.setNextMultiplexer(m)`, and `m.setPrevMultiplexer(n)`) in order to preserve the double-linked list.

- Looking up the right multiplexer: the JVM singleton multiplexer should first check to see (via implementation specific means) if it should handle the request. Each framework implementation will have its own way of determining the caller context (and hence determining whether or not "this" multiplexer should handle the method call). If the multiplexer determines that it needs to delegate the method call, it will first call the requisite method on the "next" multiplexer in the chain, if present. If there isn't a "next" multiplexer, the method call should be delegated to the legacy singleton.

5.2.2 Other VM Singletons

- System Properties – System properties are an unavoidable JVM singleton. The new framework launching mechanism defined by RFC 132 allows properties to be provided at framework creation time that may or not be backed by System properties.

- System.in, System.out, System.err – see Thread IO service defined in RFC 132 for a potential solution to these JVM singletons.

5.2.3 OSGi Singletons

- `org.osgi.framework.FrameworkUtil` is an existing class that defines a single static method used to retrieve a Filter object for looking up Service References or Dictionary objects. Framework implementations override this class in order to provide their own Filter implementations.

   - The static `FrameworkUtil.createFilter` method was modified in v4.2 to return a common Filter implementation. Framework implementations are not required provide their own implementations of `FrameworkUtil`. The `BundleContext.createFilter` method can be used to create a Filter that is optimized for particular framework implementations.

- `org.osgi.framework.AdminPermission` and `org.osgi.service.condpermadmin.BundleSignerCondition` both end up tied to particular framework implementations because there is no generic way to get the signer DN chains for a bundle.

   - A new method, `boolean matchDistinguishedNameChain(String, List)` was added (in v4.2) to `org.osgi.framework.FrameworkUtil` to provide a common DN chain matching algorithm.

   - A new method, `Map getSignerCertificates(int)`, was added (in v4.2) to `org.osgi.framework.Bundle` to facilitate retrieval of the DN chains used to sign a Bundle.

   - Final class `org.osgi.framework.AdminPermission` now uses (as of v4.2) these two methods to work with the DN chain for a bundle.

   - `org.osgi.service.condpermadmin.BundleSignerCondition` has also been updated (in v4.2) to use the new methods.

5.3 Open Issues

Other Singleton resource providers?

- `javax.xml.parsers.SAXParserFactory` / `javax.xml.parsers.DocumentBuilderFactory`
- Security Manager – there is no apparent way to multiplex the JVM Singleton security manager, nor are the implications of doing such a thing obvious. With multiple frameworks present in the same VM, the first to establish a SecurityManager effectively “wins”: the VM will call that instance of the security manager for all subsequent permission checks. In the case of parent/child frameworks of the same type (e.g. a hierarchy of Equinox frameworks), the framework implementation is free to perform additional checks to support different rules/permissions defined for each framework instance in the hierarchy. How such interactions might take place between disjoint frameworks (especially if the framework implementations differ) is undefined.
- Synchronization / locking while manipulating the linked list of multiplexers.

6 Considered Alternatives

For posterity, record the design alternatives that were considered but rejected along with the reason for rejection. This is especially important for external/earlier solutions that were deemed not applicable.

7 Security Considerations

There are several open issues regarding security considerations for using multiple frameworks in the same JVM:

- The security manager delegates to the protection domain of the individual bundles when performing permission checks. Because every bundle has its own protection domain, the presence of multiple frameworks should not impact permission checks in general.
- The presence/use of nested frameworks may impact the behavior of postponed conditions in the `ConditionalPermissionAdmin` service. More investigation and clarification is needed in this area.

8 Javadoc

8.1 Interface org.osgi.multiplexer.MultiplexSupport *- Provisional*

`public interface` **`MultiplexSupport`**

This is a simple API intended to allow multiple frameworks in the same JVM to "share" JVM singletons like the URLStreamHandlerFactory and the ContentHandlerFactory. Supporting this API makes heavy use of potentially JVM-specific reflection to manage both the JVM singleton instance and any cached state associated with previously set singletons.

Since each multiplexer could be associated with classes loaded by a different classloader, the methods on this interface would have to be discovered and invoked via reflection, rather than by casting or instanceof tests.

The presence and support of this interface (as determined by the `supportsOSGiMultiplexing` method) implies support of the following multiplexing algorithm for a given JVM singleton.

8.1.1.1 Initialization:

- Create a new multiplexer for the given JVM singleton, e.g. an implementation of a URLStreamHandlerFactory that also implements the MultiplexSupport interface.
- Retrieve the current JVM Singleton-- Does it support multiplexing?
    - If yes, get (and store) the legacy singleton from the JVM singleton multiplexer in the new multiplexer, and append the new multiplexer to the double-linked list.
    - If no, store the current singleton in the new multiplexer as the "Legacy singleton", and use Java reflection to reset the JVM singleton instance and clear any cached state before setting the new multiplexer as the JVM singleton.

8.1.1.2 Termination:

- When a framework is cleaning up, it will need to "remove" its multiplexer from the double-linked list, in a process that is the reverse of what happens during initialization.
    - If the multiplexer being cleaned up is registered as the JVM singleton, then it needs to set a new singleton: either the "next" multiplexer in the chain if other multiplexers exist, or the legacy singleton, if it exists.
    - If the multiplexer being cleaned up is not the JVM singleton, it needs only to update the previous and next links of its neighbors in order to remove itself from the list.

8.1.1.3 Determining the context:

When a method is called on a multiplexing JVM singleton, the singleton (and any subsequently called multiplexers) must decide whether or not they are the correct instance for the method call. If the current multiplexer determines that it cannot handle the current method call, it should delegate to the next multiplexer in the chain. The last multiplexer in the chain should delegate method calls to the legacy singleton if the calling context doesn't match.

Method Summary

`java.lang.Object`	**`getLegacySingleton`**`()` Obtain a reference to the pre-existing non-multiplexing JVM singleton, if present.
`java.lang.Object`	**`getNextMultiplexer`**`()`
`java.lang.Object`	**`getPrevMultiplexer`**`()`
`void`	**`setNextMultiplexer`**`(java.lang.Object multiplexer)` Set the next multiplexer in the chain.
`void`	**`setPrevMultiplexer`**`(java.lang.Object multiplexer)` Set the previous multiplexer in the chain.
`boolean`	**`supportsOSGiMultiplexing`**`()` This method allows a framework to test a pre-registered JVM singleton to see if a) the singleton is aware of the OSGi multiplexing algorithm (the method is present), and if so, b) if it can support multiplexing in the current environment.

Method Detail

8.1.2 supportsOSGiMultiplexing

`boolean` **`supportsOSGiMultiplexing`**`()`

This method allows a framework to test a pre-registered JVM singleton to see if a) the singleton is aware of the OSGi multiplexing algorithm (the method is present), and if so, b) if it can support multiplexing in the current environment.

Singleton multiplexing is heavily dependent on potentially JVM-specific reflection, so it is possible that a framework could find itself in an environment in which it can't support multiplexing, in which case, the implementation of this method would return false.

Returns:
true if multiplexing supported in the current JVM, false otherwise.

8.1.3 getLegacySingleton

`java.lang.Object` **`getLegacySingleton`**`()`

Obtain a reference to the pre-existing non-multiplexing JVM singleton, if present. As new multiplexers are added, getLegacySingleton() should be called on the current (multiplexing) singleton, and the returned value (if not null), should be saved and used for delegation.

8.1.4 getNextMultiplexer

```
java.lang.Object getNextMultiplexer()
```

Returns:
the next multiplexer in the chain, or null if not set.

8.1.5 setNextMultiplexer

```
void setNextMultiplexer(java.lang.Objectmultiplexer)
```

Set the next multiplexer in the chain.

Parameters:
multiplexer - The next multiplexer in the double-linked list.

8.1.6 getPrevMultiplexer

```
java.lang.Object getPrevMultiplexer()
```

Returns:
the previous multiplexer in the chain, or null if not set.

8.1.7 setPrevMultiplexer

```
void setPrevMultiplexer(java.lang.Objectmultiplexer)
```

Set the previous multiplexer in the chain.

Parameters:
multiplexer - The previous multiplexer in the double-linked list.

9 Document Support

9.1 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]. Kaegi, S., Modular Java Web Applications, http://www.scs.carleton.ca/~deugo/thesis/simon-kaegi/thesis-sk-final.pdf

9.2 Author's Address

Name	Erin Schnabel
Company	IBM Corporation
Address	2455 South Road, Poughkeepsie, NY 12603
Voice	+1 845-435-5648
e-mail	schnabel@us.ibm.com

9.3 Acronyms and Abbreviations

9.4 End of Document