Frequently Asked Questions

Spring Dynamic Modules for OSGi Service Platforms *

1.0-rc1

Costin Leau (SpringSource)

Chapter 1. Frequently Asked Questions

1.1. What happened to "Spring OSGi" project name?

The OSGi term is a trademark belonging to The OSGi Alliance. In order to comply with their guidelines, it was decided that the project name be changed to "Spring Dynamic Modules for OSGi Service Platforms" (aka Spring-DM). The new name is still pending final approval by the OSGi Alliance. The name change was the result of an amicable discussion between the OSGi Alliance and Interface21. Interface21 is a member of the OSGi Alliance, and the OSGi Alliance remain very supportive of the project.

1.2. Why aren't there any javadocs on `*.internal.*`?

`org.springframework.osgi.*.internal` packages are meant (as the name implies) to be private and non-public package. Thus, there is no documentation, support or compatibility guarantee for them. In fact, the Spring Dynamic Modules bundle does not even export them to prevent accidental usage.

If you find classes under this package, which you really, really depend on, then consider raising an issue on JIRA to have access opened up.

1.3. What are Spring Dynamic Modules requirements?

Spring Dynamic Modules requires at least Java 1.4, OSGi 4.0 and Spring 2.5. It might be possible to use Spring Dynamic Modules on a lower execution environment (such CDC) but it is not guaranteed to work. Both Spring and Spring Dynamic Modules rely on JavaBeans (java.beans package) which, unfortunately, is missing in most restricted environments. See this PDF for information on CDC profiles. Note that, Spring 2.5 also requires Java 1.4.

Nevertheless, experiences and feedback on running Spring-DM in restricted environments is welcomed - please use our mailing list.

1.4. What OSGi platforms are supported?

Spring-DM requires an OSGi 4.0 platform. The framework has been tested on Equinox, Felix and Knopflerfish - in fact, the test suite is ran against all of them as part of our continuous integration process.

1.5. Where can I learn about OSGi?

The best place to start is The Osgi Alliance home and developer pages which provide the OSGi specifications, introductions and many links and blogs on the topic. Please see the reference documentation appendix for more information.

In addition, all OSGi implementation websites host detailed, step-by-step tutorials and introduction.

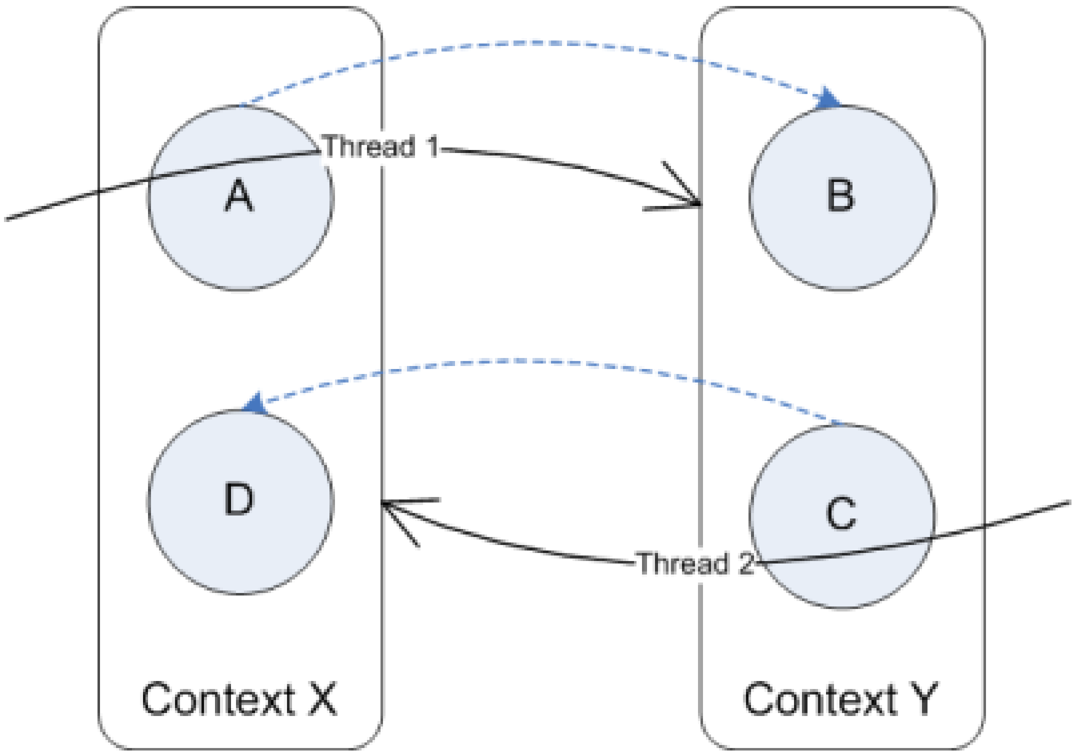
If you discover any additional materials useful for OSGi newbies, please let us know to update the list. Thank you.

1.6. I get an exception about backport-util-concurrent library being required. Why is that?

This exception is thrown only when running on Java 1.4 without backport-util-concurrent bundle installed.

OSGi platform is a concurrent environment. Beans from different application contexts can interact with each other creating cycles between their owning contexts. This means that the backing contexts have to be able to lookup and create/retrieve bean instances, all at the same time, on multiple threads. A traditional synchronised collection allows proper locking and thread coordination and prevents race conditions, but can cause very easily deadlocks.

Consider two contexts each containing two beans:



Inter-application context bean interaction

If both bean A and C are requested by two separate threads at the same time, this scenario will deadlock since each thread waits for the other one to release the "bean registry" lock even just for reading. However, when using a concurrent collection, reading doesn't require a lock so each thread can interact with the other context without waiting for a lock. Java 1.5 and upwards provide concurrent collections under `java.util` package. However, since Java 1.4 does not, backport-util-concurrent library is required.

1.7. I cannot build the sources when using JDK 6+. What can I do?

The problem is caused by Maven
<a>profiles</a>
feature which considers only a certain JDK version rather then all compatible versions. As a workaround, one can build the sources, using JDK 5.0 (if annotations are needed) or JDK 1.4.2 (without annotations or other JDK 5 specific features).

1.8. How can I use logging in OSGi?

OSGi platforms do not change the way libraries work, it just enforces tighter classloading. Thus, you can, in most of the cases, use the same logging strategy used in non-OSGi environments.

Spring (and Spring-DM) use internally the commons-logging API which acts as an "ultra-thin bridge between different logging implementations". In OSGi, just like in a non-OSGi environment, Spring and Spring-DM delegate all logging (including initialisation) to the actual commons-logging API implementation.

Out of the box, SLF4J library is provided, which shares the same purpose as commons-logging but without the classloading discovery mechanism that causes loading issues, using static wiring (see the SLF4J site for more info). Please see this question for more details on why commons-logging jar is not used.

Spring-DM uses SLF4J on top of Log4J but this can be easily changed. As part of log4j initialisation, a `log4j.properties` or `log4j.xml` configuration fille needs to be present in the bundle classpath. This means that the configuration file has to be part of your bundle or one of its attached fragments. Besides SLF4J, for another OSGi-aware solution, one can try Pax Logging.

To learn more about log4j setup process, follow this link.

1.9. If you use commons-logging API, why rely on SLF4J and not the commons-logging jar?

The commons-logging project provides the commons-logging API (`commons-logging-api-nn.jar`) along with an implementation (`commons-logging-adapters-nn.jar`) that provides a wrapper between the API and the actual logging libraries used underneath (such as log4j, java.util.logging, etc). However, in order to determine what implementation should be used, commons-logging library tries to do some classloading-based discovery that is fragile and can fail unexpectedly. In an strict classloading environment such as OSGi, this mechanism adds unnecessary complexity - that's why we decided to use SFL4J which is not just simpler and actively maintained but is also OSGi-friendly out of the box.

For more information about commons-logging classloading problems, see these links: #1 #2

1.10. Why don't you use the OSGi logging service/[insert your favourite logging library in here]?

It is completely up to you what logging implementation you want Spring-DM to use. To route log messages to the OSGi logging service, just use a commons-logging API implementation that delegates to the OSGi logging service, such as Pax Logging.

1.11. I have to use [insert name] library/framework inside. What can I do?

OSGi requires JARs to contain certain `MANIFEST.MF` entries which indicate what classes are required and shared by each archive. This means that *tradition* jars cannot be used inside an OSGi environment. To solve the problem one can:

- Use a repository of pre-wrapped libraries such as Orbit, Felix Commons or Knopflerfish repository. Spring-DM also provides a small repository for its internal usage, which you might find helpful.
- Wrap the necessary jars with proper OSGi manifest. While this can be done by hand, we strongly recommend Peter Kriens excellent bnd tool which can do this for you automatically. For Maven, see Felix maven-bundle-plugin.
- Include the jar inside your OSGi bundle and include it in the bundle classpath through *Bundle-ClassPath* directive. See the OSGi specification for more information.

1.12. *java.lang.NoClassDefFoundError: javax/transaction/...* when trying to do data access

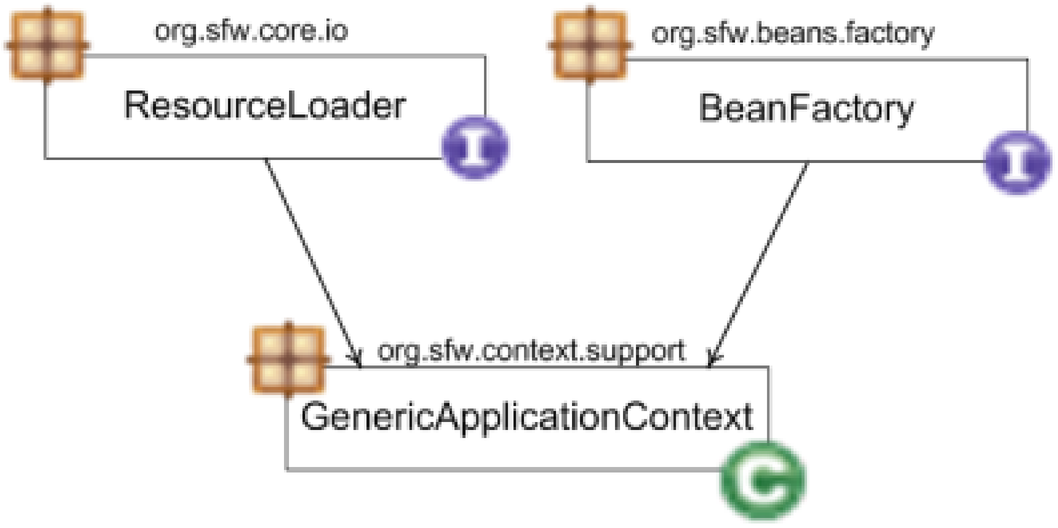
This problem is likely to be caused by bad class wiring. All 1.3+ JDKs include incomplete *javax.transaction* and *javax.transaction.xa* packages for usage inside ORB environments. To address the problem, use a proper JTA library which contains all the classes from the forementioned packages and exports them with a specific version to prevent confusion.

Spring-DM wraps JTA 1.1 library for OSGI environments which can be found at Spring snapshot repository. One can deploy this library and specify version 1.1 for *javax.transaction** packages inside *Import-Package* header. By specifying the version, one can be sure that the proper package is used.

Note that JTA 1.1 is compatible with version 1.0.1.

1.13. The autoExport option doesn't work properly!

autoExport flag, part of the service exporter, will discover and include for exporting only the *visible* interfaces/classes implemented by the service object. Consider class `GenericApplicationContext` which implements among others, interfaces `BeanFactory` (from `org.springframework.beans.factory` package) and `ResourceLoader` (`org.springframework.core.io`).



Class Hierarchy

Depending on your OSGi imports, the exporting bundle can see only one of the packages, none or both. Based on these visibility settings, the exporter will only export the classes that are 'known' to the exporting bundle.

For example, if the exporting bundle sees `org.springframework.core.io` but not `org.springframework.beans.factory`, the service will be exported as a `ResourceLoader` but not as a `BeanFactory`. In fact, exporting the object as a `BeanFactory` will fail since the bundle doesn't see this interface and thus doesn't know how to handle its contract.

1.14. When using Spring-DM integration testing I get an exception about serialVersionUID. What is the problem?

When running an integration test, Spring-DM will duplicate the test instance and execute it inside OSGi. To avoid problems like this one, make sure you are using the same libraries (with the same version) as Spring-DM when running your test. This particular problem for example is caused by a JUnit 3.8.2 vs 3.8.x serialization compatibility. Make sure that you are using at least JUnit 3.8.2 for the execution of your tests.