Frequently Asked Questions

Spring Dynamic Modules for OSGi Service Platforms *

1.2.1

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. Are there plans to support other dynamic module frameworks (such as the JSR 277 extensions in Java 7)?

There are no current plans to support other dynamic module frameworks.

1.5. Will Spring DM work in restricted environments (such as small/mobile devices)?

See the [requirements]() entry. The OSGi Service Platform is designed to run in very constrained environments however, Spring Dynamic Modules depends on the Spring Framework v2.5 which in turn depends on JDK 1.4. Thus Spring Dynamic Modules cannot run on more constrained environments (such as the OSGi Minimum Execution Environment) unless Spring itself also runs in those environments. There are no current plans to make such a version of Spring. However as existing OSGi developers adopt Spring Dynamic Modules to simplify creation of OSGi applications and the user base expands, the target audience can cover domains much broader than *enterprise Java applications*. In time this could create a large enough demand to justify the

investment needed to allow Spring and Spring DM to run in restricted environments.

1.6. 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.

1.7. 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.

1.8. I have problems building the sources. What can I do?

Please see the file called `readme-building.txt` found in the source tree.

1.9. 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:
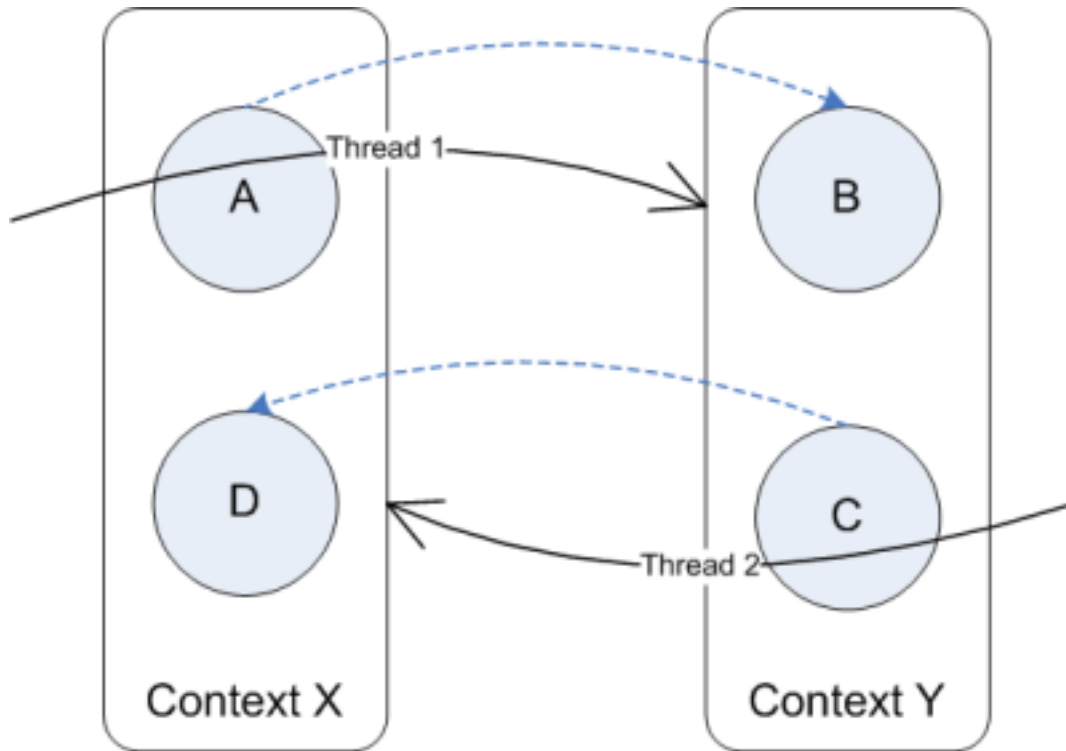
Inter-application context bean interaction

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.

1.10. 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 class loading discovery mechanism (that causes loading issues), using static wiring (see the SLF4J site for more info). To use slf4j, make sure you use: `slf4j-api-XXX.jar`, `jcl104-overslf4j-XXX.jar` and `slf4j-log4j-XXX.jar` (where XXX stands for the slf4j version). The last jar provides the static wiring between slf4j and log4j - if another implementation is desired (such as jdk14), then a different jar is required (for the jdk14 that would be slf4j-jdk14-XXX.jar) - see the official SLF4J site for more information. 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.

1.11. If you use the 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

1.12. I have to use commons-logging, what can I do?

If you have to use commons-logging (for example the jar is required by certain bundles) then try using the most recent version commons-logging version (1.1+) as it provides more options on the discovery process. Below are some settings that can be used to make commons-logging work inside an OSGi environment:

- 1.0.x

  Unfortunately, commons-logging 1.0.x uses the thread context class loader (TCCL) always for loading loggers implementations. Inside an OSGi environment, the TCCL is undefined and cannot be relied upon. Since managing the TCCL is almost impossible as most loggers are defined as static fields that need to resolved on class loading, using a different `LogFactory` is advised. One can use the *org.apache.commons.logging.LogFactory* system property to specify a different log factory however, the commons-logging bundle should be able to load this class.

- 1.1.x

  If using commons logging 1.1.x, one can turn off the tccl usage through *use_tccl* property, part of the *commons-logging.properties* file. http://commons.apache.org/logging/commons-logging-1.1/troubleshooting.html. Additionally, 1.1.x provides several system properties (such as *org.apache.commons.logging.Log.allowFlawedContext*, *org.apache.commons.logging.Log.allowFlawedDiscovery* and *org.apache.commons.logging.Log.allowFlawedHierarchy*) that can change the behaviour of the discovery process. See the LogFactoryImpl javadoc for more details.

In our tests, commons logging 1.1.x can be used with reasonable success inside OSGi. We haven't been able to find a generic configuration for commons logging 1.0.x that works and that does not rely on fragile hacks dependent on the running environment.

1.13. 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.

1.14. 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 uses the SpringSource Enterprise [Bundle Repository](#) for its dependencies, which you might find useful. Additionally, for artifacts that have not yet made it into SpringSource Repository, Spring DM provides a small, temporary (Amazon S3) Maven repository ([link](#) | [browser-friendly link](#)) for its internal usage.

- 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.

1.15. I keep getting *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.

1.16. When doing integration testing I receive *java.lang.NoClassDefFoundError: org.osgi.vendor.framework property not set*...

Remove the official OSGi jars (osgi.jar or osgi-r4-core.jar) from the classpath and use only the actual OSGi platform (Equinox/Knopflerfish/Felix) jars. The former provides only the public classes without an actual implementation and thus cannot be used during runtime, only during the compilation stage.

1.17. 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`).
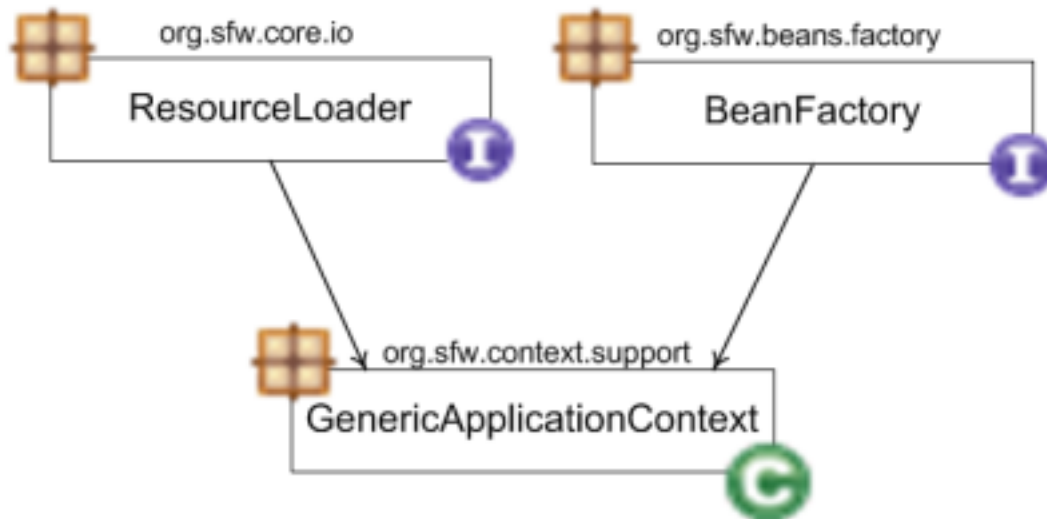
Class Hierarchy

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.

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

This problem occurs on Spring DM versions up to 1.0 - consider upgrading to 1.0.1 or better. If you are stuck with 1.0 see below. 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.

1.19. I'm using Eclipse PDE and I started getting some weird exceptions/behaviour. What's the matter?

Eclipse PDE uses Equinox OSGi platform underneath which (like other OSGi platforms) caches the bundles between re-runs. When the cache is not properly updated, one can encounter strange behaviour (such as the new services/code being picked up) or errors ranging from class versioning to linkage. Consider doing a complete clean build or, in case of Eclipse, creating a new workspace or deleting the bundle folder (depends on each project settings but most users should find it at: `[workspace_dir]\.metadata\.plugins\org.eclipse.pde.core\OSGi\org.eclipse.osgi\bundles`).

1.20. I'm upgrading to Spring DM 1.1 but now I get some ClassNotFoundExceptions. What has changed?

In Spring DM 1.1 M2, the proxy infrastructure has been refined to avoid *type leaks*, the usage of dynamic imports or exposure of class loader chain delegation. If you encounter class visibility problems during the upgrade then it's likely you have missing imports which were previously resolved as a side effect of Spring DM proxy weaving process.

1.21. I've noticed that objects imported by Spring DM are not always equal to the raw target service. Why is that?

To deal with dynamics, Spring DM creates proxies around the imported services. The proxies are classes (generated at runtime), different from the target but able to intercept the calls made to it. Since a proxy is different then its target, comparing objects against it can yield different results then when the comparison is done against the target. In most scenarios this is not a problem but there might be corner cases where this contract matters. Since 1.1, Spring DM importer proxies implement `InfrastructureProxy` interface (from Spring framework) which allow access to the raw target.

1.22. My Spring DM collection doesn't change even though the number of OSGi service changes. What's wrong?

Make sure the Spring DM collections are injected into object of compatible types (for example `list` into `java.util.List` or `java.util.Collection`). If the types are not compatible, the container will have to perform [type conversion](), transforming the Spring DM managed collection into a 'normal' one, unaware of the OSGi dynamics.

1.23. I have upgraded to Spring DM 1.2 but my custom extender/web extender configuration doesn't work anymore. What has changed?

Since Spring 2.5.6, the symbolic names of the artifacts have changed slightly. Spring DM aligned its symbolic names as well with the new patter since 1.2.0 M2. Thus the prefix `org.springframework.bundle.osgi` has been changed to `org.springframework.osgi`; for example Spring DM extender symbolic name was changed from `org.springframework.bundle.osgi.extender` to `org.springframework.osgi.extender` (notice the missing `bundle` word). To fix this problem, change the reference to the old symbolic name (usually inside the fragments manifests or LDAP filters) to the new one.

1.24. I'm using Knopflerfish 2.3.x and I have a lot of visibility exception. How can I fix this?

Since 2.3.0, Knopflerfish changed the way it does bootpath delegation which causes classes to be loaded from inside and outside OSGi. Set the system property `org.knopflerfish.framework.strictbootclassloading` to `true` before starting up the Knopflerfish platform to prevent this from happening. See Knopflerfish 2.3.x release [notes]() for more information.