Spring Dynamic Modules Reference Guide

1.0.2

Adrian M Colyer (SpringSource), Hal Hildebrand (Oracle), Costin Leau (SpringSource), Andy Piper (BEA)

Preface

Application development has seen significant changes in the last years, moving towards a simpler, more agile, POJO-based programming model in order to keep a fast pace. Dependency injection and Aspect Oriented Programming, which were once *bleeding edge* ideas, are used on a daily basis by most developers to manage and simplify the complexity of their applications.

However, in terms of deployment, things have remained mainly unchanged. Even though code bases are divided into modules, whether logical, conceptual or physical, at runtime they are seen as one monolithic application in which, making a change (be it large or small), requires a restart. OSGi aims to change this by allowing applications to be divided into *modules* that can have different life cycles, dependencies and still exist as a whole.

Spring Dynamic Modules focuses on integrating Spring Framework powerful, non-invasive programming model and concepts with the dynamics and modularity of OSGi platform. It allows transparent exporting and importing of OSGi services, life cycle management and control.

While every effort has been made to ensure that this documentation is comprehensive and there are no errors, nevertheless some topics might require more explanation and some typos might have crept in. If you do spot any mistakes or even more serious errors and you can spare a few cycles during lunch, please do bring the error to the attention of the Spring Dynamic Modules team by raising an issue. Thank you.

Part I. Introduction

This document is the reference guide for Spring Dynamic Modules. It defines Spring Dynamic Modules concepts and semantics, the syntax for the OSGi Service Platform based namespaces, the Dynamic Modules extender bundle and the OSGi manifest header entries defined by Dynamic Modules. For a tutorial introduction to building OSGi-based applications with Spring Dynamic Modules see our online page.

OSGi developers looking for an introduction to Spring should review the introductory articles on the springframework.org site.

Note: OSGi is a trademark of the OSGi Alliance. Project name is pending final approval from the Alliance.

Note: Please see the known issues page for Spring Dynamic Modules release.

Chapter 1. Why Spring Dynamic Modules?

The Spring Framework is the leading full-stack Java/JEE application framework. It provides a lightweight container and a non-invasive programming model enabled by the use of dependency injection, AOP, and portable service abstractions. The OSGi Service Platform offers a dynamic application execution environment in which modules (bundles) can be installed, updated, or removed on the fly. It also has excellent support for modularity and versioning.

Spring Dynamic Modules makes it easy to write Spring applications that can be deployed in an OSGi execution environment, and that can take advantage of the services offered by the OSGi framework. Spring's OSGi support also makes development of OSGi applications simpler and more productive by building on the ease-of-use and power of the Spring Framework. For enterprise applications, the combination of Spring Dynamic Modules and the OSGi platform provides:

- Better separation of application logic into modules, with runtime enforcement of module boundaries
- The ability to deploy multiple versions of a module (or library) concurrently
- The ability to dynamically discover and use services provided by other modules in the system
- The ability to dynamically install, update and uninstall modules in a running system
- Use of the Spring Framework to instantiate, configure, assemble, and decorate components within and across modules.
- A simple and familiar programming model for enterprise developers to exploit the features of the OSGi platform.

We believe that the combination of OSGi and Spring offers a comprehensive model for building enterprise applications.

Chapter 2. Requirements

Spring Dynamic Modules 1.0 supports JDK level 1.4 and above and OSGi R4 and above. Bundles deployed for use with Spring Dynamic Modules should specify `"Bundle-ManifestVersion: 2"` in their manifest (OSGi R4). We test against Equinox 3.2.2, Felix 1.0.1, and Knopflerfish 2.0.3 as part of our continuous integration process.

Part II. Reference Documentation

Document structure

This part of the reference documentation explains the core functionality offered by Spring Dynamic Modules.

bnd-app-ctx describes the relationship between an OSGi Bundle and a Spring Application Context, and introduces the Spring Extender Bundle support for instantiating application contexts automatically.

app-deploy describes how to deploy the Spring Framework jar files in an OSGi environment, and how to reference external APIs from your application bundles should you need to do so. This chapter also explains some of the issues to be aware of when using existing enterprise libraries not designed for OSGi in an OSGi environment.

service-registry describes how to export Spring beans as services in the OSGi service registry, and how to inject references to OSGi services into beans. This chapter also defines how the dynamic life-cycle of OSGi services and bundles is supported.

bundles describes how to declare a bean that represents an OSGi bundle, including support for installing new bundles into the OSGi platform.

testing explains the integration testing support provided by Spring Dynamic Modules. This support enables you to write simple JUnit integration tests that can start up an OSGi environment, install the bundles needed for the integration test, execute the test case(s) inside of OSGi, and return the results to the runner. This makes it easy to integrate OSGi integration testing into any environment that can work with JUnit.

Chapter 3. Bundles and Application Contexts

The unit of deployment (and modularity) in OSGi is the *bundle* (see section 3.2 of the OSGi Service Platform Core Specification). A bundle known to the OSGi runtime is in one of three steady states: installed, resolved, or active. Bundles may export services (objects) to the OSGi service registry, and by so doing make these services available for other bundles to discover and to use. Bundles may also export Java packages, enabling other bundles to import the exported types.

In Spring the primary unit of modularity is an *application context*, which contains some number of beans (objects managed by the Spring application context). Application contexts can be configured in a hierarchy such that a child application context can see beans defined in a parent, but not vice-versa. The Spring concepts of exporters and factory beans are used to export references to beans to clients outside of the application context, and to inject references to services that are defined outside of the application context.

There is a natural affinity between an OSGi bundle and a Spring application context. Using Spring Dynamic Modules, an active bundle may contain a Spring application context, responsible for the instantiation, configuration, assembly, and decoration of the objects (beans) within the bundle. Some of these beans may optionally be exported as OSGi services and thus made available to other bundles, beans within the bundle may also be transparently injected with references to OSGi services.

3.1. The Spring Dynamic Modules Extender bundle

Spring Dynamic Modules provides an OSGi bundle `org.springframework.bundle.osgi.extender`. This bundle is responsible for instantiating the Spring application contexts for your application bundles. It serves the same purpose as the ContextLoaderListener does for Spring web applications. Once the extender bundle is installed and started it looks for any existing Spring-powered bundles that are already in the *ACTIVE* state and creates application contexts on their behalf. In addition, it listens for bundle starting events and automatically creates an application context for any Spring-powered bundle that is subsequently started. app-deploy:headers describes what the extender recognizes as a "Spring-powered bundle".

3.2. Application Context Creation

The extender bundle creates applications contexts asynchronously. This behaviour ensures that starting an OSGi Service Platform is fast and that bundles with service inter-dependencies do not cause deadlock on startup. A Spring-powered bundle may therefore transition to the STARTED state before its application context has been created. It is possible to force synchronous creation of application contexts on a bundle-by-bundle basis. See app-deploy:headers for information on how to specify this behaviour.

If application context creation fails for any reason then the failure cause is logged. The bundle remains in the STARTED state. There will be no services exported to the registry from the application context in this scenario.

3.2.1. Mandatory Service Dependencies

If an application context declares mandatory dependencies on the availability of certain OSGi services (see service-registry) then creation of the application context is blocked until all mandatory dependencies can be satisfied through matching services available in the OSGi service registry. Since a service may come and go at any moment in an OSGi environment, this behaviour only guarantees that all mandatory services were available at the moment creation of the application context began. One or more services may subsequently become unavailable again during the process of application context creation. service-registry describes what happens

when a mandatory service reference becomes unsatisfied. In practice, for most enterprise applications built using Spring Dynamic Modules services, the set of available services and bundles will reach a steady state once the platform and its installed bundles are all started. In such a world the behaviour of waiting for mandatory dependencies simply ensures that bundles A and B, where bundle A depends on services exported by bundle B, may be started in any order.

A timeout applies to the wait for mandatory dependencies to be satisfied. By default the timeout is set to 5 minutes, but this value can be configured using the `timeout` directive. See app-deploy:headers for details.

It is possible to change the application context creation semantics so that application context creation fails if all mandatory services are not immediately available upon startup (see section 4.1). When configured to not wait for dependencies, a bundle with unsatisfied mandatory dependencies will be stopped, leaving the bundle in the *RESOLVED* state.

3.2.2. Application Context Service Publication

Once the application context creation for a bundle has completed, the application context object is automatically exported as a service available through the OSGi Service Registry. The context is published under the interface `org.springframework.context.ApplicationContext` (and also all of the visible super-interfaces and types implemented by the context). The published service has a service property named `org.springframework.context.service.name` whose value is set to the bundle symbolic name of the bundle hosting the application context. It is possible to prevent publication of the application context as a service using a directive in the bundle's manifest. See app-deploy:headers for details.

Note: the application context is published as a service primarily to facilitate testing, administration, and management. Accessing this context object at runtime and invoking `getBean()` or similar operations is discouraged. The preferred way to access a bean defined in another application context is to export that bean as an OSGi service from the defining context, and then to import a reference to that service in the context that needs access to the service. Going via the service registry in this way ensures that a bean only sees services with compatible versions of service types, and that OSGi platform dynamics are respected.

3.3. Bundle Lifecycle

OSGi is a dynamic platform: bundles may be installed, started, updated, stopped, and uninstalled at any time during the running of the framework.

When an active bundle is stopped, any services it exported during its lifetime are automatically unregistered and the bundle returns to the resolved state. A stopped bundle should release any resources it has acquired and terminate any threads. Packages exported by a stopped bundle continue to be available to other bundles.

A bundle in the resolved state may be uninstalled: packages that were exported by an uninstalled bundle continue to be available to bundles that imported them (but not to newly installed bundles).

A bundle in the resolved state may also be updated. The update process migrates from one version of a bundle to another version of the same bundle.

Finally of course, a resolved bundle can be started, which transitions it to the active state.

The OSGi `PackageAdmin` `refreshPackages` operation refreshes packages across the whole OSGi framework or a given subset of installed bundles. During the refresh, an application context in an affected bundle will be stopped and restarted. After a `refreshPackages` operation, packages exported by older versions of updated bundles, or packages exported by uninstalled bundles, are no longer available. Consult the OSGi specifications for full details.

When a Spring-powered bundle is stopped, the application context created for it is automatically destroyed. All services exported by the bundle will be unregistered (removed from the service registry) and the normal application context tear-down life-cycle is observed (`org.springframework.beans.factory.DisposableBean` implementors and `destroy-method` callbacks are invoked on beans in the context).

If a Spring-powered bundle that has been stopped is subsequently re-started, a new application context will be created for it.

3.4. The Resource abstraction

The Spring Framework defines a resource abstraction for loading resources within an application context (see Spring's resource abstraction). All resource loading is done through the `org.springframework.core.io.ResourceLoader` associated with the application context. The `org.springframework.core.io.ResourceLoader` is also available to beans wishing to load resources programmatically. Resource paths with explicit prefixes - for example "classpath:" are treated uniformly across all application context types (for example, web application contexts and classpath-based application contexts). Relative resource paths are interpreted differently based on the kind of application context being created. This enables easy integration testing outside the ultimate deployment environment.

OSGi 4.0.x specification defines three different spaces from which a resource can be loaded. Spring-DM supports all of them through its dedicated OSGi-specific application context and dedicated prefixes:

Table 3.1. OSGi resource search strategies

OSGi search strategy	prefix	Explanation
Class Space	`classpath:`	Search the bundle classloader (the bundle and all imported packages). Forces the bundle to be resolved.
JAR File (or JarSpace)	`osgibundlejar:`	Search only the bundle jar. Provides low-level access without requiring the bundle to be resolved.
Bundle Space	`osgibundle:`	Search the bundle jar and its attached fragments (if there are any). Will never create a class loader or resolve the bundle.

Please consult section 4.3.12 of the OSGi specification for an in depth explanation of the differences between them.

Note that if no prefix is specified, the bundle space will be used.

All of the regular Spring resource prefixes such as `file:` and `http:` are also supported, as are the pattern matching wildcards. Resources loaded using such prefixes may come from any location, they are not restricted to being defined within the resource-loading bundle or its attached fragments.

OSGi platforms may define their own unique prefixes for accessing bundle contents. For example, Equinox defines the `bundleresource:` and `bundlentry:` prefixes). These platform specific prefixes may also be used with Spring OSGi, at the cost of course of tying yourself to a particular OSGi implementation.

3.5. Accessing the BundleContext

In general there is no need to depend on any OSGi APIs when using the Spring Dynamic Modules support. If you *do* need access to the OSGi `BundleContext` object for your bundle, then Spring makes this easy to do.

The OSGi application context created by the Spring extender will automatically contain a bean of type `BundleContext` and with name `bundleContext`. You can inject a reference to this bean into any bean in the application context either by-name or by-type. In addition, Spring Dynamic Modules defines the interface `org.springframework.osgi.context.BundleContextAware`:

```
public interface BundleContextAware {
  public void setBundleContext(BundleContext context);
}
```

Any bean implementing this interface will be injected with a reference to the bundle context when it is configured by Spring. If you wish to use this facility within a bundle, remember to import the package `org.springframework.osgi.context` in your bundle manifest.

3.6. Application Context Destruction

The application context is bound to the bundle in which it lives. Thus, if the declaring bundle is being shutdown, the application context will be destroyed as well, all exported services being unregistered and all service imported dispose of.

Note however that a bundle can be closed individually or as part of a bigger event such as shutting down the entire OSGi platform. In this case or when the extender bundle is being closed down, the managed application contexts will be closed in a managed manner based on the service dependencies between them. Please see the next section for more details.

3.7. Stopping the extender bundle

If the extender bundle is stopped, then all the application contexts created by the extender will be destroyed. Application contexts are shutdown in the following order:

1. Application contexts that do not export any services, or that export services that are not currently referenced, are shutdown in reverse order of bundle id. (Most recently installed bundles have their application contexts shutdown first).

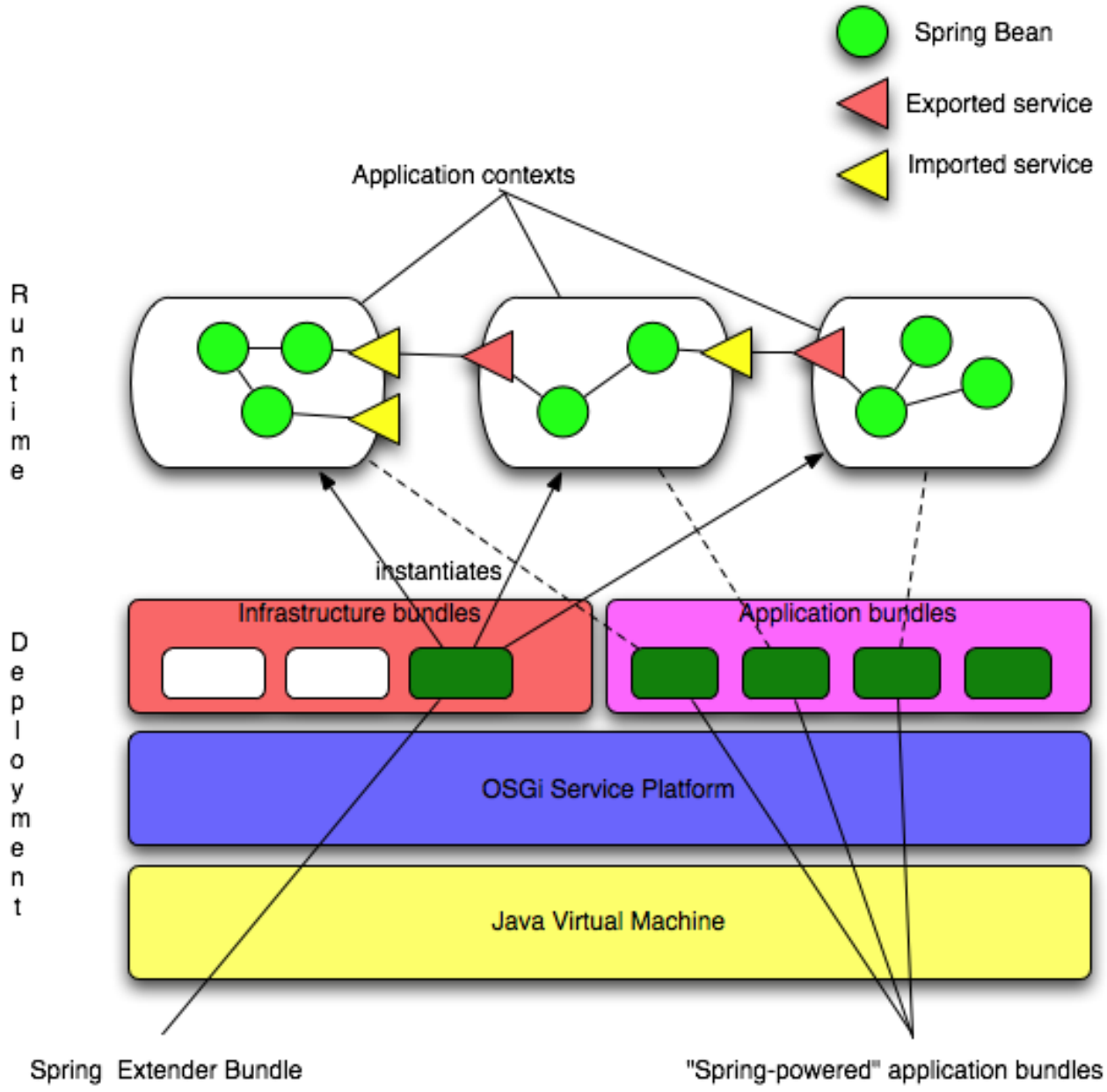
2. Shutting down the application contexts in step (1) may have released references these contexts were holding such that there are now additional application contexts that can be shutdown. If so, repeat step 1 again.

3. If there are no more active application contexts, we have finished. If there *are* active application contexts then there must be a cyclic dependency of references. The circle is broken by determining the highest ranking service exported by each context: the bundle with the lowest ranking service in this set (or in the event of a tie, the highest service id), is shut down. Repeat from step (1).

Chapter 4. Packaging and Deploying Spring-based OSGi applications

A traditional Spring application uses either a single application context, or a parent context containing service layer, data layer, and domain objects with a child context containing web layer components. The application context may well be formed by aggregating the contents of multiple configuration files.

When deploying an application to OSGi the more natural structure is to package the application as a set of peer bundles (application contexts) interacting via the OSGi service registry. Independent subsystems should be packaged as independent bundles or sets of bundles (vertical partitioning). A subsystem may be package in a single bundle, or divided into several bundles partitioned by layer (horizontal partitioning). A straightforward web application may for example be divided into four modules (bundles): a web bundle, service layer bundle, data layer bundle, and domain model bundle. Such an application would look like this:



In this example the data layer bundle yields a data layer application context that contains a number of internal components (beans). Two of those beans are made publicly available outside of the application context by

publishing them as services in the OSGi service registry.

The service layer bundle yields a service layer application context that contains a number of internal components (beans). Some of those components depend on data layer services, and import those services from the OSGi service registry. Two of the service layer components are made externally available as services in the OSGi service registry.

The web component bundle yields a web application context that contains a number of internal components (beans). Some of those components depend on application services, and import those services from the OSGi service registry. Since the domain model bundle contributes only domain model types, but does not need to create any components of its own, it has no associated application context.

4.1. Bundle format and Manifest headers

Each application module should be packaged as an OSGi bundle. A bundle is essentially a jar file with a `META-INF/MANIFEST.MF` file containing a series of headers recognized by the OSGi Service Platform. See the OSGi Service Platform Core Specification section 3.2 for details. Some OSGi implementations may support exploded jar files, but the format remains the same.

The Spring extender recognizes a bundle as "Spring-powered" and will create an associated application context when the bundle is started and one or both of the following conditions is true:

- The bundle path contains a folder `META-INF/spring` with one or more files in that folder with a '.xml' extension.
- `META-INF/MANIFEST.MF` contains a manifest header `Spring-Context`.

In addition, if the optional `SpringExtender-Version` header is declared in the bundle manifest, then the extender will only recognize bundles where the specified version constraints are satisfied by the version of the extender bundle (`Bundle-Version`). The value of the `SpringExtender-Version` header must follow the syntax for a version range as specified in section 3.2.5 of the OSGi Service Platform Core Specification.

In the absence of the `Spring-Context` header the extender expects every ".xml" file in the `META-INF/spring` folder to be a valid Spring configuration file, and all directives (see below) take on their default values.

An application context is constructed from this set of files. A suggested practice is to split the application context configuration into at least two files, named by convention *modulename*-context.xml and *modulename*-osgi-context.xml. The *modulename*-context.xml file contains regular bean definitions independent of any knowledge of OSGi. The *modulename*-osgi-context.xml file contains the bean definitions for importing and exporting OSGi services. It may (but is not required to) use the Spring Dynamic Modules OSGi schema as the top-level namespace instead of the Spring 'beans' namespace.

The `Spring-Context` manifest header may be used to specify an alternate set of configuration files. The resource paths are treated as relative resource paths and resolve to entries defined in the bundle and the set of attached fragments. When the `Spring-Context` header defines at least one configuration file location, any files in `META-INF/spring` are ignored unless directly referenced from the `Spring-Context` header.

The syntax for the `Spring-Context` header value is:

```
Spring-Context-Value ::= context ( ',' context ) *
context ::= path ( ';' path ) * (';' directive) *
```

This syntax is consistent with the OSGi Service Platform common header syntax defined in section 3.2.3 of the OSGi Service Platform Core Specification.

For example, the manifest entry:

```
Spring-Context: config/account-data-context.xml, config/account-security-context.xml
```

will cause an application context to be instantiated using the configuration found in the files `account-data-context.xml` and `account-security-context.xml` in the bundle jar file.

A number of directives are available for use with the `Spring-Context` header. These directives are:

- *create-asynchronously* (false|true): controls whether the application context is created asynchronously (the default), or synchronously.

  For example:

  ```
  Spring-Context: *;create-asynchronously=false
  ```

  Creates an application context synchronously, using all of the "*.xml" files contained in the `META-INF/spring` folder.

  ```
  Spring-Context: config/account-data-context.xml;create-asynchrously:=false
  ```

  Creates an application context synchronously using the `config/account-data-context.xml` configuration file. Care must be taken when specifying synchronous context creation as the application context will be created on the OSGi event thread, blocking further event delivery until the context is fully initialized. If an error occurs during the synchronous creation of the application context then a `FrameworkEvent.ERROR` event is raised. The bundle will still proceed to the `ACTIVE` state.

- *wait-for-dependencies* (true|false): controls whether or not application context creation should wait for any mandatory service dependencies to be satisfied before proceeding (the default), or proceed immediately without waiting if dependencies are not satisfied upon startup.

  For example:

  ```
  Spring-Context: config/osgi-*.xml;wait-for-dependencies:=false
  ```

  Creates an application context using all the files matching "osgi-*.xml" in the config directory. Context creation will begin immediately even if dependencies are not satisfied. This essentially means that mandatory service references are treated as though they were optional - clients will be injected with a service object that may not be backed by an actual service in the registry initially. See section 5.2 for more details.

- *timeout* (300): the time to wait (in seconds) for mandatory dependencies to be satisfied before giving up and failing application context creation. This setting is ignored if `wait-for-dependencies:=false` is specified. The default is 5 minutes (300 seconds).

  For example:

  ```
  Spring-Context: *;timeout:=60
  ```

  Creates an application context that waits up to 1 minute (60 seconds) for its mandatory dependencies to

appear.

- *publish-context* (true|false): controls whether or not the application context object itself should be published in the OSGi service registry. The default is to publish the context.

  For example:

```
Spring-Context: *;publish-context:=false
```

  If there is no Spring-Context manifest entry, or no value is specified for a given directive in that entry, then the directive takes on its default value.

4.2. Required Spring Framework and Spring Dynamic Modules Bundles

The Spring Dynamic Modules project provides an number of bundle artifacts that must be installed in your OSGi platform in order for the Spring extender to function correctly:

- The extender bundle itself, `org.springframework.osgi.extender`
- The core implementation bundle for the Spring Dynamic Modules support, `org.springframework.osgi.core`
- The Spring Dynamic Modules I/O support library bundle, `org.springframework.osgi.io`

In addition the Spring Framework provides a number of bundles that are required to be installed. As of release 2.5 of the Spring Framework, the Spring jars included in the Spring distribution are valid OSGi bundles and can be installed directly into an OSGi platform. The minimum required set of bundles is:

- spring-core.jar (bundle symbolic name `org.springframework.bundle.spring.core`)
- spring-context.jar (bundle symbolic name `org.springframework.bundle.spring.context`)
- spring-beans.jar (bundle symbolic name `org.springframework.bundle.spring.beans`)
- spring-aop.jar (bundle symbolic name `org.springframework.bundle.spring.aop`)

In additional the following supporting library bundles are required. OSGi-ready versions of these libraries are shipped with the Spring Dynamic Modules distribution.

- aopalliance
- backport-util (when running on JDK 1.4)
- cglib-nodep (when proxying classes rather then interfaces, needed in most cases)
- commons-logging API (SLF4J version highly recommended)
- logging implementation such as log4j

4.3. Spring XML authoring support

Spring 2.0 introduced (among other things) easier XML configuration and extensible XML authoring. The latter gives the ability of creating custom schemas that are discovered automatically (in non-OSGi environment) by the Spring XML infrastructure by including them in the classpath. Spring-DM is aware of this process and supports it in OSGi environments so that custom schemas are available to bundles that use them without any extra code or manifest declaration.

All bundles deployed in the OSGi space (whether they are `Spring-powered` or not) are scanned by Spring-DM for custom Spring namespace declaration (by checking the bundle space for`META-INF/spring.handlers` and `META-INF/spring.schemas`). If these are found, Spring-DM will make the schemas and the namespaces available through an OSGi service that will be automatically used by Spring-powered bundles. This mean that if you deploy a bundle that uses a custom schema, all you have to do is deploy the library that provides the namespace parser and the schema. Bundles that embedded inside their classpath libraries that provide custom schemas will use these over those available in the OSGi space. However, the namespaces of the embedded libraries will not shared with other bundles, that is, they will not be seen by any other bundle.

In short, with using Spring-DM, custom Spring namespaces are supported transparently without any additional work. Embedded namespace providers will have priority but will not be shared, as opposed to providers deployed as bundles which will be seen (and used) by others.

4.4. Importing and Exporting packages

Refer to the OSGi Service Platform for details of the `Import-Package` and `Export-Package` manifest headers. Your bundle will need an `Import-Package` entry for every external package that the bundle depends on. If your bundle provides types that other bundles need access to, you will need `Export-Package` entries for every package that should be available from outside of the bundle.

4.5. Considerations when using external libraries

What is the context class loader?

The thread context class loader was introduced in J2SE without much fanfare. Below is a short definition for it, quoted from one of the tutorials available on Java site:

The Java 2 platform also introduced the notion of *context class loader*. A thread's context class loader is, by default, set to the context class loader of the thread's parent. The hierarchy of threads is rooted at the primordial thread (the one that runs the program). The context class loader of the primordial thread is set to the class loader that loaded the application. So unless you explicitly change the thread's context class loader, its context class loader will be the application's class loader. That is, the context class loader can load the classes that the application can load. This loader is used by the Java runtime such as the RMI (Java Remote Method Invocation) to load classes and resources on behalf of the user application. The context class loader, like any Java 2 platform class loader, has a parent class loader and supports the same delegation model for class loading described previously.

Many enterprise application libraries assume that all of the types and resources that comprise the application are accessible through the context class loader. While most developers do not use the context class loader, the loader is used heavily by application servers, containers or applications that are multi-threaded.

In OSGi R4, the set of types and resources available through the context class loader is undefined. This means that the OSGi platform does not make a guarantee of the thread context class loader value or in other words, it does not manage it.

Thus code (for example libraries) that performs manual class loading or that generates new classes dynamically can cause problems when executed inside an OSGi environment.

Spring Dynamic Modules guarantees that during the creation of an application context on behalf of a given bundle, all of the types and resources on the bundle's classpath are accessible via the context class loader. Spring Dynamic Modules also allows you to control what is accessible through the context class loader when invoking external services and when servicing requests on exported services. See section 5 for details on this.

Work is underway in the OSGi R5 timeframe to provide standardized support for dealing with generated classes and implicit class path dependencies introduced by third-party libraries. In the interim you may need to rely on workarounds such as the `DynamicImport-Package` manifest header, or the facilities provided by specific OSGi implementations such as Equinox's buddy mechanism. The Spring Dynamic Modules documentation contains more details on known issues with common enterprise libraries and the workarounds.

4.6. Diagnosing problems

Your chosen OSGi platform implementation should be able to provide you with a good deal of information about the current status of the OSGi environment. For example, starting Equinox with the `-console` argument provides a command-line console through which you can determine which bundles are installed and their states, the packages and services exported by bundles, find out why a bundle has failed to resolve, and drive bundles through the lifecycle.

In addition, Spring itself and the Spring Dynamic Modules bundles contain extensive logging instrumentation that can help you diagnose problems. The recommended approach is to deploy the Simple Logging Facade for Java (slf4j) slf4j-api.jar and slf4j-log4j13.jar bundles (the jar files distributed by the project are valid OSGi bundles). Then you simply need to create a `log4j.properties` file in the root of your bundle classpath.

Note that Spring Dynamic Modules uses commons-logging API internally which means that its logging implementation is fully pluggable. Please see the FAQ and Resources pages for more information on other logging libraries besides log4j.

Chapter 5. The Service Registry

The OSGi service registry enables a bundle to publish objects to a shared registry, advertised via a given set of Java interfaces. Published services also have service properties associated with them in the registry.

Spring Dynamic Modules provides an `osgi` namespace for Spring (see appendix-schema) that can be used to export Spring beans as OSGi services, and to define references to services obtained via the service registry. The namespace elements may be used nested inside another top-level namespace (typically the Spring `beans` namespace), or within the top-level `osgi` element.

The following example shows the use of the `osgi` namespace within the familiar Spring beans element:

```
<?xml version="1.0" encoding="UTF-8"?>;
<beans xmlns="http://www.springframework.org/schema/beans"                    ❶
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:osgi="http://www.springframework.org/schema/osgi"                    ❷
   xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd           ❸
       http://www.springframework.org/schema/osgi
       http://www.springframework.org/schema/osgi/spring-osgi.xsd">

    <osgi:service id="simpleServiceOsgi" ref="simpleService"                  ❹
      interface="org.xyz.MyService" />
</beans>
```

❶ Use Spring Framework `beans` schema as the default namespace.
❷ Import Spring Dynamic Modules schema and associate a prefix with its namespace (`osgi` in this example).
❸ Make sure to import Spring beans schema version *2.5*.
❹ Use Spring Dynamic Modules elements using the declared namespace prefix (in this example `osgi`).

Using the OSGi namespace as a top-level namespace, the same service would be declared as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans                                                                  ❶
   xmlns="http://www.springframework.org/schema/osgi"                         ❷
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:beans="http://www.springframework.org/schema/beans"                  ❸
   xsi:schemaLocation="http://www.springframework.org/schema/osgi
       http://www.springframework.org/schema/osgi/spring-osgi.xsd
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">         ❹

    <service id="simpleServiceOsgi" ref="simpleService"                       ❺
       interface="org.xyz.MyService" />

</beans:beans>                                                                ❶
```

❶ `beans` root element has to be prefixed with Spring Framework beans schema prefix (`beans` in this example).
❷ Use Spring Dynamic Modules schema as the default namespace.
❸ Import Spring Framework `beans` schema and associate a prefix with its namespace (`beans` in this example).
❹ Make sure to import Spring beans schema version *2.5*.
❺ Use Spring Dynamic Modules elements without any prefix.

Using the OSGi namespace as a top-level namespace is particularly convenient when following the recommendation of app-deploy:headers to use a dedicated configuration file for all OSGi-related declarations.

5.1. Exporting a Spring bean as an OSGi service

The `service` element is used to define a bean representing an exported OSGi service. At a minimum you must specify the bean to be exported, and the *service interface* that the service advertises.

For example, the declaration

```
<service ref="beanToPublish" interface="com.xyz.MessageService"/>
```

exports the bean with name `beanToPublish` with interface `com.xyz.MessageService`. The published service will have a service property with the name `org.springframework.osgi.bean.name` set to the name of the target bean being registered (`beanToPublish` in this case).

The bean *defined* by the `service` element is of type `org.osgi.framework.ServiceRegistration` and is the `ServiceRegistration` object resulting from registering the exported bean with the OSGi service registry. By giving this bean an id you can inject a reference to the `ServiceRegistration` object into other beans if needed. For example:

```
<service id="myServiceRegistration" ref="beanToPublish"
    interface="com.xyz.MessageService"/>
```

As an alternative to exporting a named bean, the bean to be exported to the service registry may be defined as an anonymous inner bean of the service element. Typically the top-level namespace would be the `beans` namespace when using this style:

```
<osgi:service interface="com.xyz.MessageService">
  <bean class="SomeClass">
     ...
  </bean>
</osgi:service>
```

If the bean to be exported implements the `org.osgi.framework.ServiceFactory` interface then the `ServiceFactory` contract is honored as per section 5.6 of the OSGi Service Platform Core Specification. As an alternative to implementing this OSGi API, Spring Dynamic Modules introduces a new bean scope, the `bundle` scope. When a bean with bundle scope is exported as an OSGi service then one instance of the bean will be created for each unique client (service importer) bundle that obtains a reference to it through the OSGi service registry. When a service importing bundle is stopped, the bean instance associated with it is disposed. To declare a bean with `bundle` scope simply use the `scope` attribute of the `bean` element:

```
<osgi:service ref="beanToBeExported" interface="com.xyz.MessageService"/>

<bean id="beanToBeExported" scope="bundle" class="com.xyz.MessageServiceImpl"/>
```

5.1.1. Controlling the set of advertised service interfaces for an exported service

The OSGi Service Platform Core Specification defines the term *service interface* to represent the specification of a service's public methods. Typically this will be a Java interface, but the specification also supports registering service objects under a class name, so the phrase *service interface* can be interpreted as referring to either an interface or a class.

There are several options for specifying the service interface(s) under which the exported service is registered. The simplest mechanism, shown above, is to use the `interface` attribute to specify a fully-qualified interface

name. To register a service under multiple interfaces the nested `interfaces` element can be used in place of the `interface` attribute.

```
<osgi:service ref="beanToBeExported">
  <osgi:interfaces>
     <value>com.xyz.MessageService</value>
     <value>com.xyz.MarkerInterface</value>
  </osgi:interfaces>
</osgi:service>
```

It is illegal to use both `interface` attribute and `interfaces` element at the same time - use only one of them.

5.1.1.1. Detecting the advertised interfaces at runtime

Hierarchy visibility

Note that when using `auto-export`, only types visible to the bundle exporting the service are registered. For example, a super-interface `SI` would not be exported as a supported service interface even when using `auto-export="interfaces"` if `SI` was not on the exporting bundle's classpath.

Even if exported service class does implement `SI` transitively based on its parent, if the declaring bundle doesn't import the interface, the class is unknown to the exported service. While this might seem counter intuitive, it is actually one of the most powerful features of OSGi which give the bundle authors control over the class visibility and path.

Please see the FAQ for a more detailed explanation.

Using the `auto-export` attribute you can avoid the need to explicitly declare the service interfaces at all by analyzing the object class hierarchy and its interfaces.

The `auto-export` attribute can have one of four values:

- `disabled` : the default value; no auto-detected of service interfaces is undertaken and the `interface` attribute or `interfaces` element must be used instead.
- `interfaces` : the service will be registered using all of the Java interface types implemented by the bean to be exported
- `class-hierarchy` : the service will be registered using the exported bean's implementation type and super-types
- `all-classes` : the service will be registered using the exported bean's implementation type and super-types plus all interfaces implemented by the bean.

`auto-export` and `interface(s)` option are not exclusive; both can be used at the same time for fine grained control over the advertised interfaces if there is such a need. However, the former option should be enough for most cases.

For example, to automatically register a bean under all of the interfaces that it supports you would declare:

```
<service ref="beanToBeExported" auto-export="interfaces"/>
```

Given the interface hierarchy:

```
public interface SuperInterface {}
```

```
public interface SubInterface extends SuperInterface {}
```

then a service registered as supporting the `SubInterface` interface is *not* considered a match in OSGi when a lookup is done for services supporting the `SuperInterface` interface. For this reason it is a best practice to export all interfaces supported by the service being registered explicitly, using either the `interfaces` element or `auto-export="interfaces"`.

5.1.2. Controlling the set of advertised properties for an exported service

As previously described, an exported service is always registered with the service property `org.springframework.osgi.bean.name` set to the name of the bean being exported. Additional service properties can be specified using the nested `service-properties` element. The `service-properties` element contains key-value pairs to be included in the advertised properties of the service. The key must be a string value, and the value must be a type recognized by OSGi Filters. See section 5.5 of the OSGi Service Platform Core Specification for details of how property values are matched against filter expressions.

The `service-properties` element must contain at least one nested `entry` element from the Spring beans namespace. For example:

```
<service ref="beanToBeExported" interface="com.xyz.MyServiceInterface">
  <service-properties>
    <beans:entry key="myOtherKey" value="aStringValue"/>
    <beans:entry key="aThirdKey" value-ref="beanToExposeAsProperty"/>
  </service-properties>
</service>
```

The Spring Dynamic Modules roadmap includes support for exporting properties registered in the OSGi Configuration Administration service as properties of the registered service. See appendix-roadmap for more details.

5.1.3. The depends-on attribute

Spring will manage explicit dependencies of a service element, ensuring for example that the bean to be exported as a service is fully constructed and configured before exporting it. If a service has implicit dependencies on other components (including other service elements) that must be fully initialized before the service can be exported, then the optional `depends-on` attribute can be used to express these dependencies.

```
<service ref="beanToBeExported" interface="com.xyz.MyServiceInterface"
     depends-on="myOtherComponent"/>
```

5.1.4. The context-class-loader attribute

The OSGi Service Platform Core Specification (most current version is 4.1 at time of writing) does not specify what types and resources are visible through the context class loader when an operation is invoked on a service obtained via the service registry. Since some services may use libraries that make certain assumptions about the context class loader, Spring Dynamic Modules enables you to explicitly control the context class loader during service execution. This is achieved using the option `context-class-loader` attribute of the service element.

The permissible values for the `context-class-loader` attribute are `unmanaged` (the default) and `service-provider`. When the `service-provider` value is specified, Spring Dynamic Modules ensures that the context class loader can see all of the resources on the class path of the bundle exporting the service.
When setting `context-class-loader` to `service-provider`, the service object will be proxied to handle the

class loader. If the service advertises any concrete class then CGLIB library is required .

5.1.5. The ranking attribute

When registering a service with the service registry, you may optionally specify a service ranking (see section 5.2.5 of the OSGi Service Platform Core Specification). When a bundle looks up a service in the service registry, given two or more matching services the one with the highest ranking will be returned. The default ranking value is zero. To explicitly specify a ranking value for the registered service, use the optional `ranking` attribute.

```
<service ref="beanToBeExported" interface="com.xyz.MyServiceInterface"
  ranking="9"/>
```

5.1.6. `service` element attribute

As a summary, the following table lists the attributes names, possible values and a short description for each of them.

Table 5.1. OSGi <service> attributes

Name	Values	Description
interface	fully qualified class name (such as `java.lang.Thread`)	the fully qualified name of the class under which the object will be exported
ref	any bean name	Reference to the named bean to be exported as a service in the service registry.
context-class-loader	unmanaged | service-provider	Defines how the context class loader will be managed when an operation is invoked on the exported service. The default value is `unmanaged` which means that no management of the context class loader is attempted. A value of `service-provider` guarantees that the context class loader will have visibility of all the resources on the class path of bundle exporting the service.
auto-export	disabled | interfaces | class-hierarchy | all-classes	Enables Spring to automatically manage the set of service interfaces advertised for the service. By default this facility is `disabled`. A value of `interfaces` advertises all of the Java interfaces supported by the exported service. A value of `class-hierarchy` advertises all the Java classes in the hierarchy of the exported

Name	Values	Description
		service. A value of `all-classes` advertises all Java interfaces and classes.
ranking	any integer value	Specify the service ranking to be used when advertising the service. Default value is 0.

5.1.7. Service registration and unregistration lifecycle

The service defined by a `service` element is registered with the OSGi service registry when the application context is first created. It will be unregistered automatically when the bundle is stopped and the application context is disposed.

If you need to take some action when a service is unregistered because its dependencies are not satisfied (or when it is registered), then you can define a listener bean using the nested `registration-listener` element.

The declaration of a registration listener must use either the `ref` attribute to refer to a top-level bean definition, or declare an anonymous listener bean inline. For example:

```
<service ref="beanToBeExported" interface="SomeInterface">
  <registration-listener ref="myListener"                                    ❶
    registration-method="serviceRegistered"                                  ❷
    unregistration-method="serviceUnregistered"/>                            ❷
  <registration-listener
     registration-method="register">                                         ❸
     <bean class="SomeListenerClass"/>                                       ❹
  </registration-listener>
</service>
```

❶ Listener declaration referring to a top-level bean declaration.
❷ Indicate the `registration` and `unregistration` methods.
❸ Declare only a `registration` custom method for this listener.
❹ Nested listener bean declaration.

The optional `registration-method` and `unregistration-method` attributes specify the names of the methods defined on the listener bean that are to be invoked during registration and unregistration. A registration and unregistration callback methods must have a signature matching one of the following formats:

```
public void anyMethodName(ServiceType serviceInstance, Map serviceProperties);
```

```
public void anyMethodName(ServiceType serviceInstance, Dictionary serviceProperties);
```

where `ServiceType` can be any type compatible with the exported service interface of the service.

The register callback is invoked when the service is initially registered at startup, and whenever it is subsequently re-registered. The unregister callback is invoked during the service unregistration process, no matter the cause (such as the owning bundle stopping).

Spring-DM will use the declared `ServiceType` argument type and invoke the registration/unregistration method only when a service of a compatible type will be registered/unregistered.

`serviceProperties` represents a map holding all the properties of the registered/unregistered service. To

preserve compatibility with the OSGi specification this argument can be cast, if needed, to a `java.util.Dictionary`.

5.1.7.1. Using `OsgiServiceRegistrationListener` interface

While we discourage, it is possible to implement a Spring-DM specific interface, namely `org.springframework.osgi.service.exporter.OsgiServiceRegistrationListener` which avoids the need to declare the `registration-method` and `unregistration-method`. However, by implementing `OsgiServiceRegistrationListener`, your code becomes Spring-DM aware (which goes against the POJO philosophy).

It is possible for a listener to implement `OsgiServiceRegistrationListener` interface and declare custom methods. In this case, the Spring-DM interface methods will be called first, followed by the custom methods.

5.2. Defining references to OSGi services

Spring Dynamic Modules supports the declaration of beans that represent services accessed via the OSGi Service Registry. In this manner references to OSGi services can be injected into application components. The service lookup is made using the service interface type that the service is required to support, plus an optional filter expression that matches against the service properties published in the registry.

For some scenarios, a single matching service that meets the application requirements is all that is needed. The `reference` element defines a reference to a single service that meets the required specification. In other scenarios, especially when using the OSGi whiteboard pattern, references to *all available* matching services are required. Spring Dynamic Modules supports the management of this set of references as a `List`, `Set` collection.

5.2.1. Referencing an individual service

The `reference` element is used to define a reference to a service in the service registry.

Since there can be multiple service matching a given description, the service returned is the service that would be returned by a call to `BundleContext.getServiceReference`. This means that the service with the highest ranking will be returned, or if there is a tie in ranking, the service with the lowest service id (the service registered first with the framework) is returned (please see Section 5 from the OSGi spec for more information on the service selection algorithm).

5.2.1.1. Controlling the set of advertised interfaces for the imported service

The `interface` attribute identifies the service interface that a matching service must implement. For example, the following declaration creates a bean `messageService`, which is backed by the service returned from the service registry when querying it for a service offering the `MessageService` interface.

```
<reference id="messageService" interface="com.xyz.MessageService"/>
```

Just like the `service` declaration, when specifying multiple interfaces, use the nested `interfaces` element instead of `interface` attribute:

```
<osgi:reference id="importedOsgiService">
  <osgi:interfaces>
     <value>com.xyz.MessageService</value>
     <value>com.xyz.MarkerInterface</value>
  </osgi:interfaces>
</osgi:reference>
```

It is illegal to use both `interface` attribute and `interfaces` element at the same time - use only one of them.

The bean defined by reference element implements all of the advertised interfaces of the service that are visible to the bundle (called *greedy proxying*). If the registered service interfaces include Java class types (as opposed to interface types) then support for these types is subject to the restrictions of Spring's AOP implementation (see the Spring Reference Guide). In short, if the specified interfaces are classes (rather then interfaces), then `cglib` library must be available, and `final` methods are not supported.

5.2.1.2. The `filter` attribute

The optional `filter` attribute can be used to specify an OSGi filter expression and constrains the service registry lookup to only those services that match the given filter.

For example:

```
<reference id="asyncMessageService" interface="com.xyz.MessageService"
  filter="(asynchronous-delivery=true)"/>
```

will match only OSGi services that advertise `MessageService` interface and have the property named `asynchronous-delivery` set to value `true`.

5.2.1.3. The `bean-name` attribute

The `bean-name` attribute is a convenient short-cut for specifying a filter expression that matches on the `bean-name` property automatically set when exporting a bean using the `service` element (see service-registry:export).

For example:

```
<reference id="messageService" interface="com.xyz.MessageService"
   bean-name="defaultMessageService"/>
```

will match only OSGi services that advertise `MessageService` interface and have the property named `org.springframework.osgi.bean.name` set to value `defaultMessageService`. In short, this means finding all Spring-DM exported beans that implement interface `MessageService` and are named `defaultMessageService`.

5.2.1.4. The `cardinality` attribute

Nested <reference> declarations

In order for Spring-DM to detect mandatory dependencies, any nested/inner reference declaration will be transformed into top-level one with a generated name.

The `cardinality` attribute is used to specify whether or not a matching service is required at all times. A cardinality value of `1..1` (the default) indicates that a matching service must always be available. A cardinality value of `0..1` indicates that a matching service is not required at all times (see section 4.2.1.6 for more details). A `reference` with cardinality `1..1` is also known as a *mandatory* service reference and, by default, application context creation is deferred until the reference is satisfied.

> **Note**
>
> It is an error to declare a mandatory reference to a service that is also exported by the same bundle, this behavior can cause application context creation to fail through either deadlock or timeout.

5.2.1.5. The depends-on attribute

The `depends-on` attribute is used to specify that the service reference should not be looked up in the service registry until the named dependent bean has been instantiated.

5.2.1.6. The context-class-loader attribute

The OSGi Service Platform Core Specification (latest version is 4.1 at time of writing) does not specify what types and resources are visible through the context class loader when an operation is invoked on a service obtained via the service registry. Since some services may use libraries that make certain assumptions about the context class loader, Spring Dynamic Modules enables you to explicitly control the context class loader during service invocation. This is achieved using the option `context-class-loader` attribute of the `reference` element.

context class loader management on the importer and exporter

Spring-DM has the ability to do context class loader management on both the importer and exporter side. Normally, if Spring-DM works on both sides, only one side should have this feature enabled. However, if both sides (importer and exporter) take advantage of this capability, the last entity in the call chain will win. This means that the exporter setting, if enabled, will always override the importer setting (whatever that is).

The permissible values for the `context-class-loader` attribute are:

- `client` - during the service invocation, the context class loader is guaranteed to be able to see types on the classpath of the invoking bundle. This is the default option.
- `service-provider` - during the service invocation, the context class loader is guaranteed to be able to see types on the classpath of the bundle exporting the service.
- `unmanaged` - no context class loader management will occur during the service invocation

5.2.1.7. `reference` element attributes

As a summary, the following table lists the `reference` element attributes names, possible values and a short description for each of them.

Table 5.2. OSGi <reference> attributes

Name	Values	Description
interface	fully qualified class name (such as `java.lang.Thread`)	The fully qualified name of the class under which the object will be exported.
filter	OSGi filter expression (such as `((asynchronous-delivery=true))`)	OSGi filter expression that is used to constrain the set of matching services in the service registry.
bean-name	any string value	Convenient shortcut for specifying a filter expression that matches on the bean-name property that is

Name	Values	Description
		automatically advertised for beans published using the <service> element.
context-class-loader	client | service-provider | unmanaged	Defines how the context class loader is managed when invoking operations on a service backing this service reference. The default value is `client` which means that the context class loader has visibility of the resources on this bundle's classpath. Alternate options are `service-provider` which means that the context class loader has visibility of resources on the bundle classpath of the bundle that exported the service, and `unmanaged` which does not do any management of the context class loader.
cardinality	0..1 | 1..1	Defines the required cardinality of the relationship to the backing service. If not specified, the `default-cardinality` attribute will apply. A value is '1..1' means that a backing service must exist (this is a mandatory service reference). A value of '0..1' indicates that it is acceptable to be no backing service (an optional service reference).
timeout	any positive long	The amount of time (in milliseconds) to wait for a backing service to be available when an operation is invoked. If not specified, the `default-timeout` attribute will apply.

5.2.1.8. `reference` and OSGi Service Dynamics

The bean defined by the `reference` element is unchanged throughout the lifetime of the application context (the object reference remains constant). However, the OSGi service that backs the reference may come and go at any time. For a mandatory service reference (cardinality `1..1`), creation of the application context will block until a matching service is available. For an optional service reference (cardinality `0..1`), the reference bean will be created immediately, regardless of whether or not there is currently a matching service.

When the service backing a `reference` bean goes away, Spring Dynamic Modules tries to replace the backing service with another service matching the reference criteria. An application may be notified of a change in backing service by registering a `listener`. If no matching service is available, then the `reference` is said to be

unsatisfied. An unsatisfied mandatory service causes any exported service (`service` bean) that depends on it to be unregistered from the service registry until such time as the reference is satisfied again. See service-registry:export-import-relationship for more information.

When an operation is invoked on an unsatisfied `reference` bean (either optional or mandatory), the invocation blocks until the reference becomes satisfied. The optional `timeout` attribute of the `reference` element enables a timeout value (in milliseconds) to be specified. If a timeout value is specified and no matching service becomes available within the timeout period, an unchecked `ServiceUnavailableException` is thrown.

5.2.1.9. Getting a hold of the managed service reference

Spring-DM can automatically convert a managed OSGi service to service reference. That is, if the property into which a reference bean is to be injected, has type `ServiceReference` (instead of the service interface supported by the reference), then the managed OSGi `ServiceReference` for the service will be injected in place of the service itself:

```
public class BeanWithServiceReference {
        private ServiceReference serviceReference;
        private SomeService service;

        // getters/setters ommitted
}
```

```
<reference id="service" interface="com.xyz.SomeService"/>

<bean id="someBean" class="BeanWithServiceReference">
  <property name="serviceReference" ref="service"/>                    ❶
  <property name="service" ref="service"/>                             ❷
</bean>
```

❶ Automatic managed service to `ServiceReference` conversion.
❷ Managed service is injected without any conversion

> **Note**
>
> The injected `ServiceReference` is managed by Spring-DM and will change at the same time as the referenced backing OSGi service instance.

5.2.2. Referencing a collection of services

> **Natural vs custom ordering**
>
> Java collection API defines two interfaces for ordering objects - `Comparable` and Comparator. The first is meant to be implemented by objects for providing *natural ordering*. `String`, `Long` or `Date` are good examples of objects that implement the `Comparable` interface.
>
> However, there are cases where sorting is different then the natural ordering or, the objects meant to be sort do not implement Comparable. To address this cases, `Comparator` interface was designed.
>
> For more information on this subject, please consult the Object ordering chapter from Java collection tutorial,

Sometimes an application needs access not simply to any service meeting some criteria, but to *all* services meeting some criteria. Spring-DM allows the matching services may be held in a `List` or `Set` (optionally

sorted).

The difference between using a `List` and a `Set` to manage the collection is one of equality. Two or more services published in the registry (and with distinct service ids) may be "equal" to each other, depending on the implementation of equals used by the service implementations. Only one such service will be present in a set, whereas all services returned from the registry will be present in a list. For more details on collections, see this tutorial.

The `set` and `list` schema elements are used to define collections of services with set or list semantics respectively.

These elements support the attributes `interface`, `filter`, `bean-name`, `cardinality`, and `context-class-loader`, with the same semantics as for the `reference` element. The allowable values for the `cardinality` attribute are `0..N` and `1..N`.

A cardinality value of `0..n` indicates that it is permissible for their to be no matching services. A cardinality value of `1..n` indicates that at least one matching service is required at all times. Such a reference is considered a *mandatory* reference and any exported services from the same bundle (`service` defined beans) that depend on a mandatory reference will automatically be unregistered when the reference becomes unsatisfied, and reregistered when the reference becomes satisfied again.

The bean defined by a `list` element is of type `java.util.List`. The bean defined by a `set` element is of type `java.util.Set`.

The following example defines a bean of type `List` that will contain all registered services supporting the `EventListener` interface:

```
<list id="myEventListeners"
  interface="com.xyz.EventListener"/>
```

The members of the collection defined by the bean are managed dynamically by Spring. As matching services are registered and unregistered in the service registry, the collection membership will be kept up to date. Each member of the collection supports the service interfaces that the corresponding service was registered with and that are visible to the bundle.

Spring-DM supports sorted collections as well, both for set and list.

It is possible to specify a sorting order using either the `comparator-ref` attribute, or the nested `comparator` element. The `comparator-ref` attribute is used to refer to a named bean implementing `java.util.Comparator`. The `comparator` element can be used to define an inline bean. For example:

```
<set id="myServices" interface="com.xyz.MyService"
  comparator-ref="someComparator"/>

<list id="myOtherServices"
  interface="com.xyz.OtherService">
  <comparator>
     <beans:bean class="MyOtherServiceComparator"/>
  </comparator>
</list>
```

To sort using a natural ordering instead of an explicit comparator, you can use the `natural-ordering` element inside of `comparator`. You need to specify the basis for the natural ordering: based on the service references, following the `ServiceReference` natural ordering defined in the OSGi Core Specification section 6.1.2.3; or based on the services themselves (in which case the services must be `Comparable`).

```
<list id="myServices" interface"com.xyz.MyService">
  <comparator><natural-ordering basis="services"/></comparator>
```

```
</list>

<set id="myOtherServices"interface="com.xyz.OtherService">
  <comparator><natural-ordering basis="service-references"/></comparator>
</set>
```

For a sorted set, a `SortedSet` implementation will be created. However, since JDK API do not provide a dedicated `SortedList`interface, the sorted list will implement only the `List` interface.

5.2.2.1. Collection (`list` and `set`) element attributes

`list` and `set` elements support all the attributes available to `reference` element except the `timeout` attribute. See the following table as a summary of the `list` and `set` element attribute names, possible values and a short description for each of them.

Table 5.3. <list>/<set> attributes

Name	Values	Description
interface	fully qualified class name (such as `java.lang.Thread`)	The fully qualified name of the class under which the object will be exported.
filter	OSGi filter expression (such as `((asynchronous-delivery=true))`)	OSGi filter expression that is used to constrain the set of matching services in the service registry.
bean-name	any string value	Convenient shortcut for specifying a filter expression that matches on the bean-name property that is automatically advertised for beans published using the <service> element.
context-class-loader	client | service-provider | unmanaged	Defines how the context class loader is managed when invoking operations on a service backing this service reference. The default value is `client` which means that the context class loader has visibility of the resources on this bundle's classpath. Alternate options are `service-provider` which means that the context class loader has visibility of resources on the bundle classpath of the bundle that exported the service, and `unmanaged` which does not do any management of the context class loader.
cardinality	0..N | 1..N	Defines the required cardinality of the relationship to the backing service. If not specified, the `default-cardinality` attribute will apply. A value is '1..N' means

Name	Values	Description
		that a backing service must exist (this is a mandatory service reference). A value of '0..N' indicates that it is acceptable to be no backing service (an optional service reference).
comparator-ref	any string value	Named reference to a bean acting as comparator for the declaring collection. Declaring a comparator automatically makes the declaring collection sorted.

The table below lists the attributes available for the `comparator/natural` sub element.

Table 5.4. collection <comparator> attributes

Name	Values	Description
basis	service | service-reference	Indicate the element on which *natural ordering* should apply - `service` for considering the service instance and `service-reference` for considering the service reference instead of the service.

5.2.2.2. `list` / `set` and OSGi Service Dynamics

A collection of OSGi services will change its content during the lifetime of the application context since it needs to reflect the state of the OSGi space. As service are registered and unregistered, they will be added or removed from the collection.

While a `reference` declaration will try to find a replacement if the backing service is unregistered, the collection will simply remove the service from the collection. Like `reference`, a collection with cardinality `1..N` is said to be mandatory while a collection with cardinality `0..N` is referred to as being optional. If no matching service is available then only mandatory collections become *unsatisfied*. That is if no service is available invoking an operation on:

- mandatory collection - will throw an unchecked `ServiceUnavailableException`.
- optional collection - will *not* throw any exceptions (however the collection will be empty).

Just like `reference`, mandatory collections will trigger the unregistration of any exported service that depends upon it. See service-registry:export-import-relationship for more information.

5.2.2.3. `Iterator` contract and service collections

The recommend way of traversing a collection is by using an `Iterator`. However, since OSGi services can come and go, the content of the managed service collection will be adjusted accordingly. Spring-DM will

transparently update all `Iterators` held by the user so it is possible to safely traverse the collection while it is being modified. Moreover, the `Iterators` will reflect all the changes made to the collection, even if they occurred after the `Iterators` were created (that is during the iteration). Consider a case where a collection shrinks significantly (for example a big number of OSGi services are shutdown) right after an iteration started. To avoid dealing with the resulting 'dead' service references, Spring-DM iterators do not take collection snapshots (that can be inaccurate) but rather are updated on each service event so they reflect the latest collection state, no matter how fast or slow the iteration is.

It is important to note that a service update will only influence `Iterator` operations that are executed after the event occurred. Services already returned by the iterator will not be updated even if the backing service has been unregistered. As a side note, if an operation is invoked on such a service that has been unregistered, a `ServiceUnavailableException` will be thrown.

To conclude, while a `reference` declaration will search for candidates in case the backing service has been unregistered, a service collections will not replace unregistered services returned to the user. However, it will remove the unregistered services from the collection so future iterations will not encounter them.

Please note that the `Iterator` contract is guaranteed meaning that `next()` method *always* obey the result of the previous `hasNext()` invocation.

Table 5.5. Dynamic service collection `Iterator` contract

hasNext() returned value	next() behaviour
true	*Always* return a non-null value, even when the collection has shrunk as services when away.
false	per `Iterator` contract, `NoSuchElementException` is thrown. This applies even if other services are added to the collection

The behaviour described above, offers a consistent view over the collection even if its structure changes during iteration. To simply *refresh* the iterator, call `hasNext()` again. This will force the `Iterator` to check again the collection status for its particular entry in the iteration.

In addition, any elements added to the collection during iteration over a *sorted* collection will only be visible if the iterator has not already passed their sort point.

5.2.3. Dealing with the dynamics of OSGi imported services

Whether you are using `reference` or `set` or `list`, Spring Dynamic Modules will manage the backing service. However there are cases where the application needs to be aware when the backing service is updated.

Such applications, that need to be aware of when the service backing a `reference` bean is bound and unbound, can register one or more listeners using the nested `listener` element. This element is available on both `reference` and `set`, `list` declarations. In many respects, the service importer listener declaration is similar to the service exporter listener declaration (service-registry:export:lifecycle). The `listener` element refers to a bean (either by name, or by defining one inline) that will receive bind and unbind notifications. If this bean implements Spring-DM's `org.springframework.osgi.service.importer.OsgiServiceLifecycleListener` interface, then the `bind` and `unbind` operations in this interface will be invoked. Instead of implementing this interface (or in addition), custom bind and unbind callback methods may be named.

An example of declaring a listener that implements `OsgiServiceLifecycleListener`:

```
<reference id="someService" interface="com.xyz.MessageService">
  <listener ref="aListenerBean"/>
</reference>
```

An example of declaring an inline listener bean with custom bind and unbind methods:

```
<reference id="someService" interface="com.xyz.MessageService">
  <listener bind-method="onBind" unbind-method="onUnbind">
     <beans:bean class="MyCustomListener"/>
  </listener>
</reference>
```

If the listener bean implements the `OsgiServiceLifecycleListener` interface *and* the listener definition specifies custom bind and unbind operations then both the `OsgiServiceLifecycleListener` operation and the custom operation will be invoked, in that order.

The signature of a custom bind or unbind method must be one of:

```
public void anyMethodName(ServiceType service, Dictionary properties);

public void anyMethodName(ServiceType service, Map properties);

public void anyMethodName(ServiceReference ref);
```

where `ServiceType` can be any type. Please note that bind and unbind callbacks are invoked *only* if the backing service matches the type declared in the method signature(`ServiceType`). If you want the callbacks to be called no matter the type, use `java.lang.Object` as a `ServiceType`.

The `properties` parameter contains the set of properties that the service was registered with.

If the method signature has a single argument of type `ServiceReference` then the `ServiceReference` of the service will be passed to the callback in place of the service object itself.

When the listener is used with a `reference` declaration:

- A *bind* callback is invoked when the reference is initially bound to a backing service, and whenever the backing service is replaced by a new backing service.
- An *unbind* callback is only invoked when the current backing service is unregistered, and no replacement service is immediately available (i.e., the `reference` becomes unsatisfied).

When the listener is used with a collection declaration (`set` or `list`):

- A *bind* callback is invoked when a new service is added to the collection.
- An *unbind* callback is invoked when a service is unregistered and is removed from the collection.

Again note that service collections there is *no* notion of *service rebind*: services are added or removed from the collection.

Bind and unbind callbacks are made synchronously as part of processing an OSGi `serviceChanged` event for the backing OSGi service, and are invoked on the OSGi thread that delivers the corresponding OSGi `ServiceEvent`.

The table below lists the attributes available for the `reference` `listener` sub element.

Table 5.6. OSGi <listener> attributes

Name	Values	Description
ref	bean name reference	Name based reference to another bean acting as listener.
bind-method	string representing a valid method name	The name of the method to be invoked when a backing service is bound.
unbind-method	string representing a valid method name	The name of the method to be invoked when a backing service is bound.

5.2.4. Listener and service proxies

While the importer listener provides access to the OSGi service bound at a certain point, it is important to note that the given argument is *not* the actual service but a *proxy*. This can have subtle side effects especially with regards to service class name and identity. The reason behind using a proxy is to prevent the listener from holding strong reference to the service (which can disappear at any point). Listeners interested in tracking certain services should not rely on instance equality (==). Object equality (`equals/hashcode`) can be used but only if the backing service has exposed the aforementioned methods as part of its contract (normally by declaring them on a certain published interface/class). If these methods are not published, the proxy will invoke its own method, not the targets. This is on purpose since, while the proxy tries to be as transparent as possible, it is up to the developer to define the desired semantics.

Thus, it is recommended (especially for `reference` importers) to do tracking based on just the service interface/contract (not identity), service properties (see `org.osgi.framework.Constants#SERVICE_ID`) or service notification (bind/unbind).

5.2.5. Accessing the caller `BundleContext`

It is sometime useful for an imported service to know which bundle is using it at a certain time. To help with this scenarion, in Spring-DM imported services publish the importing bundle `BundleContext` through `LocalBundleContext` class. Each time a method on the importer is invoked, the caller `BundleContext` will be made available, using a `ThreadLocal`, through `getInvokerBundleContext()`.

Please be careful when using this class since it ties your code to Spring-DM API.

5.3. Exporter/Importer listener best practices

As mentioned above, Spring-DM exporter and importer allow listeners to be used for receiving notifications on when services are bound, unbound, registered or unregistered. Below you can find some guidance advices when working with listeners:

- Do *not* execute long activity tasks inside the listener. If you really have to, use a separate thread for executing the work. The listener are called synchronously and so try to be as fast as possible. Doing work inside the listener prevents other the event to be sent to other listeners and the OSGi service to resume

activity.

- Use listener custom declaration as much as possible - it doesn't tie your code to Spring-DM API and it doesn't enforce certain signature names.
- If find yourself repeating bind/unbind method declarations for your listener definitions, consider using Spring bean definition inheritance to define a common definition that can be reused and customized accordingly.
- Prefer `java.util.Map` instead of `java.util.Dictionary` class. The first is an interface while the latter is a deprecated, abstract class. To preserve compatibility, Spring-DM will pass to the listeners a `Map` implementation that can be casted, if needed, to a `Dictionary`.

5.3.1. Listener and cyclic dependencies

There are cases where an exporter/importer listener needs a reference back to the bean it is defined on:

```
<bean id="listener" class="cycle.Listener">                                    ❶
    <property name="target" ref="importer" />                                  ❷
</bean>

<osgi:reference id="importer" interface="SomeService">                         ❸
    <osgi:listener bind-method="bind" ref="listener" />                        ❹
</osgi:reference>
```

❶ Listener bean
❷ Dependency listener -> importer
❸ Importer declaration
❹ Dependency importer -> listener

The declaration above while valid, creates a dependecy between the `listener` and the importer it is defined upon. In order to create the importer, the `listener` has to be resolved and created but in order to do that, the importer called `service` needs to be retrieved (instantiated and configured). This cycle needs to be broken down so that at least one bean can be fully created and configured. This scenario is supported is supported by Spring-DM for both exporter and importers however, if the listener is defined as a nested bean, the cycle cannot be resolved:

```
<osgi:reference id="importer" interface="SomeService">                         ❶
    <osgi:listener bind-method="bind">                                         ❷
        <bean class="cycle.Listener">                                          ❸
            <property name="target" ref="importer" />                          ❹
        </bean>
    </osgi:listener>
</osgi:reference>
```

❶ OSGi service importer
❷ Dependency between importer -> listener
❸ Nested listener declaration
❹ Dependency nested listener -> importer

Beans and Cycles

Cyclic dependencies (A depends on B which depends back on A) increase the complexity of your configuration and in most cases indicate a design issue. What beans should be created and destroyed first for example? While they are a bad practice, the Spring container makes a best attempt to solve the cyclic

configurations when singletons are involved (since the instances can be cached). However this does not work all the time and depends heavily on your specific configuration (Can the bean class be partially initialized ? Does it rely on special `Aware` interfaces? Are `BeanPostProcessors` involved?)

The example above will fail since `service` bean cannot be initialized as it depends on the listener. The same cycle was seen before but in this case there is subtle yet big different from the container perspective - the listener is declared as a nested/inner-bean (hence the missing bean `id`). Inner beans have the same life cycle as their declaring parents and do not have any name. By definition, they are not tracked by the container and are simply created on demand. Since the importer cannot be partially created and the nested listener cannot be cached, the container cannot break the cycle and create the beans. While the two configurations shown above seem similar, one works while the other does not. Another reason to not use cycles unless you really, really have to.

To conclude, if you need inside the listener to hold a reference to the exporter/importer on which the listener is declared, either declare the listener as a *top-level* bean (as shown before) or consider doing *dependency lookup*. However, the latter approach requires extra contextual information such as the `BeanFactory` to use and the bean name and is more fragile then *dependency injection*.

Note

For those interested in the technical details, neither the exporter and importer cannot be partially initialized since they require the application context `ClassLoader` which is requested through the `BeanClassLoaderAware` which relies on a buit-in `BeanPostProcessor` which is applied only after the bean has been configured and is ready for initialization. If the `ClassLoader` was not required then the exporter/importer could have been partially initialized and the case above supported.

5.4. Service importer global defaults

The `osgi` namespace allows offers two global attributes for specifying default behaviours for all importers declared in that file.

Thus, when using the `osgi` namespace to enclose `set`, `list` or `reference` elements, one can use:

- `default-timeout` - can be used to specify the default timeout (in milliseconds) for all importer elements that do not explicitly specify one. For example:

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:osgi="http://www.springframework.org/schema/osgi"                    ❶
          osgi:default-timeout="5000">                                           ❷

  <reference id="someService" interface="com.xyz.AService" .../>                 ❸

  <reference id="someOtherService" interface="com.xyz.BService"
       timeout="1000" .../>                                                      ❹

</beans:beans>
```

❶ Declare `osgi` namespace prefix.

❷ Declare `default-timeout`(in miliseconds) on the root element. If the default is not set, it will have a value of 5 minutes. In this example, the default value is 5 seconds.

❸ This `reference` will inherit the default timeout value since it does not specify one. This service

reference will have a timeout of 5 seconds.

❹ This `reference` declares a timeout, overriding the default value. This service reference will have a timeout of 1 second.

- `default-cardinality` - can be used to specify the default cardinality for all importer elements that do not explicitly specify one. Possible values are `0..X` and `1..X` where `X` is substituted at runtime to `1` for `reference` elements or `N` for collection types such as `set` or `list`.

  Consider the following example:

```
<beans:beans
      xmlns="http://www.springframework.org/schema/osgi"                                    ❶
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:beans="http://www.springframework.org/schema/beans"                             ❷
      xmlns:osgi="http://www.springframework.org/schema/osgi"                               ❸
         osgi:default-cardinality="0..X"                                                    ❹
         default-lazy-init="false">                                                         ❺

  <reference id="someService" interface="com.xyz.AService"/>                               ❻

  <set id="someSetOfService" interface="com.xyz.BService"/>                                 ❼

  <list id="anotherListOfServices" interface="com.xyz.CService" cardinality="1..N"/>        ❽

</beans:beans>
```

❶ Declare Spring Dynamic Modules schema as the default namespace.

❷ Import Spring Framework beans schema and associate a prefix with its namespace (`beans` in this example).

❸ Import Spring Dynamic Modules schema and associate a prefix with its namespace (`osgi` in this example). This is requires since the global attributes have to be declared to an element (`beans`) belonging to another schema. To avoid ambiguity, the Spring-DM schema is imported under a specified prefix as well.

❹ Declare `default-cardinality` on the root element. If the default is not set, it will have a value of `1..N`. In this example, the default value is `0..N`. Note the `osgi` prefix added to the global attribute.

❺ `beans` element attributes (such as `default-lazy-init`) do not need a prefix since they are declared as being local and unqualified (see the beans schema for more information).

❻ The `reference` declaration will inherit the default cardinality value since it does not specify one. As `reference` represents a single service, its cardinality will be `0..1`.

❼ The `set` declaration will inherit the default cardinality value since it does not specify one. As `set` (or `list`) represents a collection of service, its cardinality will be `0..N`.

❽ The `list` declaration specifies its cardinality (`1..N`), overriding the default value.

The `default-*` attributes allow for concise and shorter declarations as well as easy propagation of changes (such as increasing or decreasing the timeout).

5.5. Relationship between the service exporter and service importer

An exported service may depend, either directly or indirectly, on other services in order to perform its function. If one of these services is considered a *mandatory* dependency (has cardinality `1..x`) and the dependency can no longer be satisfied (because the backing service has gone away and there is no suitable replacement available) then the exported service that depends on it will be automatically unregistered from the service registry - meaning that it is no longer available to clients. If the mandatory dependency becomes satisfied once more (by registration of a suitable service), then the exported service will be re-registered in the service

registry.

This automatic unregistering and re-registering of exported services based on the availability of mandatory dependencies only takes into account declarative dependencies. If exported service `S` depends on bean `A`, which in turn depends on mandatory imported service `M`, and these dependencies are explicit in the Spring configuration file as per the example below, then when `M` becomes unsatisfied `S` will be unregistered. When `M` becomes satisfied again, `S` will be re-registered.

```
<osgi:service id="S" ref="A" interface="SomeInterface"/>

<bean id="A" class="SomeImplementation">
   <property name="helperService" ref="M"/>
</bean>

<!-- the reference element is used to refer to a service
     in the service registry -->
<osgi:reference id="M" interface="HelperService"
     cardinality="1..1"/>
```

If however the dependency from `A` on `M` is not established through configuration as shown above, but instead at runtime through for example passing a reference to `M` to `A` without any involvement from the Spring container, then Spring Dynamic Modules will *not* track this dependency.

Chapter 6. Working with Bundles

Spring-DM offers a dedicated schema element for interacting with existing bundles or for installing new ones. While it is not intended to be used as a replacement for proper OSGi services, the `bundle` element offers a very easy way of executing actions on bundles based on the lifecycle of the application context.

The `bundle` element defines a bean of type `org.osgi.framework.Bundle`. It provides a simple way to work directly with bundles, including driving their lifecycle. In the simplest case all you need to do is specify the `symbolic-name` of the bundle you are interested in:

```
<bundle id="aBundle" symbolic-name="org.xyz.abundle"/>
```

The bean `aBundle` can now be injected into any property of type `Bundle`.

If the needed bundle is not installed, one can use `location` attribute to indicate install or/and the `action`/`destroy-action` attributes provide declarative control over the bundle's lifecycle. The `location` attribute is used to specify a URL where the bundle jar file artifact can be found. The `action` attribute specifies the lifecycle operation to be invoked on the bundle object. The supported action values are `install`, `start`, `update`, `stop`, and `uninstall`. These actions have the same semantics as the operations of the corresponding names defined on the `Bundle` interface (see the OSGi Service Platform Core Specification), with the exception that pre-conditions are weakened to allow for example a *start* action to be specified against a bundle that is not currently installed (it will be installed first).

The following table shows how actions are interpreted for the given Bundle states:

Table 6.1. <bundle> `action` values

Action	UNINSTALLED	INSTALLED/RESOLVED
START	installs and starts the bundle	starts the bundle
UPDATE	installs the bundle and then updates it (`Bundle.update()`)	updates the bundle
STOP	no action taken	no action taken
UNINSTALL	no action taken	bundle is uninstalled

For example:

```
<!-- ensure this bundle is installed and started -->;
<bundle id="aBundle" symbolic-name="org.xyz.abundle"
   location="http://www.xyz.com/bundles/org.xyz.abundle.jar"
   action="start"/>
```

The following table lists the `bundle` element attributes names, possible values and a short description for each of them.

Table 6.2. <bundle> attributes

Name	Values	Description
symbolic-name	any valid symbolic-name String	The bundle symbolic name of the bundle object. Normally used

Name	Values	Description
		when interacting with an already installed bundle.
location	String that can be converted into an `URL`	Location used to install, update or/and identify a bundle.
action	start | stop | install | uninstall | update	Lifecyle action to drive on the bundle. The action is executed at startup.
destroy-action	(same as action)	Lifecyle action to drive on the bundle. The action is executed at shutdown.

The samples that ship with the Spring Dynamic Modules project include further support for a `virtual-bundle` element that can be used to create and install OSGi bundles on the fly from existing artifacts.

Chapter 7. Testing OSGi based Applications

By following best practices and using the Spring Dynamic Modules support, your bean classes should be easy to unit test as they will have no hard dependencies on OSGi, and the few OSGi APIs that you may interact with (such as `BundleContext`) are interface-based and easy to mock. Whether you want to do unit testing or integration testing, Spring-DM can ease your task.

7.1. OSGi Mocks

Mocks vs Stubs

There are various strategies to unit test code that requires colaborators. The two most popular strategies are *stubs* and *mocks*.

See this article by Martin Fowler which describes in detail the difference between them.

Even though most OSGi API are interfaces and creating mocks using a specialized library like EasyMock is fairly simple, in practice the amount of code of setting the code (especially on JDK 1.4) becomes cumbersome. To keep the tests short and concise, Spring-DM provides OSGi mocks under `org.springframework.osgi.mock` package.

It's up to you to decide whether they are useful or not however, we make extensive use of them inside Spring-DM test suite. Below you can find a code snippet that you are likely to encounter in our code base:

```
private ServiceReference reference;
private BundleContext bundleContext;
private Object service;

protected void setUp() throws Exception {
        reference = new MockServiceReference();
        bundleContext = new MockBundleContext() {

                public ServiceReference getServiceReference(String clazz) {
                        return reference;
                }

                public ServiceReference[] getServiceReferences(String clazz, String filter)
                                throws InvalidSyntaxException {
                        return new ServiceReference[] { reference };
                }

                public Object getService(ServiceReference ref) {
                    if (reference == ref)
                       return service;
                    super.getService(ref);
                }
        };

        ...
}

public void testComponent() throws Exception {
    OsgiComponent comp = new OsgiComponent(bundleContext);

    assertSame(reference, comp.getReference());
    assertSame(object, comp.getTarget());
}
```

As ending words, experiment with them and choose whatever style or library you feel most confortable with. In our test suite we use the aforementioned mocks, EasyMock library and plenty of integration testing (see below).

7.2. Integration Testing

What about JUnit4/TestNG?

While JUnit4/TestNG overcome the class inheritance problem that appears when building base JUnit classes, by decoupling the runner from the test through annotations, Spring-DM cannot use them since it has to support Java 1.4.

However, it is planned for the future to provide an optional, JVM 5-based testing extension to integrate the existing testing framework with the aforementioned libraries.

In a restricted environment such as OSGi, it's important to test the visibility and versioning of your classes, the manifests or how your bundles interact with each other (just to name a few).

To ease integration testing, the Spring Dynamic Modules project provides a test class hierarchy (based on `org.springframework.osgi.test.AbstractOsgiTests`) that provides support for writing regular `JUnit` test cases that are then automatically executed in an OSGi environment.

In general, the scenario supported by Spring-DM testing framework is:

- start the OSGi framework (Equinox, Knopflerfish, Felix)
- install and start any specified bundles required for the test
- package the test case itself into a `on the fly` bundle, generate the manifest (if none is provided) and install it in the OSGi framework
- execute the test case inside the OSGi framework
- shut down the framework
- passes the test results back to the originating test case instance that is running outside of OSGi

By following this sequence it is trivial to write JUnit-based integration tests for OSGi and have them integration into any environment (IDE, build (ant, maven), etc.) that can work with JUnit.

The rest of this chapter details (with examples) the features offered by Spring-DM testing suite.

7.2.1. Creating a simple OSGi integration test

While the testing framework contains several classes that offer specific features, it is most likely that your test cases will extend `org.springframework.osgi.test.AbstractConfigurableBundleCreatorTests` (at least this is what we use in practice).

Let's extend this class and interact with the OSGi platform through the `bundleContext` field:

```
public class SimpleOsgiTest extends AbstractConfigurableBundleCreatorTests {

public void testOsgiPlatformStarts() throws Exception {
        System.out.println(bundleContext.getProperty(Constants.FRAMEWORK_VENDOR));
        System.out.println(bundleContext.getProperty(Constants.FRAMEWORK_VERSION));
        System.out.println(bundleContext.getProperty(Constants.FRAMEWORK_EXECUTIONENVIRONMENT));
}
}
```

Simply execute the test as you normally do with any JUnit test. On Equinox 3.2.x, the output is similar to:

```
Eclipse
1.3.0
OSGi/Minimum-1.0,OSGi/Minimum-1.1,JRE-1.1,J2SE-1.2,J2SE-1.3,J2SE-1.4}
```

It's likely that you will see other log statements made by the testing framework during your test execution by these can be disabled and have only an informative value as they don't affect your test execution.

Note that you did not have to create any bundle, write any MANIFEST or bother with imports or exports, let alone starting and shutting down the OSGi platform. The testing framework takes care of these automatically when the test is executed.

Let's do some quering and figure out what the environment in which the tests run is. A simple way to do that is to query the `BundleContext` for the installed bundles:

```
public void testOsgiEnvironment() throws Exception {
        Bundle[] bundles = bundleContext.getBundles();
        for (int i = 0; i < bundles.length; i++) {
                System.out.print(OsgiStringUtils.nullSafeName(bundles[i]));
                System.out.print(", ");
        }
        System.out.println();
}
```

The output should be similar to:

```
OSGi System Bundle, asm.osgi, log4j.osgi, spring-test, spring-osgi-test, spring-osgi-core,
    spring-aop, spring-osgi-io, slf4j-api,
spring-osgi-extender, etc... TestBundle-testOsgiPlatformStarts-com.your.package.SimpleOsgiTest,
```

As you can see, the testing framework installs the mandatory requirements required for running the test such as the Spring, Spring-DM, slf4j jars among others.

7.2.2. Installing test prerequisites

OSGi-friendly libraries

To work on OSGi environments, jars need to declare in their `MANIFEST.MF`, Export or Import packages; that is declare what classes they need or offer to other bundles. Most libraries are OSGi unaware and do not provide the proper manifest entries which means they are unusable in an OSGi environment.

At the moment, there are several initiatives in the open source space to provide the proper manifest - please see the FAQ for more information.

Besides the Spring-DM jars and the test itself is highly likely that you depend on several libraries or your own code for the integration test.

Consider the following test that relies on Apache Commons Lang:

```
import org.apache.commons.lang.time.DateFormatUtils;
    ...
          public void testCommonsLangDateFormat() throws Exception {
                System.out.println(DateFormatUtils.format(new Date(), "HH:mm:ssZZ"));
        }
}
```

Running the test however yields an exception:

```
java.lang.IllegalStateException: Unable to dynamically start generated unit test bundle
     ...
Caused by: org.osgi.framework.BundleException: The bundle could not be resolved.
Reason: Missing Constraint: Import-Package: org.apache.commons.lang.time; version="0.0.0"
    ...
        ... 15 more
```

The test requires `org.apache.commons.lang.time` package but there is no bundle that exports it. Let's fix this by installing a commons-lang bundle.

One can specify the bundles that she wants to be installed using `getTestBundlesNames` or `getTestBundles` method. The first one returns an array of String that indicate the bundle name, package and versioning through as a String while the latter returns an array of `Resources` that can be used directly for installing the bundles. That is, use `getTestBundlesNames` when you rely on somebody else to locate (the most common case) the bundles and `getTestBundles` when you want to locate the bundles yourself.

By default, the test suite uses the local maven2 repository to locate the artifacts. The locator expects the bundle String to be a comma separated values containing the artifact group, name, version and (optionally) type. It's likely that in the future, various other locators will be available. One can plug in their own locator through the `org.springframework.osgi.test.provisioning.ArtifactLocator` interface.

Let's fix our integration test by installing the required bundle (and some extra osgi libraries):

```
protected String[] getTestBundlesNames() {
        return new String[] { "org.springframework.osgi, cglib-nodep.osgi, 2.1.3-SNAPSHOT",
                "org.springframework.osgi, jta.osgi, 1.1-SNAPSHOT",
                "org.springframework.osgi, commons-lang.osgi, 2.3-SNAPSHOT" };
        };
}
```

Rerunning the test should show that these bundles are now installed in the OSGi platform.

> **Note**
>
> The artifacts mentioned above have to exist in your local maven repository.

7.2.3. Advanced testing framework topics

The testing framework allows a lot of customization to be made. This chapter details some of the existing hooks that you might want to know about. However, these are advanced topics as they increase the complexity of your test infrastructure.

7.2.3.1. Customizing the test manifest

There are cases where the auto-generated test manifest does not suite the needs of the test. For example the manifest requires some different headers or a certain package needs to be an optional import.

For simple cases, one can work directly with the generated manifest - in the example below, the bundle class path is being specified:

```
protected Manifest getManifest() {
      // let the testing framework create/load the manifest
      Manifest mf = super.getManifest();
      // add Bundle-Classpath:
      mf.getMainAttributes().putValue(Constants.BUNDLE_CLASSPATH, ".,bundleclasspath/simple.jar");
```

```
      return mf;
}
```

Another alternative is to provide your own manifest by overriding `getManifestLocations()`:

```
protected String getManifestLocation() {
      return "classpath:com/xyz/abc/test/MyTestTest.MF";
}
```

However each manifest needs the following entry:
“Bundle-Activator: org.springframework.osgi.test.JUnitTestActivator”

since without it, the testing infrastructure cannot function properly. Also, one needs to import JUnit, Spring and Spring-DM specific packages used by the base test suite:

```
Import-Package: junit.framework,
  org.osgi.framework,
  org.apache.commons.logging,
  org.springframework.util,
  org.springframework.osgi.service,
  org.springframework.osgi.util,
  org.springframework.osgi.test,
  org.springframework.context
```

Failing to import a package used by the test class will cause the test to fail with a `NoDefClassFoundError` error.

7.2.3.2. Customizing test bundle content

By default, for the on-the-fly bundle, the testing infrastructure uses all the classes, xml and properties files found under `./target/test-classes` folder. This matches the project layout for maven which is used (at the moment by Spring-DM). These settings can be configured in two ways:

1. programmatically by overriding `AbstractConfigurableBundleCreatorTests` `getXXX` methods.
2. declaratively by creating a properties file having a similar name with the test case. For example, test `com.xyz.MyTest` will have the properties file named `com/xyz/MyTest-bundle.properties`. If found, the following properties will be read from the file:

Table 7.1. Default test jar content settings

Property Name	Default Value	Description
root.dir	file:./target/test-classes	the root folder considered as the jar root
include.patterns	/**/*.class, /**/*.xml, /**/*.properties	Comma-separated string of Ant-style patterns
manifest	(empty)	manifest location given as a String. By default it's empty meaning the manifest will be created by the test framework rather then being supplied by the user.

This option is handy when creating specific tests that need to include certain resources (such as localization

files or images).

Please consult `AbstractConfigurableBundleCreatorTests` and `AbstractOnTheFlyBundleCreatorTests` tests for more customization hooks.

7.2.3.3. Understanding the `MANIFEST.MF` generation

A useful feature of the testing framework represents the automatic creation of the test manifest based on the test bundle content. The manifest creator component uses byte-code analysis to determine the packages imported by the test classes so that it can generate the proper OSGi directives for them. Since the generated bundle is used for running a test, the creator will use the following assumptions:

- No packages will be exported.

  The *on-the-fly* bundle is used for running a test which (usually) consumes OSGi packages for its execution. This behaviour can be changed by customizing the manifest.

- Split packages (i.e. classes from the same package can come from different bundles) are not supported.

  This means that packages present in the test framework are considered complete and no `Import-Package` entry will be generated for them. To avoid this problem, consider using sub-packages or moving the classes inside one bundle. Note that split packages are discouraged due to the issues associated with them (see the OSGi Core spec, Chapter 3.13 - Required Bundles).

- The test bundle contains only test classes.

  The byte-code parser will look only at the test classes hierarchy. Any other class included in the bundle, will not be considered so no imports will be generated for it. Consider customizing the manifest yourself or attaching the extra code as inner classes to the test class. A future version of Spring-DM might alleviate this problem by inspecting all classes included in the test bundle.

The reason behind *the lack of such features* is the byte-code parser is aimed to be simple and fast at creating test manifests - it is not meant as a general-purpose tool for creating OSGi artifacts.

7.2.4. Creating an OSGi application context

Spring-DM testing suite builds on top of Spring testing classes. To create an application context (OSGi specific), one should just override `getConfigLocations[]` method and indicate the location of the application context configuration. At runtime, an OSGi application context will be created and cached for the lifetime of the test case.

```
protected String[] getConfigLocations() {
   return new String[] { "/com/xyz/abc/test/MyTestContext.xml" };
}
```

7.2.5. Specifying the OSGi platform to use

The testing framework supports out of the box, three OSGi 4.0 implementations namely: Equinox, Knopflerfish and Felix. To be used, these should be in the test classpath. By default, the testing framework will try to use Equinox platform. This can be configured in several ways:

1. programmatically through `getPlatformName()` method

   .

   Override the aforementioned method and indicate the fully qualified name of the `Platform` interface implementation. Users can use the `Platforms` class to specify one of the supported platforms:

```
protected String getPlatformName() {
   return Platforms.FELIX;
}
```

2. declaratively through `org.springframework.osgi.test.framework` system property.

   If this property is set, the testing framework will use its value as a fully qualified name of a Platform implementation. It that fails, it will fall back to Equinox after logging a warning message. This option is useful for building tools (such as ant or maven) since it indicates a certain target environment without changing and test code.

7.2.6. Waiting for the test dependencies

A built-in feature of the testing framework is the ability to wait until all dependencies are deployed before starting the test execution. Since the OSGi platforms are concurrent by nature, installing a bundle doesn't mean that all its services are running. By running a test before its dependency services are fully initialized can cause sporadic errors that pollute the test results. By default, the testing framework inspects all bundles installed by the user and, if they are Spring-powered bundles, waits until they are fully started (that is their application context is published as an OSGi service). This behaviour can be disabled by overriding `shouldWaitForSpringBundlesContextCreation` method. Consult `AbstractSynchronizedOsgiTests` for more details.

7.2.7. Testing framework performance

Considering all the functionality offered by the testing framework, one might wonder if this doesn't become a performance bottleneck. First, it's worth noting that all the work done automatically by the testing infrastructure has to be done anyway (such as creating the manifest or creating a bundle for the test or installing the bundles). Doing it manually simply simply does not work as it's too error prone and time consuming. In fact, the current infrastructure started as way to do efficient, automatic testing without worrying about deployment problems and redundancy.

As for the numbers, the current infrastructure has been used internally for the last half a year - our integration tests (around 120) run in about 3:30 on a laptop. Most of this time is spent on starting and stopping the OSGi platform: the "testing framework" takes around 10% (as shown in our profiling so far). For example, the manifest generation has proved to take less then 0.5 seconds in general, while the jar creation around 1 second.

However, we are working on making it even faster and smarter so that less configuration options are needed and the contextual information available in your tests is used as much as possible. If you have any ideas or suggestion, feel free to use our issue tracker or/and mailing list.

Hopefully this chapter showed how Spring-DM testing infrastructure can simplify OSGi integration testing and how it can be customized. Consider consulting the javadocs for more information.

Part III. Appendixes

Document structure

Various appendixes outside the reference documentation.

appendix-compendium describes the support provided for the OSGi Compendium Services, including the Configuration Admin service. While the support is included in the 1.0 distribution, it is not guaranteed (yet) to be maintained in a backward-compatible form in future point releases.

appendix-extensions describes extensions that are included in the 1.0 distribution, but are not guaranteed to be maintained in a backward-compatible form in future point releases. We anticipate these features moving into the core specification over time.

appendix-pde-integration describes how to integrate Spring-DM with Eclipse Plug-in Development Environment.

appendix-archetype describes the Spring-DM Maven 2 archetype usage.

appendix-roadmap describes some features that are included in the 1.0 distribution but are still considered early-access. The externals of these features may change in future releases. This appendix also discusses other planned features for which no implementation yet exists.

appendix-schema defines the schemas provided by Spring Dynamic Modules.

Appendix A. Compendium Services

The OSGi Service Platform Service Compendium specification defines a number of additional services that may be supported by OSGi implementations. Spring Dynamic Modules supports an additional "compendium" namespace that provides support for some of these services. By convention, the prefix `osgix` is used for this namespace.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/osgi"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:osgix="http://www.springframework.org/schema/osgi-compendium"
   xmlns:beans="http://www.springframework.org/schema/beans"
   xsi:schemaLocation="http://www.springframework.org/schema/osgi
        http://www.springframework.org/schema/osgi/spring-osgi.xsd
        http://www.springframework.org/schema/osgi-compendium
        http://www.springframework.org/schema/osgi-compendium/spring-osgi-compendium.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

   <!-- use the OSGi namespace elements directly -->
   <service id="simpleServiceOsgi" ref="simpleService"
       interface="org.xyz.MyService" />

   <!-- qualify compendium namespace elements -->
   <osgix:property-placeholder persistent-id="com.xyz.myapp"/>

</beans:beans>
```

At present this namespace provides support for the Configuration Admin service. Support for other compendium services may be added in future releases.

A.1. Configuration Admin

A.1.1. Property placeholder support

> **Future directions**
>
> Please note that in future releases, the property-placeholder element will be changed to be properly align with Spring Framework classes and namespace. That is, rather then provide a Spring-DM specific declaration, the *traditional* Spring Framework declaration can be used.

Spring Dynamic Modules provides support for sourcing bean property values from the OSGi Configuration Administration service. This support is enabled via the `property-placeholder` element. The property placeholder element provides for replacement of delimited string values (placeholders) in bean property expressions with values sourced from the configuration administration service. The required `persistent-id` attribute specifies the persistent identifier to be used as the key for the configuration dictionary. The default delimiter for placeholder strings is "${...}".

Given the declarations:

```
<osgix:property-placeholder persistent-id="com.xyz.myapp"/>

<bean id="someBean" class="AClass">
  <property name="timeout" value="${timeout}"/>
</bean>
```

Then the `timeout` property of `someBean` will be set using the value of the `timeout` entry in the configuration dictionary registered under the `com.xyz.myapp` persistent id.

The placeholder strings are evaluated at the time that the bean is instantiated. Changes to the properties made via Configuration Admin subsequent to the creation of the bean do *not* result in re-injection of property values. See the `managed-service` and `managed-service-reference` elements documented in appendix B if you require this level of integration. The `placeholder-prefix` and `placeholder-suffix` attributes can be used to change the delimiter strings used for placeholder values.

It is possible to specify a default set of property values to be used in the event that the configuration dictionary does not contain an entry for a given key. The `defaults-ref` attribute can be used to refer to a named bean of `Properties` or `Map` type. Instead of referring to an external bean, the `default-properties` nested element may be used to define an inline set of properties.

```
<osgix:property-placeholder persistent-id="com.xyz.myapp">
  <default-properties>
     <property name="productCategory" value="E792"/>
     <property name="businessUnit" value="811"/>
  </default-properties>
</osgix:property-placeholder>
```

The `persistent-id` attribute must refer to the persistent-id of an OSGi `ManagedService`, it is a configuration error to specify a factory persistent id referring to a `ManagedServiceFactory`.

A.1.2. Configuration Dictionaries

Support for directly accessing configuration objects and their associated dictionaries, and for instantiating beans directly from configuration objects is on the Spring Dynamic Modules road map. See appendix-roadmap for more information.

Appendix B. Extensions

This appendix describes extensions to the core functionality that are shipped with the 1.0 distribution, but are not guaranteed to have backwards compatibility across point releases. We anticipate these features migrating into the core specification in future releases.

B.1. Annotation-based injection

The `org.springframework.osgi.extensions.annotation` bundle that ships with Spring Dynamic Modules provides early access to annotation-based support for injecting references to OSGi services. JDK 1.5 or above is required to use this feature.

Bean class (setter) methods may be annotated with `org.springframework.osgi.extensions.annotation.ServiceReference`. By default the property type of the annotated property is used to look up a service with a matching service interface in the OSGi service registry and inject the result. For example, given the configuration:

```
<bean id="annotationDriven" class="MyAnnotationDrivenBeanClass">
```

and the class declaration:

```
public class MyAnnotationDrivenBeanClass {

  @ServiceReference
  public void setMessageService(MessageService aService) { ... }

}
```

then a service lookup for services implementing the `MessageService` interface will be performed, and the best match (using the same algorithm as documented for the `reference` element) will be injected.

The `ServiceReference` annotation class has a number of attributes that can be used to control the service lookup (for example, to specify a filter string) that mirror the options provided by the `reference` element. See the javadoc for more information.

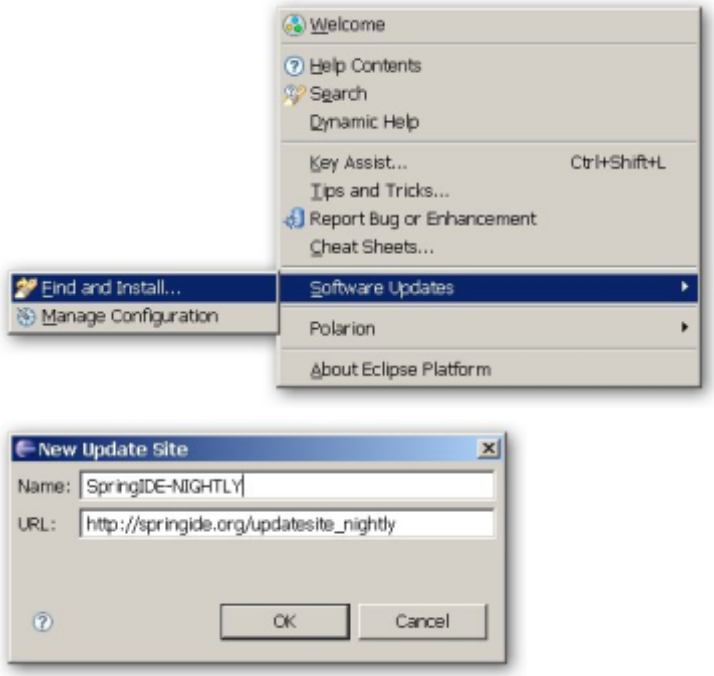
Appendix C. Eclipse Plug-in Development integration

Eclipse PDE "provides comprehensive OSGi tooling, which makes it an ideal environment for component programming, not just Eclipse plug-in development". In fact, Eclipse IDE is built on top of OSGi and uses at its core the Equinox OSGi implementation. Moreover, all the Eclipse plug-ins are OSGi bundles. This makes Eclipse with PDE a very attractive tool for creating OSGi bundles. While Spring Dynamic Modules artifacts can be integrated as *normal* libraries, through Spring IDE, Spring-DM can be installed as a target platform ready to be used with PDE.

The following steps illustrate how to install Spring IDE extension for OSGi and how to use it in your project. Please see Spring IDE installation page for information on its requirement and install process.

1. **Set up nightly update site**

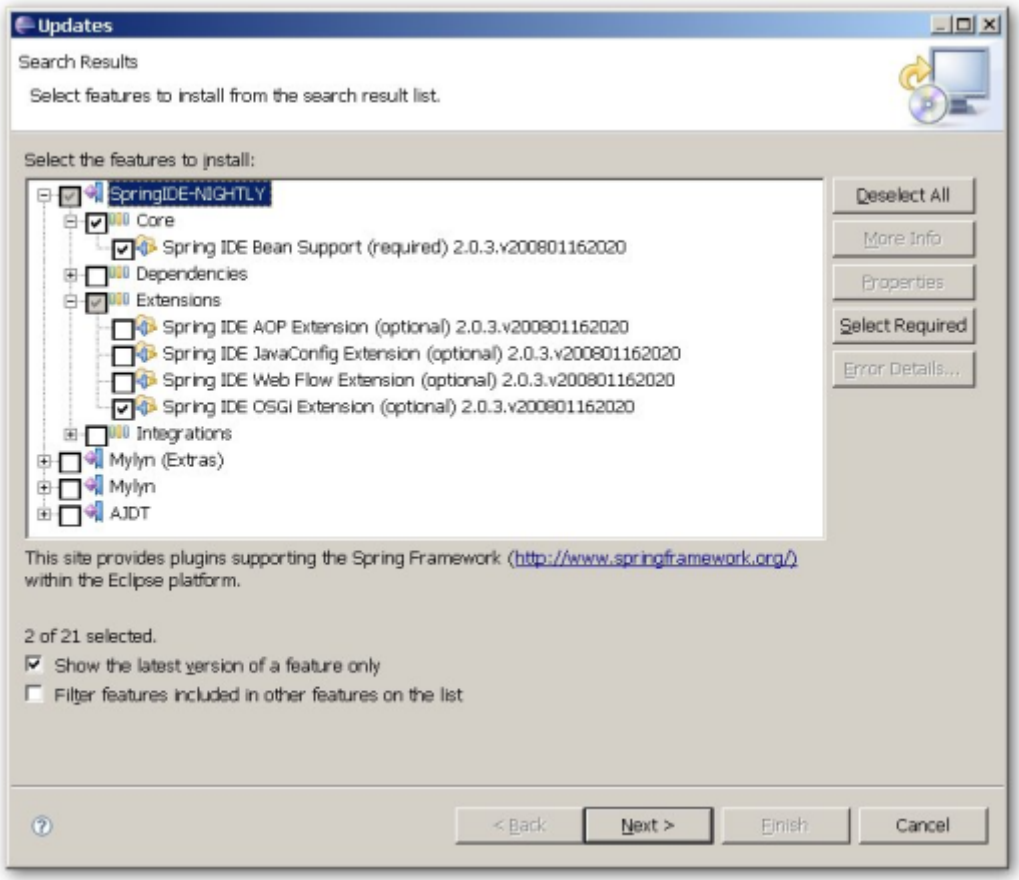
   At the moment, the OSGi extension is available only on Spring-IDE nightly builds update site. Add it to the Eclipse configuration by opening the software update menu:



   and create a new update site pointing to `http://www.springide.org/updatesite_nightly`

2. **Select Spring IDE OSGi extension**

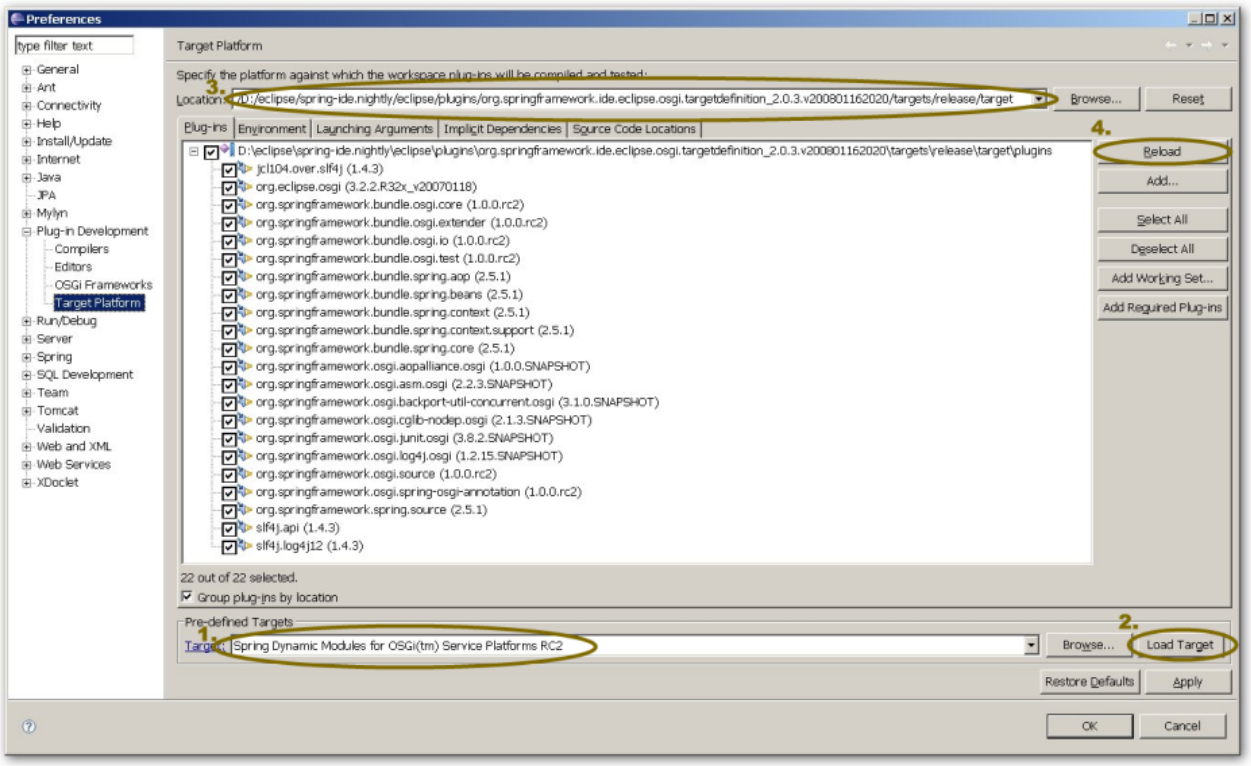
   After using the nightly update site and performing the update, Eclipse will show the search results. Unfold the *Extension* menu and select `Spring IDE OSGi Extension`:



and proceed with the instalation.

3. **Select Spring Dynamic Modules Target Platform**

Once the plug-in has been installed, Spring Dynamic Modules can be selected as a PDE target platform. Select Window/Preferences/Plug-in Development and then Target Platform.



Select the Spring-DM version that you desire from the Pre-defined Target (1) drop box and press Load Target (2). Eclipse will load the target and all bundles defined by it - this includes Spring-DM bundles and all of its dependencies (SLF4J is used for logging). The configuration can be customised appropriately by removing and adding bundles.

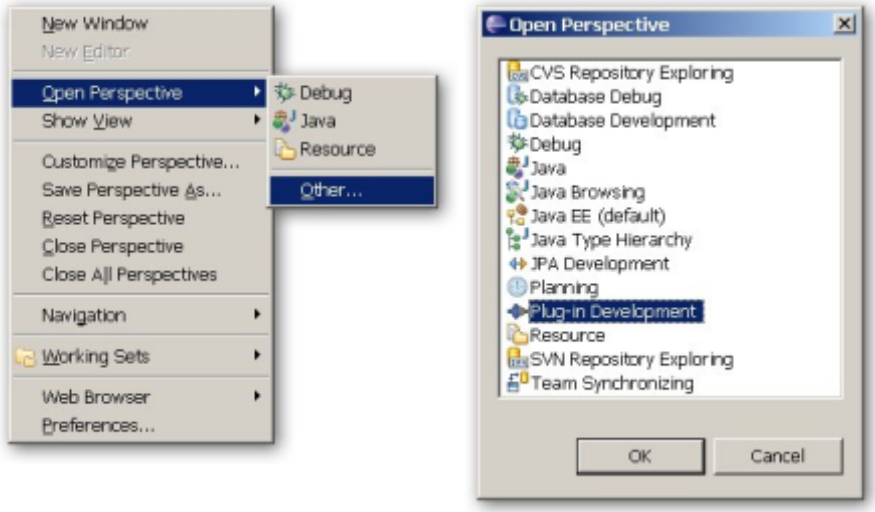
In its current form, the plug-in offers two predefined targets - one for the stable released versions and one for the SNAPSHOT/nightly Spring DM jars. The latter does not contain any jars as it is expected for the user to download them manually. Simply locate the path where the plug-ins should be located (3), enter that folder and do a

```
mvn install
```

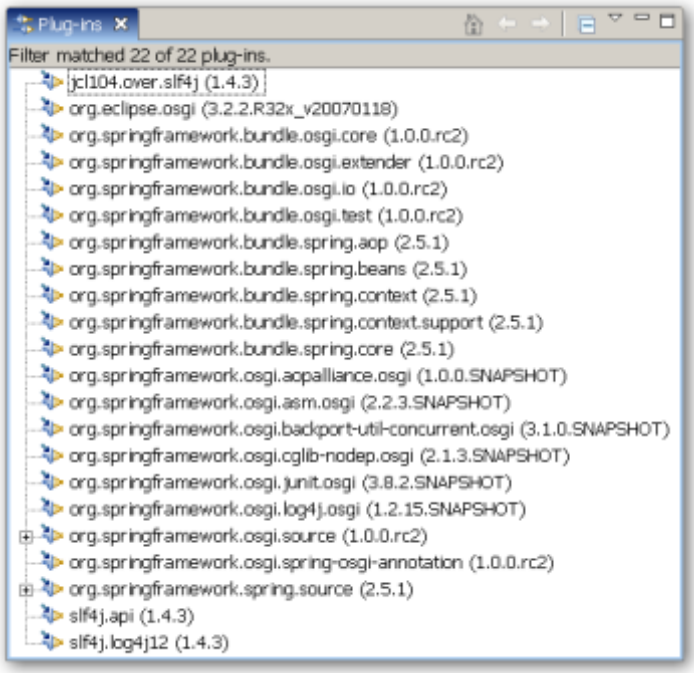
The latest Spring-DM SNAPSHOT will be downloaded along with all of its dependencies. Simply click on the reload button (4) and Eclipse will pick up the bundles.

4. **Select PDE perspective**

Once the installation is completed just select the PDE perspective:



and the Spring-DM and its dependencies should be available in the plug-ins view:


Plug-ins
Filter matched 22 of 22 plug-ins.
jcl104.over.slf4j (1.4.3)
org.eclipse.osgi (3.2.2.R32x_v20070118)
org.springframework.bundle.osgi.core (1.0.0.rc2)
org.springframework.bundle.osgi.extender (1.0.0.rc2)
org.springframework.bundle.osgi.io (1.0.0.rc2)
org.springframework.bundle.osgi.test (1.0.0.rc2)
org.springframework.bundle.spring.aop (2.5.1)
org.springframework.bundle.spring.beans (2.5.1)
org.springframework.bundle.spring.context (2.5.1)
org.springframework.bundle.spring.context.support (2.5.1)
org.springframework.bundle.spring.core (2.5.1)
org.springframework.osgi.aopalliance.osgi (1.0.0.SNAPSHOT)
org.springframework.osgi.asm.osgi (2.2.3.SNAPSHOT)
org.springframework.osgi.backport-util-concurrent.osgi (3.1.0.SNAPSHOT)
org.springframework.osgi.cglib-nodep.osgi (2.1.3.SNAPSHOT)
org.springframework.osgi.junit.osgi (3.8.2.SNAPSHOT)
org.springframework.osgi.log4j.osgi (1.2.15.SNAPSHOT)
org.springframework.osgi.source (1.0.0.rc2)
org.springframework.osgi.spring-osgi-annotation (1.0.0.rc2)
org.springframework.spring.source (2.5.1)
slf4j.api (1.4.3)
slf4j.log4j12 (1.4.3)

Appendix D. Spring Dynamic Modules Maven Archetype

As part of the distribution, Spring-DM provides a Maven archetype which creates the basic structure of a Java project that uses Spring DM, especially useful to new users. To run the archetype (and create the new project), simply run the following command line:

```
mvn archetype:create \
-DarchetypeGroupId=org.springframework.osgi \
-DarchetypeArtifactId=spring-osgi-bundle-archetype \
-DarchetypeVersion=${version}   \
-DgroupId=<your-project-groupId>  \
-DartifactId=<your-project-artifactId> \
-Dversion=<your-project-version>
```

> **Note**
>
> The command above should be invoked as one line - the `\` is used as a convenience to break the long line into smaller pieces

The result of the command, is a Maven 2 project that defines a simple bean, configures it using `src/main/resources/META-INF/spring/bundle-context.xml` and `src/main/resources/META-INF/spring/bundle-context-osgi.xml` and provides unit and (out of the OSGi container) integration tests. The project is packaged as an OSGi bundle.

Notice that by default, the project does not contain a MANIFEST.MF for your project. The Maven infrastructure will generate it, through Apache Felix bundle plug-in. To do that, run the following (from the project root):

```
mvn package
```

> **Note**
>
> To avoid the confusion between the generated artifacts and maintained files, the manifest file resides under `META-INF` folder while Spring configuration files under `src/main/resources/META-INF` directory.

This will compile your project, pack it as a jar and create the OSGi manifest based on your classes under `/META-INF` folder (so that users running Eclipse PDE can use it right away. To generate the manifest, without creating the OSGi bundle, simply run:

```
mvn org.apache.felix:maven-bundle-plugin:manifest
```

D.1. Generated Project Features at-a-glance

- Packaged as an OSGi bundle
- `META-INF/MANIFEST.MF` for OSGi bundle automatically generated

- Simple bean interface and implementation defined. Interface and implementation types are in different packages, only the interface package is exported by the bundle.
- `src/main/resources/META-INF/spring/bundle-context.xml` is a Spring configuration file that defines the simple bean.
- `src/main/resources/META-INF/spring/bundle-context-osgi.xml` is a spring configuration file ready for you to add bean definitions from the osgi namespace (services, references etc.)
- `BeanImplTest` case defined to unit test the simple bean
- `BeanIntegrationTest` defined to fire up the non-osgi portions of the application context configuration and test *outside* of OSGi
- `BeanOgiIntegrationTest` defined to fire up the osgi portions of the application context configuration and test *inside* of OSGi
- `.project`, `.classpath`, and `build.properties` files created to enable use of this project directly inside eclipse as a PDE plugin project

Appendix E. Roadmap

This appendix documents features on the Spring Dynamic Modules roadmap. The design of these features specified here is subject to change. As a most up to date source, please see our issue tracker.

E.1. Enhanced Configuration Admin Support

E.1.1. Managed Services

It should be possible to easily instantiate a bean from the configuration information stored for a `ManagedService` , and a set of beans from the configuration information stored for a `ManagedServiceFactory` . In addition, updates to the configuration information should be propagated to beans created in this way.

The `managed-service` element is used to define a bean based on the configuration information stored under a given persistent id. It has two mandatory attributes, `class` and `persistent-id` . The `persistent-id` attribute is used to specify the persistent id to be looked up in the configuration administration service, `class` indicates the Java class of the bean that will be instantiated.

A simple declaration of a managed service bean would look as follows:

```
<osgix:managed-service id="myService" class="com.xyz.MessageService"
   persistent-id="com.xyz.messageservice"/>
```

The properties of the `managed-service` bean are autowired by name based on the configuration found under the given persistent id. It is possible to declare regular Spring bean `property` elements within the `managed-service` declaration. If a property value is defined both in the configuration object stored in the Configuration Admin service, and in a nested `property` element, then the value from Configuration Admin takes precedence. Property values specified via `property` elements can therefore be treated as default values to be used if none is available through Configuration Admin.

It is possible for the configuration data stored in Configuration Admin to be updated once the bean has been created. By default, any updates post-creation will be ignored. To receive configuration updates, the `update-strategy` attribute can be used with a value of either `bean-managed` or `container-managed`.

The default value of the optional `update-strategy` attribute is `none`. If an update strategy of `bean-managed` is specified then the `update-method` attribute must also be used to specify the name of a method defined on the bean class that will be invoked if the configuration for the bean is updated. The update method must have one of the following signatures:

```
public void anyMethodName(Map properties)
public void anyMethodName(Map<String,?> properties); // for Java 5
public void anyMethodName(Dictionary properties);
```

When an update strategy of `container-managed` is specified then the container will autowire the bean instance by name based on the new properties received in the update. For `container-managed` updates, the bean class must provide setter methods for the bean properties that it wishes to have updated. Container-managed updates cannot be used in conjunction with constructor injection. Before proceeding to autowire based on the new property values, a lock is taken on the bean instance. This lock is released once autowiring has completed. A class may therefore synchronize its service methods or otherwise lock on the bean instance in order to have atomic update semantics.

E.1.2. Managed Service Factories

The `managed-service-factory` element is similar to the `managed-service` element, but instead defines a set of beans, one instance for each configuration stored under the given factory pid. It has two mandatory attributes, `factory-pid` and `class`.

A simple `managed-service-factory` declaration would look as follows:

```
<osgix:managed-service-factory id="someId" factory-pid="org.xzy.services"
           class="MyServiceClass"/>
```

This declaration results in the creation of zero or more beans, one bean for each configuration registered under the given factory pid. The beans will have synthetic names generated by appending "-" followed by the persistent id of the configuration object as returned by Configuration Admin, to the value of the `id` attribute used in the declaration of the `managed-service-factory`. For example, `someId-config.admin.generated.pid`.

Over time new configuration objects may be added under the factory pid. A new bean instance is automatically instantiated whenever a new configuration object is created. If a configuration object stored under the factory pid is deleted, then the corresponding bean instance will be disposed (this includes driving the `DisposableBean` callback if the bean implements `DisposableBean`). The option `destroy-method` attribute of the `managed-service-factory` element may be used to specify a destroy callback to be invoked on the bean instance. Such a method must have a signature:

```
public void anyMethodName();
```

It is also possible for the configuration of an existing bean to be updated. The same `update-strategy` and `update-method` attributes are available as for the `managed-service` element and with the same semantics (though obviously only the bean instance whose configuration has been updated in Configuration Admin will actually be updated). The same client-locking semantics also apply when using the `container-managed` update strategy

E.1.3. Direct access to configuration data

If you need to work directly with the configuration data stored under a given persistent id or factory persistent id, the easiest way to do this is to register a service that implements either the `ManagedService` or `ManagedServiceFactory` interface and specify the pid that you are interested in as a service property. For example:

```
<service interface="org.osgi.service.cm.ManagedService" ref="MyManagedService">
  <service-properties>
    <entry key="service.pid" value="my.managed.service.pid"/>
  </service-properties>
</service>

<bean id="myManagedService" class="com.xyz.MyManagedService"/>
```

and where the class `MyManagedService` implements `org.osgi.service.cm.ManagedService`.

E.1.4. Publishing configuration administration properties with exported services

We intend to provide support for automatic publication of service properties sourced from the Configuration

Admin service under a given persistent id. This support has yet to be designed but may look as follows:

```
<service ref="toBeExported" interface="SomeInterface">
   <osgix:config-properties persistent-id="pid"/>
</service>
```

Issues to be considered are scoping of a subset of properties to be published (public/private), and automatic updates to published service properties if the value is updated via config admin.

Note that named properties can easily be published as service properties already without this support, simply by using the `property-placeholder` support.

E.2. Access to Service References for Collections

The current specification does not provide for access to the `ServiceReference` objects for services in a managed collection (i.e. obtained via a `set` or `list` declaration). A future release of Spring Dynamic Modules will provide an easy means of gaining access to these references.

E.3. Start level integration

A future release of Spring Dynamic Modules may offer the following additional guarantee with respect to application context creation and start levels:

Application context creation happens asynchronously. However, the extender bundle does guarantee that the creation of all application contexts for bundles at start level `n` will be complete before the creation of any application context at start level `m`, where `m > n`. Care must therefore be taken not to introduce any mandatory dependencies on services exported by bundles with higher start levels or a deadlock will be introduced.

In a similar vein, when shutting down the extender bundle, application contexts at start level `m` will be shut down before application contexts at start level `n`, where `m > n`.

E.4. Web application support

A key part of the roadmap for the next release of Spring Dynamic Modules is built-in support for building web applications out of OSGi bundles. This will be supported using both an `HttpService` provided by a bundle installed into the running OSGi framework, and also by embedding an OSGi container inside an existing servlet container. Pioneering work has been done by Martin Lippert, Bernd Kolb, and Gerd Wutherich amongst others. Please see our mailing list archive for more information.

E.5. ORM/Persistence support

Care needs to be taken when using JPA or Hibernate under OSGi as the persistence engines must have visibility of the persistent types and mapping files. The Spring Dynamic Modules project will be investigating an extension model to make managing this easier when persistent configuration is split across several bundles. See Peter Krien's blog entry on the topic for an insight into the issues.

Also, the project aims to simplify deployment of JDBC drivers and pooling libraries that at the moment require special `DynamicImport-Package`.

E.6. OSGi standards

While OSGi 4.0 is currently required, work is underway to take advantage of the new features available in 4.1. SpringSource is an active participant in the OSGi Enterprise Expert Group and we hope to help many of the ideas found in the Spring Dynamic Modules project to make their way into the OSGi R5 specification. Spring Dynamic Modules would obviously seek to support any such standards at that point in time.

Appendix F. Spring Dynamic Modules Schema

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns="http://www.springframework.org/schema/osgi"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:beans="http://www.springframework.org/schema/beans"
            xmlns:tool="http://www.springframework.org/schema/tool"
            targetNamespace="http://www.springframework.org/schema/osgi"
            elementFormDefault="qualified"
            attributeFormDefault="unqualified"
            version="1.0">

    <xsd:import namespace="http://www.springframework.org/schema/beans"/>
    <xsd:import namespace="http://www.springframework.org/schema/tool"/>

    <xsd:annotation>
        <xsd:documentation><![CDATA[
            Namespace support for the core services provided by Spring Dynamic Modules.
        ]]></xsd:documentation>
    </xsd:annotation>

    <xsd:attributeGroup name="defaults">
            <xsd:annotation>
                    <xsd:documentation><![CDATA[Defaults for Spring-DM OSGi declarations.]]>
                    </xsd:documentation>
            </xsd:annotation>
            <!-- attributes -->
            <xsd:attribute name="default-timeout" type="xsd:long" default="30000">
                    <xsd:annotation>
                            <xsd:documentation><![CDATA[
                            Default timeout (in milliseconds) for all reference (service importers) elements tha
                            Default value is 300000 ms (5 minutes).
                            ]]></xsd:documentation>
                    </xsd:annotation>
            </xsd:attribute>
            <xsd:attribute name="default-cardinality" type="TdefaultCardinalityOptions" default="1..X">
                    <xsd:annotation>
                            <xsd:documentation><![CDATA[
                    Default cardinality (of the relationship to the backing service(s)) for all OSGi references
                    elements that do not explicitly specify one.
                    Default value is '1..X' (resolved to '1..1' for osgi:reference and '1..N' for osgi:list/set)
                    service must exist (this is a mandatory service reference). A value of '0..X' (resolved to '
                    and '0..N' for osgi:list/set) indicates that it is acceptable to be no backing service (an o
                        ]]></xsd:documentation>
                </xsd:annotation>
        </xsd:attribute>
    </xsd:attributeGroup>

    <xsd:simpleType name="TdefaultCardinalityOptions">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="1..X">
                    <xsd:annotation>
                            <xsd:documentation><![CDATA[
                            A backing service must exist (this is a mandatory service reference).
                            ]]></xsd:documentation>
                    </xsd:annotation>
            </xsd:enumeration>
            <xsd:enumeration value="0..X">
                    <xsd:annotation>
                            <xsd:documentation><![CDATA[
                            A backing service can be missing (this is an optional service reference).
                            ]]></xsd:documentation>
                    </xsd:annotation>
            </xsd:enumeration>
        </xsd:restriction>
    </xsd:simpleType>

    <!-- reference -->
    <xsd:element name="reference" type="TsingleReference">
        <xsd:annotation>
            <xsd:documentation><![CDATA[
            Defines a reference to a service obtained via the OSGi service registry.
```

```
            ]]></xsd:documentation>
        </xsd:annotation>
    </xsd:element>

    <xsd:complexType name="Treference">
        <xsd:complexContent>
            <xsd:extension base="beans:identifiedType">
                <xsd:sequence minOccurs="0" maxOccurs="unbounded">
                        <xsd:element name="interfaces" type="beans:listOrSetType" minOccurs="0" maxOccurs="1">
                            <xsd:annotation>
                                    <xsd:documentation><![CDATA[
                                    The set of service interfaces to advertise in the service registry.
                                    ]]></xsd:documentation>
                            </xsd:annotation>
                    </xsd:element>
                    <xsd:element name="listener" type="Tlistener" minOccurs="0" maxOccurs="unbounded">
                      <xsd:annotation>
                              <xsd:documentation><![CDATA[
                                Defines a listener that will receive notification when a service backing this re
                                bound or unbound.
                              ]]></xsd:documentation>
                      </xsd:annotation>
                    </xsd:element>
                </xsd:sequence>
                <xsd:attribute name="interface" use="optional" type="xsd:token">
                        <xsd:annotation>
                                <xsd:documentation><![CDATA[
                                The service interface that the services obtained via the registry are required t
                                By convention this is a Java interface type, but may also be a (non-final) class
                                ]]></xsd:documentation>
                                <xsd:appinfo>
                                        <tool:annotation>
                                                <tool:expected-type type="java.lang.Class" />
                                        </tool:annotation>
                                </xsd:appinfo>
                        </xsd:annotation>
                </xsd:attribute>
                <xsd:attribute name="filter" use="optional" type="xsd:string">
                      <xsd:annotation>
                              <xsd:documentation><![CDATA[
                                Defines an OSGi filter expression that is used to constrain the set of matching
                                in the service registry.
                              ]]></xsd:documentation>
                      </xsd:annotation>
                </xsd:attribute>
                <xsd:attribute name="depends-on" type="xsd:string" use="optional">
                                        <xsd:annotation>
                                                <xsd:documentation><![CDATA[
                                                        Used to refer to the name of another bean that this bean
                                                        service registry look-up does not happen until after the
                                                        (most commonly used to refer to a bundle bean).
                                                ]]></xsd:documentation>
                                        </xsd:annotation>
                </xsd:attribute>
                <xsd:attribute name="bean-name" type="xsd:string" use="optional">
                        <xsd:annotation>
                                <xsd:documentation><![CDATA[
                                        Convenient shortcut for specifying a filter expression that matches on t
                                        that is automatically advertised for beans published using the service e
                                ]]></xsd:documentation>
                        </xsd:annotation>
                </xsd:attribute>
                <xsd:attribute name="context-class-loader" type="TreferenceClassLoaderOptions" default="client">
                        <xsd:annotation>
                                <xsd:documentation><![CDATA[
                                        Defines how the context class loader is managed when invoking operations
                                        backing this service reference. The default value is 'client' which mean
                                        class loader has visibility of the resources on this bundle's classpath.
                                        options are 'service-provider' which means that the context class loader
                                        resources on the bundle classpath of the bundle that exported the servic
                                        which does not do any management of the context class loader.
                                ]]></xsd:documentation>
                        </xsd:annotation>
                </xsd:attribute>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
```

```
    <xsd:simpleType name="TreferenceClassLoaderOptions">
        <xsd:restriction base="xsd:NMTOKEN">
            <xsd:enumeration value="client"/>
            <xsd:enumeration value="service-provider"/>
            <xsd:enumeration value="unmanaged"/>
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:complexType name="Tlistener">
            <xsd:annotation>
                    <xsd:documentation><![CDATA[
                            Defines a listener that will be notified when the service backing the enclosing serv
                            unbound. Use either the 'ref' attribute or a nested bean declaration for the listene
                    ]]></xsd:documentation>
            </xsd:annotation>
        <xsd:sequence minOccurs="0" maxOccurs="1">
            <!-- nested bean declaration -->
            <xsd:any namespace="##other" minOccurs="1" maxOccurs="1" processContents="skip"/>
        </xsd:sequence>

        <!-- shortcut for bean references -->
        <xsd:attribute name="ref" type="xsd:string" use="optional">
                <xsd:annotation>
                        <xsd:documentation><![CDATA[
                                Refers by name to the bean that will receive bind and unbind events
                        ]]></xsd:documentation>
                </xsd:annotation>
        </xsd:attribute>
        <xsd:attribute name="bind-method" type="xsd:token" use="optional">
                <xsd:annotation>
                        <xsd:documentation><![CDATA[
                                The name of the method to be invoked when a backing service is bound.
                        ]]></xsd:documentation>
                </xsd:annotation>
        </xsd:attribute>
        <xsd:attribute name="unbind-method" type="xsd:token" use="optional">
                <xsd:annotation>
                        <xsd:documentation><![CDATA[
                                The name of the method to be invoked when a backing service is unbound.
                        ]]></xsd:documentation>
                </xsd:annotation>
        </xsd:attribute>
    </xsd:complexType>

    <!-- single reference -->
    <xsd:complexType name="TsingleReference">
        <xsd:complexContent>
            <xsd:extension base="Treference">
                 <xsd:attribute name="cardinality" use="optional" type="TsingleReferenceCardinality">
                        <xsd:annotation>
                              <xsd:documentation><![CDATA[
                                Defines the required cardinality of the relationship to the backing service. If
                                the default-cardinality attribute will apply. A value is '1..1' means that a bac
                                must exist (this is a mandatory service reference). A value of '0..1' indicates
                                acceptable to be no backing service (an optional service reference).
                              ]]></xsd:documentation>
                        </xsd:annotation>
                </xsd:attribute>
                              <xsd:attribute name="timeout" use="optional" type="xsd:long">
                    <xsd:annotation>
                              <xsd:documentation><![CDATA[
                                For a 'reference' element, the amount of time (in milliseconds) to wait for a ba
                                available when an operation is invoked. If not specified, the default-timeout at
                                See also the default-timeout attribute of the osgi element.
                              ]]></xsd:documentation>
                        </xsd:annotation>
                </xsd:attribute>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:simpleType name="TsingleReferenceCardinality">
        <xsd:restriction base="xsd:token">
            <xsd:enumeration value="1..1"/>
            <xsd:enumeration value="0..1"/>
        </xsd:restriction>
    </xsd:simpleType>
```

```
	<!-- reference collections (set, list) -->
	<xsd:element name="list" type="TreferenceCollection">
		<xsd:annotation>
			<xsd:documentation source="java:org.springframework.osgi.service.importer.support.OsgiSe
			  Defines a bean of type 'List' that contains all of the services matching the given cri
			  The list members are managed dynamically as matching backing services come and go.
			]]></xsd:documentation>
			<xsd:appinfo>
				<tool:annotation>
					<tool:exports type="java.util.List"/>
				</tool:annotation>
			</xsd:appinfo>
		</xsd:annotation>
	</xsd:element>

	<xsd:element name="set" type="TreferenceCollection">
		<xsd:annotation>
			<xsd:documentation source="java:org.springframework.osgi.service.importer.support.OsgiSe
			  Defines a bean of type 'Set' that contains all of the services matching the given crit
			  The set members are managed dynamically as matching backing services come and go.
			]]></xsd:documentation>
			<xsd:appinfo>
				<tool:annotation>
					<tool:exports type="java.util.Set"/>
				</tool:annotation>
			</xsd:appinfo>
		</xsd:annotation>
	</xsd:element>

	<xsd:complexType name="TreferenceCollection">
		<xsd:complexContent>
			<xsd:extension base="Treference">
				<xsd:sequence minOccurs="0" maxOccurs="1">
					<xsd:element name="comparator" type="Tcomparator">
						<xsd:annotation>
							<xsd:documentation source="java:java.util.Comparator"><!
							  Used to define an inline bean of type Comparator that
							]]></xsd:documentation>
							<xsd:appinfo>
								<tool:annotation>
									<tool:expected-type type="java.util.Comp
								  </tool:annotation>
							</xsd:appinfo>
						</xsd:annotation>
					</xsd:element>
				</xsd:sequence>
				<xsd:attribute name="comparator-ref" type="xsd:string" use="optional">
					<xsd:annotation>
						<xsd:documentation><![CDATA[
							Used to refer to a named bean implementing the Comparato
							sort the matching services.
						]]></xsd:documentation>
					</xsd:annotation>
				</xsd:attribute>

			    <xsd:attribute name="cardinality" use="optional" type="TcollectionCardinality">
				<xsd:annotation>
				  <xsd:documentation><![CDATA[
				    Defines the required cardinality of the relationship to the backing services. If
				    the default-cardinality attribute will apply. A value of '1..N' means that at le
				    service must exist (this is a mandatory service reference. A value of '0..N' ind
				    is acceptable for there to be no backing service (an optional service reference)
				  ]]></xsd:documentation>
			      </xsd:annotation>
		</xsd:attribute>
				</xsd:extension>
		</xsd:complexContent>
	</xsd:complexType>

	<xsd:complexType name="Tcomparator">
		<xsd:annotation>
			<xsd:documentation source="java:java.util.Comparator"><![CDATA[
			  Used to define an inline bean of type Comparator that will be used to sort the matchin
			]]></xsd:documentation>
			<xsd:appinfo>
				<tool:annotation>
					<tool:exports type="java.util.Comparator" />
				  </tool:annotation>
```

```
                    </xsd:appinfo>
                </xsd:annotation>
                <xsd:choice>
                   <xsd:element name="natural" type="TnaturalOrdering"/>
                <xsd:sequence minOccurs="1" maxOccurs="1">
                    <!-- nested bean declaration -->
                        <xsd:any namespace="##other" minOccurs="1" maxOccurs="1" processContents="skip"/>
                </xsd:sequence>
        </xsd:choice>
        </xsd:complexType>

        <xsd:complexType name="TnaturalOrdering">
                <xsd:attribute name="basis" type="TorderingBasis" use="required"/>
        </xsd:complexType>

        <xsd:simpleType name="TorderingBasis">
                <xsd:restriction base="xsd:token">
            <xsd:enumeration value="service"/>
            <xsd:enumeration value="service-reference"/>
        </xsd:restriction>
        </xsd:simpleType>

        <xsd:simpleType name="TcollectionCardinality">
        <xsd:restriction base="xsd:token">
            <xsd:enumeration value="1..N"/>
            <xsd:enumeration value="0..N"/>
        </xsd:restriction>
    </xsd:simpleType>

     <!-- service -->

     <xsd:element name="service" type="Tservice"/>

     <xsd:complexType name="Tservice">
            <xsd:annotation>
                    <xsd:documentation source="java:org.springframework.osgi.service.exporter.support.OsgiServic
                            Exports the reference bean as a service in the OSGi service registry. The bean defin
                            type org.osgi.framework.ServiceRegistration.
                    ]]></xsd:documentation>
                        <xsd:appinfo>
                                <tool:annotation>
                                        <tool:exports type="org.osgi.framework.ServiceRegistration"/>
                                </tool:annotation>
                        </xsd:appinfo>
            </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="beans:identifiedType">
                <xsd:sequence minOccurs="0" maxOccurs="1">
                    <xsd:element name="interfaces" type="beans:listOrSetType" minOccurs="0">
                            <xsd:annotation>
                                    <xsd:documentation><![CDATA[
                                            The set of service interfaces to advertise in the service registry.
                                    ]]></xsd:documentation>
                            </xsd:annotation>
                    </xsd:element>
                    <xsd:element name="service-properties" minOccurs="0" type="beans:mapType">
                               <xsd:annotation>
                                    <xsd:documentation><![CDATA[
                                            Defines the service properties.
                                    ]]></xsd:documentation>
                            </xsd:annotation>
                    </xsd:element>
                    <xsd:element name="registration-listener" type="TserviceRegistrationListener" minOccurs="0"
                            <xsd:annotation>
                                    <xsd:documentation><![CDATA[
                                            Defines a listener that will be notified when this service is regist
                                            OSGi service registry.
                                    ]]></xsd:documentation>
                            </xsd:annotation>
                    </xsd:element>

                    <!-- nested bean declaration -->
                    <xsd:any namespace="##other" minOccurs="0" maxOccurs="1" processContents="skip"/>
                </xsd:sequence>
                <xsd:attribute name="interface" type="xsd:token" use="optional">
                        <xsd:annotation>
                                <xsd:documentation><![CDATA[
                                        Defines the interface to advertise for this service in the service regis
```

```
                        ]]></xsd:documentation>
                                <xsd:appinfo>
                                        <tool:annotation>
                                                <tool:expected-type type="java.lang.Class" />
                                          </tool:annotation>
                                </xsd:appinfo>
                        </xsd:annotation>
                </xsd:attribute>
                <xsd:attribute name="ref" type="xsd:string" use="optional">
                        <xsd:annotation>
                                <xsd:documentation><![CDATA[
                                        Refers to the named bean to be exported as a service in the service regi
                                ]]></xsd:documentation>
                        </xsd:annotation>
                </xsd:attribute>
                <xsd:attribute name="depends-on" type="xsd:string" use="optional">
                        <xsd:annotation>
                                <xsd:documentation><![CDATA[
                                        Used to ensure that the service is not exported to the registry before t
                                        has been created.
                                ]]></xsd:documentation>
                        </xsd:annotation>
                </xsd:attribute>
                <xsd:attribute name="context-class-loader" type="TserviceClassLoaderOptions" default="unmanaged"
                        <xsd:annotation>
                                <xsd:documentation><![CDATA[
                                        Defines how the context class loader will be managed when an operation i
                                        exported service. The default value is 'unmanaged' which means that no m
                                        the context class loader is attempted. A value of 'service-provider' gua
                                        the context class loader will have visibility of all the resources on th
                                        bundle exporting the service.
                                ]]></xsd:documentation>
                        </xsd:annotation>
                </xsd:attribute>
                <xsd:attribute name="auto-export" type="TautoExportModes" default="disabled">
                        <xsd:annotation>
                                <xsd:documentation><![CDATA[
                                        Enables Spring to automatically manage the set of service interfaces adv
                                        service. By default this facility is disabled. A value of 'interfaces' a
                                        of the Java interfaces supported by the exported service. A value of 'cl
                                        advertises all the Java classes in the hierarchy of the exported service
                                        'all-classes' advertises all Java interfaces and classes.
                                ]]></xsd:documentation>
                        </xsd:annotation>
                </xsd:attribute>
                <xsd:attribute name="ranking" type="xsd:int" default="0">
                        <xsd:annotation>
                                <xsd:documentation><![CDATA[
                                        Specify the service ranking to be used when advertising the service.
                                ]]></xsd:documentation>
                        </xsd:annotation>
                </xsd:attribute>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="TserviceRegistrationListener">
        <xsd:annotation>
                <xsd:documentation><![CDATA[
                        Defines a listener that will be notified when the bean is registered or unregistered
                        Use either the 'ref' attribute or a nested bean declaration for the listener bean.
                ]]></xsd:documentation>
        </xsd:annotation>
    <xsd:sequence minOccurs="0" maxOccurs="1">
        <!-- nested bean declaration -->
        <xsd:any namespace="##other" minOccurs="1" maxOccurs="1" processContents="skip"/>
    </xsd:sequence>

    <!-- shortcut for bean references -->
    <xsd:attribute name="ref" type="xsd:string" use="optional">
            <xsd:annotation>
                    <xsd:documentation><![CDATA[
                            Refers by name to the bean that will receive register and unregister events.
                    ]]></xsd:documentation>
            </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="registration-method" type="xsd:token" use="optional">
            <xsd:annotation>
```

```
                    <xsd:documentation><![CDATA[
                            The name of the method to be invoked when the service is registered.
                    ]]></xsd:documentation>
            </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="unregistration-method" type="xsd:token" use="optional">
            <xsd:annotation>
                    <xsd:documentation><![CDATA[
                            The name of the method to be invoked when the service is unregistered.
                    ]]></xsd:documentation>
            </xsd:annotation>
    </xsd:attribute>
</xsd:complexType>

<xsd:simpleType name="TserviceClassLoaderOptions">
    <xsd:restriction base="xsd:token">
        <xsd:enumeration value="service-provider"/>
        <xsd:enumeration value="unmanaged"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="TautoExportModes">
    <xsd:restriction base="xsd:token">
        <xsd:enumeration value="disabled"/>
        <xsd:enumeration value="interfaces"/>
        <xsd:enumeration value="class-hierarchy"/>
        <xsd:enumeration value="all-classes"/>
    </xsd:restriction>
</xsd:simpleType>

<!-- bundle -->

<xsd:element name="bundle" type="Tbundle">
        <xsd:annotation>
                <xsd:documentation source="java:org.springframework.osgi.bundle.BundleFactoryBean"><![CDATA[
                        Defines a bean representing a Bundle object. May be used to drive bean lifecycle tra
                ]]></xsd:documentation>
                    <xsd:appinfo>
                            <tool:annotation>
                                    <tool:exports type="org.osgi.framework.Bundle"/>
                            </tool:annotation>
                    </xsd:appinfo>
        </xsd:annotation>
</xsd:element>

<xsd:complexType name="Tbundle">
    <xsd:complexContent>
        <xsd:extension base="beans:identifiedType">
                  <!-- optional nested bean declaration -->
            <xsd:sequence minOccurs="0" maxOccurs="1">
                                <xsd:any namespace="##other" minOccurs="1" maxOccurs="1" processContents="la
                                   <xsd:annotation>
                                <xsd:documentation><![CDATA[
                                        OSGi bundle to work with.
                                ]]></xsd:documentation>
                                                        <xsd:appinfo>
                                                                <tool:annotation>
                                                                        <tool:expected-type type="org.osgi.fra
                                                                  </tool:annotation>
                                                        </xsd:appinfo>
                        </xsd:annotation>
                        </xsd:any>
                    </xsd:sequence>

            <xsd:attribute name="symbolic-name" type="xsd:string" use="optional">
                    <xsd:annotation>
                            <xsd:documentation><![CDATA[
                                    The bundle symbolic name of the bundle object. Normally used when intera
                                    installed bundle.
                            ]]></xsd:documentation>
                    </xsd:annotation>
            </xsd:attribute>
            <xsd:attribute name="depends-on" type="xsd:string" use="optional">
                    <xsd:annotation>
                            <xsd:documentation><![CDATA[
                                    Indicates that this bundle object should not be created until the named
                            ]]></xsd:documentation>
                    </xsd:annotation>
```

```
                </xsd:attribute>

                <xsd:attribute name="location" type="xsd:string" use="optional">
                        <xsd:annotation>
                                <xsd:documentation><![CDATA[
                                        Location used to install, update or/and identify a bundle.
                                ]]></xsd:documentation>
                        </xsd:annotation>
                </xsd:attribute>
                <xsd:attribute name="action" type="TbundleAction" use="optional">
                        <xsd:annotation>
                                <xsd:documentation><![CDATA[
                                Lifecyle action to drive on the bundle. 'start' starts the bundle, installing if
                                'stop' stops the bundle if it is currently ACTIVE. 'install' installs the bundle
                                currently uninstalled. 'uninstall' stops the bundle if needed, and then uninstal
                                'update' installs the bundle if needed, and then invokes the Bundle.update() ope
                                ]]></xsd:documentation>
                        </xsd:annotation>
                </xsd:attribute>
                <xsd:attribute name="destroy-action" type="TbundleAction" use="optional">
                        <xsd:annotation>
                                <xsd:documentation><![CDATA[
                                Lifecyle action to drive on the bundle. 'start' starts the bundle, installing if
                                'stop' stops the bundle if it is currently ACTIVE. 'install' installs the bundle
                                currently uninstalled. 'uninstall' stops the bundle if needed, and then uninstal
                                'update' installs the bundle if needed, and then invokes the Bundle.update() ope
                                ]]></xsd:documentation>
                        </xsd:annotation>
                </xsd:attribute>
                <xsd:attribute name="start-level" type="xsd:int" use="optional" default="0">
                        <xsd:annotation>
                                <xsd:documentation><![CDATA[
                                        Start level to set for the bundle.
                                ]]></xsd:documentation>
                        </xsd:annotation>
                </xsd:attribute>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:simpleType name="TbundleAction">
        <xsd:restriction base="xsd:token">
            <xsd:enumeration value="start"/>
            <xsd:enumeration value="stop"/>
            <xsd:enumeration value="install"/>
            <xsd:enumeration value="uninstall"/>
            <xsd:enumeration value="update"/>
        </xsd:restriction>
    </xsd:simpleType>


</xsd:schema>
```

Appendix G. Acknowledgments

Spring Dynamic Modules would like to thank (in alphabetical order) to : Bill Gallagher, Olivier Gruber, Richard S. Hall, BJ Hargrave, Peter Kriens, Martin Lippert, Jeff McAffer, Glyn Normington, Gerd Wuetherich for their contributions in the development of this specification.

Part IV. Other Resources

In addition to this reference documentation, there exist a number of other resources that may help you learn how to use OSGi and Spring Dynamic Modules. These additional, third-party resources are enumerated in this section.

Chapter 8. Useful links

- *Getting Started with OSGi* - by Neil Bartlett here and here.
- *Equinox Documents* - here
- *Felix-related presentations* - various presentations hosted by Apache Felix project.
- *Launching Spring Dynamic Modules using pax-runner* - screencast
- *OSGi Alliance Blog* - here
- *SpringSource OSGi blog* - here