

Virgo Web Server Programmer Guide

Ramnivas Laddad

Colin Yates

Sam Brannen

Rob Harrop

Christian Dupuis

Andy Wilkinson



2.1.0.M01

Table of Contents

Preface	v
1. Prerequisites	1
1.1. Runtime Environment	1
1.2. References	1
2. Introduction to VWS	3
2.1. Overview	3
2.2. What is the Virgo Web Server?	3
2.3. Why the Virgo Web Server?	5
3. Deployment Architecture	7
3.1. Supported Deployment Formats	7
3.2. Dependency Types	12
3.3. A guide to forming bundles	13
4. Developing Applications	17
4.1. Anatomy of a bundle	17
4.2. Creating PARs and WARs	18
4.3. Creating Plans	20
4.4. Creating and Using Configuration Artifacts	25
4.5. Programmatic Access to Personality-specific Features	27
4.6. Automatic Imports	28
4.7. Working with dependencies	29
4.8. Application trace	33
4.9. Application versioning	33
5. Migrating to OSGi	35
5.1. Migrating Web Applications	35
5.2. Migrating to a Plan or a PAR	36
6. Migrating Form Tags	39
6.1. Overview of the Form Tags Sample Application	41
6.2. Form Tags WAR	41
6.3. Form Tags Shared Libraries WAR	43
6.4. Form Tags Shared Services WAR	45
6.5. Form Tags PAR	52
6.6. Summary of the Form Tags Migration	54
6.7. Form Tags as a plan	55
7. Tooling	57
7.1. Installation	57
7.2. Running a Virgo Web Server instance within Eclipse	57
7.3. Bundle and Library Provisioning	59
7.4. Setting up Eclipse Projects	60
7.5. Developing OSGi Bundles	61
7.6. Deploying Applications	65
8. Common Libraries	67
8.1. Working with Hibernate	67
8.2. Working with DataSources	67
8.3. Weaving and Instrumentation	67

8.4. JSP Tag Libraries	68
9. Known Issues	69
9.1. JPA Entity Scanning	69
9.2. ClassNotFoundException When Creating a Proxy	69
9.3. Creating proxies with CGLIB for package-protected types	69
9.4. Tomcat Restrictions	69

Preface

Increasing complexity in modern enterprise applications is a fact of life. You not only have to deal with complex business logic, but also a myriad of other concerns such as security, auditing, exposing business functionality to external applications, and managing the evolution of that functionality and technologies. The Spring Framework and Spring Portfolio products address these needs by offering a Plain-Old Java Object (POJO) based solution that lets you focus on your business logic.

Complex applications pose problems that go beyond using the right set of technologies. You need to take into account other considerations such as a simplified development process, easy deployment, monitoring deployed applications, and managing changes in response to changing business needs. This is where the Virgo Runtime Environment comes into play. It offers a simple yet comprehensive platform to develop, deploy, and service enterprise applications. In this Programmer Guide, we explore the runtime portion of the Virgo Runtime Environment, the Virgo Web Server, and learn how to develop applications to benefit from its capabilities.

1. Prerequisites

1.1 Runtime Environment

The Virgo Web Server requires Java SE 6 or later to be installed. Java is available from [Sun](#) and elsewhere.

1.2 References

To make effective use of the Virgo Web Server, you should also refer to the following guides:

- [Virgo Web Server User Guide](#)
- [Spring Dynamic Modules Reference Guide](#)
- [Spring Framework Reference Guide](#)

2. Introduction to the Virgo Web Server

2.1 Overview

In this chapter, we provide an overview of the Virgo Web Server focusing on what it is, what benefits it provides to developers and administrators, and why you should use it.

2.2 What is the Virgo Web Server?

The Virgo Web Server, or VWS for short, is the runtime portion of the Virgo Runtime Environment. It is a lightweight, modular, OSGi-based runtime that provides a complete packaged solution for developing, deploying, and managing enterprise applications. By leveraging several best-of-breed technologies and improving upon them, the VWS offers a compelling solution to develop and deploy enterprise applications.

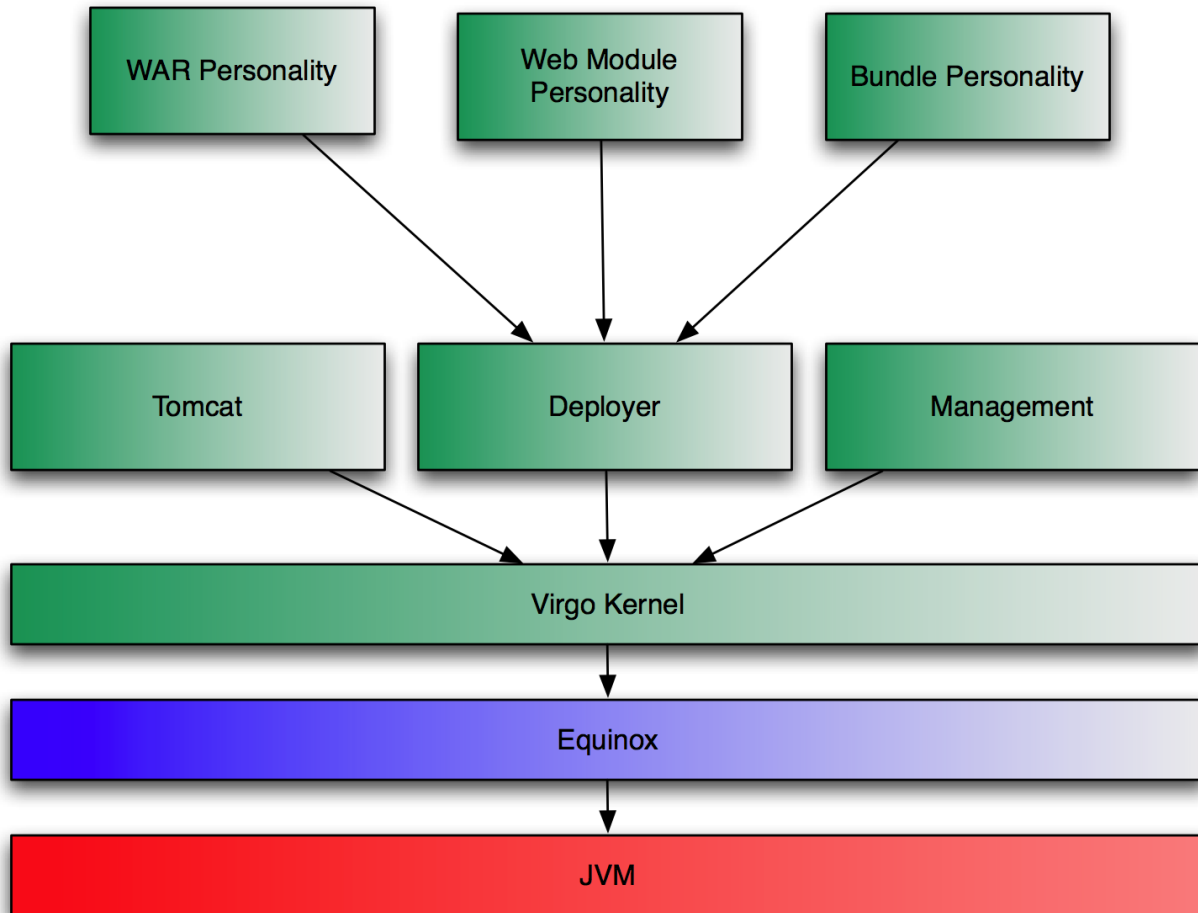
What makes up the Virgo Web Server?

The Virgo Web Server is built on top of the following core technologies:

- [Spring Framework](#), obviously!
- [Tomcat](#) as the web container.
- [OSGi R4.2](#).
- [Equinox](#) as the OSGi implementation.
- [Spring Dynamic Modules for OSGi](#) for working with OSGi in a Spring application.
- [SpringSource Tool Suite](#) for developing applications.

Note, however, that the Virgo Web Server isn't just a combination of these technologies. Rather, it integrates and extends these technologies to provide many features essential for developing, deploying, and managing today's enterprise Java applications.

The following diagram presents a high-level overview of the VWS's architecture.



At the heart of the Virgo Web Server is the Virgo Kernel or VK. The VK is an OSGi-based kernel that takes full advantage of the modularity and versioning of the OSGi platform. The VK builds on Equinox and extends its capabilities for provisioning and library management, as well as providing core functionality for the VWS.

To maintain a minimal runtime footprint, OSGi bundles are installed on demand by the VK provisioning subsystem. This allows for an application to be installed into a running VWS and for its dependencies to be satisfied from an external repository. Not only does this remove the need to manually install all your application dependencies, which would be tedious, but it also keeps memory usage to a minimum.

As shown in the figure, VK runs on top of Equinox within a standard Java Virtual Machine. Above the VK is a layer of subsystems which contribute functionality to the VWS. Subsystems are configured to run for various *profiles* and typically provide additional services to the basic OSGi container such as serviceability, management, and personality-specific deployment.

In the Virgo Web Server, applications are modular and each module has a personality that describes what kind of module it is: web, batch, web service, etc. The VWS deploys modules of each personality in a personality-specific manner. For example, web modules are configured in Tomcat with web context. Each module in the application can be updated independently of the other modules whilst retaining the identity of being part of the larger application. Whatever kind

of application you are building, the programming model remains standard Spring and Spring DM.

Version 2.1.0.M01 of the Virgo Web Server supports the *bundle*, *web*, and *WAR* personalities, which enable you to build sophisticated web applications. The WAR personality includes support for standard Java EE WARs, "shared library" WARs, and "shared services" WARs, each of which will be covered in greater detail in Chapter 3, *Deployment Architecture*. Future releases will include support for more personalities such as batch, web services, etc.

2.3 Why the Virgo Web Server?

You could deploy a web application in a stand-alone servlet engine or application server. Or you could even deploy directly in an OSGi container such as Equinox. However, deploying in the Virgo Web Server offers a number of key benefits that make it both more appealing and more suitable for enterprise application development.

Deployment options and migration paths

While many applications deployed in the Virgo Web Server will take advantage of OSGi capabilities, not all applications need such sophistication. For example, development teams may initially choose to continue packaging existing web applications as standard WAR files and then gradually migrate toward a fully OSGi-based packaging and deployment model. The Virgo Web Server makes such migrations easy for developers by supporting multiple packaging and deployment formats. These formats and migration strategies are discussed in greater detail in Chapter 5, *Migrating to OSGi* and Chapter 6, *Case study: Migrating the Form Tags sample application*.

Simplified development and deployment of OSGi-based applications

Prior to the release of the Virgo Web Server, developing and deploying OSGi applications involved inherent complexity such as:

- *Obtaining OSGi bundles for popular Java libraries:* For optimal benefits, every technology you use in an OSGi application must be packaged as OSGi bundles. Currently, this involves manually converting JAR files into bundles and making sure that any libraries needed by those bundles are also available as OSGi bundles.
- *Package management complexity:* OSGi bundles use other bundles through `Import-Package` manifest headers. Many applications use a set of common technologies (e.g., an ORM solution, a web framework, etc.). Combining these two characteristics leads to duplicated configuration in the form of repeated and verbose `Import-Package` statements.
- *Lack of application-level isolation:* In OSGi everything is a bundle, and all bundles share the

same OSGi Service Registry. To highlight how conflicts can arise between applications and their services in this shared service registry, consider the following scenarios.

- Application A is comprised of bundles B and C. In a standard OSGi environment, if you attempt to install two instances of the same version of application A (i.e., two sets of bundles B and C), a clash will occur, because you cannot deploy multiple bundles with the same `Bundle-SymbolicName` and `Bundle-Version` combination.
- Application A1 is comprised of bundles B1 and C1. Similarly, application A2 is comprised of bundles B2 and C2. Each bundle has a unique combination of `Bundle-SymbolicName` and `Bundle-Version`. Bundles B1 and B2 both export service S which is imported by both C1 and C2. In contrast to the previous example, there is no conflict resulting from duplicate `Bundle-SymbolicName/Bundle-Version` combinations; however, there is a clash for the exported service S. Which service S will bundles C1 and C2 end up using once they are installed? Assuming bundles B1 and C1 are intended to work together, you would not want bundle C1 to get a reference to service S from bundle B2, because it is installed in a different logical application. On the contrary, you typically want bundle C1 to get a reference to service S exported by bundle B1, but in a standard OSGi environment this may not be the case.

Furthermore, since standard OSGi does not define a notion of an application as a set of bundles, you cannot deploy or undeploy an application and its constituent bundles as a single unit.

The Virgo Web Server introduces a number of features to solve these issues:

- A full set of OSGi bundles for many popular Java libraries to get you started quickly with creating OSGi applications.
- An OSGi library concept that obviates the need to duplicate verbose `Import-Package` statements.
- The PAR packaging format which offers application-level isolation and deployment.
- The concept of a plan, which is an XML file that lists a collection of bundles that Virgo Web Server should load together as a single application. Conceptually, plans are very like PARs, except that a plan describes the contents of the application rather than a PAR that actually contains them.

Enhanced diagnostics during deployment and in production

Identifying why an application won't deploy or which particular library dependencies are unsatisfied is the cause of many headaches! Similarly, production time errors that don't identify the root cause are all too familiar to Java developers. The VWS was designed from the ground up to enable tracing and First Failure Data Capture (FFDC) that empower developers with precise information at the point of failure to fix the problem quickly.

3. Deployment Architecture

The Virgo Web Server offers several choices when it comes to deploying applications. Each choice offers certain advantages, and it is important to understand those in order to make the right choice for your application. In this chapter, we take a closer look at the choices offered, compare them, and provide guidelines in choosing the right one based on your specific needs.

The VWS supports standard self-contained WAR files thus allowing you to use the Virgo Web Server as an enhanced web server. The VWS also supports the *Shared Libraries* WAR format which allows for slimmer WAR files that depend on OSGi bundles instead of including JAR files inside the WAR. The *Shared Services* WAR format allows developers to further reduce the complexity of standard WARs by deploying services and infrastructure bundles alongside the WAR. A shared services WAR will then consume the services published by those bundles. To complete the picture, the VWS supports the new OSGi-standard *Web Bundle* deployment format for web applications that builds on the benefits provided by a shared services WAR. In addition to this VWS provides additional conveniences for developing and deploying Spring MVC-based web applications.

For applications consisting of multiple bundles and web applications, plans and the PAR format are the primary deployment models that take advantage of OSGi capabilities. We will explore all of these formats and their suitability later in this guide.

3.1 Supported Deployment Formats

The Virgo Web Server supports applications packaged in the following formats:

1. [Raw OSGi Bundles](#)
2. [Java EE WAR](#)
3. [Web Bundles](#)
4. [PAR](#)
5. [Plans](#)

When you deploy an application to the VWS, each deployment artifact (e.g., a single bundle, WAR, PAR, or plan) passes through a deployment pipeline. This deployment pipeline supports the notion of personality-specific deployers which are responsible for processing an application with a certain personality (i.e., application type). The 2.1.0.M01 release of the VWS natively supports personality-specific deployers analogous to each of the aforementioned packaging options. Furthermore, the deployment pipeline can be extended with additional personality deployers, and future releases of the VWS will provide support for personalities such as Batch, Web Services, etc.

Let's take a closer look now at each of the supported deployment and packaging options to

explore which one is best suited for your applications.

Raw OSGi Bundles

At its core, the Virgo Web Server is an OSGi container. Thus any OSGi-compliant bundle can be deployed directly on the VWS unmodified. You'll typically deploy an application as a single bundle or a set of stand-alone bundles if you'd like to publish or consume services globally within the container via the OSGi Service Registry.

WAR Deployment Formats

For Web Application Archives (WAR), the Virgo Web Server provides support for the following three formats.

1. [Standard WAR](#)
2. [Shared Libraries WAR](#)
3. [Shared Services WAR](#)

Each of these formats plays a distinct role in the incremental migration path from a standard Java EE WAR to an OSGi-ified web application.

Standard WAR

Standard WAR files are supported directly in the VWS. At deployment time, the WAR file is transformed into an OSGi bundle and installed into Tomcat. All the standard WAR contracts are honored, and your existing WAR files should just drop in and deploy without change. Support for standard, unmodified WAR files allows you to try out the Virgo Web Server on your existing web applications and then gradually migrate toward the *Shared Libraries WAR* and *Shared Services WAR* formats.

In addition to the standard support for WARs that you would expect from Tomcat, the VWS also enables the following features:

1. Spring-driven load-time weaving (see Section 6.8.4, "Load-time weaving with AspectJ in the Spring Framework").
2. Diagnostic information such as FFDC (first failure data capture)

The main benefit of this application style is familiarity -- everyone knows how to create a WAR file! You can take advantage of the VWS's added feature set without modifying the application. The application can also be deployed on other Servlet containers or Java EE application servers.

You may choose this application style if the application is fairly simple and small. You may also prefer this style even for large and complex applications as a starting point and migrate to the

other styles over time as discussed in Chapter 5, *Migrating to OSGi*.

Shared Libraries WAR

If you have experience with developing and packaging web applications using the standard WAR format, you're certainly familiar with the pains of library bloat. So, unless you're installing shared libraries in a common library folder for your Servlet container, you have to pack all JARs required by your web application in `/WEB-INF/lib`. Prior to the release of the Virgo Web Server, such library bloat has essentially been the norm for web applications, but now there is a better solution! The Shared Libraries WAR format reduces your application's deployment footprint and eradicates library bloat by allowing you to declare dependencies on libraries via standard OSGi manifest headers such as `Import-Package` and `Require-Bundle`. The VWS provides additional support for simplifying dependency management via the `Import-Library` and `Import-Bundle` manifest headers which are essentially macros that get expanded into OSGi-compliant `Import-Package` statements.



Tip

For detailed information on which libraries are already available, check out the [SpringSource Enterprise Bundle Repository](#).

Shared Services WAR

Once you've begun taking advantage of declarative dependency management with a Shared Libraries WAR, you'll likely find yourself wanting to take the next step toward reaping further benefits of an OSGi container: sharing services between your OSGi-compliant bundles and your web applications. By building on the power and simplicity of Spring-DM, the *Shared Services* WAR format puts the OSGi Service Registry at your finger tips. As a best practice you'll typically publish services from your domain, service, and infrastructure bundles via `<osgi:service ... />` and then consume them in your web application's `ApplicationContext` via `<osgi:reference ... />`. Doing so promotes programming to interfaces and allows you to completely decouple your web-specific deployment artifacts from your domain model, service layer, etc., and that's certainly a step in the right direction. Of the three supported WAR deployment formats, the Shared Services WAR is by far the most attractive in terms of modularity and reduced overall footprint of your web applications.

WARs and the OSGi Web Container (RFC66)

Virgo Web Server fully supports the OSGi Web Container (RFC66) standard. In fact, the reference implementation for RFC66 was developed by SpringSource from an offshoot of the original VWS codebase. This RI is now fully integrated in VWS as the basis of the support for web application deployment.

The Web Container specification introduces the concept of a *web bundle*, which is a WAR that is also a bundle. The specification defines how WAR files are transformed into bundles automatically as needed.

The Web Container specification is not yet publicly available, but you can find an introduction to the Web Container in blog entries written by the VWS team [here](#) and [here](#).

Extensions to the Web Container

Virgo Web Server provides a variety of extensions to the Web Container that allow you to construct sophisticated applications. The table below, summarizes the extensions that are available or in development.

Table 3.1.

Feature	Description
Auto-import of system packages	All packages exported by the system bundle are automatically imported by web bundles
Instrumentable ClassLoaders	All web bundle ClassLoaders are instrumentable by Spring's load-time weaving infrastructure.
Support for exploded bundles/WARs	Bundles/WARs in directory form can be deployed as web bundles
Support for scanning TLDs in dependencies	As per the Web Container specification, all TLDs located inside a web bundle are located using the rules defined in the JSP 2.1 specification. In VWS, the dependencies of a web bundle are also scanned for TLDs following the rules outlined in JSP 2.1

Web Modules

Web Modules have been removed in favor of war files and web bundles following the OSGi Web Container specification. We believe our users will benefit more from a standard model than one that is VWS-specific.

PAR

A PAR is a standard JAR which contains all of the modules of your application (e.g., service, domain, and infrastructure bundles as well as a WAR or web module for web applications) in a single deployment unit. This allows you to deploy, refresh, and undeploy your entire application as a single entity. If you are familiar with Java EE, it is worth noting that a PAR can be considered a replacement for an EAR (Enterprise Archive) within the context of an OSGi container. As an added bonus, modules within a PAR can be refreshed independently and on-the-fly, for example via the Virgo Web Server Tool Suite (see Chapter 7, *Tooling*).

Many of the benefits of the PAR format are due to the underlying OSGi infrastructure, including:

- Fundamentally modularized applications: instead of relying on fuzzy boundaries between logical modules in a monolithic application, this style promotes physically separated modules in the form of OSGi bundles. Then each module may be developed separately, promoting parallel development and loose coupling.
- Robust versioning of various modules: the versioning capability offered by OSGi is much more comprehensive than any alternatives. Each module can specify a version range for each of its dependencies. Bundles are isolated from each other in such a way that multiple versions of a bundle may be used simultaneously in an application.
- Improved serviceability: each bundle may be deployed or undeployed in a running application. This allows modifying the existing application to fix bugs, improve performance, and even to add new features without having to restart the application.

Furthermore, PARs scope the modules of your application within the VWS. Scoping provides both a physical and logical application boundary, effectively shielding the internals of your application from other PARs deployed within the VWS. This means your application doesn't have to worry about clashing with other running applications (e.g., in the OSGi Service Registry). You get support for load-time weaving, classpath scanning, context class loading, etc., and the VWS does the heavy lifting for you to make all this work seamlessly in an OSGi environment. If you want to take full advantage of all that the Virgo Web Server and OSGi have to offer, packaging and deploying your applications as a PAR is a good choice, although plans are an even better one, as described in the next section.

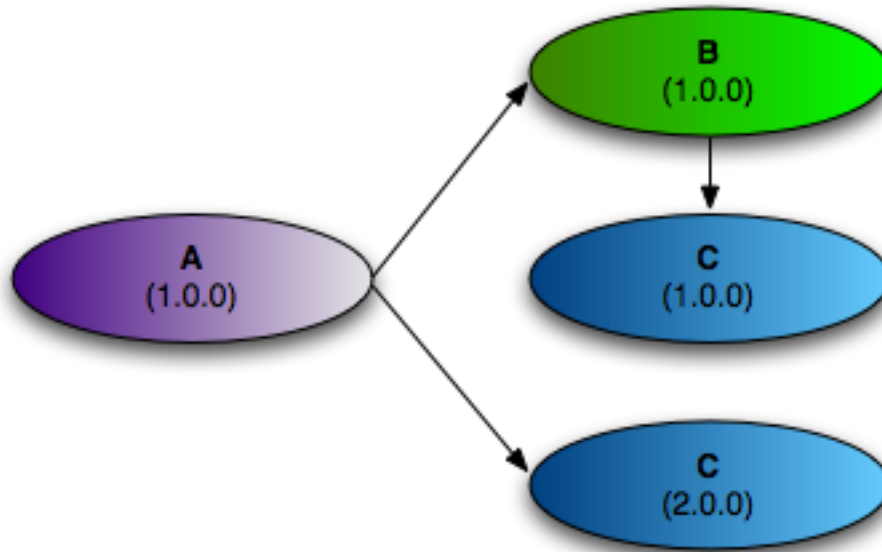


OSGi != multiple JARs

Note that while physically separated modules can, in theory, be implemented simply using multiple JARs, complex versioning requirements often make this impractical. For example, consider the situation depicted in the diagram below.

- Bundle A depends on version 1.0.0 of bundle B and version 2.0.0 of bundle C.
- Bundle B depends on version 1.0.0 of bundle C.

Suppose that versions 1.0.0 and 2.0.0 of bundle C are neither backward nor forward compatible. Traditional monolithic applications cannot handle such situations: either bundle A or bundle B would need reworking which undermines truly independent development. OSGi's versioning scheme enables this scenario to be implemented in a robust manner. If it is desirable to rework the application to share a single version of C, then this can be planned in and is not forced.



Plans

A plan is similar to a PAR in that it encapsulates all of the artifacts of your application in a single deployment unit. The main difference, however, is that a plan is simply an XML file that lists the artifacts of your application; a PAR, by contrast, is an actual JAR file that physically contains the artifacts. Just like a PAR, you deploy, refresh, and undeploy a plan as a single entity. We highly recommend the use of plans for creating applications.

When you create a plan, you can specify that the included bundles and services are in a scope that isolates them from the rest of Virgo Web Server and its deployments. This scoping ensures that the bundles wire to each other and see each other's services in preference to services from outside the scope. Scoping also prevents application code from leaking into the global scope or scope of another application. In addition, a plan can link the lifecycle of a group of bundles together atomically, which ensures that install, start, stop, and uninstall events on a single artifact in the plan are escalated to all artifacts in the plan. You can, however, disable both of these features by simply updating an attribute in the plan.

The general benefits of using plans are similar to those of using PARs; see [PAR](#) for details. Plans offer added benefits, however, such as the ability to control the deployment order of your application: the order in which you list artifacts in the plan's XML file is the order in which VWS deploys them. Additionally, because plans specify the artifacts that make up an application by reference, it is easier to share content between plans as well as update individual parts of a plan without having to physically repackage (re-JAR) it.

3.2 Dependency Types

In an OSGi environment, there are two kinds of dependencies between various bundles: *type* dependency and *service* dependency.

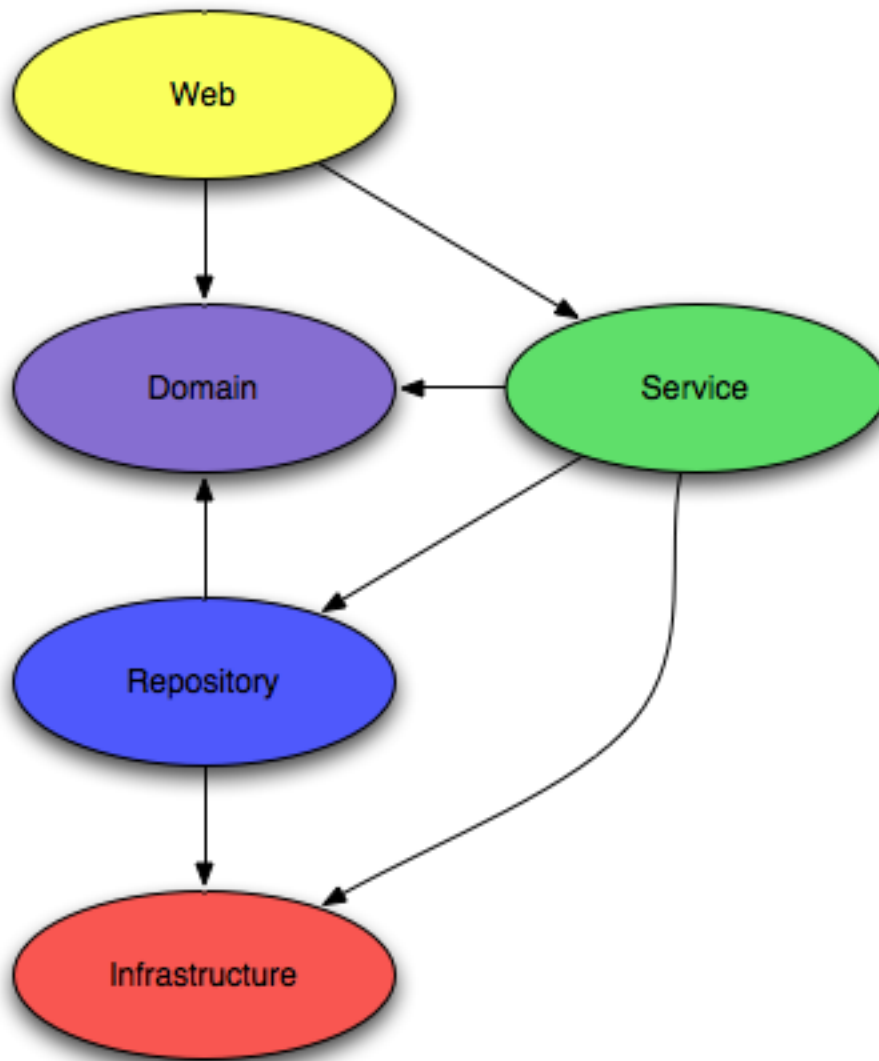
- **Type dependency:** A bundle may depend on a type exported by another bundle thus creating a type dependency. Type dependencies are managed through `Import-Package` and `Export-Package` directives in the OSGi manifest. This kind of dependency is similar to a JAR file using types in other JAR files from the classpath. However, as we've seen earlier, there are significant differences.
- **Service dependency:** A bundle may also publish services (preferably using Spring-DM), and other bundles may consume those services. If two bundles depend on the same service, both will be communicating effectively to the same object. More specifically, any state for that service will be shared between all the clients of that service. This kind of arrangement is similar to the commonly seen client-server interaction through mechanisms such as RMI or Web Services.

3.3 A guide to forming bundles

So what makes a good application suitable for deployment on the Virgo Web Server? Since OSGi is at the heart of the VWS, modular applications consisting of bundles, which each represent distinct functionality and well-defined boundaries, can take maximum advantage of the OSGi container's capabilities. The core ideas behind forming bundles require following good software engineering practices: separation of concerns, minimum coupling, and communication through clear interfaces. In this section, we look at a few approaches that you may use to create modular applications for Virgo Web Server deployment. Please consider the following discussion as guidelines and not as rules.

Bundles can be formed along horizontal slices of layering and vertical slices of function. The objective is to enable independent development of each bundle and minimize the skills required to develop each bundle.

For example, an application could have the following bundles: *infrastructure*, *domain*, *repository*, *service*, and *web* as shown in the following diagram.



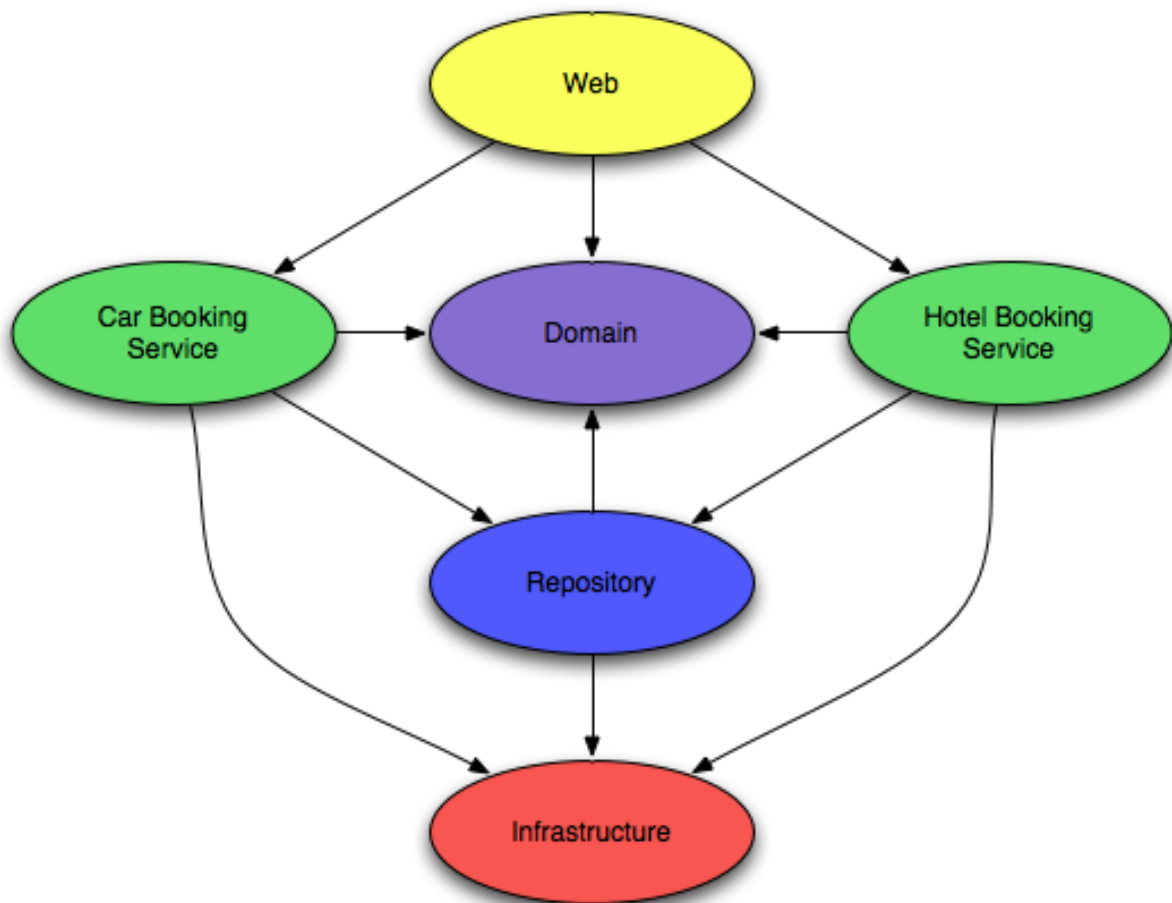
Each bundle consists of types appropriate for that layer and exports packages and services to be used by other layers. Let's examine each bundle in more detail:

Table 3.2. Bundles across layers

Bundles	Imported Packages	Exported Packages	Consumed Services	Published Services
Infrastructure	Third-party libraries	Infrastructure interfaces	None	None
Domain	Depends: for example, if JPA is used to annotate persistent types, then JPA packages.	Public domain types	None	None
Web	Domain, Service	None	Service beans	None
Service	Domain, Infrastructure,	Service	Repository	Service beans

Bundles	Imported Packages	Exported Packages	Consumed Services	Published Services
	Repository	interfaces	beans	
Repository	Domain, Third-party libraries, ORM bundles, etc.	Repository interfaces	DataSources, ORM session/entity managers, etc.	Repository beans

Within each layer, you may create bundles for each subsystem representing a vertical slice of business functionality. For example, as shown in the following figure, the service layer is divided into two bundles each representing separate business functionalities.



You can similarly separate the repositories, domain classes, and web controllers based on the business role they play.

4. Developing Applications

Applications that take advantage of the OSGi capabilities of the Virgo Web Server are typically comprised of multiple bundles. Each bundle may have dependencies on other bundles. Furthermore, each bundle exposes only certain packages and services. In this chapter, we look at how to create bundles, import and export appropriate functionality, and create artifacts to deploy web applications on the Virgo Web Server.

4.1 Anatomy of a bundle



Tip

This is an abbreviated introduction to OSGi bundles. Please refer to the [Spring Dynamic Modules for OSGi documentation](#) for full details.

An OSGi bundle is simply a jar file with metadata that describe additional characteristics such as version and imported and exported packages.

A bundle exports types and publishes services to be used by other bundles:

- **Types:** via the OSGi `Export-Package` directive,
- **Services:** via Spring-DM's `<service ... />` XML namespace element.

A bundle may import types and services exported by other bundles:

- **Types:** via the OSGi `Import-Package` directive,
- **Services:** via Spring-DM's `<reference ... />` XML namespace element.

Let's see an example from the PetClinic sample application. The following listing shows the `MANIFEST.MF` file for the `org.springframework.petclinic.infrastructure.hsqldb` bundle.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: PetClinic HSQL Database Infrastructure
Bundle-SymbolicName: org.springframework.petclinic.infrastructure.hsqldb
Bundle-Version: 1.0
Import-Library: org.springframework.spring;version="[2.5,2.6]"
Import-Bundle: com.springsource.org.apache.commons.dbcp;version="[1.2.2.osgi,1.2.2.osgi]",
               com.springsource.org.hsqldb;version="[1.8.0.9,1.8.0.9]"
Import-Package: javax.sql
Export-Package: org.springframework.petclinic.infrastructure
```

The `org.springframework.petclinic.infrastructure.hsqldb` bundle expresses its dependencies on the `javax.sql` package, the Commons DBCP and HSQLDB bundles, and the Spring library (we will examine the details of the library artifact in the section

called “Defining libraries”). The Commons DBCP bundle is imported at a version of exactly 1.2.2.osgi and the HSQLDB bundle is imported at a version of exactly 1.8.0.9. The Spring library is imported at a version between 2.5 inclusive and 2.6 exclusive.

Note that you do not specify the bundle that will provide the imported packages. The Virgo Web Server will examine the available bundles and satisfy the required dependencies.

The following `osgi-context.xml` file from the PetClinic sample’s `org.springframework.petclinic.repository.jdbc` bundle declares a service published by the bundle and references a service published by another bundle.

```
<?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: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/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

  <service id="osgiClinic" ref="clinic" interface="org.springframework.petclinic.repository.Clinic" />
  <reference id="dataSource" interface="javax.sql.DataSource"/>

</beans:beans>
```

The `service` element publishes the `clinic` bean (a regular Spring bean declared in the `module-context.xml` file) and specifies `org.springframework.petclinic.repository.Clinic` as the type of the published service.

The `reference` elements define a `dataSource` bean that references a service published by another bundle with an interface type of `javax.sql.DataSource`.

4.2 Creating PARs and WARs

The Virgo Web Server supports two OSGi-oriented ways of packaging applications: the PAR format and application modules (including personality-specific modules). The VWS also supports three distinct WAR deployment and packaging formats: standard Java EE WAR, Shared Libraries WAR, Shared Services WAR.

The VWS also supports plans as a way to describe an application. This method is similar to a PAR in that it encapsulates all the artifacts of an application as a single unit, but differs in that a plan simply lists the bundles in an XML file rather than packaging all the bundles in a single JAR file. The use of plans offers additional benefits to using PARs; for this reason, we recommend their use. For details, see [Creating Plans](#).

PARs

An OSGi application is packaged as a JAR file, with extension `.par`. A PAR artifact offers several benefits:

- A PAR file has an application name, version, symbolic name, and description.

- The modules of a PAR file are scoped so that they cannot be shared accidentally by other applications. The scope forms a boundary for automatic propagation of load time weaving and bundle refresh.
- The modules of a PAR have their exported packages imported by the synthetic context bundle which is used for thread context class loading. So, for example, hibernate will be able to load classes of any of the exported packages of the modules in a PAR file using `Class.forName()` (or equivalent).
- The PAR file is visible to management interfaces.
- The PAR file can be undeployed and redeployed as a unit.

A PAR includes one or more application bundles and its manifest specifies the following manifest headers:

Table 4.1. PAR file headers

Header	Description
Application-SymbolicName	Identifier for the application which, in combination with Application-Version, uniquely identifies an application
Application-Name	Human readable name of the application
Application-Version	Version of the application
Application-Description	Short description of the application

The following code shows an example MANIFEST.MF in a PAR file:

```
Application-SymbolicName: com.example.shop
Application-Version: 1.0
Application-Name: Online Shop
Application-Description: Example.com's Online Shopping Application
```

Web Modules

As discussed earlier, Web Modules are no longer supported in VWS. Instead, we recommend that you use Shared Service WARs or Web Bundles that are compliant with the OSGi Web Container specification.

Migrating to a Web Bundle from a Web Module

To move from a Web Module to a Web Container-compliant Web Bundle you need to follow these four steps:

1. Remove the `Module-Type` manifest header

2. Replace any `Web-DispatcherServletUrlPatterns` header with the corresponding servlet entries in `web.xml`
3. Replace any `Web-FilterMappings` header with the corresponding filter entries in `web.xml`
4. Move all content in `MODULE-INF` to the root of the WAR

Removing `Web-DispatcherServletUrlPatterns`

To remove a `Web-DispatcherServletUrlPatterns` header such as `Web-DispatcherServletUrlPatterns: *.htm`, start by declaring a `DispatcherServlet` in `web.xml`:

```
<servlet>
  <servlet-name>dispatcher.myapp</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>
```

For every mapping in the `DispatcherServletUrlPatterns` header, create the corresponding servlet-mapping:

```
<servlet-mapping>
  <servlet-name>dispatcher.myapp</servlet-name>
  <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

Removing `Web-FilterMappings`

To remove a `Web-FilterMappings` header such as `Web-FilterMappings: myfilter;url-patterns:= "*.htm"`, start by declaring `DelegatingFilterProxy` in `web.xml` for each filter listed:

```
<filter>
  <filter-name>myfilter</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
```

For every mapping listed for the filter create the corresponding filter-mapping:

```
<filter-mapping>
  <filter-name>myfilter</filter-name>
  <url-pattern>*.htm</url-pattern>
</filter-mapping>
```

4.3 Creating Plans

Plans are similar to PARs in that they encapsulate the artifacts of an application as a single unit. As a consequence, they have similar benefits; for details of the benefits common to PARs and plans, see [PARs](#).

Plans have the following additional benefits, which is why we recommend that you use plans rather than PARs when defining an application:

- Virgo Web Server deploys the artifacts in the plan in the order in which they are listed in the XML file, which gives you complete control over deployment order. With a PAR, the order of deployment of the included artifacts is not guaranteed.
- Plans describe their contents by reference (using an XML file) as opposed to PARs that are JAR files that physically contain the included artifacts. For this reason, it is easier to share content between plans as well as update individual parts of a plan without having to physically repack (re-JAR) it.
- You can enable or disable whether a plan is scoped or atomic; PARs are always scoped and atomic.

Plans always get their dependencies from the VWS repository. This means, for example, that if you drop one of the plan's dependencies in the `pickup` directory rather than adding it to the repository, the plan will fail to deploy because it will not find the dependency.

Creating the Plan XML File

Plans are XML files that have a `.plan` file extension, such as `multi-artifact.plan`. The structure of the XML file is simple: the root element is `<plan>` with attributes specifying the name of the plan, the version, atomicity, and scoping. Then, for each artifact that makes up your application, you add a `<artifact>` element, using its attributes to specify the type of artifact and its name and version. The following is a simple example of a plan's XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<plan name="multi-artifact.plan" version="1.0.0" scoped="true" atomic="true"
      xmlns="http://www.springframework.org/schema/dm-server/plan"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="
        http://www.springframework.org/schema/dm-server/plan
        http://www.springframework.org/schema/dm-server/plan/springsource-dm-server-plan.xsd">

  <artifact type="configuration" name="app-properties" version="1.0.0"/>
  <artifact type="bundle" name="com.springframework.exciting.app" version="[2.0.0, 3.1.0)"/>

</plan>
```

In the preceding example, the name of the plan is `multi-artifact.plan` and its version is `1.0.0`. The plan is both scoped and atomic. The plan contains two artifacts: one is a bundle called `com.springframework.exciting.app` and the other is a configuration file called `app-properties`.

The following table describes the attributes of the `<plan>` element.

Table 4.2. Attributes of the `<plan>` Element

Attribute	Description	Required?
name	Specifies the name of this plan. Virgo Web Server uses the name as one component of the unique identifier of this plan.	Yes.
version	Specifies the version of this plan. You must use OSGi version specification syntax, such as <code>2.1.0</code> . Virgo	Yes.

Attribute	Description	Required?
	Web Server uses the version as one component of the unique identifier of this plan.	
scoped	Specifies whether Virgo Web Server should install the artifacts into plan-specific scope so that only the application described by this plan has access to the artifacts. If you disable scoping, then Virgo Web Server installs the artifacts into the global scope, which means they are then available for access by all other deployed artifacts. Set the attribute to <code>true</code> to enable scoping or <code>false</code> to disable it.	Yes.
atomic	Specifies whether you want to tie together the lifecycle of the artifacts in this plan. Making a plan atomic means that if you install, start, stop, or uninstall a single artifact in the plan, Virgo Web Server escalates the event to all artifacts in the plan. Also, in an atomic plan, Virgo Web Server prevents artifacts from being in inconsistent states. For example, if one artifact should fail to start, then Virgo Web Server stops all artifacts in the plan. Set this attribute to <code>true</code> to enable atomicity or <code>false</code> to disable it.	Yes.

The following table describes the attributes of the `<artifact>` element.

Table 4.3. Attributes of the `<artifact>` Element

Attribute	Description	Required?
type	Specifies the type of the artifact. Valid values are: <ul style="list-style-type: none"> <code>bundle</code>: Specifies an OSGi bundle. Use this artifact type for WAR files. <code>configuration</code>: Specifies that the artifact is a configuration file. Configuration files contain name/value pairs that set initial values for configuration properties of a bundle. <code>plan</code>: Specifies that the artifact is a plan. <code>par</code>: Specifies that the artifact is a PAR. 	Yes.
name	Specifies the name of the artifact. See Artifact Names for guidelines for determining the name of an artifact.	Yes.

Attribute	Description	Required?
version	Specifies the version or range of versions of this artifact that VWS should look up in its repositories and then install and deploy. You must use OSGi version specification syntax, such as <code>[1.0.0, 2.0.0)</code> .	No. If not specified, defaults to 0, which in OSGi means 0 to infinity, or any version.

Artifact Names

When you create a plan, you use the name attribute of the `<artifact>` element to specify the name of all the plan's dependencies. This section describes how to determine the name of an artifact, which is not always obvious.

Use the following guidelines to determine the name of an artifact:

- **Bundle:** In this context, a *bundle* refers to a standard OSGi bundle as well as a Web application WAR file. The name of a bundle is the value of the `Bundle-SymbolicName` header in the `META-INF/MANIFEST.MF` file of the `*.jar` or `*.war` file. The following `MANIFEST.MF` snippet shows a bundle with name `com.springsource.exciting.app`:

```
Bundle-SymbolicName: com.springsource.exciting.app
```

If the bundle does not contain a `META-INF/MANIFEST.MF` file, then the name of the bundle is its filename minus the `.jar` or `.war` extension.

- **Configuration File:** The name of a configuration file is its filename minus the `.properties` extension.
- **Plan:** The name of a plan is the value of the required name attribute of the `<plan>` element in the plan's XML file. In the following XML snippet, the plan name is `multi-artifact.plan`:

```
<?xml version="1.0" encoding="UTF-8"?>
<plan name="multi-artifact.plan" version="1.0.0" scoped="true" atomic="true"
      xmlns="http://www.springsource.org/schema/dm-server/plan"
...

```

- **PAR:** The name of a PAR is the value of the `Application-SymbolicName` header in the `META-INF/MANIFEST.MF` file of the `*.par` file. The following `MANIFEST.MF` snippet shows a PAR with name `com.springsource.my.par`:

```
Application-SymbolicName: com.springsource.my.par
```

If the PAR does not contain a `META-INF/MANIFEST.MF` file, then the name of the PAR is its filename minus the `.par` extension.

Using the Plan

Because a plan is a list of artifacts, rather than a physical file that contains the artifacts, there are a few additional steps you must perform before you deploy it to VWS.

1. Copy the artifacts that make up the plan to the `usr` repository, which by default is the `$DMS_HOME/repository/usr` directory, where `$DMS_HOME` refers to the top-level installation directory of VWS. Note that you might have configured the server differently; in which case, copy the artifacts to your custom repository directory.
2. Restart VWS.
3. After the server has started, either use the Admin Console to deploy the plan, or manually deploy it by copying the plan's XML file into the `$DMS_HOME/pickup` directory.

This results in VWS deploying the plan with the same semantics as a PAR file.

4. To undeploy the plan, use the Admin Console, or simply delete it from the `$DMS_HOME/pickup` directory.

Plans and Scoping

As described in previous sections, you can specify that a plan be *scoped*. This means that Virgo Web Server installs the artifacts that make up the plan into a plan-specific scope so that only the application described by the plan has access to the artifacts. If you disable scoping, then Virgo Web Server installs the artifacts into the global scope, which means they are available for access by all other deployed artifacts. This section describes scoping in a bit more detail. It also describes how you can change the default behavior of scoping, with respect to services, so that a service that is in a scope can be made globally available.

If a bundle in a given scope imports a package and a bundle in the same scope exports the package, then the import may only be satisfied by the bundle in the scope, and not by any bundles outside the scope, including the global scope. Similarly, package exports from bundles in a scope are not visible to bundles in the global scope.

If a bundle in a scope uses Spring DM (or the blueprint service) to obtain a service reference and a bundle in the same scope uses Spring DM (or the blueprint service) to publish a matching service, then the service reference may only bind to the service published in the scope (and not to any services outside the scope). Services published by bundles in a scope are not visible to bundles in the global scope.

However, sometimes it is useful to make a service in a scope globally available to artifacts outside the scope. To do this, publish the service with the `com.springsource.service.scope` service property set to `global`. Use the `<service-properties>` child element of `<service>`, as shown in the following example:

```
<service id="publishIntoGlobal" interface="java.lang.CharSequence">
  <service-properties>
    <beans:entry key="com.springsource.service.scope" value="global" />
  </service-properties>
  <beans:bean class="java.lang.String">
    <beans:constructor-arg value="foo"/>
  </beans:bean>
</service>
```

4.4 Creating and Using Configuration Artifacts

Applications typically include some sort of configuration data that might change depending on the environment in which the application is deployed. For example, if an application connects to a database server using JDBC, the configuration data would include the JDBC URL of the database server, the JDBC driver, and the username and password that the application uses to connect to the database server. This information often changes as the application is deployed to different computers or the application moves from the testing phase to the production phase.

Virgo Web Server provides a feature called *configuration artifacts* that makes it very easy for you to manage this configuration data. A configuration artifact is simply a properties file that is made available at runtime using the OSGi ConfigurationAdmin service. When you create this properties file, you set the values of the properties for the specific environment in which you are going to deploy your application, and then update the metadata of your Spring application to use the properties file. You then deploy the application and properties file together, typically as a [plan](#). Virgo Web Server automatically creates a configuration artifact from the properties file, and you can manage the lifecycle of this configuration artifact in the same way you manage the lifecycle of PARs, bundles, and plans, using both the Admin Shell and Admin Console. Additionally, VWS subscribes your application for notification of any refresh of the configuration artifact and the application can then adapt accordingly, which means you can easily *change* the configuration of your application without redeploying it.

In sum, configuration artifacts, especially when combined with plans, provide an excellent mechanism for managing external configuration data for your applications.

The following sections describe the format of the configuration artifact, how to update the Spring application context file of your application so that it knows about the configuration artifact, and finally how to include it in a plan alongside your application.

As an example to illustrate the configuration artifact feature, assume that you have a Spring bean called `PropertiesController` whose constructor requires that four property values be passed to it, as shown in the following snippet of Java code:

```
@Controller
public class PropertiesController {

    private final String driverClassName;
    private final String url;
    private final String username;
    private final String password;

    public PropertiesController(String driverClassName, String url, String username, String password) {
        this.driverClassName = driverClassName;
        this.url = url;
        this.username = username;
        this.password = password;
    }
}
```

In the preceding example, the `PropertiesController` constructor requires four property values: `driverClassName`, `url`, `username`, and `password`. Note that the example shows just one way that a class might require property values; your application may code it another way.

Additionally, assume that the following snippet of the associated Spring application context XML file shows how the `PropertiesController` bean is configured:

```
<bean class="com.springsource.configuration.properties.PropertiesController">
    <constructor-arg value="${driverClassName}" />
    <constructor-arg value="${url}" />
    <constructor-arg value="${username}" />
    <constructor-arg value="${password}" />
</bean>
```

The rest of this section describes how the bean can get these property values using a configuration artifact.

Creating the Properties File

To create a properties file that in turn will become a configuration artifact when deployed to VWS from which a Spring bean, such as the `PropertiesController` bean, will get the actual property values, follow these guidelines:

- Create a text file in which each property is listed as a name/value pair, one pair per line. Precede comments with a `#`. For example:

```
# Properties for the com.springsource.configuration.properties sample
driverClassName = org.w3.Driver
url             = http://www.springsource.com
username        = joe
password        = secret
```

The example shows four properties whose name correspond to the constructor arguments of the `PropertiesController` Spring bean.

- Name the file anything you want, as long as it has a `.properties` extension, such as `app-properties.properties`.

Updating Your Application

To update your application so that it "knows" about the configuration artifact, you update the application's Spring application context XML file, typically located in the `WEB-INF` directory.

You use the `<context:property-placeholder>` element to specify that you want to use the VWS mechanism for substituting values into bean properties. The `properties-ref` attribute of this element points to a `<osgi-compendium:cm-properties>` element which you use to specify the configuration artifact that contains the property values. You set the value of the `persistent-id` attribute of this element equal to the name of the configuration artifact, which is the name of the properties file *minus* the `.properties` extension.

The following sample Spring application context XML file shows everything wired together; only relevant parts of the file are shown:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:osgi-compendium="http://www.springframework.org/schema/osgi-compendium"
  xsi:schemaLocation="http://www.springframework.org/schema/osgi
    http://www.springframework.org/schema/osgi/spring-osgi-1.2.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-2.5.xsd
    http://www.springframework.org/schema/osgi-compendium
    http://www.springframework.org/schema/osgi-compendium/spring-osgi-compendium-1.2.xsd">

  ...

  <bean class="com.springsource.configuration.properties.PropertiesController">
    <constructor-arg value="${driverClassName}"/>
    <constructor-arg value="${url}"/>
    <constructor-arg value="${username}"/>
    <constructor-arg value="${password}"/>
  </bean>

  <context:property-placeholder properties-ref="configAdminProperties"/>

  <osgi-compendium:cm-properties id="configAdminProperties" persistent-id="app-properties"/>

  ...
</beans>
```

The preceding example shows how the id `configAdminProperties` wires the `<context:property-placeholder>` and `<osgi-compendium:cm-properties>` elements together. Based on the value of the `persistent-id` attribute, you must also deploy a properties file called `app-properties.properties` which VWS installs as a configuration artifact.

Adding the Configuration Artifact to a Plan

Although you can always deploy your application and associated configuration artifact using the `pickup` directory, we recommend that you group the two together in a plan, add the two artifacts to the repository, and then deploy the plan using the `pickup` directory. The following sample plan includes the two artifacts:

```
<?xml version="1.0" encoding="UTF-8"?>
<plan name="multi-artifact.plan" version="1.0.0"
  scoped="false" atomic="false"
  xmlns="http://www.springsource.org/schema/dm-server/plan"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://www.springsource.org/schema/dm-server/plan
    http://www.springsource.org/schema/dm-server/plan/springsource-dm-server-plan.xsd">

  <artifact type="configuration" name="app-properties" version="0"/>
  <artifact type="bundle" name="org.eclipse.virgo.configuration.properties" version="1.0.0"/>
</plan>
```

For additional information about plans, see [Creating Plans](#).

4.5 Programmatic Access to Personality-specific Features

Module personalities typically provide automatic access to features specific to the personality via custom manifest headers or other configuration mechanisms. There may be situations, however, for which programmatic access to such features is desirable or necessary. This section describes how to programmatically access personality-specific features from application code in a module.

Programmatic Access to Web Personality Features

Programmatic Access to the `WebApplicationContext`

The Virgo Web Server automatically creates a `WebApplicationContext` for Web Bundles and WAR files. When used in conjunction with an auto-configured Spring MVC `DispatcherServlet`, there is generally no need to access the `WebApplicationContext` programmatically, since all components of the web application are configured within the scope of the `WebApplicationContext` itself. However, if you wish to access the `WebApplicationContext` you can do so via the web application's `ServletContext`. The Web Personality subsystem stores the bundle's `WebApplicationContext` in the `ServletContext` under the attribute name "BSN-ApplicationContext", where BSN is the `Bundle-SymbolicName` of your WAR or Web Bundle.

Alternatively, since the Web Personality subsystem also stores the `WebApplicationContext` under the attribute name with the value of the `WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE` constant, you may choose to use Spring MVC's `WebApplicationContextUtils`' `getWebApplicationContext(servletContext)` or `getRequiredWebApplicationContext(servletContext)` methods to access the `WebApplicationContext` without providing an explicit attribute name.

Programmatic Access to the `BundleContext`

As required by the OSGi Web Container specification, you can access the `BundleContext` of your WAR or Web Bundle via the web application's `ServletContext`. The bundle context is stored in the `ServletContext` under the attribute name `osgi-bundlecontext`.

4.6 Automatic Imports

The Virgo Web Server generates automatic package imports (i.e., via the `Import-Package` manifest header) for various module personalities. This section lists which packages are automatically generated for each personality.

Automatic Imports for the Web Personality

As required by the OSGi Web Container specification all WARs and Web Bundles will automatically import the following packages:

- `javax.servlet;version="2.5"`
- `javax.servlet.http;version="2.5"`
- `javax.servlet.jsp;version="2.1"`
- `javax.servlet.jsp.el;version="2.1"`
- `javax.servlet.jsp.tagext;version="2.1"`
- `javax.el;version="1.0"`

In addition to the above-described imports, VWS will also generate automatic imports for all of the packages that are exported by the system bundle, unless an import for the package already exists in the artifact's manifest, or the artifact contains the package, i.e. within `WEB-INF/classes`, or in a jar file in `WEB-INF/lib`. When an import is generated, it is versioned such that it exactly matches the version or versions of the package that are exported from the system bundle. For example, a package that's exported only at version `1.0.0` will generate an import with a version of `[1.0.0, 1.0.0]`, and a package that's exported at version `1.0.0` and version `2.0.0` will generate an import with a version of `[1.0.0, 2.0.0]`.



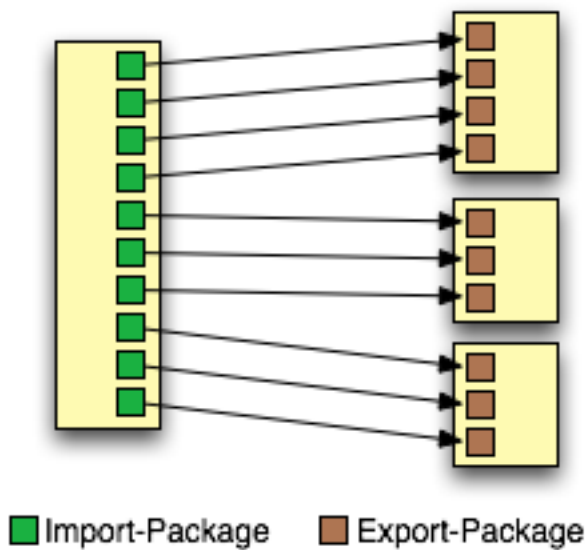
System Bundle Package Exports

For further details on which packages are exported by the OSGi system bundle, consult the `java6-server.profile` file located in the `SERVER_HOME/lib` directory.

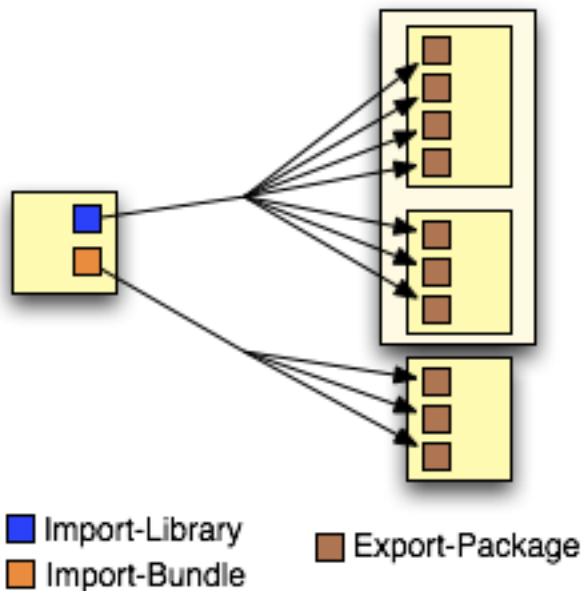
4.7 Working with dependencies

Complex enterprise frameworks such as Spring and Hibernate are typically divided into many, many different packages. Traditionally, if an OSGi bundle wished to make extensive use of such a framework its manifest would have to import a huge number of different packages. This can be an error-prone and tedious process. Furthermore, application developers are used to thinking in terms of their application using a framework, such as Spring, as a whole, rather than a long list of all the different packages that comprise the framework.

The following figure provides a simple illustration of the complexity of only using `Import-Package`:



The Virgo Web Server reduces the need for long lists of imported packages by introducing two new manifest headers; `Import-Bundle` and `Import-Library`. The following figure provides an illustration of the simplification that these new headers offer:



As you can see, use of `Import-Bundle` and `Import-Library` can lead to a dramatic reduction in the number of imports that you need to include in an application bundle's manifest. Furthermore, `Import-Bundle` and `Import-Library` are simply aliases for `Import-Package`; at deployment time `Import-Bundle` and `Import-Library` header entries are automatically expanded into numerous `Import-Package` entries. This means that you retain the exact same semantics of using `Import-Package`, without having to go through the labourious process of doing so.

Importing libraries

A bundle in an application can declare a dependency on a library by using the Virgo Web Server-specific `Import-Library` header. This header specifies a comma-separated list of library symbolic names and version ranges that determine which libraries are imported. By default a dependency on a library is mandatory but this can be controlled through use of the resolution directive in exactly the same way as it can with `Import-Package`.

```
Import-Library: org.springframework.spring;version="[2.5.4, 3.0)",  
org.aspectj;version="[1.6.0,1.6.0]";resolution="optional"
```

This example `Import-Library` header declares a mandatory dependency on the Spring library at a version from 2.5.4 inclusive to 3.0 exclusive. It also declares an optional dependency on the AspectJ library at exactly 1.6.0.

Importing bundles

A bundle in an application can declare a dependency on a bundle by using the Virgo Web Server-specific `Import-Bundle` header. The header specifies a comma-separated list of bundle symbolic names, version ranges, and scope declarations that determine which bundles are imported and the scope of their dependency. By default a dependency on a bundle is mandatory but this can be controlled through use of the resolution directive in exactly the same way as it can with `Import-Package`.

```
Import-Bundle: com.springsource.org.apache.commons.dbcp;version="[1.2.2.osgi, 1.2.2.osgi]"
```

This example `Import-Bundle` header declares a mandatory dependency on the Apache Commons DBCP bundle at exactly 1.2.2.osgi.

Scoping Bundles in an Application

When working with a scoped application, such as a PAR file or a plan, you might run into a situation where one of the bundles in the application (call it `bundleA`) depends on another bundle (`bundleB`) that performs a runtime task (such as class generation) that a third bundle (`bundleC`) might need to know about, although `bundleC` does not explicitly depend on `bundleB`.

For example, Hibernate uses CGLIB (code generation library) at runtime to generate proxies for persistent classes. Assume that a domain bundle in your application uses Hibernate for its persistent objects, and thus its `Import-Bundle` manifest header includes the Hibernate bundle. Further assume that a separate Web bundle uses reflection in its data-binding code, and thus needs to reflect on the persistent classes generated by Hibernate at runtime. The Web bundle now has an indirect dependency on the Hibernate bundle because of these dynamically generated classes, although the Web bundle does not typically care about the details of how these classes

are persisted. One way to solve this dependency problem is to explicitly add the Hibernate bundle to the `Import-Bundle` header of the Web bundle; however, this type of explicit-specified dependency breaks the modularity of the application and is not a programming best practice.

A better way to solve this problem is to specify that Virgo Web Server itself dynamically import the bundle (Hibernate in the example above) to all bundles in the application at runtime. You do this by adding the `import-scope:=application` directive to the `Import-Bundle` header of the bundle that has the direct dependency (the domain bundle in our example). At runtime, although the Web bundle does not explicitly import the Hibernate bundle, Virgo Web Server implicitly imports it and thus its classes are available to the Web bundle. This mechanism allows you to declare the dependencies you need to make your application run, without having to make changes to your application that might limit its flexibility.

The following example shows how to use the `import-scope` directive with the `Import-Bundle` header:

```
Import-Bundle: com.springsource.org.hibernate;version="[3.2.6.ga,3.2.6.ga]";import-scope:=application
```

You can also set the `import-scope` directive to the (default) value `bundle`; in this case, the scope of the bundle is just the bundle itself and thus Virgo Web Server does not perform any implicit importing into other bundles of the application.

Note that use of the `import-scope:=application` directive of the `Import-Bundle` header only makes sense when the bundle is part of a scoped application (PAR or plan); if the bundle is not part of a scoped application, then this directive has no effect.

Finally, because `import-scope:=application` implicitly adds a bundle import to each bundle of the PAR or plan, the impact of subsequently refreshing the imported bundle is, in general, broader than it would have been if you had not used `import-scope:=application`. This may well affect the performance of refresh.

Defining libraries

Libraries are defined in a simple text file, typically with a `.libd` suffix. This file identifies the library and lists all of its constituent bundles. For example, the following is the library definition for Spring 2.5.4:

```
Library-SymbolicName: org.springframework.spring
Library-Version: 2.5.4
Library-Name: Spring Framework
Import-Bundle: org.springframework.core;version="[2.5.4,2.5.5)",
org.springframework.beans;version="[2.5.4,2.5.5)",
org.springframework.context;version="[2.5.4,2.5.5)",
org.springframework.aop;version="[2.5.4,2.5.5)",
org.springframework.web;version="[2.5.4,2.5.5)",
org.springframework.web.servlet;version="[2.5.4,2.5.5)",
org.springframework.jdbc;version="[2.5.4,2.5.5)",
org.springframework.orm;version="[2.5.4,2.5.5)",
org.springframework.transaction;version="[2.5.4,2.5.5)",
org.springframework.context.support;version="[2.5.4,2.5.5)",
org.springframework.aspects;version="[2.5.4,2.5.5)",
com.springsource.org.aopalliance;version="1.0"
```

The following table lists all of the headers that may be used in a library definition:

Table 4.4. Library definition headers

Header	Description
Library-SymbolicName	Identifier for the library
Library-Version	Version number for the library
Import-Bundle	A comma separated list of bundle symbolic names. Each entry may optionally specify a version (using the <code>version=</code> directive) and the scope of the import (using the <code>import-scope</code> directive).
Library-Name	Optional. The human-readable name of the library
Library-Description	Optional. A human-readable description of the library

Installing dependencies

Rather than encouraging the packaging of all an application's dependencies within the application itself, Virgo Web Server uses a local provisioning repository of bundles and libraries upon which an application can depend. When the Virgo Web Server encounters an application with a particular dependency, it will automatically provide, from its provisioning repository, the appropriate bundle or library.

Making a dependency available for provisioning is simply a matter of copying it to the appropriate location in the VWS's local provisioning repository. By default this is `SERVER_HOME/repository/bundles/usr` for bundles, and `SERVER_HOME/repository/libraries/usr` for libraries. A more detailed discussion of the provisioning repository can be found in the [User Guide](#).

4.8 Application trace

As described in the [User Guide](#) Virgo Web Server provides support for per-application trace. Virgo Web Server provides SLF4J with Logback logging for Event Logging and Tracing. Application trace is configured in the `serviceability.xml` file. See the [User Guide](#) for more details.

4.9 Application versioning

In much the same way that individual OSGi bundles can be versioned, Virgo Web Server allows applications to be versioned. How exactly you do this depends on how you have packaged the application:

- If you package your application using a PAR, you version the application by using the `Application-Version` header in the `MANIFEST.MF` file of the PAR file.

- If you use a plan to describe the artifacts that make up your application, you version it by using the `version` attribute of the `<plan>` root element of the plan's XML file.
- If your application consists of a single bundle, you version it in the standard OSGi way: by using the `Bundle-Version` header of the `MANIFEST.MF` file of the bundle.

Virgo Web Server uses an application's version to prevent clashes when multiple versions of the same application are deployed at the same time. For example, the application trace support described in Section 4.8, "Application trace", includes the application's name and version in the file path. This ensures that each version of the same application has its own trace or logging file.

5. Migrating to OSGi

Taking on a new technology such as OSGi may seem a bit daunting at first, but a proven set of migration steps can help ease the journey. Teams wishing to migrate existing applications to run on the Virgo Web Server will find that their applications typically fall into one of the following categories.

- **Web Application:** for web applications, this chapter provides an overview of the steps required to migrate from a Standard WAR to a Shared Services WAR. Furthermore, the following chapter provides a detailed case study involving the migration of the Spring 2.0 Form Tags show case application.
- **Anything else:** for any other type of application, you will typically either deploy your application as multiple individual bundles, as a single PAR file, or as a plan, which is the recommended approach for deploying applications on the Virgo Web Server. See Section 5.2, “Migrating to a Plan or a PAR” for details.

5.1 Migrating Web Applications

Many applications may start with the standard WAR format for web applications and gradually migrate to a more OSGi-oriented architecture. Since the Virgo Web Server offers several benefits to all supported deployment formats, it provides a smooth migration path. Of course, depending on your application’s complexity and your experience with OSGi, you may choose to start immediately with an OSGi-based architecture.

Standard WAR

If you are not yet familiar with OSGi or simply want to deploy an existing web application on the Virgo Web Server, you can deploy a standard WAR and leverage the VWS with a minimal learning curve. In fact reading the [Virgo Web Server User Guide](#) is pretty much all that you need to do to get started. Furthermore, you will gain familiarity with the Virgo Web Server, while preparing to take advantage of the other formats.

Shared Libraries WAR

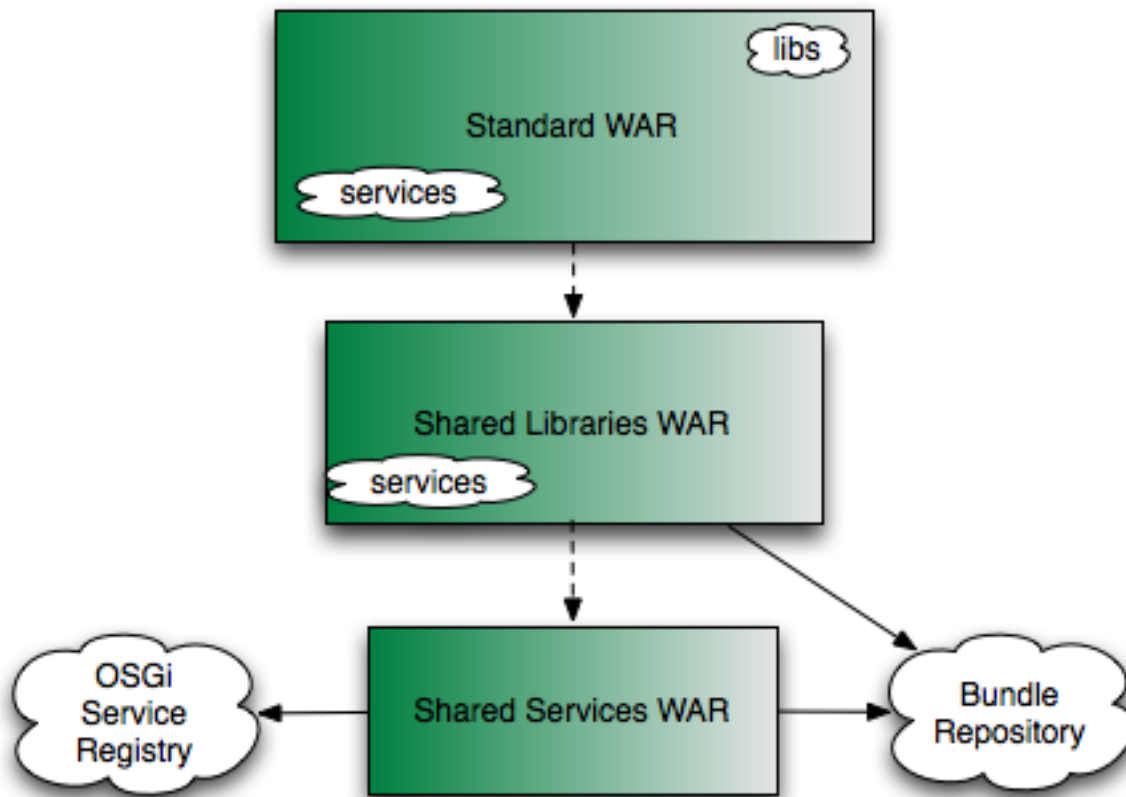
The *Shared Libraries WAR* format is the first step to reaping the benefits of OSGi. In this phase, you dip your toes into OSGi-based dependency management by removing JAR files from the WAR and declaring dependencies on corresponding OSGi bundles.

Shared Services WAR

In this phase, you take the next step toward a fully OSGi-based architecture by separating your web artifacts (e.g., Servlets, Controllers, etc.) from the services they depend on.

Web Migration Summary

The following diagram graphically depicts the migration path from a Standard WAR to a Shared Services WAR. As you can see, the libraries (*libs*) move from within the deployment artifact to the Bundle Repository. Similarly, the services move from within the WAR to external bundles and are accessed via the OSGi Service Registry. In addition, the overall footprint of the deployment artifact decreases as you move towards a Shared Services WAR.



5.2 Migrating to a Plan or a PAR

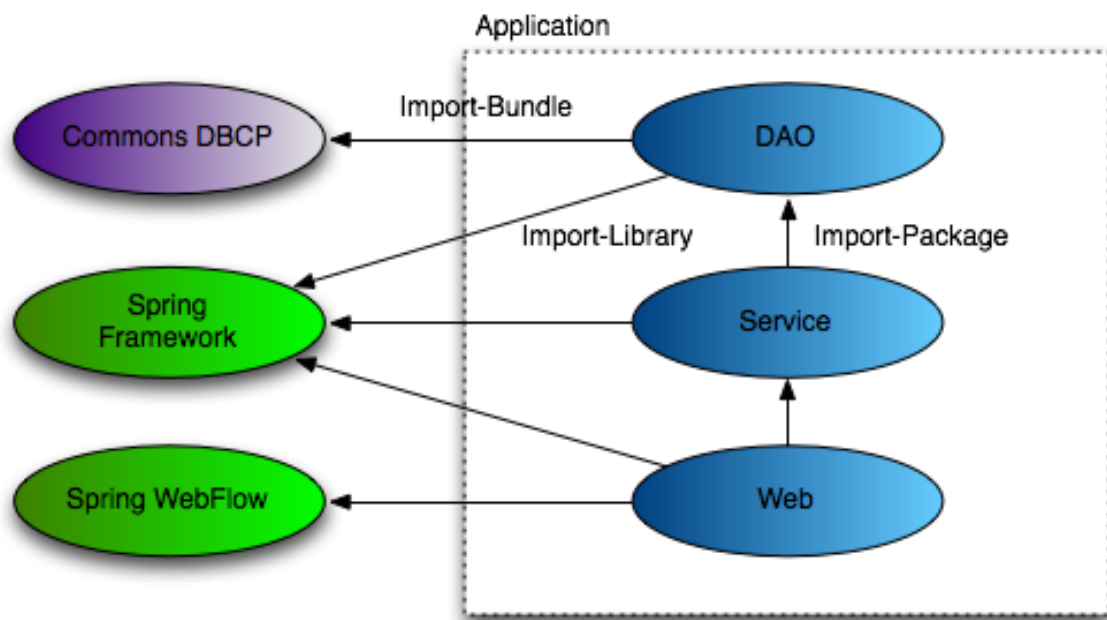
The first steps to migrating an existing application to a plan or a PAR are the same: deciding on the bundles that make up the application and ensuring that their `Import-Package`, `Import-Library`, and `Import-Bundle` manifest headers are correct. Once you have the list of bundles that make up your application, you then decide whether you want to JAR them all into a single application file (PAR) or create a plan that simply lists the bundles by reference. Creating a plan is the recommend way to create an application, although PARs also have benefits that might suit your needs better, as described in the section called “Plan or PAR?”.

Creating the Application Bundles

When migrating an existing application to the PAR packaging and deployment format or a plan, you consider modularity as the prime objective. Following the ideas discussed in Section 3.3, “A guide to forming bundles”, you refactor the application into multiple bundles. You may start conservatively with a small number of bundles and then further refactor those bundles.

If the original code is crafted following good software practices such as separation of concerns and use of well-defined interfaces, migration may involve modifying only configuration and packaging. In other words, your Java sources will remain unchanged. Even configuration is likely to change only slightly.

For example, the following diagram depicts a typical web application that has been refactored and packaged as a PAR. The blue elements within the *Application* box constitute the bundles of the application. Each of these bundles imports types from other bundles within the PAR using `Import-Package`. The green elements in the left column represent *libraries* installed on the VWS. The PAR’s bundles reference these libraries using `Import-Library`. The purple element in the left column represents a bundle within the VWS’s bundle repository which is imported by the DAO bundle using `Import-Bundle`. In contrast to a traditional, monolithic WAR deployment, the PAR format provides both a logical and physical application boundary and simultaneously allows the application to benefit from both the OSGi container and the Virgo Web Server.



Plan or PAR?

Once you have refactored your existing application into separate OSGi bundles, you then must decide whether to package the bundles into a single PAR file or create a plan that lists the bundles by reference. As described in more detail in preceding sections of this guides, PARs and

plans have similar benefits, such as:

- Scoping
- Atomicity, or the ability to deploy and control the bundles as a single unit
- Versioning
- Improved serviceability

Plans, the method most recommended by us to create your application, has the following added benefits:

- Guaranteed order of deployment, based on the order in which they are listed in the plan's XML file
- Ease of sharing content between plans and updating individual plans without having to physically repack, due to the artifacts being listed by reference.
- Ability to disable scoping and atomicity, if desired.

The main benefit of PARS is that, because they physically contain all the required artifacts, you know exactly what bundles are deployed when you deploy the PAR file, in contrast to plans that allow content to be substituted or lost.

For details about creating plans and PARs, see Section 4.3, “Creating Plans” and Section 4.2, “Creating PARs and WARs”, respectively.

6. Case study: Migrating the Form Tags sample application

In this chapter we will walk through the steps needed to migrate the Form Tags sample application from a standard Java EE WAR to a fully OSGi compliant *Shared Services WAR* within a PAR. The migration involves four packaging and deployment formats:

1. [Standard WAR](#)
2. [Shared Libraries WAR](#)
3. [Shared Services WAR](#)
4. [PAR with a shared services WAR](#)

Each of these migration steps will produce a web application that can be deployed and run on the VWS.

After summarising the process, an example `plan` is shown which is another way of packaging and deploying the application.

The following image displays the directory structure you should have after installing the Form Tags sample. Note however that the release tag will typically resemble `2.0.0.RELEASE`.

Name
▶ about_files
▶ About.html
▼ dist
▶ formtags-par-2.0.1.BUILD-20100608135114.par
▶ formtags-shared-libs-2.0.1.BUILD-20100608135114.war
▶ formtags-shared-services-service-2.0.1.BUILD-20100608135114.jar
▶ formtags-shared-services-war-2.0.1.BUILD-20100608135114.war
▶ formtags-war-2.0.1.BUILD-20100608135114.war
▶ epl-v10.html
▶ notice.html
▼ projects
▶ build-formtags
▶ build.properties
▶ build.versions
▼ par
▶ org.springframework.showcase.formtags
▶ org.springframework.showcase.formtags.domain
▶ org.springframework.showcase.formtags.service
▶ org.springframework.showcase.formtags.web
▶ README.TXT
▼ shared-libs
▶ formtags-shared-libs
▶ README.TXT
▼ shared-services
▶ build-shared-services
▶ formtags-shared-services-service
▶ formtags-shared-services-war
▶ README.TXT
▶ virgo-build
▼ war
▶ formtags-war
▶ README.TXT
▶ README.TXT

The `dist` directory contains the distributables, and the `projects` directory contains the source code and build scripts.

For simplicity, this chapter will focus on the distributables—which are built using Spring-Build—rather than on configuring a project in an IDE.



Tip

Pre-packaged distributables are made available in the `dist` directory; however, if you would like to modify the samples or build them from scratch, you may do so using Spring-Build. Take a look at the `README.TXT` file in each of the folders under the `projects` directory in the Form Tags sample for instructions.

6.1 Overview of the Form Tags Sample Application

The sample that we will be using is the Form Tags show case sample which was provided with Spring 2.0. The Form Tags application has been removed from the official Spring 2.5.x distributions; however, since it is relatively simple but still contains enough ingredients to demonstrate the various considerations required during a migration, we have chosen to use it for these examples.

The purpose of the Form Tags show case sample was to demonstrate how the Spring specific `form: tags`, released in Spring 2.0, make view development with JSPs and tag libraries easier. The Form Tags application consists of a single `UserService` which returns a list of `Users`. Furthermore, the application demonstrates how to list, view, and edit `Users` in a simple Spring MVC based web application using JSP and JSTL.

6.2 Form Tags WAR

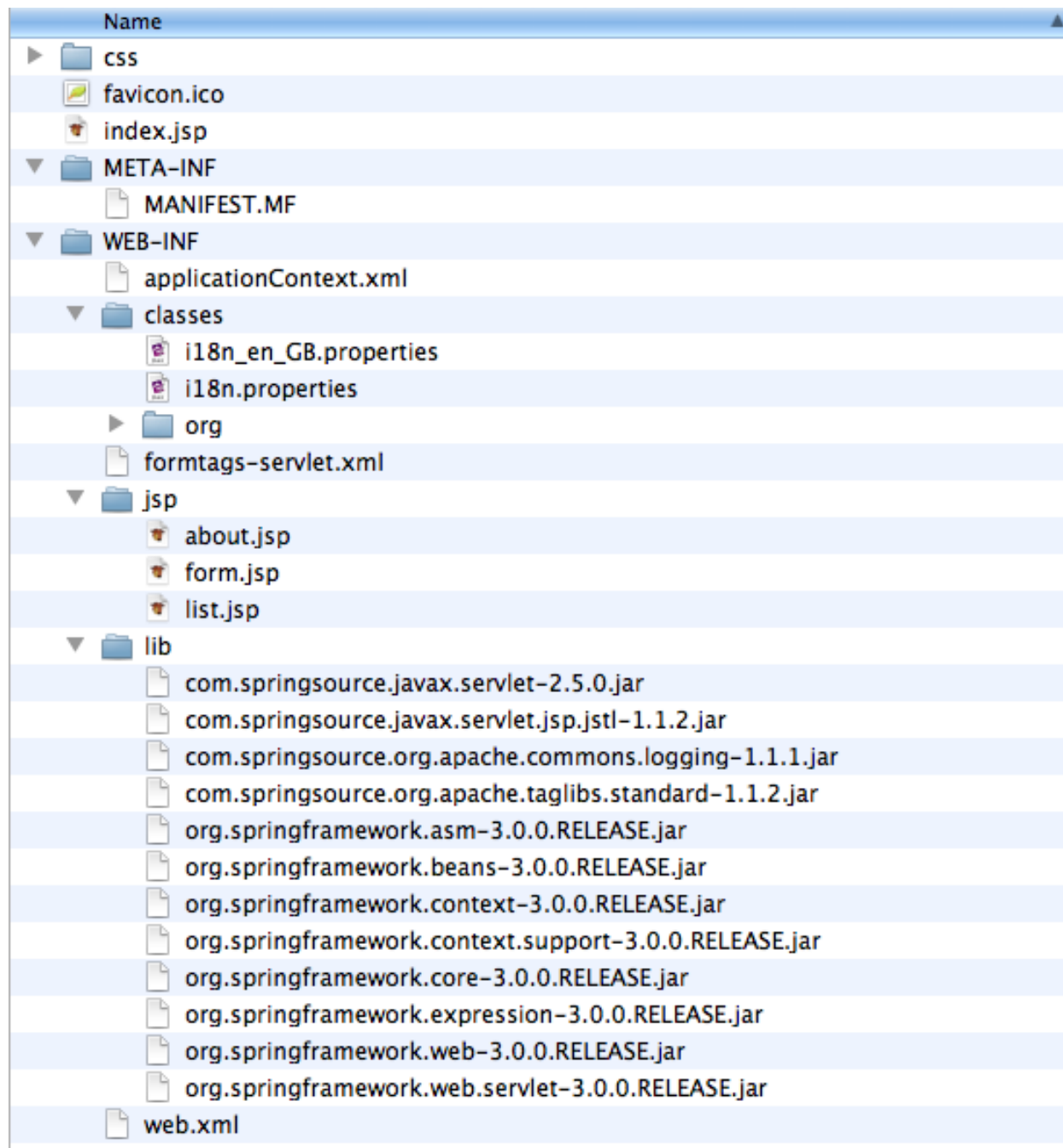
We begin with a standard WAR deployment.



Note

The Virgo Web Server supports the standard Java EE WAR packaging and deployment format as a first-class citizen, and there are many benefits to deploying a standard WAR file on the VWS including, but not limited to: tooling support, runtime error diagnostics, FFDC (first failure data capture), etc. In addition, support for standard WAR deployment provides an easy on-ramp for trying out the Virgo Web Server with existing web applications.

The following screen shot displays the directory structure of the Form Tags application using the standard WAR format. As you can see, there is no deviation from the standard structure and layout, and as you would expect, all of the web application's third-party dependencies (for example: Spring, Commons Logging) are packaged as JARs in `WEB-INF/lib`.



To deploy this application, simply copy `dist/formtags-war-2.0.0.*.war` to the `SERVER_HOME/pickup` directory for hot deployment.

You should then see the VWS produce console output similar to the following:



Note

The console output has been reformatted to fit this document.


```
[2009-07-01 14:54:45.135] fs-watcher
<SPDE0048I> Processing 'CREATED' event for file 'formtags-war-2.0.0.RELEASE.war'.
[2009-07-01 14:54:45.797] fs-watcher
<SPDE0010I> Deployment of 'formtags-war-2.0.0.RELEASE.war' version '0' completed.
[2009-07-01 14:54:45.797] Thread-20
<SPWE0000I> Starting web bundle '/formtags-war-2.0.0.RELEASE'.
[2009-07-01 14:54:46.380] Thread-20
<SPWE0001I> Started web bundle '/formtags-war-2.0.0.RELEASE'.
```

Navigate to `http://localhost:8080/` plus the web application context path, which in the above case is `formtags-war-2.0.0.RELEASE`. Thus navigating to `http://localhost:8080/formtags-war-2.0.0.RELEASE` should render the sample application's welcome page, as displayed in the screen shot below.



Tip

For WARs, the default web context path is the name of the WAR file without the `.war` extension. You can optionally specify a context path using the `Web-ContextPath` bundle manifest header, which will be described in further detail later.



6.3 Form Tags Shared Libraries WAR

As mentioned above, a standard WAR file typically packages all its required dependencies in `WEB-INF/lib`. The servlet container will then add all of the JARs in `WEB-INF/lib` to the application's classpath.

The first step of the migration towards benefiting from an OSGi container is to retrieve the dependencies from the VWS's bundle repository at runtime. This can significantly reduce the time it takes to build and deploy the application. It also enables the enforcement of policies regarding the use of third-party libraries.

The way in which dependencies are declared in an OSGi environment is via manifest headers in a bundle's `/META-INF/MANIFEST.MF`. As mentioned in Chapter 4, *Developing Applications*, there are three ways of expressing dependencies: `Import-Package`, `Import-Bundle` and

Import-Library.

The Form Tags application uses JSTL standard tag libraries. Thus, you need to choose a JSTL provider, for example the Apache implementation which comes with the VWS. To use the Apache implementation of JSTL, you need to express your dependency as outlined in the following manifest listing. Because it is a single bundle, Import-Bundle is the simplest and therefore preferred manifest header to use.

The Form Tags application requires commons-logging and Spring. It would be very painful to have to list all the Spring packages one by one. Equally, considering the number of bundles that make up the Spring framework, it would be verbose to list each bundle. Therefore Import-Library is the preferred approach for expressing the dependency on the Spring framework.



Tip

How do you determine the name of a library definition provided by the Virgo Web Server? Use the [SpringSource Enterprise Bundle Repository](#).

Examine the /META-INF/MANIFEST.MF in /dist/formtags-shared-libs-*.war:

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.0
Created-By: 1.5.0_13-119 (Apple Inc.)
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.springframework.showcase.formtags-shared-libs
Import-Library: org.springframework.spring;version="[2.5.4,3.1.0)"
Import-Bundle: com.springsource.org.apache.taglibs.standard;version="1
.1.2"
```

You can see the Import-Library and Import-Bundle directives that instruct the VWS to add the appropriate package imports to the bundle classpath used by this WAR file.

Deploying the shared libraries WAR onto the VWS should result in console output similar to the following:



Note

The console output has been reformatted to fit this document.

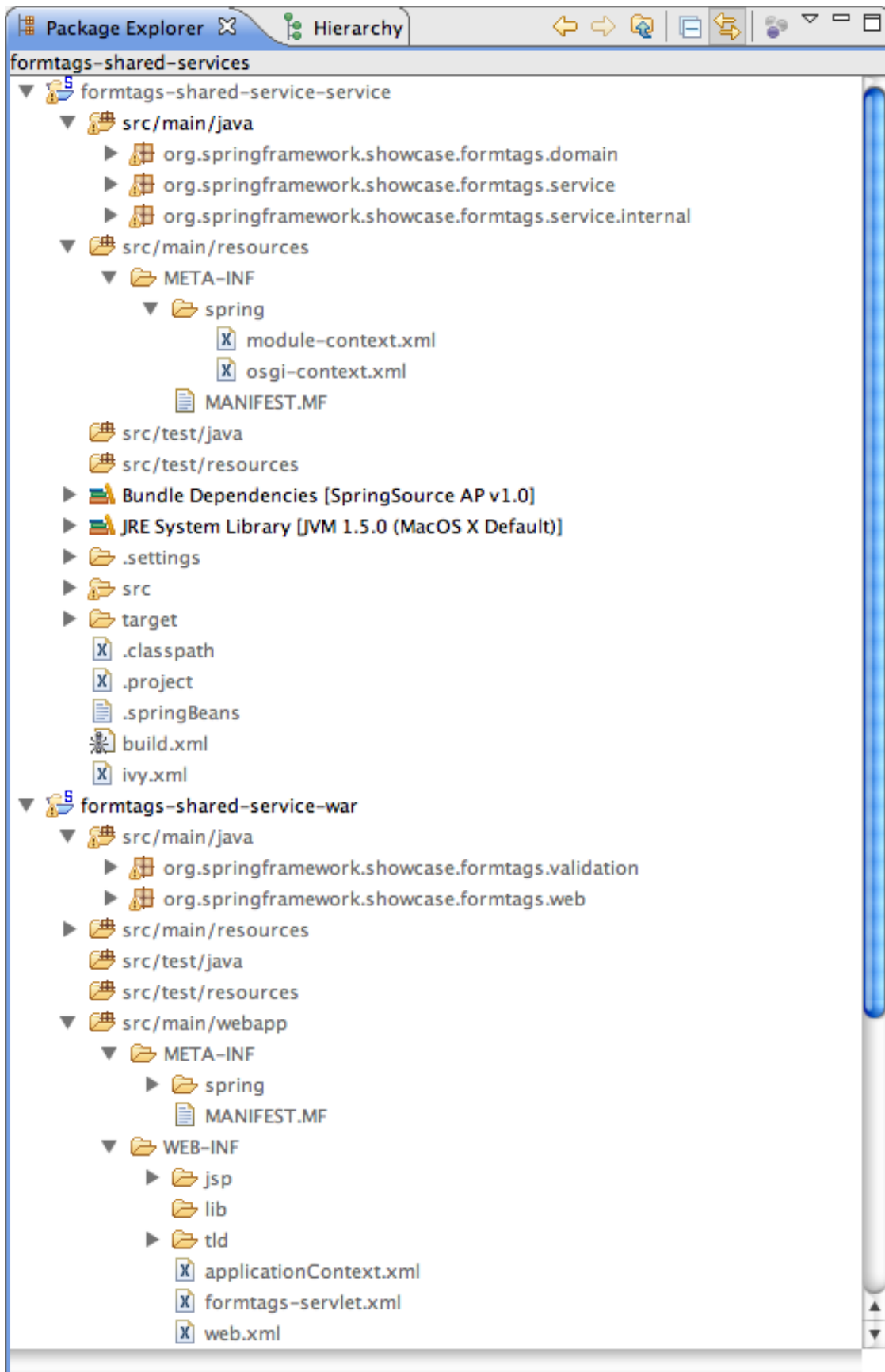
```
[2009-07-01 15:00:14.953] fs-watcher
<SPDE0048I> Processing 'CREATED' event for file 'formtags-shared-libs-2.0.0.RELEASE.war'.
[2009-07-01 15:00:15.363] fs-watcher
<SPDE0010I> Deployment of 'org.springframework.showcase.formtags_shared_libs' version '2' completed.
[2009-07-01 15:00:15.364] Thread-20
<SPWE0000I> Starting web bundle '/formtags-shared-libs-2.0.0.RELEASE'.
[2009-07-01 15:00:15.816] Thread-20
<SPWE0001I> Started web bundle '/formtags-shared-libs-2.0.0.RELEASE'.
```

Navigating to `http://localhost:8080/formtags-shared-libs-BUILDTAG` should render the welcome page. Note that for the pre-packaged distributable, the BUILDTAG should be similar to `2.0.0.RELEASE`; whereas, for a local build the `-BUILDTAG` may be completely omitted. Please consult the console output, web-based admin console, or log to determine the exact context path under which the web application has been deployed.

6.4 Form Tags Shared Services WAR

The next step in the migration is to deploy the services as a separate OSGi bundle which the WAR then references. The Form Tags sample has a single service `UserManager`.

This scenario has two separate deployables, the `service` bundle and the WAR file. The following image shows the two separate source trees:



**Note**

Note that the WAR does not contain the `.domain` or `.service` packages as these will be imported from the separate service bundle.

The Service Bundle

The responsibility of the first bundle (`formtags-shared-services-service`) is to provide the API of the `formtags` service. This includes both the domain and the service API. In the same way that imports are defined in the `/META-INF/MANIFEST.MF`, so are exports. The following is the `/META-INF/MANIFEST.MF` listing from the service bundle.

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.0
Created-By: 1.5.0_13-119 (Apple Inc.)
Bundle-ManifestVersion: 2
Bundle-Name: FormTags Service (and implementation)
Bundle-SymbolicName: org.springframework.showcase.formtags.service-shared-services
Export-Package: org.springframework.showcase.formtags.service,org.springframework.showcase.formtags.domain
Import-Library: org.springframework.spring;version="[2.5.4,3.1.0)"
```

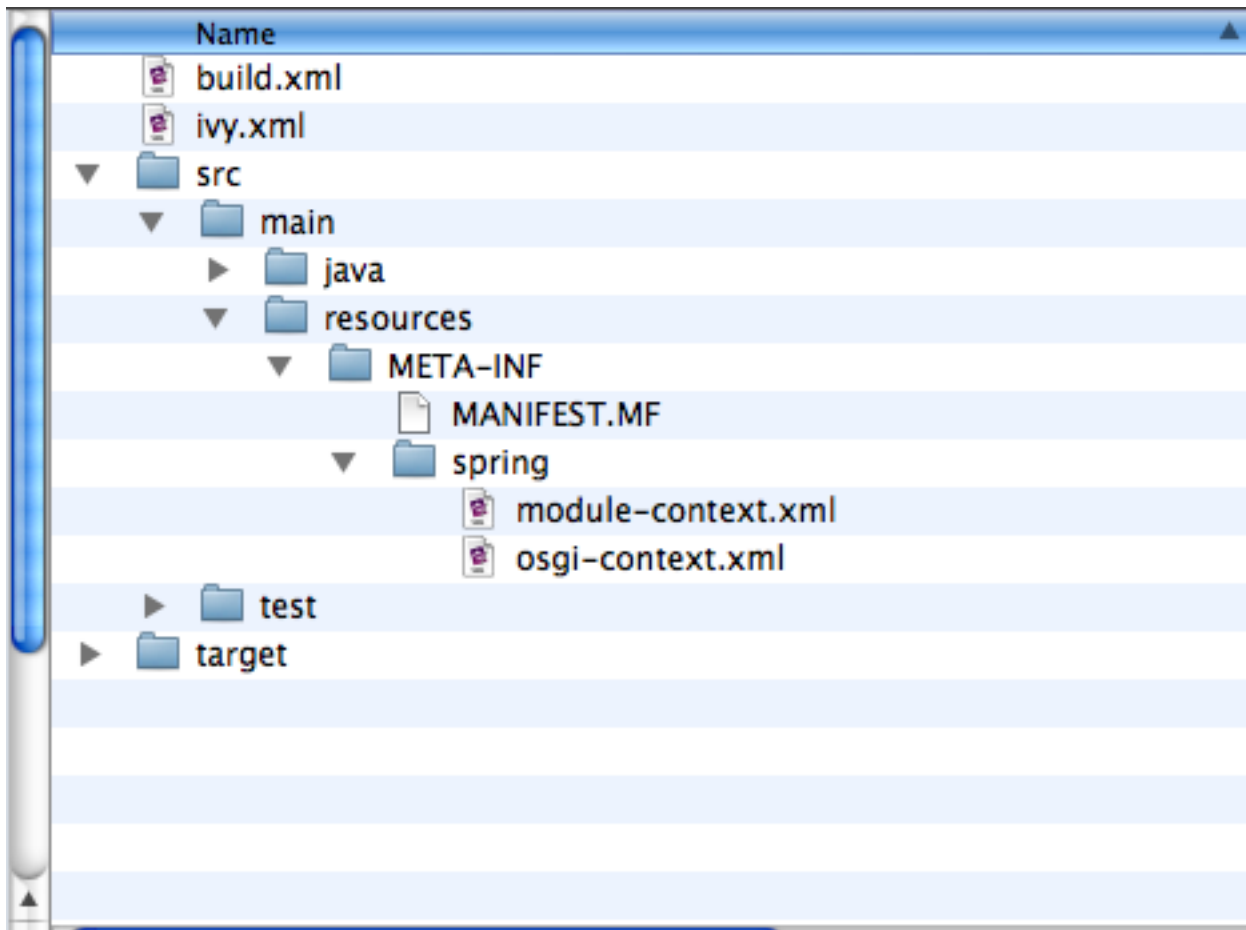
The symbolic name of this bundle is `org.springframework.showcase.formtags.service-shared-services`. Note that the name of the bundle typically describes the package that the bundle primarily exports. If you take a look at the `repository/bundles/ext` in the VWS directory, you'll see that names are almost always indicative of the contents of the bundle. For this example, however, we have also appended `"-shared-services"` in order to avoid possible clashes with other bundle symbolic names. You will see later that the PAR also contains a service bundle.

**Note**

In OSGi, the combination of `Bundle-SymbolicName` and `Bundle-Version` is used to uniquely identify a bundle within the OSGi container. Furthermore, when you deploy a bundle to the Virgo Web Server, for example via the `pickup` directory, a bundle's filename is also used to uniquely identify it for the purpose of supporting *hot deployment* via the file system.

As well as exporting types (i.e. the domain classes and service API), the service bundle also publishes an implementation of the `UserManager`. The actual implementation is `StubUserManager`; however, that should remain an implementation detail of this bundle.

The fact that this bundle publishes a service is not captured in the `/META-INF/MANIFEST.MF`, as it is a Spring-DM concept. The following image is of `src/main/resources/spring`.



As you can see there are two Spring configuration files: `module-context.xml` and `osgi-context.xml`.



Tip

These names are arbitrary; however, they follow an informal convention: `module-context.xml` typically bootstraps the Spring context (usually delegating to smaller fine grained context files inside another directory), whilst `osgi-context.xml` contains all the OSGi service exports and references.

The following is a listing of `module-context.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <bean id="userManager"
        class="org.springframework.showcase.formtags.service.internal.StubUserManager"/>

</beans>
```

As you can see, this simply defines a bean called `userManager`. The following is a listing of `osgi-context.xml`.

```
<?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: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/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

  <service ref="userManager"
    interface="org.springframework.showcase.formtags.service.UserManager"/>

</beans:beans>
```

This single bean definition exports the userManager defined in module-context.xml to the OSGi service registry and makes it available under the public `org.springframework.showcase.formtags.service.UserManager` API.

The service bundle should now be ready to deploy on the VWS. So copy `/dist/formtags-shared-services-services*` to the `SERVER_HOME/pickup` directory. Output similar to the following should appear in the VWS's console:



Note

The console output has been reformatted to fit this document.

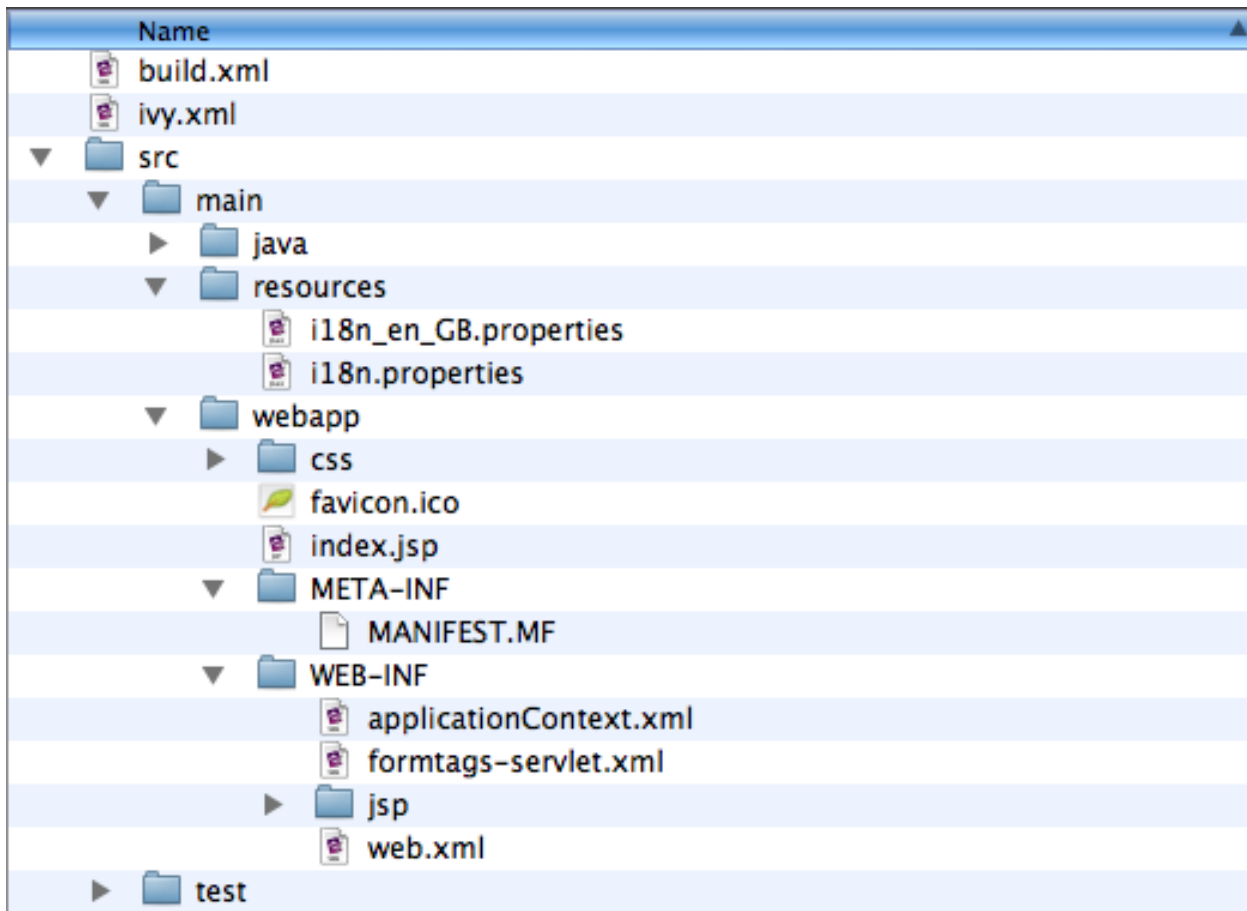
```
[2009-07-01 15:05:03.511] fs-watcher
<SPDE0048I> Processing 'CREATED' event for file 'formtags-shared-services-service-2.0.0.RELEASE.jar'.
[2009-07-01 15:05:03.688] fs-watcher
<SPDE0010I> Deployment of 'org.springframework.showcase.formtags.service_shared_services' version '2.0.0.RELEASE' co
```

Accessing the Service and Types from the WAR

The WAR file now needs to access the types and service exported by the service bundle. The following listing is the WAR's `/META-INF/MANIFEST.MF` which imports the types exported by the service bundle. The `Import-Bundle` statement has also been extended to import `org.springframework.osgi.core`, which is necessary in order to load an OSGi-enabled `WebApplicationContext`.

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.0
Created-By: 1.5.0_13-119 (Apple Inc.)
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.springframework.showcase.formtags.web-shared-
services
Import-Package: org.springframework.showcase.formtags.domain,org.sprin
gframework.showcase.formtags.service, org.eclipse.virgo.server.web.dm;version="[1.0,2.1)"
Import-Library: org.springframework.spring;version="[2.5.4,3.1.0)"
Import-Bundle: com.springsource.org.apache.taglibs.standard;version="1
.1.2",org.springframework.osgi.core
```

In addition to importing the exported types of the service bundle, the WAR must also obtain a reference to the UserManager published by the service bundle. The following image shows the directory structure of the Shared Services WAR.



As you can see in the above image, the Form Tags Shared Services WAR's /WEB-INF/web.xml directory contains a standard web.xml deployment descriptor, applicationContext.xml which defines the configuration for the *root* WebApplicationContext, and formtags-servlet.xml which defines the configuration specific to the configured *formtags* DispatcherServlet.

As is typical for Spring MVC based web applications, you configure a ContextLoaderListener in web.xml to load your root WebApplicationContext; however, to enable your WebApplicationContext to be able to reference services from the OSGi Service Registry, you must explicitly set the contextClass Servlet context parameter to the fully qualified class name of a ConfigurableWebApplicationContext which is OSGi-enabled. When deploying Shared Services WARs to the Virgo Web Server, you should use

org.eclipse.virgo.server.web.dm.ServerOsgiBundleXmlWebApplicationContext. This will then enable the use of Spring-DM's <reference ... /> within your root WebApplicationContext (i.e., in applicationContext.xml). The following listing is an excerpt from /WEB-INF/web.xml.

```
<context-param>
  <param-name>contextClass</param-name>
  <param-value>org.eclipse.virgo.server.web.dm.ServerOsgiBundleXmlWebApplicationContext</param-value>
</context-param>

<listener>
```



```
<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

The Form Tags Shared Services WAR contains a `/WEB-INF/applicationContext.xml` file which is the default configuration location used to create the *root* `WebApplicationContext` for Spring MVC's `ContextLoaderListener`.



Note

As already mentioned, in the OSGi world, bundle configuration takes place in the root `/META-INF/` directory. Typically Spring-DM powered configuration files will live there as well (e.g., in `/META-INF/spring/*.xml`). In a WAR, however, the root `WebApplicationContext` loaded by `ContextLoaderListener` and the `DispatcherServlet`'s application context typically live in `/WEB-INF/`.

The following is the listing of the WAR's `/WEB-INF/applicationContext.xml`.

```
<?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: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/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

  <reference id="userManager"
    interface="org.springframework.showcase.formtags.service.UserManager"/>

</beans:beans>
```

The single bean declaration is retrieving a service that implements the `org.springframework.showcase.formtags.service.UserManager` API from the OSGi Service Registry.



Tip

You might have been expecting a reference to the service bundle, but that isn't how OSGi works. OSGi provides a service registry, and this bean definition is accessing a service in that registry that meets the specified restriction (i.e. implements the specified interface). This leads to a very loosely coupled programming model: the WAR really doesn't care where the implementation comes from.



Tip

What happens if there is no service at runtime? What if there are multiple services that match the criteria? Spring-DM provides a lot of configuration options, including whether or not the reference is *mandatory*, how long to wait for a service reference, etc. Please consult the [Spring Dynamic Modules for OSGi](#) home page for further information.

One of the benefits of programming to interfaces is that you are decoupled from the actual implementation; Spring-DM provides a proxy. This has enormous benefits including the ability to dynamically refresh individual bundles without cascading that refresh to unrelated bundles.

To deploy the WAR, copy `/dist/formtags-shared-services-war*` to the

SERVER_HOME/pickup directory. You should then see console output similar to the following:



Note

The console output has been reformatted to fit this document.

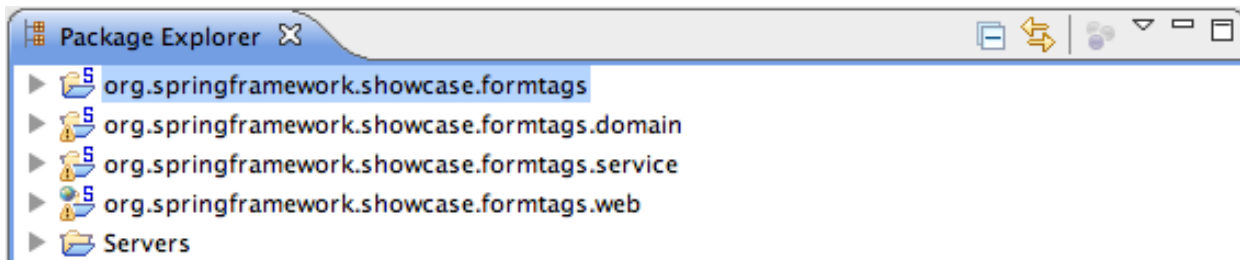
```
[2009-07-01 15:09:19.819] fs-watcher
<SPDE0048I> Processing 'CREATED' event for file 'formtags-shared-services-war-2.0.0.RELEASE.war'.
[2009-07-01 15:09:20.167] fs-watcher
<SPDE0010I> Deployment of 'org.springframework.showcase.formtags.web_shared_services' version '2' completed.
[2009-07-01 15:09:20.168] Thread-20
<SPWE0000I> Starting web bundle '/formtags-shared-services-war-2.0.0.RELEASE'.
[2009-07-01 15:09:20.647] Thread-20
<SPWE0001I> Started web bundle '/formtags-shared-services-war-2.0.0.RELEASE'.
```

Navigating to the appropriate link should render the welcome page.

6.5 Form Tags PAR

The final step in the migration is that of a full blown OSGi application with web support. The Virgo Web Server introduces a new packaging and deployment format: the PAR. A PAR is a standard JAR with a ".par" file extension which contains all of the modules of your application (e.g., service, domain, and infrastructure bundles as well as a WAR for web applications) in a single deployment unit. Moreover, a PAR defines both a physical and logical application boundary.

The PAR sample is comprised of four directories, as shown below.



The formtags-par directory is a build project that understands how to create the PAR from its constituent bundles.

Granularity of the PAR

Achieving the appropriate level of granularity for your OSGi application is more of an art than a science. It helps to look at the different requirements:

Table 6.1. Granularity drivers

Requirement	Description
Domain/Technical Layering	Applications can be split either by domain (i.e., by use

Requirement	Description
	case or <i>vertically</i>) or by their technical layers (i.e., <i>horizontally</i>). Since the Form Tags application essentially has only a single use case, the bundles are split by technical layering (i.e., domain, service, and web).
Refreshability	A major benefit of OSGi is that of refreshability: if one bundle is changed, only bundles that have a dependency upon the exported types need to be refreshed. This has a high impact on development time costs as well as production costs. However, this can lead to lots of smaller, fine grained bundles. An example of this granularity would be to separate out the service API and implementation into two different bundles. This means that a change in the implementation wouldn't require any other bundles to be refreshed.

Ultimately the right level of granularity will depend upon your particular application and team.



Note

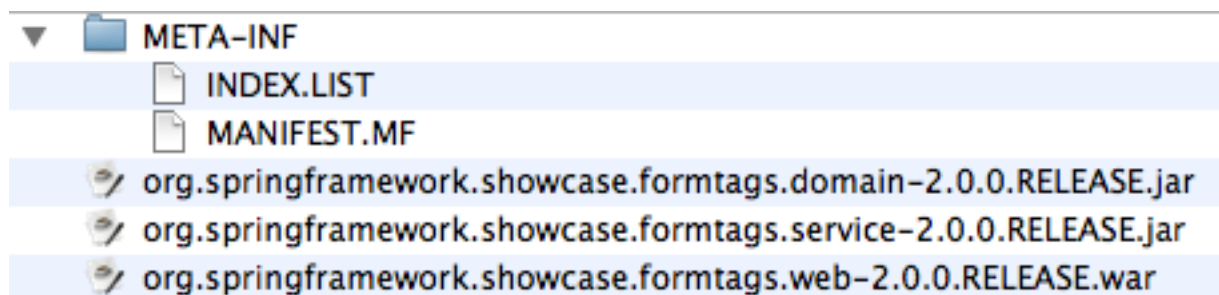
This topic will be revisited in greater detail later in the Programmer Guide in a chapter covering how to build a PAR from scratch.

Domain and Service Bundles

The service bundle is identical (except for the `Bundle-SymbolicName`) to that in the shared-services variation of the sample. The PAR has also separated out the domain classes into their own bundle. When layering by technical considerations, it is again somewhat of an unofficial convention to have a `.domain` bundle.

Constructing the PAR

Finally we need to construct the PAR itself. The following are the contents of the exploded PAR.



You can see that the PAR itself doesn't contain any resources or Java classes: it simply packages together a related set of bundles as a single, logical unit.

The PAR does however, contain its own `/META-INF/MANIFEST.MF`.

```
Manifest-Version: 1.0
Application-SymbolicName: org.springframework.showcase.formtags-par
Application-Version: 1.0.0
Application-Name: FormTags Showcase Application (PAR)
```

For more information on the contents of the PAR's `/META-INF/MANIFEST.MF`, please consult Chapter 4, *Developing Applications*.

You can now deploy the PAR on the VWS, for example by copying `/dist/formtags-par*.par` to the VWS's `pickup` directory. You should then see console output similar to the following:



Note

The console output has been reformatted to fit this document.

```
[2009-07-01 15:13:43.306] fs-watcher
<SPDE0048I> Processing 'CREATED' event for file 'formtags-par-2.0.0.RELEASE.par'.
[2009-07-01 15:13:44.060] fs-watcher
<SPDE0010I> Deployment of 'formtags-par' version '2.0.0.RELEASE' completed.
[2009-07-01 15:13:44.068] Thread-20
<SPWE0000I> Starting web bundle '/formtags-par'.
[2009-07-01 15:13:45.212] Thread-20
<SPWE0001I> Started web bundle '/formtags-par'.
```

Navigate to <http://localhost:8080/formtags-par> to see the welcome page.



Tip

Note that the web application's context path is explicitly defined via the `Web-ContextPath` manifest header in `/META-INF/MANIFEST.MF` of the web module within the PAR.

6.6 Summary of the Form Tags Migration

The Virgo Web Server provides out-of-the-box support for deploying standard Java EE WAR files. In addition support for *Shared Libraries* and *Shared Services* WAR formats provides a logical migration path away from standard, monolithic WARs toward OSGi-enable Web applications. The PAR packaging and deployment format enables truly fine-grained, loosely-coupled, and efficient application development. In general, the migration steps presented in this chapter are fairly straightforward, but developers should set aside time for some up-front design of the bundles themselves.

It is recommended that you take another sample application or indeed your own small application and go through this migration process yourself. This will help you better understand the concepts and principles at work. In addition, it is highly recommended that you familiarize

yourself with the Eclipse-based *Virgo Web Server Tools* support which is discussed in Chapter 7, *Tooling*.

6.7 Form Tags as a plan

Plans (see Section 4.3, “Creating Plans”) allow us to package and deploy the Form Tags application in a more flexible way. Instead of packaging all the bundles of the application into a single PAR file, each bundle can be placed in the repository and referred to in a *plan*.

The bundles to be placed in a repository in the chain (for example, `repository/usr`) are:

```
org.springframework.showcase.formtags.domain-2.0.0.RELEASE.jar
org.springframework.showcase.formtags.service-2.0.0.RELEASE.jar
org.springframework.showcase.formtags.web-2.0.0.RELEASE.war
```

which are just those files which were part of the PAR.

Here is the contents of a suitable plan file for the Form Tags example:

```
<?xml version="1.0" encoding="UTF-8"?>
<plan name="formtags.plan" version="2.0.0" scoped="true" atomic="true"
      xmlns="http://www.springsource.org/schema/dm-server/plan"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="
        http://www.springsource.org/schema/dm-server/plan
        http://www.springsource.org/schema/dm-server/plan/springsource-dm-server-plan.xsd">

  <artifact type="bundle" name="org.springframework.showcase.formtags.domain_par" version="[2.0,2.1)"/>
  <artifact type="bundle" name="org.springframework.showcase.formtags.service_par" version="[2.0,2.1)"/>
  <artifact type="war" name="org.springframework.showcase.formtags.web_par" version="[2.0,2.1)"/>

</plan>
```

where we have chosen to use any of the artifacts in the version range [2.0,2.1). This plan (as a file called, for example, `formtags.plan`) can be deployed in any of the normal ways (for example, dropped in the `pickup` directory).

When the plan is deployed, the artifacts it references are installed from the repository and deployed in the order given in the plan file. Because this plan is scoped and atomic, the collection is given an application scope and is started and stopped as a single unit.

7. Tooling

SpringSource provides a set of plug-ins for the Eclipse IDE that streamline the development lifecycle of OSGi bundles and PAR applications. The Virgo Web Server Tools build on top of the Eclipse Web Tools Project (WTP) and Spring IDE, the open-source Spring development tool set.

The Virgo Web Server Tools support the creation of new OSGi bundle and PAR projects within Eclipse, and the conversion of existing projects into OSGi bundle projects. Projects can then be deployed and debugged on a running VWS from within Eclipse.

7.1 Installation

Currently the Tools support Eclipse 3.4 and Eclipse 3.5 with the corresponding version of WTP. Downloading and unzipping the [Eclipse IDE for Java EE Developers](#) is the easiest way to start.

Execute the following steps to install the Tools into your Eclipse environment.

1. Install Spring IDE 2.1.0 from <http://springide.org/updatesite/> using the Eclipse Update Manager.



Note

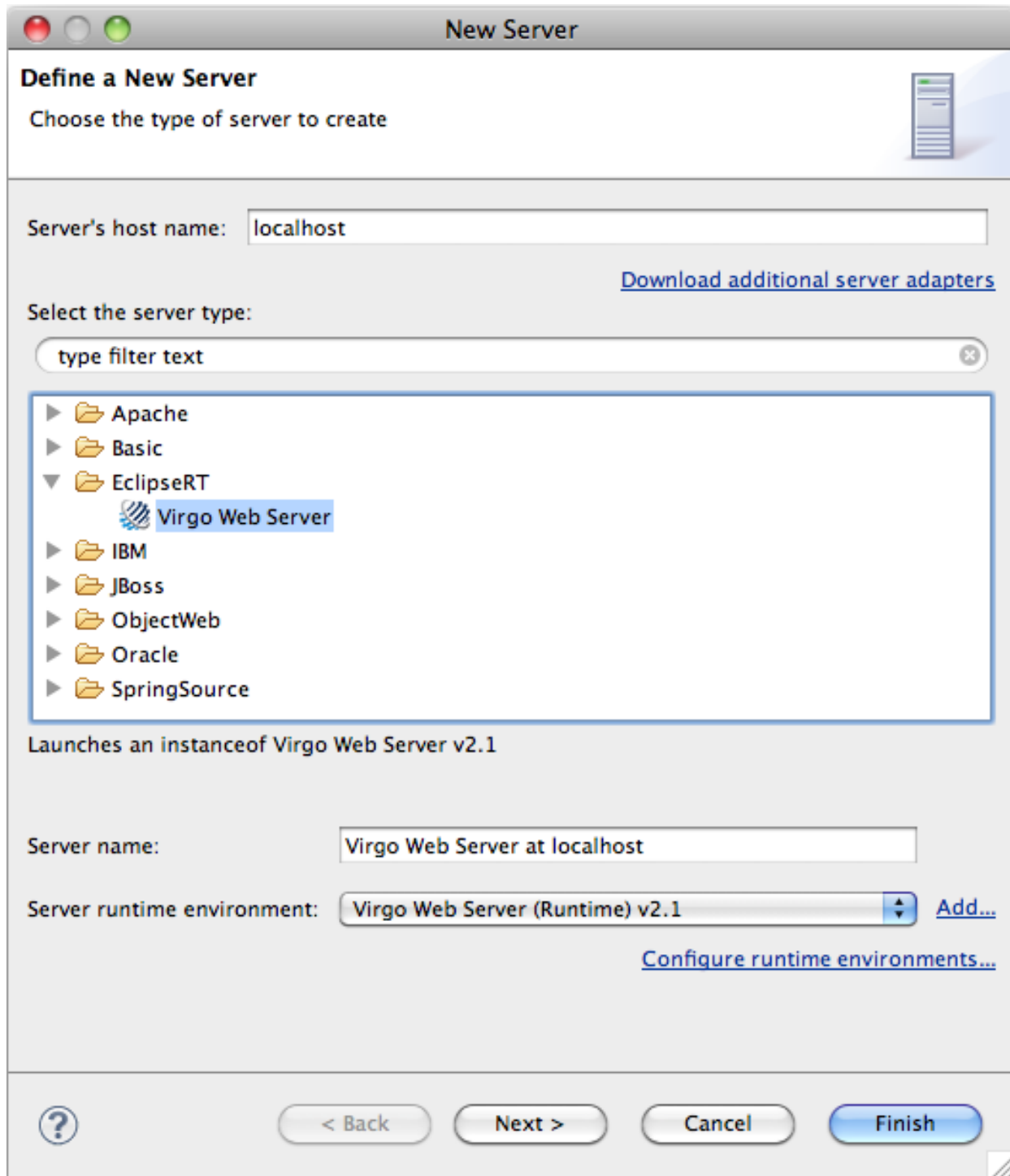
Don't try to install the "Spring IDE Dependencies (only for Eclipse 3.2.x)" from the "Dependency" category on Eclipse 3.3. This feature is intended only for Eclipse 3.2 and is to keep Spring IDE backward-compatible. You will not be able to continue with the installation if you select this feature on Eclipse 3.3.

2. Install the Tools from <http://static.springsource.com/projects/sts-dm-server/update/> using the Eclipse Update Manager.

7.2 Running a Virgo Web Server instance within Eclipse

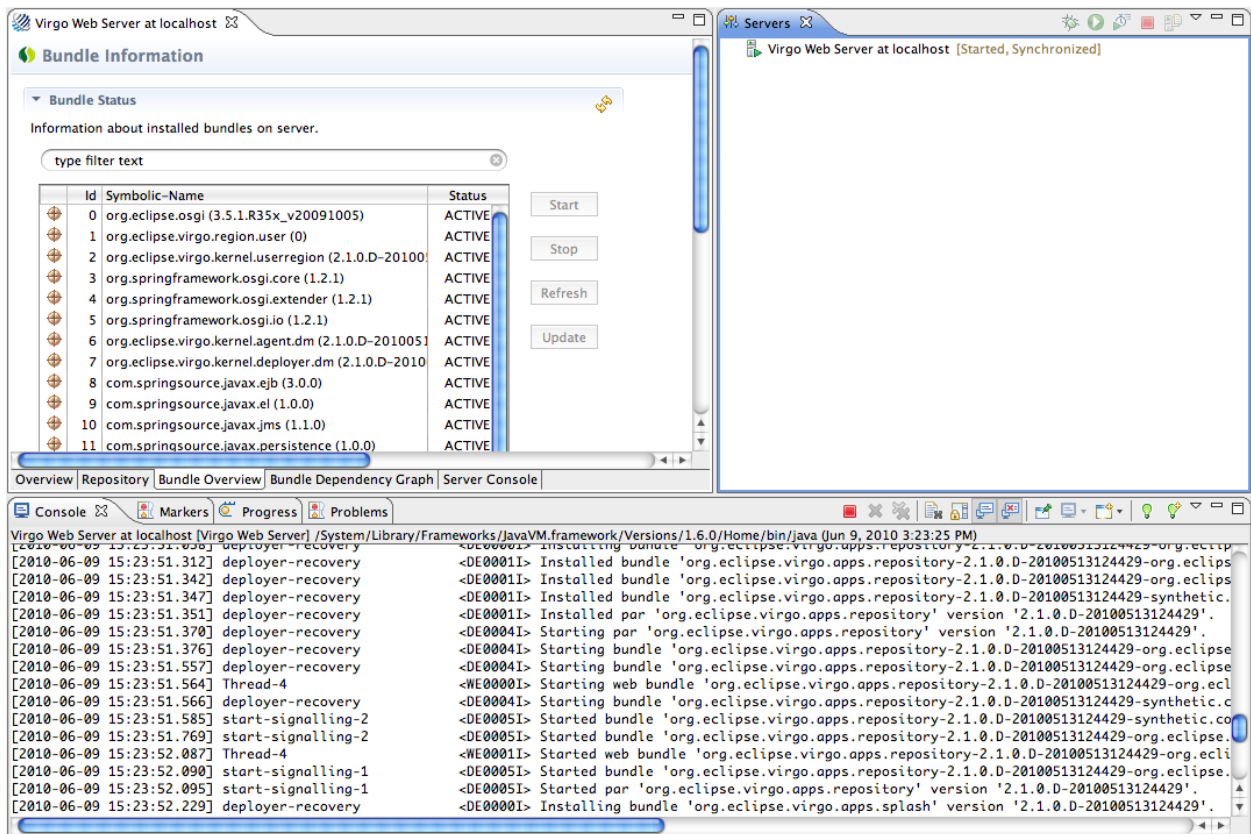
After installing the Tools from the update site outlined in the previous section, you will be able to configure an instance of the VWS inside Eclipse.

To do so bring up the WTP Servers view (i.e., Window → Show View → Other → Server → Servers). You can now right-click in the view and select "New → Server". This will bring up a "New Server" dialog. Select "Virgo Web Server v2.0 Server" in the "Virgo" category and click "Next".



Within the "New Server Wizard" point to the installation directory of the Virgo Web Server and finish the wizard. After finishing the wizard you should see a Virgo Web Server entry in the Servers view.

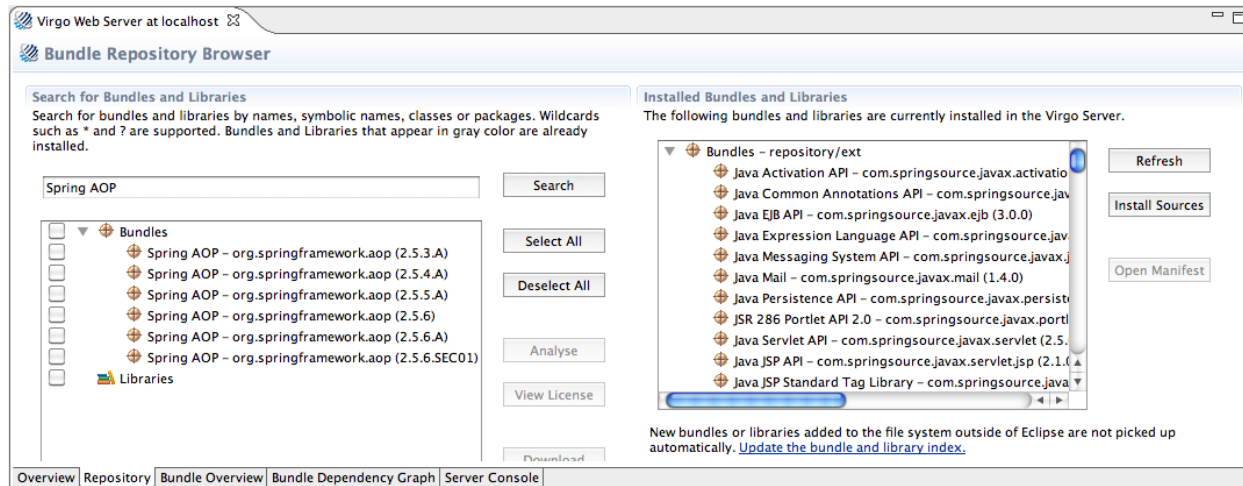
To start, stop, and debug the created Virgo Web Server instance use the toolbar or the context menu actions of the Servers view.



7.3 Bundle and Library Provisioning

After successful configuration of an instance of the Virgo Web Server in Eclipse you can use the Repository Browser to very easily install bundles and libraries from the remote SpringSource Enterprise Bundle Repository.

To open the Repository Browser double-click a Virgo Web Server instance in the Servers view and select the "Repository" tab in the server editor. Please note that opening of the Editor may take a few seconds as the contents of the local repository needs to be indexed before opening.



The left section of the Repository Browser allows the user to run searches against the SpringSource Enterprise Bundle Repository and displays matching results. The search can take parts of bundle symbolic names, class or package names and allows wildcards such as “?” and “*”. By selecting the checkbox left to a matching bundle and/or library and clicking the "Download" button it is very easy to install new bundles in the Virgo Web Server. For your convenience JARs containing the bundle source code can be automatically downloaded as well.

Clicking the "Download" button will trigger an Eclipse background job that will download the selected repository artifacts and -- if desired -- the source JARs one after another.

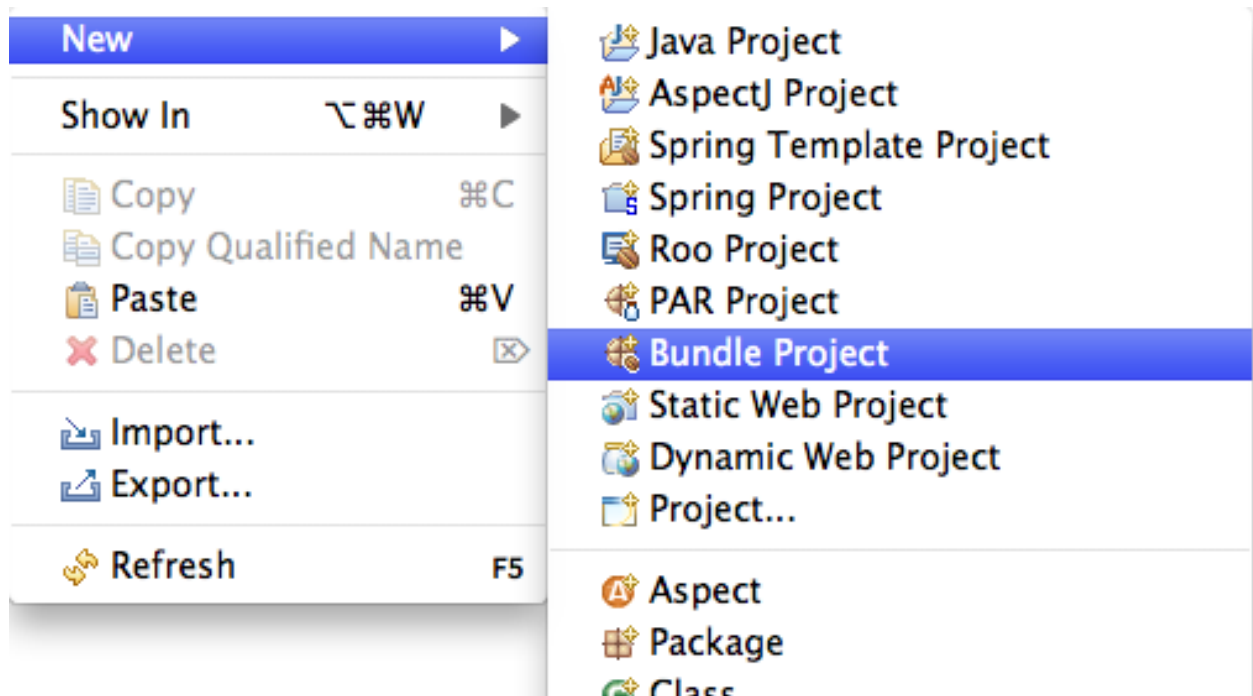
The section on the right displays the currently installed bundles and libraries. Bundles with available sources are visually marked. You can very easily download missing source JARs by using the "Install Sources" button.

7.4 Setting up Eclipse Projects

The Virgo Web Server supports different deployment units as discussed earlier in this guide. The Tools define specific project types to support the development of OSGi and PAR projects.

Creating New Projects

There are two New Project Wizards available within Eclipse that allow for creating new OSGi bundle and PAR projects. The projects created by the wizards are deployable to the integrated VWS instance without requiring any additional steps.



Those wizards create the required `MANIFEST.MF` file and appropriate manifest headers.

Migrating existing Java Projects

To migrate an existing Java Project to be used with the VWS, the Tools provide a migration action that adds the required meta data to the project. The migration will not change your project's source layout.

Use the context menu action of a project in the Package or Project Explorer and select "Spring Tools → Convert to OSGi bundle project".

7.5 Developing OSGi Bundles

The Tools provide functionality that makes developing OSGi bundles, especially the editing of `MANIFEST.MF` files, easier.

Resolving Bundle Dependencies

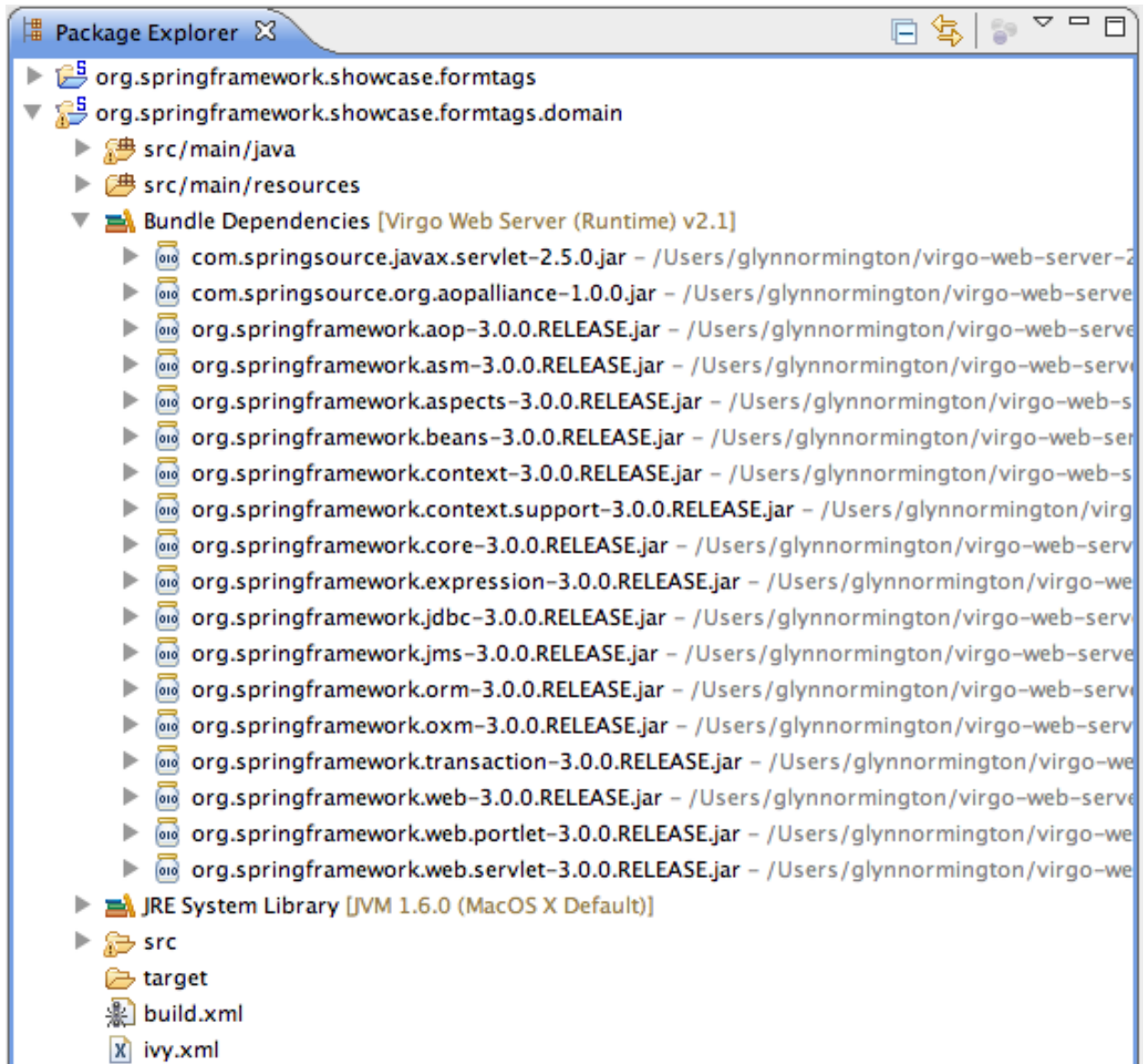
While working with OSGi bundles, one of the most interesting and challenging aspects is defining the package, bundle, and library imports in the manifest and then keeping this in sync with your compile classpath either in Ant and Maven or Eclipse. In most cases you would typically be required to manually set up the Eclipse classpath. Ultimately, the Eclipse compile classpath is still different from the bundle runtime classpath, as normally an entire JAR file is being made available on the Eclipse classpath but not necessarily at runtime due to the explicit visibility rules defined in `Import-Package` headers.

The Tools address this problem by providing an Eclipse classpath container that uses an Virgo Web Server-specific dependency resolution mechanism. This classpath container makes resolved dependencies available on the project's classpath but allows only access to those package that are imported explicitly (e.g., via `Import-Package`) or implicitly by using `Import-Library` or `Import-Bundle`.

To use the automatic dependency resolution, an OSGi bundle or PAR project needs to be targeted to a configured Virgo Web Server instance. This can be done from the project's preferences by selecting the runtime on the "Targeted Runtimes" preference page.

**Note**

In most scenarios it is sufficient to target the PAR project to a runtime. The nested bundles will then automatically inherit this setting.



After targeting the project or PAR you will see a "Bundle Dependencies" classpath container in your Java project. It is now safe to remove any manually configured classpath entries.

The classpath container will automatically attach Java source code to the classpath entries by looking for source JARs next to the binary JARs in the Virgo Web Server's repository. You can also manually override the source code attachment by using the properties dialog on a single JAR entry. This manual attachment will always override the convention-based attachment.

Editing the Manifest

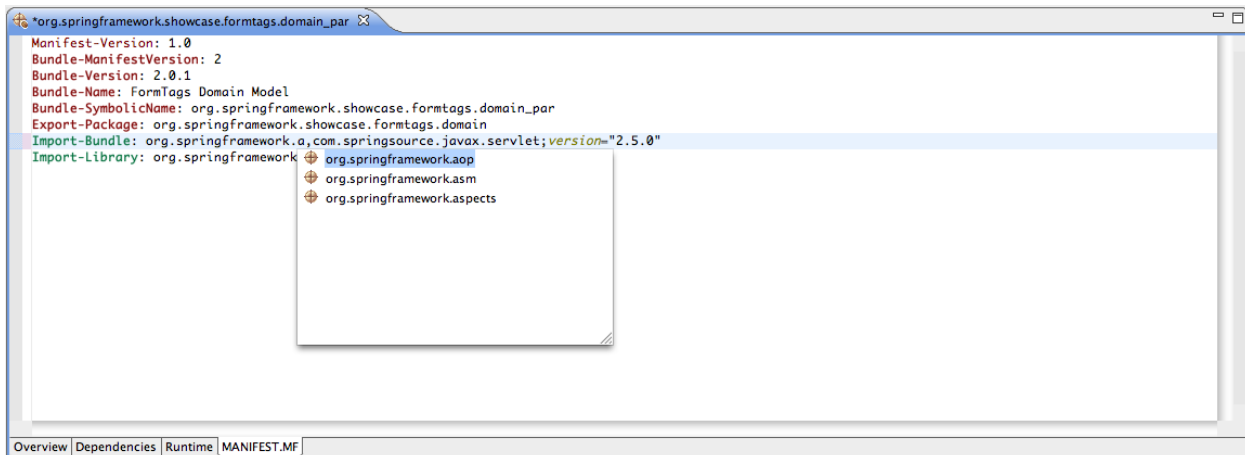
The Tools provide a Bundle Manifest Editor that assists the developer to create and edit MANIFEST.MF files. The editor understands the Virgo Web Server specific headers like `Import-Library` and `Import-Bundle` and provides content assist features while editing source code. Furthermore a Eclipse Form-based UI is also available.

To open the Bundle Manifest Editor right click a MANIFEST.MF file and select "Bundle Manifest Editor" from the "Open With" menu.



Note

Please note that the Virgo Web Server specific manifest headers appear in green color to distinguish them from those headers defined in the OSGi specification. This also makes navigating much easier.

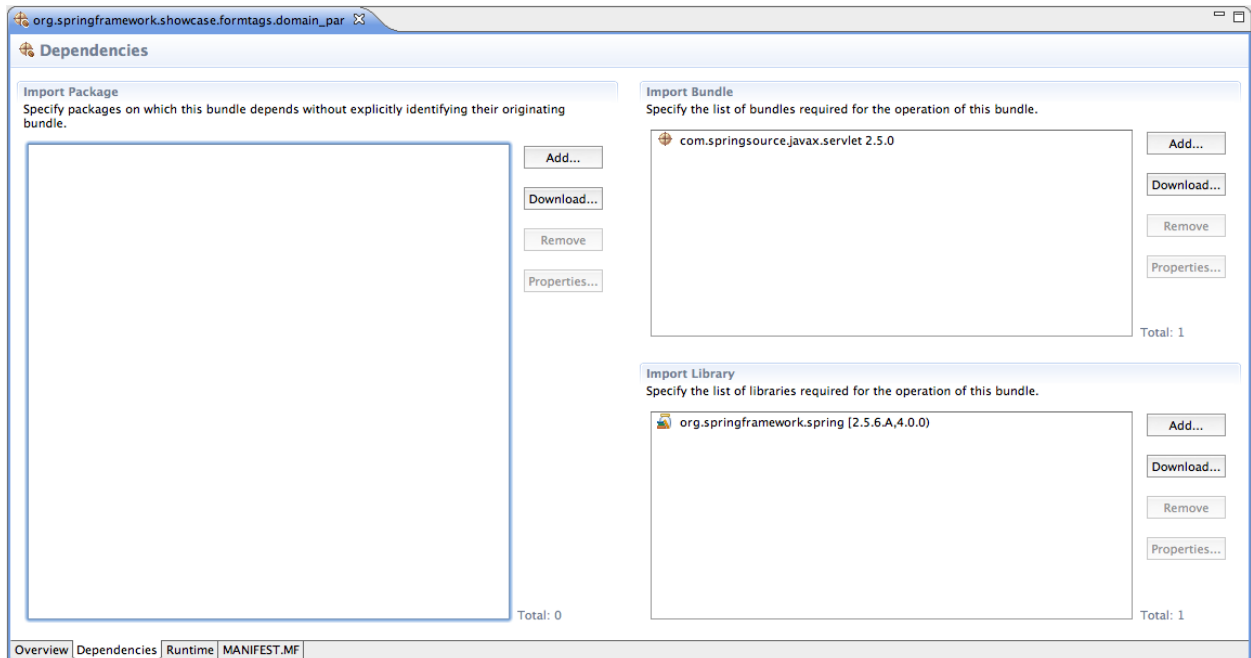


The content assist proposals in the source tab as well as in the UI-based tabs are resolved from the bundle and library repository of an installed and configured Virgo Web Server. Therefore it is important to target the project or PAR to a specific VWS instance to indicate to the tooling which bundle repository to use.



Note

If a OSGi bundle project is not targeted to a VWS instance, either directly or indirectly via a PAR project's targeting, the manifest editor will not be able to provide content assist for importing packages, bundles, and libraries.

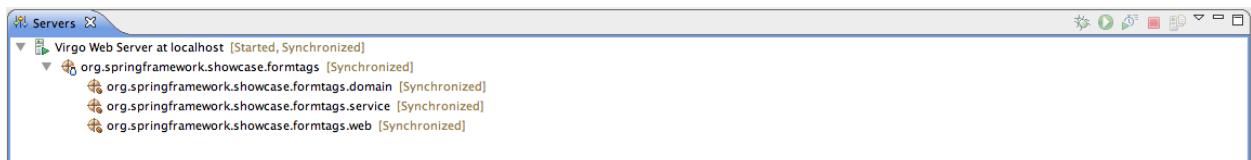


The Dependencies tab of the Bundle Manifest Editor enables the user to easily download and install bundles and libraries from the SpringSource Enterprise Bundle Repository by using the "Download..." buttons next to the "Import Bundle" and "Import Library" sections.

7.6 Deploying Applications

Currently the Tools support direct deployment of WTP Dynamic Web Projects, OSGi bundle and PAR projects to the VWS from directly within Eclipse.

To deploy an application to the Virgo Web Server just bring up the context menu on the configured VWS runtime in the Servers view and choose "Add or Remove Projects...". In the dialog, select the desired project and add it to the list of "Configured projects".



Note

Deploying and undeploying an application from the VWS certainly works while the Virgo Web Server is running, but you can also add or remove projects if the VWS is not running.

Once an application is deployed on the Virgo Web Server the tooling support will automatically pick up any change to source files -- for example, Java and XML context files -- and refresh the

deployed application on the VWS.

The wait time between a change and the actual refresh can be configured in the configuration editor of the runtime. To bring up that editor, double-click on the configured Virgo Web Server instance in the Servers view.

8. Working with Common Enterprise Libraries

8.1 Working with Hibernate

Importing Hibernate

Hibernate uses CGLIB to dynamically create subclasses of your entity types at runtime. To guarantee that Hibernate and CGLIB can correctly see the types, you must add an `Import-Library` or `Import-Bundle` for the Hibernate library or bundle into any bundle that uses Hibernate directly.

Additionally, if other bundles in your application contain types to be persisted by Hibernate, then be sure to specify the `import-scope` directive of the `Import-Bundle` header in the bundle that uses Hibernate directly. The `import-scope` directive tells Virgo Web Server to implicitly import the bundle into all other bundles that make up the application; this ensures that bundles that indirectly depend on the generated Hibernate classes have access to them, but you do not have to explicitly update their `Import-Bundle` header, ensuring modularity. For example:

```
Import-Bundle: com.springsource.org.hibernate;version="[3.2.6.ga,3.2.6.ga]";import-scope:=application
```

The `import-scope` directive works only for the bundles in a scoped application (PARs or plans.)

8.2 Working with DataSources

Many `DataSource` implementations use the `DriverManager` class which is incompatible with typical OSGi class loading semantics. To get around this, use a `DataSource` implementation that does not rely on `DriverManager`. Versions of the following `DataSources` that are known to work in an OSGi environment are available in the [SpringSource Enterprise Bundle Repository](#).

- [Apache Commons DBCP](#)
- `SimpleDriverDataSource` available in [Spring JDBC](#) 2.5.5 and later

8.3 Weaving and Instrumentation

When using a library that performs bytecode weaving or instrumentation, such as AspectJ, OpenJPA or EclipseLink, any types that are woven must be able to see the library doing the

weaving. This is accomplished by adding an `Import-Library` for the weaving library into all bundles that are to be woven.

Weaving is often used by JPA implementations to transform persisted types. When using a JPA provider that uses load-time weaving, an `Import-Library` for the provider is needed in the bundles containing the persisted types.

8.4 JSP Tag Libraries

When using tag libraries within a WAR or Web Bundle, be sure to include an `Import-Bundle` or `Import-Library` for the tag library bundle(s). This will ensure that your module can see the TLD definition and implementing types. For example, to use the Apache implementation of JSTL, add the following to your bundle's `/META-INF/MANIFEST.MF`:

```
Import-Bundle: com.springsource.org.apache.taglibs.standard;version="1.1.2"
```

9. Known Issues

9.1 JPA Entity Scanning

Classpath scanning for JPA entities annotated with `@Entity` does not work. Describing entities with `@Entity` will work, but the entities need to be listed explicitly.

9.2 `ClassNotFoundException` When Creating a Proxy

When creating proxies at runtime, there are circumstances where `ClassNotFoundException`s can be generated. These errors happen because the proxy creating bundle does not have visibility into every type on the interface of the proxy. You can either put in import statements for all the relevant types or add use a service (with visibility of all pertinent types) to create the proxy. Please see [this blog entry](#) for more details.

9.3 Creating proxies with CGLIB for package-protected types

In traditional Java EE applications user types are loaded by the same `ClassLoader` as CGLIB. This allows CGLIB to proxy package-protected types. In OSGi environments, user types and CGLIB will most likely be packaged in separate bundles. This results in the user types and CGLIB being loaded by different `ClassLoaders`. This prevents CGLIB from proxying any package-protected types.

The workaround for this issue is to make all types that require proxying public.

9.4 Tomcat Restrictions

The following Tomcat features are not supported.

- `<Context>` elements.

