

Jan Machacek
Aleksa Vukotic
Anirvan Chakraborty

SpringSource dm Server IN ACTION



Unedited Draft

 MANNING





MEAP Edition
Manning Early Access Program

Copyright 2008 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Please post comments or corrections to the Author Online forum at:
<http://www.manning-sandbox.com/forum.jspa?forumID=508>

Table of Contents

Part One: Introduction

1. Why Spring AP?
2. Getting started with Spring AP
3. Components/architecture of the Spring AP

Part Two: Deploying applications in Spring AP

4. Packaging & deployment formats
5. Modularization strategies
6. Deploying applications
7. Personalities
8. Dependencies
9. Migrating to Spring AP

10. J2EE application migration
11. Common Enterprise components
12. Externalized configuration

Part Three: Development, management & monitoring, performance tuning

13. Tooling
14. Management & monitoring

Why SpringSource dm Server?

The SpringSource dm Server is an OSGi-based Java EE application server. The dm Server is part of the SpringSource Application Platform. In addition to the dm Server, the Application Platform contains tools and components that simplify development and management of Java enterprise applications.

Before we explore dm Server and components of the Application Platform in more detail, let's take a look at why you should leave your tried and tested Java EE application development tools and application servers behind and join the SpringSource dm Server revolution!

1.1 Life without the dm Server

We know that we should be starting with a positive outlook – you know, there are no problems, there are only solutions and so on, but let's be honest for a moment and explore the issues we face in our Java EE application development and delivery.

When we implement our Java EE applications, we usually take extra care not to introduce too many dependencies. We layer the applications and we use the development environments and build scripts to enforce the layering rules. We use circular dependency and complexity analysis tools to ensure our code is as clean as possible.

Then we take our well-designed and cleanly structured code and build a deployment archive. The archive and the deployed application do not maintain the structure we have worked so hard to create.

We also try to keep the libraries in our applications consistent. When we start building our application, we make sure that we satisfy the dependencies of all components in our application exactly. When we use Hibernate 3.3.0 SP1, we need to include Commons Collections 3.1. All is well until some other component needs Commons Collections 3.2.1. We cannot keep both the 3.1 and the 3.2.1 Commons Collections jars, so we only keep the latest version and hope for the best.

Problems that are even more serious arise when we need to make changes to our application. In the current Java EE application servers, it is difficult to update components of running applications without restarting them. As a result, even a simple change means downtime and even if we exhaustively test the application in a pre-production environment, the deployment may still fail.

Finally, when our deployed application is running, it becomes a black box. To alleviate this problem, we include management and logging code in our application. Unfortunately, each application uses slightly different logging infrastructure. Moreover, the logging that the management and monitoring infrastructure does not always contain the information the developers need.

To summarize, the list of issues we face the traditional Java EE application deployment includes:

- Monolithic deployment archives. When we build our carefully designed application, all its structure is lost in the blob of the deployment archive.
- Complex re-use of services. It is difficult to simply refer to services from other already running applications. Even if we use framework services to reduce the complexity of exposing a service in a traditional Java EE application, we still face class loading and versioning problems. It not possible to have two versions of the same library in the application server's shared libraries. This forces us to use other transport formats, such as XML, which only adds to the complexity of our applications.
- Complex change control. Whenever we want to deploy a new version of even a small

Please post comments or corrections to the Author Online forum at:

<http://www.manning-sandbox.com/forum.jspa?forumID=508>

component in a monolithic application, we have to deploy the entire application. Moreover, it makes it more difficult to determine the scope of the change we introduced. Without modularity, we have to re-test the entire application.

- Lack of versioning. It is impossible to reference a specific version of a library in our applications. We cannot deploy multiple versions of the same library (class) in our application server and then specify which version our application needs. We end up deploying many common libraries in each application; and even so, we may run into problems caused by the application server's class loading behavior. Ultimately, we find ourselves in a deploy & pray situation, where we deploy the application and hope that it would somehow use the right library.
- Complex management and monitoring of running applications. Over time, we have perfected the logging and monitoring of our Java EE applications. Still, time and time again, we find ourselves complaining that the logs do not contain enough information to identify an error.

We got used to living with these problems: we automatically package our applications into a monolithic archives; we hope that the different library versions will be compatible enough not to cause any problems in our application. We accept that to design a flexible service-oriented application, we have to jump through technological hoops. We are forced to consider the complexity of the logging framework and the application's performance in order to record enough information in the log to allow us to successfully diagnose the problem. Often, we do not record enough state information about the failures and we record the topmost failure, rather than recording the first failure (the root cause of the problem).

Now, forget all those limitations and imagine a perfect platform. Such a perfect development and deployment platform would help us:

- Enforce application's structure at runtime. We would like to maintain the packaging and layering structure of our application even when we deploy it.
- Use versioning of the components. We would like to eliminate the possibility of incompatible libraries in our applications. We would like to be able to reference a library by its name *and* a version.
- Be able to change components of our applications without changing the rest of the application. If we could do this, we could reduce the downtime. More importantly, if we could keep the application modular, we'd focus more on loose coupling. We could also more easily control the development process in larger teams. Finally, we could reduce testing time, because we would know the specific components that have been changed.
- Closely monitor our application's health and performance. If all our applications shared the same management and monitoring interface, we could simplify the work for the production team. In addition, the management and monitoring information should allow the developers to easily identify the problem.
- Explore different architectural and deployment models. Re-using services from one application in another involves solving technical problems: we need to design the applications in a particular way to be able to achieve this re-use. We would like to forget the technical detail and concentrate on the code that "makes us money".

Such perfect platform is the SpringSource Application Platform and in this book, we will explore how you can use the tools and applications that make up the Platform to achieve these goals.

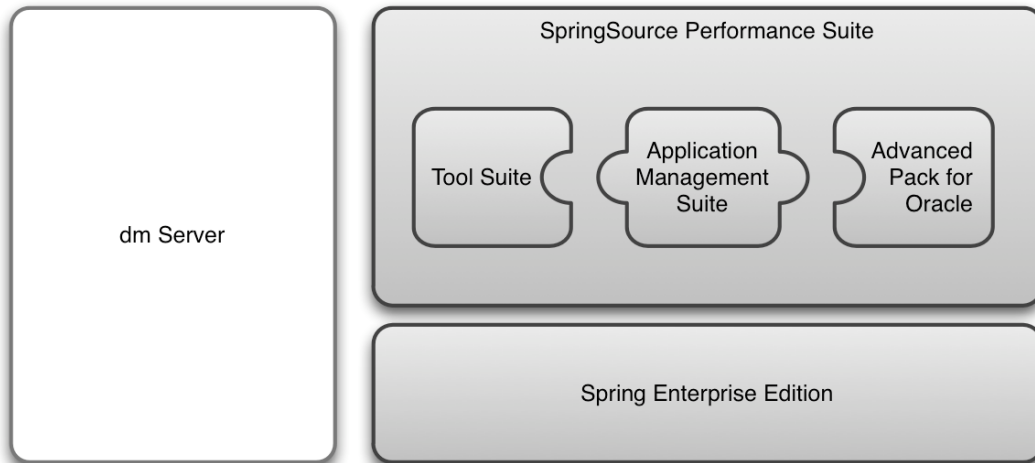
1.2 How does the dm Server fit in?

The dm Server sets out to resolve the problems we outlined in the previous paragraph. Together with the tools in the SpringSource Performance Suite, we can write better code faster. We will begin with an illustration on Figure 1.1, which shows the Platform's components.

Figure 1.1 SpringSource Application Platform components

Please post comments or corrections to the Author Online forum at:

<http://www.manning-sandbox.com/forum.jspa?forumID=508>



Let's follow this diagram and take a more detailed look at the elements that make up the Platform.

TOOL SUITE

The Tool Suite is a comprehensive package of Spring development tools. It includes the Spring IDE to assist Spring Framework development; dm Server tools, which help you configure the dm Server instances and deploy and debug your dm Server projects in the dm Server.

In addition to development tools, the Tool Suite brings architectural and code review tools to help the developers make the best of the Spring Framework and the dm Server. In addition to the review tools, the Tool Suite can connect to SpringSource's knowledge base, which gives you access to the SpringSource's consultants' experience.

APPLICATION MANAGEMENT SUITE

The Application Management Suite unifies monitoring of deployed applications and brings consistent interface to the production team. The Application Management Suite can monitor a number of application servers and other services, but it performs best when it monitors applications that use the Spring Enterprise Edition.

ADVANCED PACK FOR ORACLE

To achieve the best performance in mission critical applications that use the Oracle RDBMS, you can take advantage of the SpringSource Advanced Pack for Oracle. The Advanced Pack for Oracle allows you to use features of the Oracle RDBMS that are not available in JDBC, for example custom data types, Oracle messaging and failover support.

SPRING ENTERPRISE EDITION

Spring Enterprise Edition is a quality-assured version of the Spring Portfolio, which includes the Spring Framework, Spring Security, Spring Web Flow, Spring Web Services and many others. The Spring Enterprise Edition keeps the packaging structure of the Spring Framework, but it includes application-monitoring code. This monitoring code exposes information about your application's runtime. The Application Management Suite can use the monitoring information that the Spring Enterprise Edition exposes to keep an eye on your Spring applications.

DM SERVER

To keep the advantages of the clean design and modern frameworks even when you deploy your applications, use the dm Server. The dm Server is a new generation of Java EE application servers: it brings together traditional Java EE and OSGi worlds.

Now that you know how the components of the Application Platform fit together, we are going to take a more detailed look at each of them.

1.3 Introducing the SpringSource dm Server

The dm Server is the central component of the Application Platform. It is the only OSGi R4.1-based application server that lets the developers access the OSGi functionality. Other application servers that use OSGi use it only internally and allow you to deploy and run only standard Java EE applications. Finally, even though the dm Server is built on and exposes the OSGi model, your

Please post comments or corrections to the Author Online forum at:

<http://www.manning-sandbox.com/forum.jspa?forumID=508>

applications do not have to use the OSGi model at all. In fact, you can deploy legacy Java EE web applications without any major modifications to their code. However, in your new applications you can take full advantage of the dm Server's extended OSGi environment. The benefits of the OSGi platform include:

- Modularization and versioning. The OSGi platform allows you to keep and enforce the modular structure of your applications. In addition to this, the OSGi platform supports versioning of the modules.
- Manageability. The OSGi platform includes module management tools that allow you to install, uninstall, start and stop bundles without writing a line of code.
- Non-intrusiveness. The OSGi platform does not force you to write your application's classes and interfaces in any special way. There is no special contract that your bundles must use to be able to export or import a service.
- Laziness. The applications that use the OSGi platform can take advantage of lazy initialization of the required bundles and services. The platform can delay loading and initialization of a bundle (and all bundles it depends on) until it is needed.

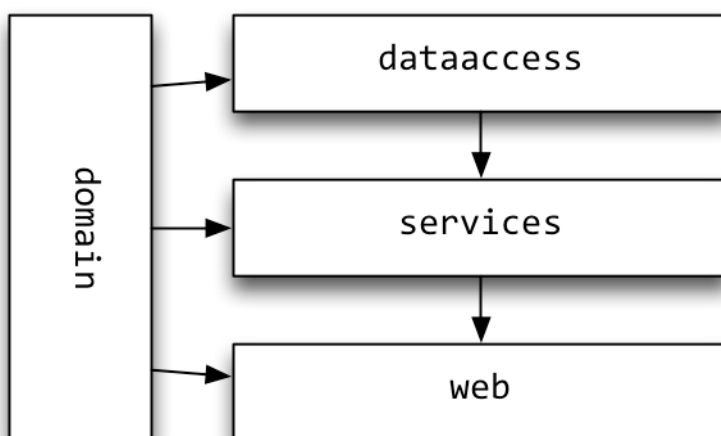
The benefits of the OSGi environment include the ability to keep the modularization of your applications even when deployed, versioning of the modules and the ability to specify dependencies between the modules.

1.3.1 OSGi in dm Server

Let's now focus on the main difference between the dm Server and other Java EE application servers: OSGi used internally as well as made available for the developers. Even though the OSGi model was first introduced 10 years ago, it has not been very visible to the majority of Java developers. Because the OSGi model is an unfamiliar approach to packaging and deploying applications, we need to take a look at its principles before we can delve into the depths of the dm Server.

When you want to deploy Java EE application in non-OSGi Java EE application server, you have to prepare a deployment archive. The deployment archive obliterates the structure of your application modules' source code. The only exception is the EAR deployment archive, which preserves the topmost module structure (for example, an EAR may contain an EJB JAR, a resource adapter RAR and a web application in a WAR). The structure of the code within the archives in an EAR or standalone WAR is completely lost. Let's take application whose architecture is shown in Figure 1.2.

Figure 1.2 Architecture of a sample Java EE application



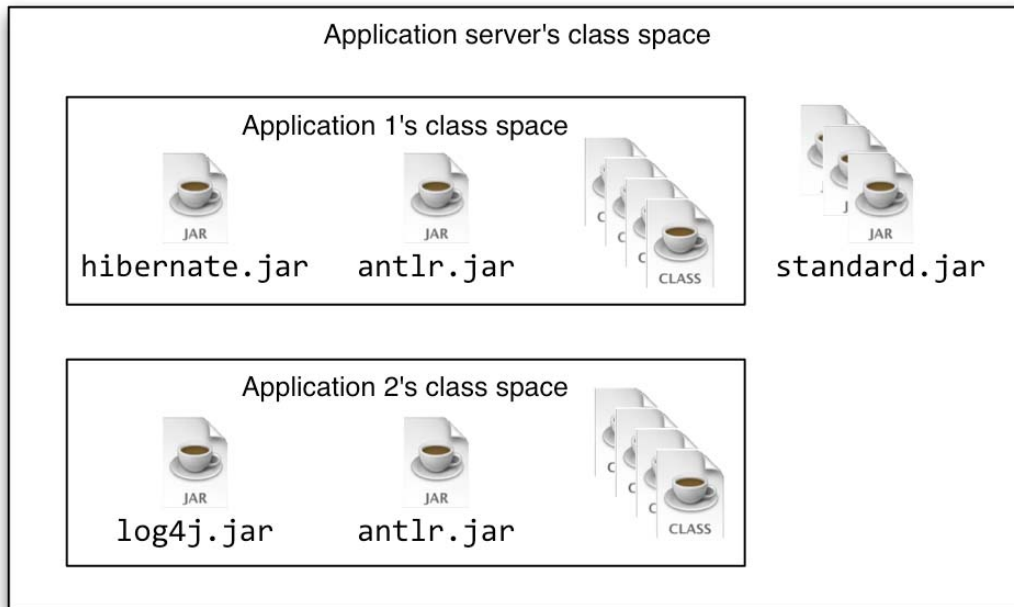
You can see that the web module uses the services module, which uses the dataaccess module. All three modules use the domain module. We can enforce this structure in our IDE and in the build process, but when we create the deployment archive, the structure is lost. We end up with one archive that contains all classes from all packages. Further, if we cannot easily re-deploy a single module, we always have to re-deploy the entire application.

Please post comments or corrections to the Author Online forum at:

<http://www.manning-sandbox.com/forum.jspa?forumID=508>

When we deploy the application from Figure 1.2, the boundaries between the layers are lost. Our application's classes as well as classes in our application's libraries are in one class loading space. Depending on the application server, the shared libraries (i.e. the libraries that are included in the application server's distribution) are included in another class loading space (see Figure 1.3).

Figure 1.3 Runtime view of the application



You can see that there is no concept of modules or versions, there is just one big class space and that there is no library versioning problem. We had to deploy the `antlr.jar` in both applications in our example, because we cannot ensure that there is only one version of the `antlr.jar` library in the application server's class space. Usually, the application server favors the libraries from the application's class space before loading the class from the application server's class space. This causes a problem when two applications should share objects between each other. Our code can still run in our unit tests, but may fail in the production environment, because the test environment's `ClassLoader` behaves differently than the production's `ClassLoader`. Tracking these errors (typically `NoSuchMethod` or `ClassCastException`s) is very difficult.

Finagle's law

Finagle's law of dynamic negatives describes this situation perfectly: Anything that can go wrong, will—at the worst possible moment. We can directly apply this to our enterprise application. It will fail when it is running the most critical process (on Sunday evening).

Finally, if a Martian were to review our Java EE deployment strategies, they would certainly be surprised to see that we deploy the same libraries in each application. "Why," the Martian Java guru would say, "do you not share the libraries between the applications?" Because we do not know that we'd get the right version and because we cannot deploy the different version of the same library.

Enter OSGi. In OSGi, there are no complex Java EE applications in large deployment archives. There aren't even any applications, there are only bundles. Each bundle can export and import packages, register and reference services in the OSGi service platform's registry. If we wanted to deploy the application from Figure 1.2 in an OSGi environment, we would deploy it as four bundles: `dataaccess`, `services`, `web` and `domain`. The `web` bundle would import packages and reference services from the `services` bundle, the `services` bundle would import packages and reference services from the `dataaccess` bundle, the `web`, `services` and `dataaccess` bundles would import packages from the `domain` bundle.

Please post comments or corrections to the Author Online forum at:

<http://www.manning-sandbox.com/forum.jspa?forumID=508>

However, we should create as few modules as possible. There are two reasons for this: the first reason is that the fewer modules we have, the less work we have to do to deploy them. The second reason is more important. It is unlikely that some other application will need to use our application's data access module. Similarly, if this is a standalone application, its domain is specific to this application alone. With these principles in mind, we should implement the application in just two bundles: `services`, which includes all code from the domain, `dataaccess` and `services` packages and the `web` bundle that deals with the presentation tier.

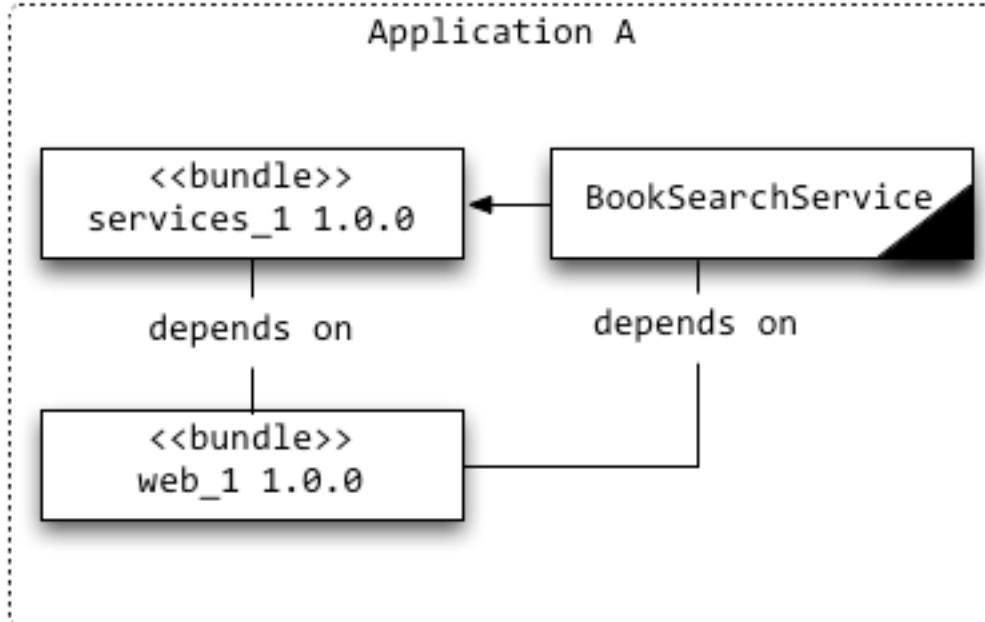
It is important to remember that deploying a bundle is not the same as including a jar on an application's classpath. The OSGi execution environment uniquely identifies each bundle using its symbolic name and version; you will never end up in the "jar hell" of today's enterprise applications.

The OSGi execution environment allows you to deploy, start, stop and undeploy bundles dynamically. This means that we can update the `services` bundle without stopping the `web` bundle.

It would seem that OSGi alone is the ideal solution. Unfortunately, it brings strict restrictions to the structure of the applications. As we said earlier, there are no applications or jars in OSGi; there are only bundles. You can no longer "just include the `hibernate-3.2.2-ga.jar` in your application's `WEB-INF/lib` directory"; you have to turn it into an OSGi bundle. While turning a jar into a bundle is not a complicated process, it is enough to put most developers off the idea of using OSGi. Finally, the OSGi execution environment can deploy and run bundles, but it does not deal with typical Java EE application requirements. For example, Equinox, an OSGi execution environment implementation is not aware of servlets. Another limitation of OSGi is that there is no concept of applications: there are only bundles and all bundles share the same namespace. There are two scenarios where a single OSGi namespace causes a problem and two more reasons to use the dm Server, which extends the OSGi execution environment's features to overcome these limitations.

The first scenario is when you want to deploy two instances of the same set of bundles. Figure 1.3 shows this situation.

Figure 1.3 Sample application structure



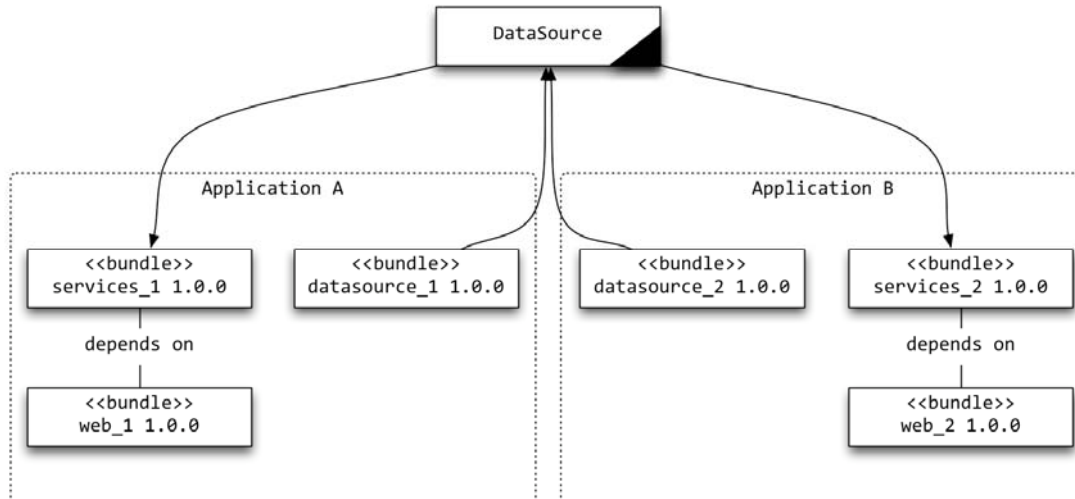
We show Application A to emphasize that the two bundles logically belong into the application unit, but the OSGi execution environment is not aware of this. Therefore, the first deployment of `services_1` and `web_1` bundles will succeed. However, we cannot deploy a second instance of the `services_1` and `web_1` bundles, because bundle with the same name are already deployed.

The second scenario where a single OSGi registry causes problems is a situation where bundles with different names export the same service (see Figure 1.4).

Figure 1.4 Different bundles exporting the same service

Please post comments or corrections to the Author Online forum at:

<http://www.manning-sandbox.com/forum.jspa?forumID=508>



In this scenario, we would like to have two `DataSource`s, one for each application. However, there is no guarantee that `services_1` will receive the `DataSource` exported from `datasource_1` and that `services_2` will receive `DataSource` exported from `datasource_2`. There is no guarantee because the OSGi model has no notion of applications.

The last limitation (or perhaps just annoyance) of OSGi is that you can only import packages. For example, if we wanted to import all packages from the Spring Framework OSGi bundles, we would need to write 249 `Import-Package` directives.

Note

In most cases, you could use the OSGi standard `Require-Bundle`. Unfortunately, `Require-Bundle` comes with a subtle, but serious implication. When you use `Require-Bundle`, the OSGi platform may order the referenced packages in an unexpected order.

If bundle A depends on bundles B, C and D and if bundle B depends on bundle D, then using `Require-Bundle B, C, D` in bundle A will cause the dependency list for A to become B, D, C instead of B, C, D. This means that types from C may be hidden by the same types from D.

Another issue is that if you use `Require-Bundle`, the OSGi platform may split packages. This means that the packages will come partly from one module and partly from another and each module's package may contain different types.

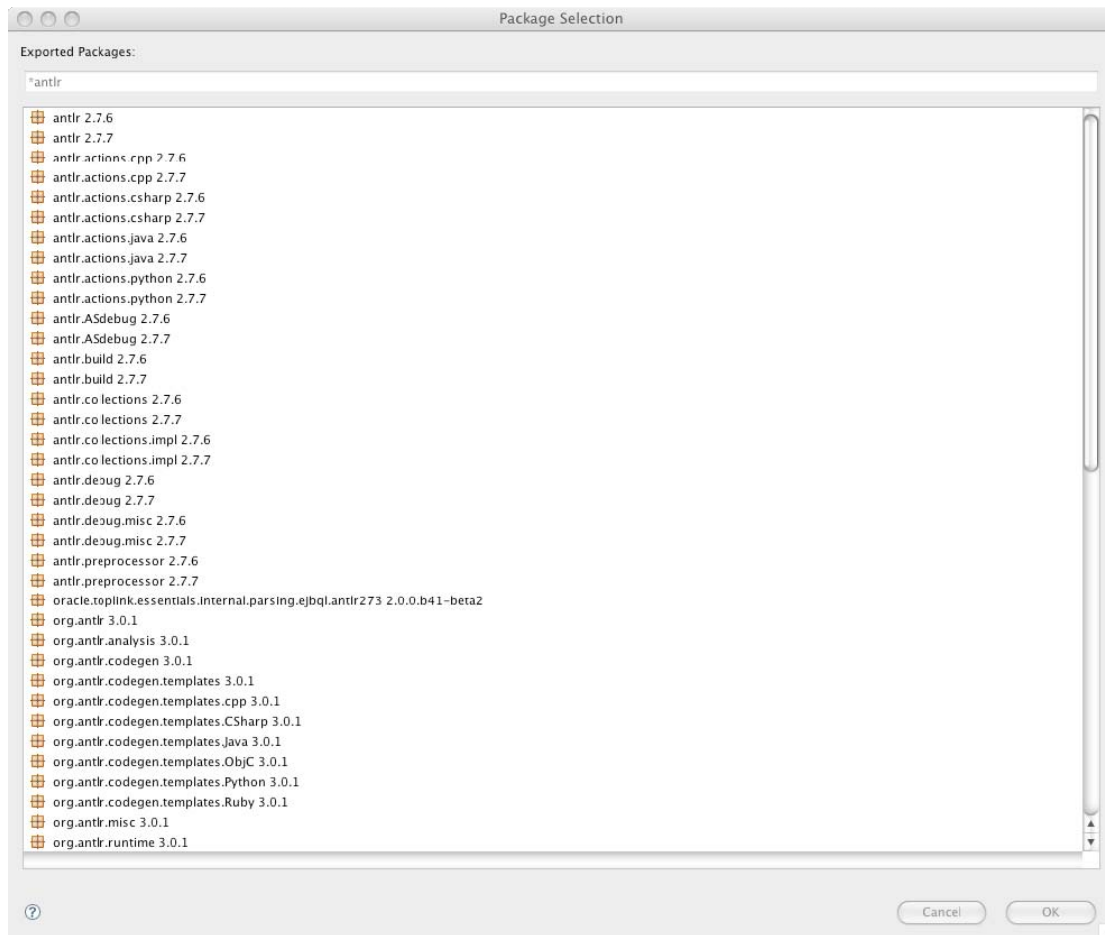
Put simply, if you use `Require-Bundle`, you can end up with the same package containing types from different modules.

The SpringSource dm Server addresses the issues we have outlined. It is an OSGi R4.1 execution environment, but it allows you to deploy legacy applications with minimal impact, which will allow you to gradually migrate existing Java EE applications to dm Server. To help you write new OSGi-based applications, its distribution includes a comprehensive list of bundles for typical Java EE libraries. If you cannot find a particular bundle in the dm Server's distribution, you can use SpringSource's bundle repository, which contains hundreds of Java libraries converted into OSGi bundles. Figure 1.5 shows a fragment of the repository's content.

Figure 1.5 Fragment of the SpringSource Enterprise Bundle repository

Please post comments or corrections to the Author Online forum at:

<http://www.manning-sandbox.com/forum.jspa?forumID=508>



The SpringSource Enterprise Bundle Repository contains OSGi bundles for common (and even some more exotic) Java EE libraries. This is extremely helpful when migrating your applications to dm Server. Even a small Java EE web application easily uses 30 libraries. At the lowest level, the difference between a JAR and an OSGi bundle is the META-INF/MANIFEST.MF file. However, the MANIFEST.MF's structure is not trivial. Listing 1.1 shows the MANIFEST.MF file from the Hibernate 3.2.6 GA bundle.

Listing 1.1 MANIFEST.MF file from the Hibernate bundle

```
Manifest-Version: 1.0
Created-By: 1.5.0_13-119 (Apple Inc.)
Import-Package: antlr;version="[2.7.6, 3.0.0)",antlr.collections;versi
on="[2.7.6, 3.0.0)",antlr.collections.impl;version="[2.7.6, 3.0.0)",c
# 255 more bundles
rnate.dialect,org.hibernate.dialect.function,org.hibernate.engine,org
.hibernate.property,org.hibernate.type,org.xml.sax"
Bundle-Version: 3.2.6.ga
Bundle-Name: JBoss Hibernate Object-Relational Mapper
Ant-Version: Apache Ant 1.7.0
Bundle-Vendor: SpringSource
Bundle-ManifestVersion: 2
Implementation-Title: Hibernate3
Bundle-SymbolicName: com.springsource.org.hibernate
Implementation-Version: 3.2.6.ga
Hibernate-Version: 3.2.6.ga
Implementation-Vendor: hibernate.org
```

As you can see, the work that SpringSource have put into creating the SpringSource Enterprise Repository is enormous. All you have to do is to search the repository for the bundle you need or, if you use Ivy or Maven to build your applications, you use the <dependency> element to download the appropriate bundle.

Apart from the vast collection of OSGi bundles representing common Java EE libraries, the dm Server includes support for applications as set of bundles. It also allows you to specify the type of code the bundle contains. For example, you can mark a bundle as web bundle. The dm Server will

Please post comments or corrections to the Author Online forum at:

<http://www.manning-sandbox.com/forum.jspa?forumID=508>

direct HTTP requests to the web bundles and you will be able to easily create OSGi web applications. Finally, the dm Server allows you to import all packages from a bundle: all 249 `Import-Package` lines can now be replaced by only 15 `Import-Bundle` lines.

1.3.2 Core components of the dm Server

Now that we have covered the reasons why the OSGi extensions in dm Server make application development easier, we need to take a look under the dm Server's hood. The dm Server makes the most of existing Java frameworks and tools. The core libraries the dm Server uses are the Spring Framework, Apache Tomcat, OSGi R4.1, Equinox and Spring Dynamic Modules for OSGi. Because these are the core components of the dm Server, we are going to take a look at every one of them in more detail.

SPRING FRAMEWORK

At its core, the Spring Framework is a lightweight inversion of control container. Applications that follow the IoC principle use configuration that describes the dependencies between its components. It is then up to the IoC framework to satisfy the configured dependencies. The "inversion" means that the application does not control its structure; it is up to the IoC framework to do that. Consider an example where an instance of class `Foo` depends on an instance of class `Bar` to perform some kind of processing. Traditionally, `Foo` creates an instance of `Bar` using the `new` operator or obtains one from some kind of factory class. Using the IoC technique, an instance of `Bar` (or a subclass) is provided to `Foo` at runtime by some external process. This injection of dependencies at runtime has sometimes led to IoC being given the much more descriptive name dependency injection (DI).

Spring's DI implementation puts focus on loose coupling: the components of your application should assume as little as possible about other components. The easiest way to achieve loose coupling in Java is to code to interfaces. Imagine your application's code as a system of components: in a web application, you will have components that handle the HTTP requests and then use the components that contain the business logic of the application. The business logic components, in turn, use the data access objects (DAOs) to persist the data to a database. The important concept is that each component does not know what concrete implementation it is using; it only sees an interface. Because each component of the application is aware only of the interfaces of the other components, we can switch the implementation of the components (or entire groups or layers of components) without affecting the components that use the changed components. Spring's DI core uses the information from your application's configuration files to satisfy the dependencies between its components.

Apart from the dependency injection core, the Spring Framework brings extensive support for aspect-oriented programming as well as support for major components of Java EE applications. The enterprise component support includes data access, remoting, scheduling, Enterprise Java Beans, JMS, JMX. A large component of today's Java EE applications use web interface. You can take advantage of Spring's own Model-View-Controller implementation as well as Spring WebFlow in the implementation of your web applications.

We are not going to cover the Spring Framework in detail in this book; if you want to refresh your knowledge of Spring, consider reading *Pro Spring 2.5* by Jan Machacek, Aleksa Vukotic, Anirvan Chakraborty and Jessica Ditt.

TOMCAT

Tomcat implements the Java Servlet and JavaServer Pages specifications. It is widely used throughout the Java world; Tomcat contributors claim that it runs in at least 100,000 production environments. Because of its reliability and performance, the dm Server uses Tomcat to deploy the web applications.

The dm Server uses Tomcat to deploy web bundles. As you can imagine, a web bundle is a bundle that contains web application code (controllers, JSP files) and it processes HTTP requests and produces HTTP responses.

OSGI R4.1

There were many efforts to keep the modularization from the source code to deployment and runtime. The most notable one in the Java world is the OSGi framework (which used to stand for Open Services Gateway initiative, but became just OSGi). The dm Server uses the latest specification of the OSGi release.

We are only going to cover the basic concepts of the OSGi platform in Chapter 2. If you are interested in reading about OSGi in detail, take a look at *OSGi Service Platform* by OSGi Alliance,

Please post comments or corrections to the Author Online forum at:

<http://www.manning-sandbox.com/forum.jspa?forumID=508>

which covers OSGi Release 3, but will give you good background knowledge of the specification.

EQUINOX

There are several implementations of the OSGi platform: the most notable ones are Felix and Equinox. Felix is an OSGi R4.1 implementation from Apache. Equinox is an implementation of the same specification of the OSGi platform by the Eclipse Foundation.

The dm Server uses Equinox, whose aim, apart from being the OSGi platform implementation, is to provide additional bundles and services to allow other projects to build on it. The Eclipse platform as well as the dm Server use Equinox as their OSGi platform implementations.

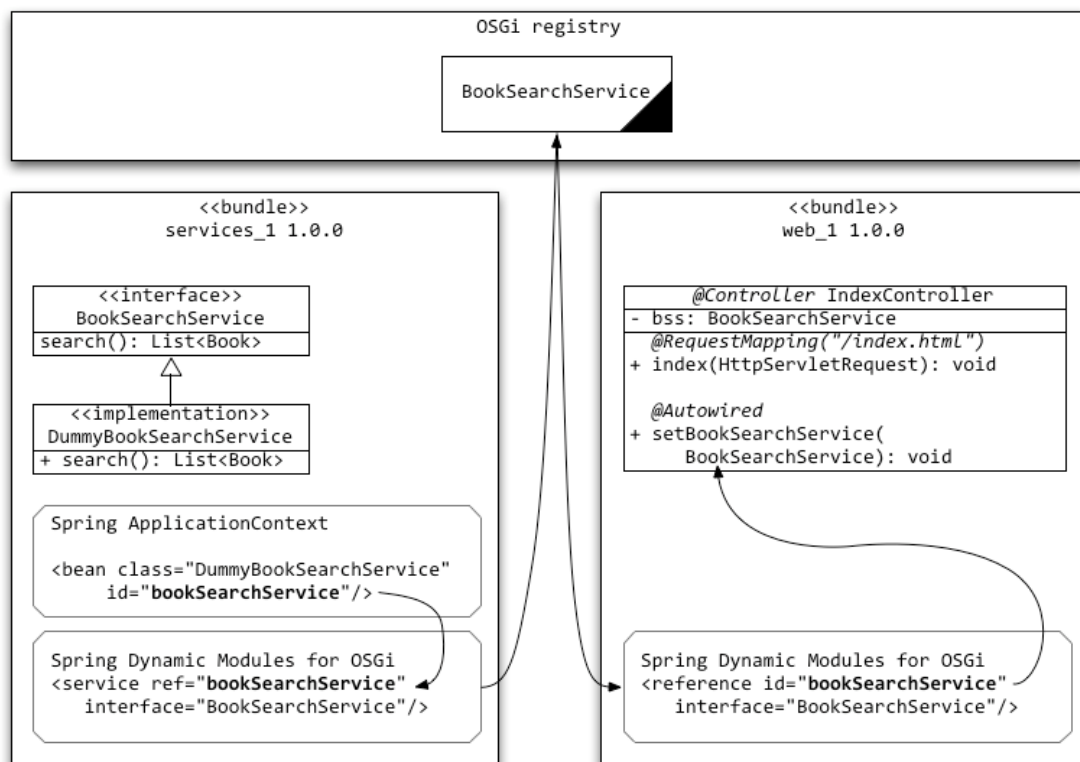
SPRING DYNAMIC MODULES FOR OSGi

The Spring Dynamic Modules for OSGi simplifies development of applications written using the Spring Framework in an OSGi runtime. The combination of OSGi execution environment and SDM allows us to:

- Keep the logical structure of our application at runtime,
- Use and control the versioning of the bundles that make up our applications,
- Discover services exposed from other bundles and to use these services,
- Make the services available as regular Spring beans in Spring applications.

In short, Spring Dynamic Modules for OSGi allows you to export a Spring bean as OSGi service and to import an OSGi service and expose it as Spring bean. Let's take the web and services bundles in Figure 1.6.

Figure 1.6 Services and web OSGi bundles with Spring Dynamic Modules for OSGi



As you can see, both bundles use Spring Dynamic Modules for OSGi. The services bundle's Spring application context contains the `bookSearchService` bean. The SDM exposes the `bookSearchService` bean in the OSGi registry. The SDM in the web bundle discovers a service that implements the `BookSearchService` interface in the OSGi registry and exposes it as the `bookSearchService` Spring bean.

You can fine-tune the SDM's behaviour. We will mention just one to wet your appetite: we can set a timeout in which the referenced service must become available. If the referenced service is unavailable (or becomes unavailable), then the SDM will wait the specified time before failing. This

Please post comments or corrections to the Author Online forum at:

<http://www.manning-sandbox.com/forum.jspa?forumID=508>

allows us to dynamically remove, update and start dependent services at runtime, without the need to restart all bundles that make up the application. The application that uses the dependent bundle will simply wait until the dependent bundle becomes available again or the timeout expires.

Dynamic updates

This is the magic you saw in many presentations of the Spring Dynamic Modules for OSGi: the web bundle was using a `services` bundle. The administrators then stopped the `services` bundle and replaced it with another implementation. While they were doing that, the web bundle was still running, but the calls to the service referenced from the `services` bundle were being queued. As soon as the new `services` bundle became available and the referenced service appeared in the OSGi registry, the web bundle started using the updated service.

The combination of Spring Dynamic Modules for OSGi and the dm Server gives you the ability to create flexible and manageable applications.

1.3.3 dm Server's licensing model

Apart from the technological aspect, the second most important aspect of selecting a new application server is its licensing model.

The source code of the community edition of the dm Server uses the GNU Public License Version 3.0 ("GPL"). So, what does this mean to you as developers and what does it mean if you wanted to use the community edition of the dm Server in production environment?

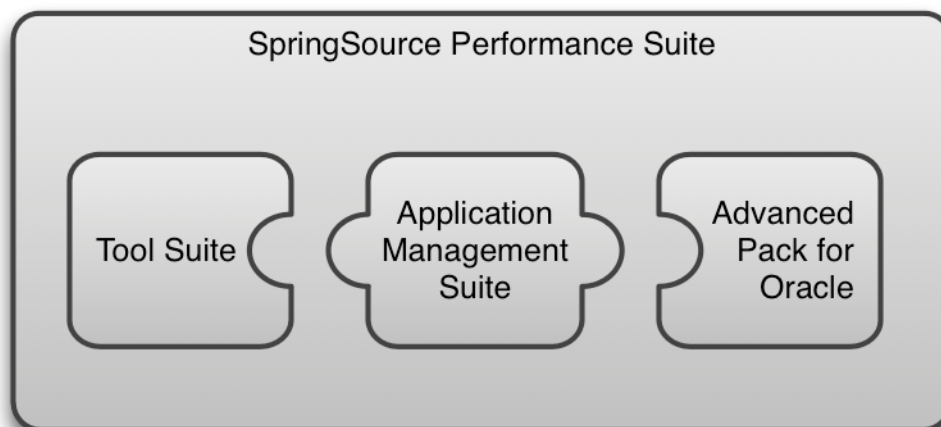
Focusing on the dm Server, the implications of the GPL license mean that the dm Server community edition is free for you to use and modify, however, if you release the modified version of the dm Server, you must also use the GPL license. Remember that you do not have to make your modified version of the dm Server publicly available, if that is the case, you do not need to release its source code.

Therefore, you can use the community edition of the dm Server in production in a commercial environment without having to worry about licensing costs. You can also modify the dm Server, but you cannot commercial applications that you release on its source code.

1.4 SpringSource Performance Suite

Let's leave the dm Server and continue with the next component in the SpringSource Application Platform: the SpringSource Performance Suite. Figure 1.7 shows the components of the Performance Suite.

Figure 1.7 SpringSource Performance Suite and the dm Server



The suite brings easier application development, improved application performance, reliability and monitoring support. Its three components are the SpringSource Tool Suite, SpringSource Application Monitoring Suite and SpringSource Advanced Pack for Oracle.

Please post comments or corrections to the Author Online forum at:

<http://www.manning-sandbox.com/forum.jspa?forumID=508>

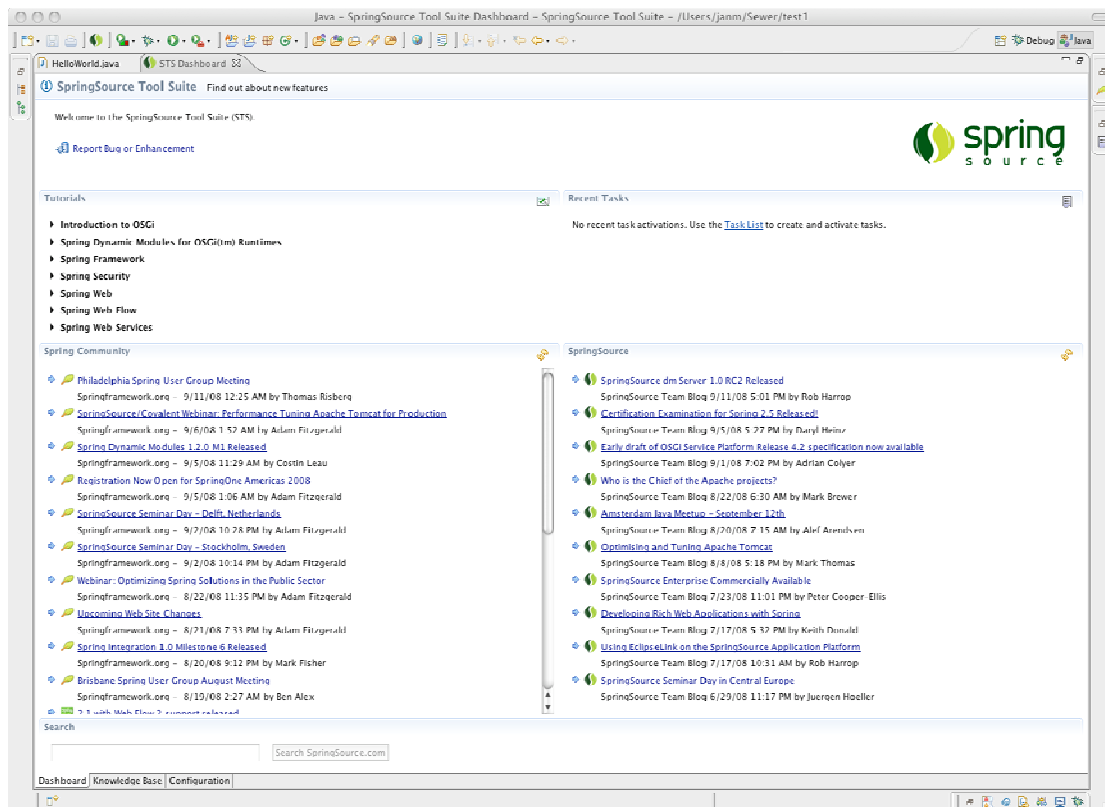
1.4.1 SpringSource Tool Suite

Let's begin with the Tool Suite – a tool developers will use most frequently. It is a Spring-focused IDE built on the Eclipse platform. It assists developers with implementation of their Spring Framework applications. SpringSource Tool Suite adds design, architectural review and runtime problem analysis tools to the Eclipse environment. The key benefits of the Tool Suite are:

- Familiar and easy to use. Because the Tool Suite builds on the well-known Eclipse platform, most Java EE developers will be able to start using it immediately.
- Mylyn task-based approach. The task-based approach allows you to focus your attention even if you are working on complex enterprise applications. The Tool Suite also automatically adds the necessary tasks when you use its wizards. Mylyn is an existing addition to the Eclipse platform. It keeps track of things to do (in source code as well as in the Eclipse platform; more generally, it is aware of the context of each task). This helps you stay on top of the sometimes complex series of tasks you need to complete.
- Architectural and best practice reviews. You can take advantage of the SpringSource's experience in application design and implementation.
- Access to the community knowledge. The Tool Suite's newsfeeds and knowledge base search tools allow you to solve the problems you may face quickly and efficiently.
- Wizards and visual development. The wizards and visual tools will give the beginners the boost they need to start writing Spring applications. The architects will be able to get a high-level overview of even the most complex applications.
- Excellent learning resource. The Tool Suite includes tutorials and examples that cover all aspects of programming using the Spring Portfolio.

Figure 1.8 shows the community and tutorial sections concentrated in the STS Dashboard: a view that shows the latest Spring community and SpringSource news. It also provides a selection of tutorials to get you started with the Spring Framework, Spring Dynamic Modules for OSGi and many others. Finally, it shows a view of the active tasks (see Figure 1.8).

Figure 1.8 SpringSource Tool Suite STS Dashboard



We will be using the Tool Suite throughout the book to develop the sample applications.

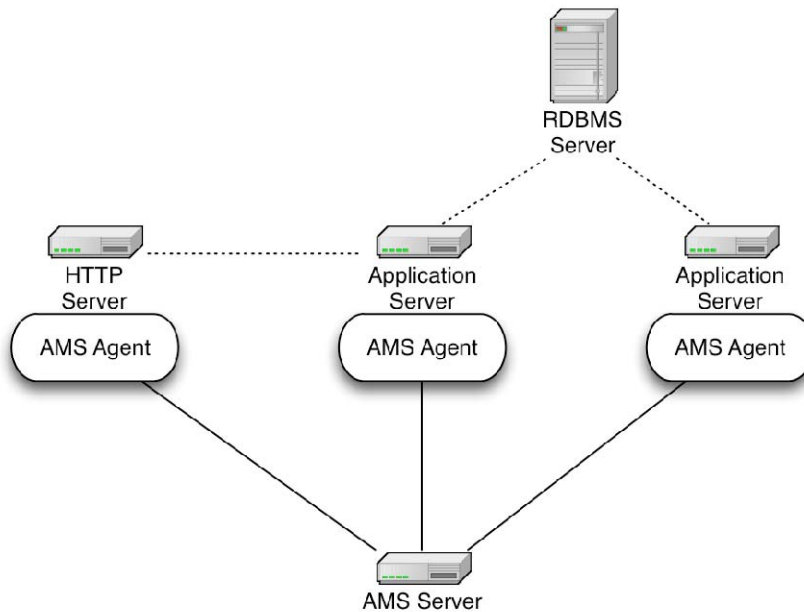
Please post comments or corrections to the Author Online forum at:

<http://www.manning-sandbox.com/forum.jspa?forumID=508>

1.4.2 SpringSource Application Management Suite

The Application Management Suite (AMS) is a tool that offers the administrators and developers a view on the performance and health of the platforms running their applications. The AMS consists of the server and the agents. To install the AMS, you need to install a server and then install an agent on each server that you want to monitor. The server communicates with the agents to set up the monitoring parameters; the agents then report back to the server. Figure 1.9 shows the logical setup of the AMS.

Figure 1.9 AMS platform setup



This figure shows that the AMS server agents can be installed on any server, even if it is not running a Java EE application server. In the example in Figure 1.8, you can see that there is the AMS Agent on a server that is running the Apache HTTP server.

The AMS Server can therefore monitor a number of servers and a number of parameters on each server. The AMS can monitor servers and services. Table 1.1 summarizes the servers AMS can monitor.

Table 1.1 Supported server types

Server	Description
Apache HTTP Server	Apache HTTP server is the world's most popular HTTP server; it often acts as front-facing HTTP server for Java EE web applications
Apache Tomcat	Apache Tomcat is popular implementation of the Java Servlet and JavaServer Pages. It is an excellent alternative to most heavy-weight Java EE application servers
SpringSource ERS	The Apache Enterprise Ready Server is an enterprise edition of the Apache HTTP Servers 1.3 and 2.x; it adds failover and clustering features to the standard Apache HTTP server.
ColdFusion	Adobe's ColdFusion is an application server that uses the ColdFusion Markup Language to implement server-side functionality
Geronimo	Apache Geronimo is a fully certified Java EE 5 application server.

Please post comments or corrections to the Author Online forum at:

<http://www.manning-sandbox.com/forum.jspa?forumID=508>

	The latest release of Geronimo brings clustering and excellent GShell for server's administration
iPlanet Application Server	iPlanet is a Java EE application server based on the Netscape Application Server and NetDynamics Application Server
JBoss	JBoss is an open-source project that combines a Java EE application server with other enterprise frameworks such as workflow engine and a rules engine
JRun	Adobe's JRun is a Java EE application server. Along the standard Java EE application server features, it comes with Adobe Flash Remoting support
Resin	Caucho Resin is a high-performance Java EE application server with interesting support for virtual hosts and clustering
ServletExec	New Atlanta's ServletExec is an Java EE application server with JSF 1.2 support
dm Server	SpringSource's own dm Server is an OSGi-based Java EE application server
Sun JVM 1.5	Sun JVM is not an application server, but the JVM provides comprehensive management interface that the AMS can use
WebLogic (including Admin and NodeManager)	WebLogic is a complex Java EE application server with support for many enterprise features and its own JRockit JVM.
WebSphere (including Admin)	WebSphere is IBM's implementation of a Java EE application server

You can see that the servers are not limited to Java EE application servers. In fact, the only thing these servers have in common is that they provide some management interface. However, monitoring servers only cannot give you a full overview of the monitored platform. The AMS completes the view by allowing you to monitor services (see Table 1.2).

Table 1.2 Service monitoring support

Operating System services	Network Services
CPU load	Reply from a ping to an internet address
Filesystem (directory, directory tree, file and mounts)	Open TCP socket
Network interface	Standard TCP protocols (DHCP, DNS, FTP, HTTP, IMAP, LDAP, POP3, RPC, SMTP, SSH)
One or more processes	Standard UDP protocols (DHCP, DNS, HTTP, IMAP, RPC, NTP, SNMP)

Because you can monitor low-level services as well as the various servers, the AMS gives you an excellent overview of your running applications. Figure 1.10 shows AMS dashboard with an overview of the monitored services.

Figure 1.10 AMS Dashboard

Please post comments or corrections to the Author Online forum at:

<http://www.manning-sandbox.com/forum.jspa?forumID=508>

SpringSource AMS

Recent Alerts : (There are no alerts in the last 2 hours.)

hadmin - Logout Help

Dashboard Resources Analyze Administration

Platforms > MacOSX > macpro Platform

Description: Mac OS X Leopard
 Default Gateway: 192.168.1.254
 IP Address: 192.168.1.66
 OS Version: 10.5.5

Owner: HQ Administrator (hadmin) - Change...
 Vendor: Apple
 Primary DNS: 192.168.1.254
 RAM: 2048 MB

Vendor Version: 10.5
 CPU Speed: 4 @ 2660 MHz
 Architecture: i386

Tools Menu

MONITOR INVENTORY ALERT VIEWS

RESOURCES INDICATORS METRIC DATA

Platform Services Health Avail

CPU
 FileServer Mount
 NetworkServer Interface

Deployed Servers Health Avail

dm Server
 macpro AMS Agent 1.0.1
 macpro AMS Tomcat 5.5

Show Metrics of Categories

Availability Utilization
 Throughput Performance

Show Metrics with Value Types

Dynamic Trends Up
 Trends Down Static

Keyword Search:

Metric Display Range: Last 8 Hours Advanced Settings...

Alerts	OOB	LOW	AVG	PEAK	LAST	Collection Interval
0	0		100%		✓	00:01:00
0	0	31.9 MB	31.9 MB	31.9 MB	31.9 MB	00:05:00
0	0	1.3	1.3	1.3	1.3	00:05:00
0	0	2.0 GB	2.0 GB	2.0 GB	2.0 GB	00:05:00

The AMS does not only perform passive monitoring of the servers, it also allows you to control them. Figure 1.11 shows that you can use the AMS server to start an instance of SpringSource dm Server.

Figure 1.11 Using AMS console to start dm Server

Current Status

Control Action: start
 Command State: ✓ Completed
 Command Status: macpro AMS Tomcat 5.5
 Elapsed Time: 4.096s
 Clear Status Detail

Description:
 Date Started: 09/20/2008 03:15:38 PM
 Date Scheduled: 09/20/2008 03:15:37 PM

Quick Control - Quick Control Actions will occur after the current Control Action.

Control Action: Select...
 Control Arguments (optional):
 Quick Control Actions will be done in parallel to all resources.

Control Action Schedule - Click "New..." below to schedule a Control Action.

Control Action	Next Fire	Date Scheduled	Description
NEW...			

Total: 0 Items Per Page: 15

In addition to manual control, you can configure automatic error recovery procedures. For example, you can configure the AMS to restart an instance of dm Server should it fail. This way, if you have clustered deployment of dm Servers, the AMS can ensure that your application can recover from failure. If your applications use the Oracle RDBMS, you can improve their robustness by using the SpringSource Advanced Pack for Oracle.

1.4.3 SpringSource Advanced Pack for Oracle

The SpringSource Advanced Pack for Oracle brings support for the features in Oracle 10g and above that are not accessible using standard JDBC calls. This means that with the Advanced Pack for Oracle, you can now use Oracle's message queuing, XML types and custom data types. The Advanced Pack for Oracle also allows you to replace the standard DataSource implementation with the OracleDataSource. The OracleDataSource brings additional features, for example fast connection failover. The Advanced Pack for Oracle includes aspect-oriented programming support for transparently re-trying same operation in a new transaction should the original transaction fail.

You do not need the Advanced pack for Oracle if you are not using Oracle (naturally!) or if you don't need to use any of the advanced features Oracle supports. You won't need the Advanced Pack for Oracle to compile and run the examples in this book

1.5 Spring Enterprise Edition

The last component of the SpringSource Application Platform is the Spring Enterprise Edition. It is an

Please post comments or corrections to the Author Online forum at:

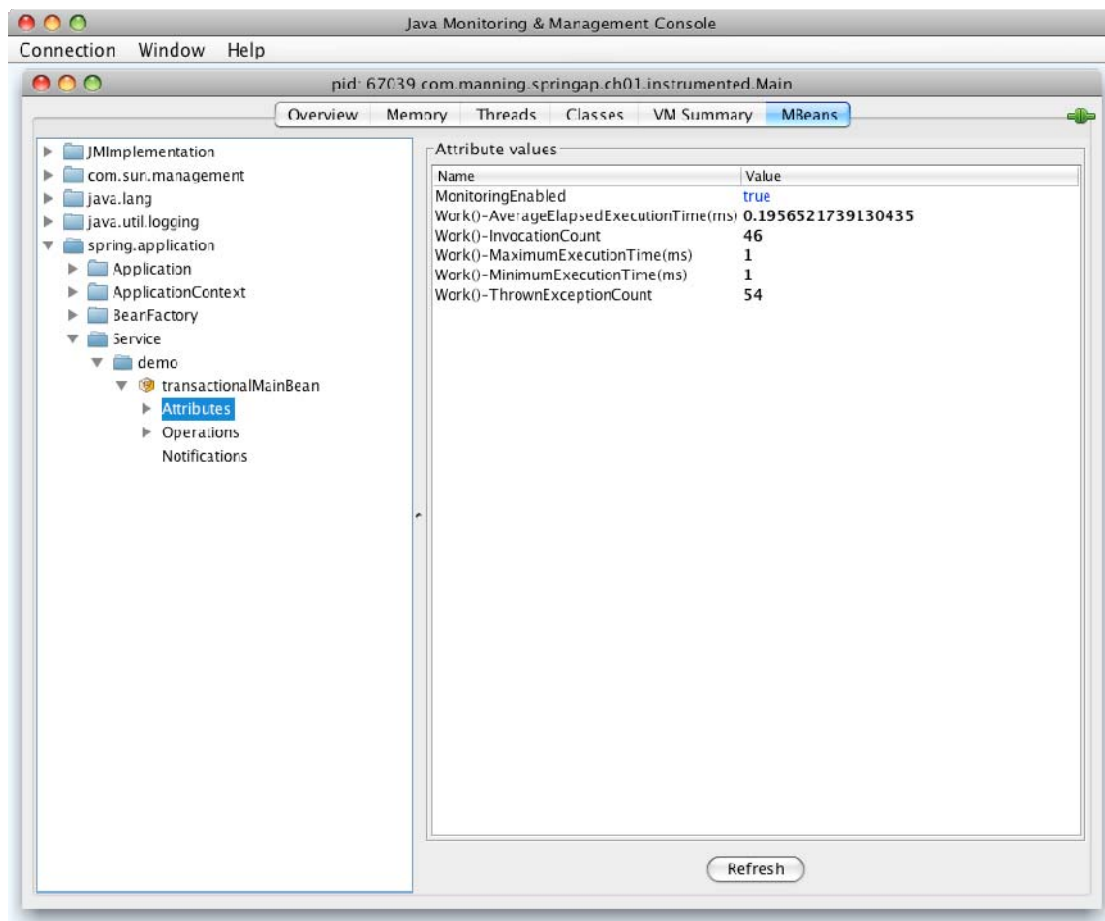
<http://www.manning-sandbox.com/forum.jspa?forumID=508>

instrumented and quality-assured version of the core projects in the Spring Portfolio: the Spring Framework, Spring Security, Spring Web Flow, Spring Web Services, Spring Dynamic Modules for OSGi, Spring Batch, Spring LDAP and Spring Integration.

The instrumented code is like scaffolding around the Spring Framework in your application. The non-instrumented, standard version of the Spring Framework does not record any information about its execution. The instrumentation code exports management information about your application as JMX MBeans. The Application Management Suite uses the JMX MBeans to give you a view on your application's status. If you do not want to use the Application Management Suite, you can use any JMX client.

The instrumented Spring Framework applies the monitoring code (advices) to your stereotyped components (components with the `@Controller`, `@Repository`, `@Transactional`, `@Component` and `@Service` annotations). Under the hood, when the instrumentation discovers these components, it wraps them in a Spring AOP proxy (or adds a `MethodInterceptor` to an existing proxy). The advice collects common statistics for method execution. Figure 1.12 shows the JMX console with a trivial Spring application.

Figure 1.12 JMX console view of an instrumented Spring application



You can see that JConsole can display details about the Spring application. You can find statistics on performance and health of your application's stereotyped components and you can see details about the Spring Framework's `BeanFactory` and `ApplicationContext`.

To get this detailed window into your Spring applications, you only need to replace the un-instrumented Spring Framework by the instrumented version. The combination of the dm Server and applications that use instrumented Spring Framework will give the Application Management Suite all information it needs to actively monitor your applications. While it is not necessary to use the Application Management Suite or the instrumented Spring Framework, it will help managing mission-critical systems.

Please post comments or corrections to the Author Online forum at:

<http://www.manning-sandbox.com/forum.jspa?forumID=508>

1.6 Summary

In this chapter, we looked at the features of the SpringSource Application Platform. We explored the components of the Application Platform: the dm Server and the SpringSource Performance Suite, which contains the SpringSource Tool Suite, SpringSource Application Management Suite and Spring Enterprise Edition.

You know that the dm Server is an OSGi application server that can help you overcome most of the challenges of traditional Java EE programming and deployment models. To help you take full advantage of dm Server's features, you can use the SpringSource Enterprise. The SpringSource Enterprise is a set of tools and libraries for Java enterprise development.

The SpringSource Enterprise component you will use most frequently is the SpringSource Tool Suite – an Eclipse-based development environment. Once your applications are running in the dm Server, you can monitor them in another component of the SpringSource Enterprise: the SpringSource Application Management Suite. To allow the Application Management Suite gather as much information about your application's health and performance, you can use the Spring Enterprise Edition. The Spring Enterprise Edition is a version of the Spring Framework that includes performance and health monitoring code. We have mentioned the SpringSource Advanced Pack for Oracle, which gives you access to the advanced features of the Oracle RDBMS that are not available in JDBC.

Read on to find out how to download and install the SpringSource Application Platform and how to get the dm Server set up. In the next chapter, we will write our first dm Server applications.