SAMPLE CHAPTER

OSGi IN DEPTH

Alexandre de Castro Alves

FOREWORD BY David Bosschaert

MANNING

OSGi in Depth

by Alexandre de Castro Alves

Chapter 1

brief contents

1 OSGi as a new platform for application development

This chapter covers

- Underlying concepts of development platforms
- OSGi technology, including the framework and the enterprise services
- The benefits of using OSGi for the development of enterprise-grade applications
- The relation of Enterprise OSGi to Java Standard Edition and Java Enterprise Edition
- The current OSGi players in the market

We've all used development platforms in the past, such as Java Enterprise Edition (JEE), and even though there have been great advances in this industry, we're still building large complex systems, which are hard to develop, maintain, and extend.

OSGi provides a new development platform, based on modular decoupled components and a pluggable dynamic service model. In this book, you'll learn that OSGi is the ideal platform for the development of full-fledged, enterprise-grade, maintainable applications. Furthermore, we'll look in depth at how OSGi applications

can use a plethora of carrier-grade infrastructure services, such as HTTP, configuration, deployment, event handling, transactions, persistence, RMI, naming and directory services, and management.

We'll start this chapter by exploring development platforms and the benefits of using such platforms to develop software. We'll then discuss the requirements of a platform intended for the development of enterprise-grade applications. Next, we'll focus on the OSGi technology, expanding into its core pieces, called the OSGi framework, and its enterprise services. Finally, you'll learn why OSGi is a good fit as a development platform, particularly in light of existing solutions, such as JEE. Let's start by examining the basics.

1.1 What are development platforms and application frameworks?

In the context of software development, a *development platform* is a set of software libraries and tools that aid in the development of software components, and the corresponding runtime environment that can host these developed components, as shown in figure 1.1.

The *runtime environment* may consist of the hardware, operating system (OS), and supporting runtime libraries. One example of a runtime environment is the Java Runtime Environment (JRE), which includes the Java virtual machine (JVM) that isolates the developer from the details of the underlying OS and hardware.

Software frameworks are specialized types of a development platform's libraries and tools. Wikipedia defines a software framework as an "abstraction providing generic functionality that can be selectively specialized to provide specific functionality."

Particularly interesting to us are application frameworks. An application framework is a type of software framework whose purpose is to provide a structure for the creation of software applications. Applications are programs that allow users to perform related tasks together. Examples of software applications are document editors and antivirus software.

Putting it all together, a development platform allows a developer to create applications and to host these applications so that end users can use them. Throughout this book, it's important to keep these two players in mind: the developer (you) and

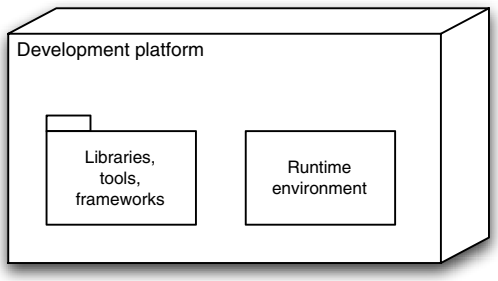

Figure 1.1 A development platform consists of a software framework and its supporting runtime environment.

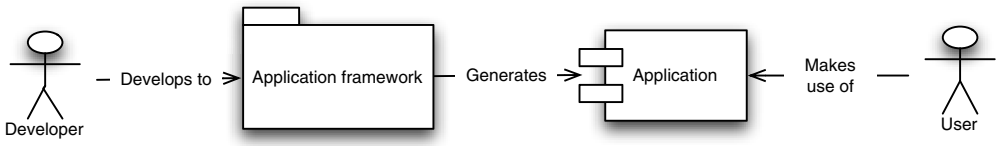

Figure 1.2 A developer creates an application, which is used by a user, and develops to a framework.

the end user, as illustrated in figure 1.2. Software development platforms are also called toolkits or SDKs (software development kits).

Historically, development platforms have always played an import role in software. The Java platform, also known as Java Standard Edition (JSE), is one example. In this case, the Java development kit (JDK) provides the software framework, and the Java Runtime Environment (JRE) provides the runtime environment. The OSGi Service Platform, which is the subject of this book, is another example of a development platform. The OSGi Service Platform uses the JRE as its runtime environment. In addition, it provides an application framework layered on top of the JDK. We'll look into its details throughout this book, but first let's see why it's important to use a development platform to begin with.

1.1.1 Why use a development platform?

Consider the following definition of the word *framework*:

> *"an essential supporting structure of a building, vehicle, or object"*

Why is the supporting structure of a vehicle important? It clearly sounds like it's important, but let's see if I can articulate why that's the case. I can think of two main reasons:

- It guarantees that I'm sitting on top of something that's solid—something that has been designed properly, implemented suitably, and tested thoroughly. A framework helps to decrease defects. This is a runtime characteristic.
- It gives the manufacturer an opportunity to reuse the frame for different vehicles. A framework helps to improve productivity through reuse. This is a design-time characteristic.

It's no different for development platforms. Development frameworks allow the creation of new applications in a form that's both efficient and has a high degree of quality.

1.1.2 Enterprise platforms

Enterprise platforms are development platforms that support the creation of enterprise applications—applications that implement business processes, business logic, or business integration to an enterprise. Examples of enterprise applications are a loan-approval application, an order-processing application, a customer relationship management (CRM) application, and a travel management application.

The following two aspects characterize enterprise platforms:

- Enterprise platforms provide a collection of infrastructure-level utilities and services common to many businesses and industries, such as management, directory service, monitoring, and distribution.
- Enterprise platforms must scale, perform efficiently, and be robust and fault tolerant.

Following up on our previous example, Java also has an enterprise version, which is called Java Enterprise Edition (JEE). Other examples of enterprise platforms are Microsoft's .NET Framework, SpringSource's tc Server Development Edition, and to some extent, Google's Web Toolkit. Recently, a new enterprise platform has been developed, the OSGi Service Platform Enterprise Specification.

As you can see by the number of players, enterprise technology is quite mature, so why is there a need for a new platform, such as the one being provided by OSGi? We'll address this question in section 1.3, but before we can do that, you need to understand OSGi a bit better.

1.2 The OSGi technology

The Open Service Gateway initiative (OSGi) was formed in March 1999 by a consortium of leading technology companies with the mission to define a *universal integration platform for the interoperability of applications and services.*

When I first read their mission, it gave me the impression of being both overly complex and somewhat outdated. Hadn't people already created a universal platform for applications? As you'll learn, no one has been able to do it successfully.

1.2.1 The problem domain

First, let's investigate the underlying problem that these companies were facing. The initial members of the OSGi alliance were in a large part telecommunication equipment manufacturers and service providers. They were interested in deploying software applications on small-memory devices. For example, consider a mobile phone as the device and a location-tracking application and an advertisement application as the software applications being deployed to the mobile phone. The location-tracking application uses the mobile phone to verify the current location of the subscriber and informs the advertisement application of the location. The app then retrieves selected advertisements that are suitable to the current location of the subscriber, such as promotions from nearby restaurants, as shown in figure 1.3.

This seemingly simple interaction caused the equipment and service providers several interesting problems. First, the devices tend to have different hardware and thus different programming APIs. Hence, each vendor had to program its applications to a specific device and then port to other devices. Second, not only do the various hardware devices use different programming APIs, but there's also a large variation in their functions and capabilities. Some have more memory than others; some have a disk whereas others are completely diskless; some have GPS and some do not. Third, the

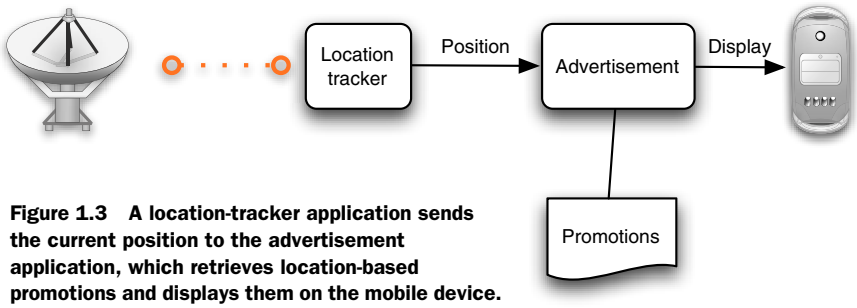

Figure 1.3 A location-tracker application sends the current position to the advertisement application, which retrieves location-based promotions and displays them on the mobile device.

lifetime of these devices is generally between one to two years, which means that new applications are likely to be created during this period, partly because of changing market demands. These new applications need to be dynamically deployed to the devices and join the existing collaborating applications that are currently running in the devices. And finally, because of the scarcity of the resources, these applications needed to closely cooperate with each other in a concise and, more important, lightweight manner.

This is no simple matter after all. Is there an existing universal platform that could help us, or does one need to be created?

Again, let's go through the problems.

PROBLEM: COPING WITH DIVERSE PROGRAMMING APIS

The first problem is portability. We need a single programming platform that abstracts the application from the underlying operating system and the hardware. In other words, we need a virtual machine. Does anything come to your mind? Yes, of course. Let's use Java to solve our first problem.

PROBLEM: VARYING DEVICE CAPABILITY

The second problem is subtler. Let's consider a specific case. The advertisement application can retrieve the available promotions from different sources. If the device has ample bandwidth, the data source could be remote. If the device doesn't have enough bandwidth but has a disk, then the promotions could be retrieved in the background and cached in the local disk as the subscriber enters a location, as shown in figure 1.4.

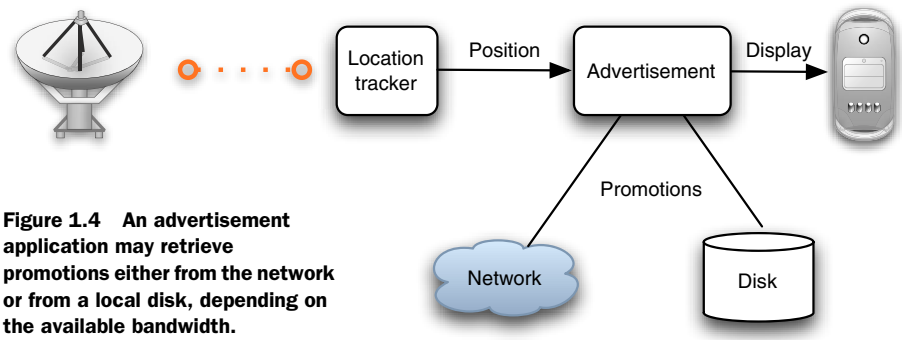

Figure 1.4 An advertisement application may retrieve promotions either from the network or from a local disk, depending on the available bandwidth.

There's a clear service contract between the advertisement application and the data source, but the implementation of this contract will vary depending on the device. This universal platform must make it easy for applications to decouple service contracts from the service implementation. The Standard Edition of Java (JSE) doesn't have a service registry or a service management framework that could help us with this. Hence, this facility either needs to be implemented from scratch, or we could try to borrow something from the Enterprise Edition (JEE) of Java. Let's hold on to this thought and tackle it after going through the other problems.

PROBLEM: SUPPORTING DYNAMIC CHANGES

The third problem can be summarized by the following requirement: the platform must allow the dynamic deployment and undeployment of applications in a secure form. Does JSE have support for this? Not really. You could try to solve this with the Java class loaders, but it wouldn't be easy, and there's no simple way of unloading classes after they've been loaded, not to mention that there's no concept of an application deployment unit. The closest concept to an application deployment unit is the idea of JARs (Java Archives), but JARs by themselves don't provide all the metadata that's needed, such as a unique naming schema for the applications.

As with the previous problem, we can implement our own solution for dynamic deployment of applications or try to leverage something from JEE. For example, web servers do have the concept of web applications, which are defined as part of a WAR (Web Archive) deployment unit file.

PROBLEM: PROVIDING A LIGHTWEIGHT SYSTEM

This brings us to the last problem. Whatever solution we pick, it must be lightweight. Yes, we could try to leverage a directory service such as JNDI from JEE or leverage the architecture from web servers, but these solutions would fail to consider the size and memory constraints enforced by the devices onto the platform, making it less suitable for embedded solutions and not a viable option.

1.2.2 The solution: a dynamic module system for Java

Java addresses some of the problems we've discussed, such as portability, but not all of them. For instance, it lacks proper support for dynamic service management.

Enter the OSGi Service Platform. In its most succinct definition, the OSGi Service Platform, or OSGi platform for short, is a dynamic module system for Java. In OSGi terminology, a Java module is called a *bundle*.

The OSGi Service Platform is composed of two main components, the OSGi framework and the OSGi services, as shown in figure 1.5.

THE OSGI FRAMEWORK

The OSGi framework provides its users with all the pieces that we discussed in the previous section:

- A portable and secure execution environment based on Java
- A service management system, which can be used to register and share services across bundles and decouple service providers from service consumers

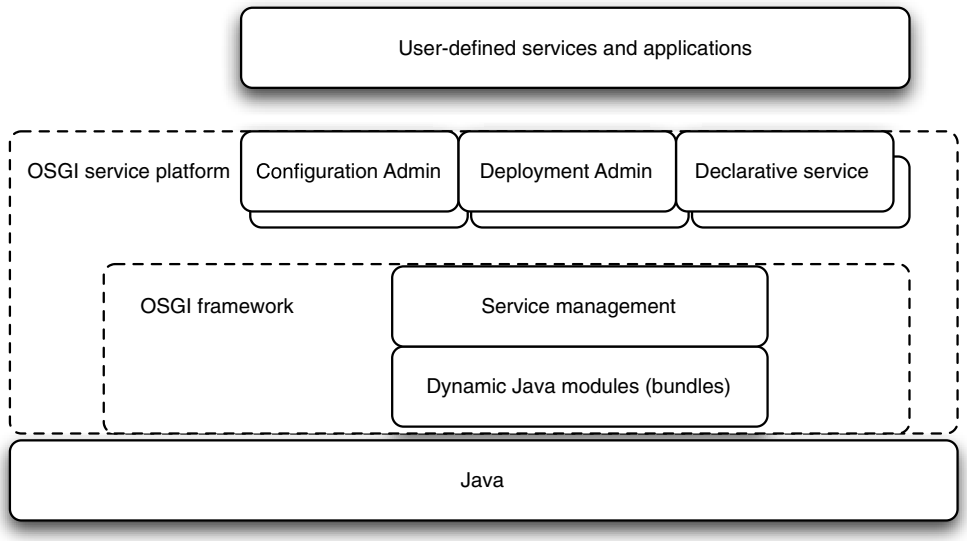

Figure 1.5 The OSGi Service Platform comprises the OSGi framework and the OSGI services.

- A dynamic module system, which can be used to dynamically install and uninstall Java modules, which OSGi calls bundles
- A lightweight and scalable solution

The OSGi framework is the core structure of the OSGi Service Platform. It can be seen as a backplane that hosts bundles, possibly containing services. If you consider that a bundle can be an application, then the definition of the OSGi framework is in accordance with our definition of application frameworks. That is to say, the OSGi framework is an example of an application framework.

Right now, we'll leave the definition of the OSGi framework somewhat loose. Let's not worry about what exactly bundles and services are. We'll discuss the framework, its concepts, and its APIs in detail in the next chapters.

THE OSGI SERVICES

Alongside the OSGI framework, the OSGI Service Platform includes several general-purpose services. You can think of these services as native applications of the OSGi Service Platform.

Some of these services are horizontal functions that are mostly always needed, such as a logging service and a configuration service.

Some are protocol related, such as an HTTP service, which could be used by a web-based application.

And finally, some services are intrinsically tied to the framework, which won't work without them. Examples of these are the bundle wiring, which manages the dynamic module system itself, and the start-level service, which manages the bootstrap process of the framework.

The focus of the initial releases of the OSGi Service Platform had been on the OSGi framework, but gradually we see the OSGi services playing a more prominent role. This trend toward other components, such as services, built on top of the core OSGi framework is a reflection of the increasing popularity of the technology.

1.2.3 The Enterprise OSGi

As can be deduced from its history, OSGi was initially employed in the embedded market. But with its growing popularity and maturity, OSGi is moving to the enterprise market. To address this requirement, the OSGi Enterprise Expert Group (EEG) created the OSGi Service Platform Enterprise Specification (Enterprise OSGi).

This specification combines OSGi services that can selectively be used to provide enterprise functionality. These services can be grouped into enterprise features. Examples of enterprise features, as shown in figure 1.6, are the following:

- *Management and configuration*—This group includes the Configuration Admin service as well as the JMX Management Model service and the Metatype service.
- *Distribution*—This feature allows the communication of end points between remote instances of OSGi framework instances. Some of the services in this group are the Remote Service and the SCA Configuration Type.
- *Data access*—This feature includes support for JDBC, JPA, and JTA and allows the manipulation of persisted objects.

Before getting into the details of the OSGi technology itself, you should understand the high-level benefits of using OSGi as your development platform. We address this topic in the next section.

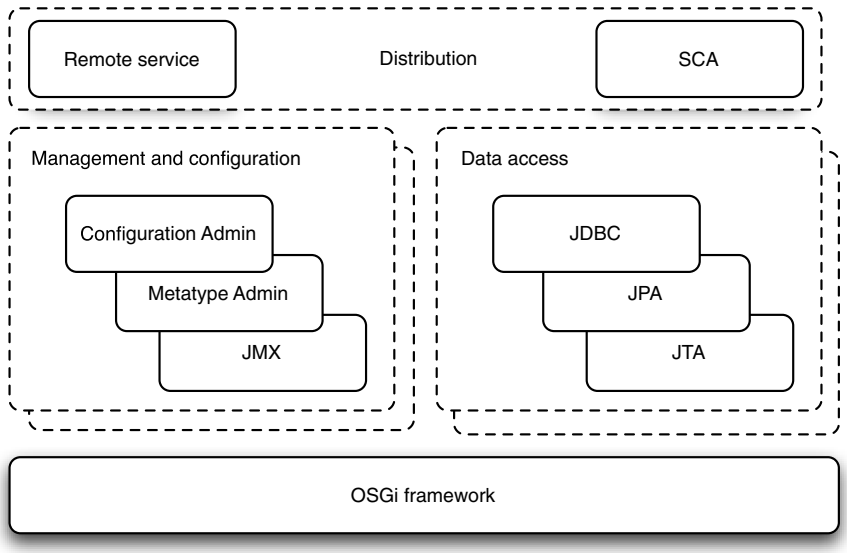

Figure 1.6 Enterprise OSGi consists of the OSGi framework and several OSGi services; together they provide enterprise features.

1.3 Benefits of using the OSGi platform

Why is the OSGi technology a good development platform for Java applications? Even further to the point, why or when is it better than the existing platforms on the market? To answer this question, we need to consider the problems we face when we develop full-fledged carrier-grade applications today.

For simplification, we can categorize these problems into two groups: problems intrinsic to the development of complex applications and problems related to existing development platforms. The problems intrinsic to applications include the following:

- As applications become larger and more complex, they become harder to maintain, sometimes exponentially so!
- Applications are difficult to extend without causing their erosion.

The problems related to existing platforms include these:

- Existing platforms are large, heavyweight systems and thus are complex to learn and use.
- There's a lack of portability among software vendors, making it difficult to reuse or share vendor components, even at the API level.

We'll explore each of these problems individually in the next sections. If the OSGi technology is able to help us address the problems of both of these categories, then not only is it suitable for the development of applications, but it's also a better tool for doing so.

1.3.1 OSGi manages the complexity of large systems

As developers, we've one time or another all faced the problem of complexity. Things are good when we're working by ourselves, on a separate, isolated piece of code. But as the team grows, from one person to ten, and the code grows from a few thousand lines to several hundred thousand, so does the complexity of working with the code and the team, and the bad news is that the increase isn't linear.

I'm positive all of the following will sound familiar:

- A simple change to the implementation of one component causes breakages throughout the application, at apparently dissociated locations.
- No one on the development team knows with certainty whether an interface can be changed without breaking existing clients.
- There are several closely related versions of the same utility functions in the application's source code.

How is the problem of managing large systems related to enterprise applications? Enterprise applications are by their very nature complex systems because of the following two factors:

- Business processes and business logic are inherently complex.

- Enterprise applications need to deal with complex issues such as resilience, management, and distribution.

How can OSGi help you manage the complexity of large systems? OSGi decreases complexity by allowing you to efficiently modularize your code and thus deal with smaller problems one at a time. By designing your code as independent modules that interact collectively to achieve the application's goal, rather than as a single monolithic structure, you're able to apply the millennia-old strategy of *divide and conquer* to your solutions.

Remember that the OSGi framework allows you to define Java modules, or bundles. These bundles have formal versioned interfaces, which must be explicitly referenced by any consuming client. In fact, by defining formal contracts between producers and consumers of code, you're able to decrease the likelihood of experiencing the three problems raised in the preceding paragraphs.

For example, let's look at the first problem again in detail. Consider bundle `B`, which contains three packages: `p`, `q`, and `r`. Packages `p` and `q` contain only implementation classes and don't need to be public, whereas `r` contains public user interfaces, as shown in figure 1.7. There's a bug in class `C` of package `p` (`p.C`) that needs to be fixed. If you were using the OSGi framework, you could have specified that the packages `p` and `q` of bundle `B` are not public. This means that the only code that has visibility to these packages would be within the bundle itself, which would allow you to restrict testing to the bundle and to any consumers of the public package `r` when `p.C` is changed. By modularizing your code, you have better control over it and know what the impact will be when something changes.

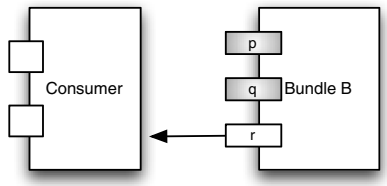

Figure 1.7 Bundle `B` with private packages `p` and `q` and public package `r`, which is being used by a consumer

In this particular case, could you have achieved the same results by meticulously coding and making sure that all Java classes are final, using the least open accessibility modifier (for example, private members), and so on? Perhaps, but would it be efficient or even possible to do these tasks on a large scale, involving several people and thousands of lines of code? Most certainly, it would not.

Furthermore, keep in mind that in OSGi the contract between producers and consumers is specified declaratively, that is, not in Java. This gives you enormous potential for tooling. For example, you could find out the transitive closure of all classes that should be tested when a class is changed.

Nevertheless, as brilliantly stated by Fred Brooks in 1986, there are no silver bullets. It's still the developer's responsibility to design adequately. For example, there'd be no point in using the OSGi framework to achieve modularization and then make everything public. We'll address modularity in the following chapters, so don't worry if the details aren't clear yet. The main point to understand is that the OSGi framework improves modularity, which in turn decreases the complexity of managing large projects and increases reuse.

1.3.2 OSGi provides extensibility without eroding the system

Successful applications commonly need to be extended throughout their lifecycle; this is largely driven by changes to business requirements in today's fast-paced markets.

The problem with extensions is that they open up your system. Extensions are like public APIs, but they're more problematic because people have greater flexibility with extensions than with public APIs. You'll find that people sometimes do the unexpected with extensions.

Extending a system slowly helps erode it.

This is similar to software maintenance. As software ages, fixing bugs becomes harder and harder. Every code change takes longer to make and has a greater potential of causing other problems in the software. The reason for this erosion is that both when adding extensions and fixing bugs you're incorporating new code that wasn't made by the original authors of the software.

How can you restrain the erosion? You have to bind and control the new code. How can OSGi help you with binding and controlling extensions? As you've seen, OSGi defines the concepts of services, service consumers, and service providers.

A service consists of an interface and an implementation. The service consumer only sees and uses the service interfaces, whereas the service provider supplies the service implementations and doesn't interact directly with the consumers.

Generally, extensions of an application framework play the role of service providers, and the actual framework plays the role of the service consumer, as shown in figure 1.8. The service interface defines the contract of the extension; in other words, it's the extension point.

In this case the extension hooks into the lifecycle of the framework, and the framework calls back into the extension when appropriate. One example is an extension that wants to be notified when events of a certain type are received by the framework.

Sometimes extensions also act as the service consumers. The framework still defines the service interface, but it also provides the service implementation, which is then used by extensions.

Regardless of the approach, by keeping the extensions decoupled from the framework as separate service providers or consumers and by having a formal extension contract, you're able to isolate the extension code and thus decrease the overall erosion of the system.

Furthermore, OSGi allows you to dynamically manage the service providers. For example, OSGi supports the shutdown of a service provider that might be misbehaving without impacting the rest of the system.

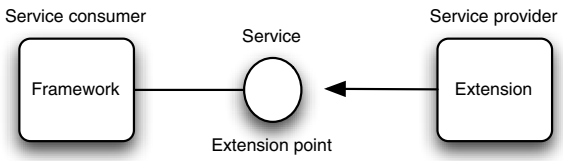

Figure 1.8 Extensions provide services through extension points. The framework consumes the services.

1.3.3 OSGi is lightweight and customizable

As you've seen, the applications you develop can become quite complex. This complexity has historically also been reflected in the development platforms. To support the complexity of full-fledged applications, the development platforms comprise collections of features, APIs, and tools and therefore have become heavyweight and complex themselves. They try to be a one-size-fits-all solution to all the requirements of all businesses.

For example, a loan-approval application may need to interact with a credit-checking system using web services, whereas an order-processing application interacts with its partner using some messaging middleware, such as JMS. Regardless of these different requirements, the existing development platforms include both web services and JMS technologies for both applications' runtimes. Even though the order-processing application uses only JMS, its runtime also ends up paying the price of a web services stack.

This may seem inconsequential initially, but consider that there are dozens of different enterprise technologies, as you've seen in the OSGi services section, and thus a simple hello world application may end up having a runtime that takes megabytes of memory and seconds to minutes to start. No matter what, this complexity leaks out to us developers in different forms. Our iterative development lifecycle becomes slow, we must learn more APIs than we need to use, the programming model becomes complex, and so on. We can all relate to how complicated it is to deploy the simplest of applications to any enterprise platform today.

The OSGi platform addresses this by providing a bare-bones framework, the OSGi framework, to which services can be added a la carte. For example, if you need a web services stack, you can install it; otherwise, it's not present. The OSGi platform can be customized to be as lightweight or as complex as needed. Furthermore, this flexibility shows in different ways; for example, being able to install features dynamically means that the lifecycle of new features provided by software vendors can be shortened. You don't need to wait a year or two for the next version of your application servers; instead, you can download and install new enterprise features by themselves as soon as they become available.

1.3.4 OSGi allows for portability

Java Enterprise Edition and some of the other enterprise platforms are standardized. This means that in theory you should be able to move a JEE application from one JEE application server to another, albeit hinging on the fact that you must use only standard APIs and no vendor extensions. This capability of being able to host an application on a different vendor's application server is called *application portability*.

Yet, there's another level of portability, not always mentioned, which is that of vendors' features themselves. For example, wouldn't it be nice to be able to use a vendor's JMS implementation with another vendor's web services stack in a single container? Why would you want to do this? Among other reasons, here are the three main ones:

- It allows you to pick and choose the best-of-breed implementations of different features across all vendors. For example, one vendor might be known for its messaging implementation, whereas another might have more experience with persistence service.
- It allows you to make use of new features that may be available only from certain vendors.
- Being able to move a particular feature to a different container means that you can use vendor extensions and still achieve application portability, because you can migrate the container's features alongside the application.

So, whereas most enterprise platforms try to standardize their entire APIs as a single unit, OSGi standardizes the APIs piecemeal, in modules, allowing the vendors to provide smaller pieces and the developers to select and reuse the pieces they find to be better.

OSGi also has another major advantage over other standards such as the JEE specification: simplicity. As you'll see, you can create an OSGi-compliant Java module by adding a few lines to a Java's MANIFEST.MF file. Conversely, a JEE-compliant module needs several JEE configuration files, annotations, and Java classes that implement technology-specific Java interfaces. Simplicity plays well with standardization. Vendors are more apt to invest in standardization if the cost isn't prohibitive. Take a look at the Apache Software Foundation projects at http://www.apache.org/. You'll notice that several projects, such as Derby, have already been made into OSGi modules. Would this have happened if the process of making a library into an OSGi module was costly? I doubt it.

So far we've looked at four benefits of using OSGi. In the next section, we'll sum these up in one practical example.

1.4 Building blocks: the essence of OSGi

Let's reconsider the location-specific advertisement application of section 1.2.1. There are several features or services you could add to it:

- You could persist the selections of the promotions preferred by the mobile client, which later could be used for data mining.
- You could inform the correspondent of the promotion (for example, the store) that the mobile client has received the promotion and spent more than some considerable amount of time looking at it, perhaps through the use of an event-dispatching service.
- You could provide a mechanism that allows configuring of the application.

Figure 1.9 depicts the advertisement application and the services it uses.

Put simply, the most important thing to learn in this chapter is that the OSGi platform provides you with the means to quickly, efficiently, and easily build applications just by putting together *building blocks*. OSGi allows you to use and integrate building blocks that are being provided by different vendors, unknown to them. As you'll see,

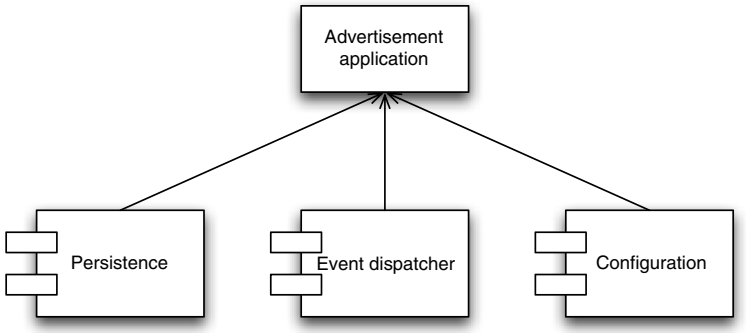

Figure 1.9 Application makes use of enterprise-based building blocks

you can even replace a building block dynamically, if a newer, better version is available from the same vendor or from a different vendor.

In this book, you'll learn how to use OSGi to create modular enterprise-grade applications by reusing infrastructure-based building blocks. You'll also learn how to perform the role of a building block vendor, providing your own reusable infrastructure services.

Collectively, all of these benefits have triggered the dissemination of public libraries of features—repositories of OSGi modules. We'll go through a list of such repositories in the next section.

1.5 Players

A multitude of vendors have implemented some aspect of the OSGi Service Platform, be it just the framework or selected services. This is no doubt a testament to OSGi's simplicity and its modularization.

Some of the most commonly used implementations of OSGi frameworks are these:

- Eclipse Equinox: http://www.eclipse.org/equinox/
- Apache Felix: http://felix.apache.org/site/index.html
- Knopflerfish: http://www.knopflerfish.org/

Most of these framework vendors also publish a public repository of services. For example, the following URLs point to these different vendors' repositories:

- Felix Repository: http://felix.apache.org/obr/releases.xml
- Knopflerfish Repository: http://www.knopflerfish.org/repo/repository.xml

In addition, several new projects have been created to tackle the implementation of the Enterprise OSGi. The goal is to allow developers to build enterprise applications using standard Enterprise OSGi services and the framework implementation of their choosing. Following are two projects that are working on implementing the Enterprise OSGi:

- Eclipse Gemini: http://www.eclipse.org/proposals/gemini/
- Apache Aries: http://incubator.apache.org/aries/

Finally, diverse development platforms, which have distinct goals, are using OSGi internally to implement their solutions. Some of these completely hide OSGi, whereas some do actually expose OSGi in some form or other. For example, a common exposure is to treat the OSGi module as the deployment unit of their platforms. Regardless, here are some examples of development platforms that are OSGi based:

- IBM WebSphere Application Server
- Oracle (formerly Sun) GlassFish Application Server
- Eclipse Virgo (SpringSource dm Server)
- JBoss Application Server
- Apache Camel
- Apache Sling
- Apache ServiceMix
- Apache Karaf

This is a reasonable number, especially if you consider that OSGi is still an emerging technology. Furthermore, because you can mix and match the components from different vendors, the overall gains are even larger! For example, you can use the Felix OSGi framework with the Eclipse service implementations and perhaps implement a few services of your own along the way.

By now, you're probably convinced and excited to use OSGi, but does this mean you need to start from scratch? We'll tackle this issue in the next section.

1.6 Are we starting from scratch?

If you're going to develop to the OSGi platform, does it mean that you need to drop your existing development platforms, such as JEE, and learn a new technology from scratch yet again? Or, even more important, can you leverage code from any of your existing enterprise applications?

Fortunately, the OSGi platform builds heavily upon JEE. As you'll see, there'll be several cases where we use, in some form or other, existing specifications defined by JEE, just in a slightly more modularized and isolated manner. This means that you're able to use the skill sets you learned in the past.

In addition, because existing JEE applications have been coded to APIs that also exist in the OSGi platform, you can move over some of the code from JEE containers to OSGi containers; however, this isn't always possible, and even when it is possible, some rewriting may be needed.

Finally, the reverse is also true; you should be able to migrate OSGi applications to JEE in some form. Furthermore, there's an even more appealing option in this case, where OSGi applications can be hosted in existing JEE application servers. This is done by hosting the full OSGi framework on top of JEE, and it's possible because OSGi is, after all, lightweight and modular.

As you'll see, the benefits of using OSGi are numerous, but in the end it all hinges on the fact that OSGi allows you to modularize both the static as well as the dynamic

structure of a program in a productive and efficient form. You may point out that the concept of modularization has been used in software engineering for decades, so why is this any different now? The difference is that OSGi allows you to apply modularization in a systematic form to software systems at their very foundation, which was never attempted before.

1.7 Summary

Development platforms consist of an application framework and a supporting runtime environment. The Java platform, with the JDK and JRE, is one example of a development platform. Enterprise platforms, such as Java Enterprise Edition (JEE), add enterprise features to the platform.

The OSGi Service Platform provides a dynamic Java module system for Java. It allows Java code to be modularized and to be managed as services. The OSGi Service Platform consists of the OSGi framework and the OSGi services. The OSGi Service Platform Enterprise Specification (Enterprise OSGi) was created to support enterprise use cases. It defines a collection of OSGi services that can be used together for enterprise features.

OSGi provides the means to achieve modularization, which helps decrease and manage the complexity of large systems. OSGi provides a native extensibility mechanism through the use of services. Finally, a consortium of several large companies is driving the OSGi specifications, hence aiding with its adoption as a standard.

In the next chapter, we'll drill down into the OSGi framework in detail. In chapter 3, we'll employ what you've learned so far to build your first OSGi-powered application. But this application will lack several useful features, such as persistence. Following up on this, we'll take an in-depth look into how we can use configuration, event handling, persistence, RMI, transactions, naming and directory services, and management to develop full-fledged carrier-grade OSGi solutions.

Finally, you must keep in mind that the main purpose of OSGi is to provide you with an efficient framework for creating and integrating software building blocks.

JAVA

OSGi IN DEPTH

Alexandre de Castro Alves

OSGi is a mature framework for developing modular Java applications. Because of its unique architecture, you can modify, add, remove, start, and stop parts of an application without taking down the whole system. You get a lot of benefit by mastering the basics, but OSGi really pays off when you dig in a little deeper.

OSGi in Depth presents practical techniques for implementing OSGi, including enterprise services such as management, configuration, event handling, and software component models. You'll learn to custom-tailor the OSGi platform, which is itself modular, and discover how to pick and choose services to create domain-specific frameworks for your business. Also, this book shows how you can use OSGi with existing JEE services, such as JNDI and JTA.

What's Inside

- Deep dives into modularization, implementation decoupling, and class-loading
- Practical techniques for using JEE services
- Customizing OSGi for specific business domains

Written for Java developers who already know the basics, *OSGi in Depth* picks up where *OSGi in Action* leaves off.

Alexandre Alves is the architect for Oracle CEP, coauthor of the WS-BPEL 2.0 specification, and a member of the steering committee of EP-TS.

For access to the book's forum and a free eBook for owners of this book, go to manning.com/OSGiinDepth

Free eBook
SEE INSERT

"Gives you a deep understanding of OSGi."
—From the Foreword by David Bosschaert, Red Hat

"Crucial if you're developing OSGi-based systems."
—Steve Gutz, IBM

"A thorough book on an increasingly important topic."
—Rick Wagner, Red Hat

"Alex has loads of experience to back up his words."
—Mike Keith, Oracle

"A jumpstart resource."
—Benjamin Muschko
Webs Inc.

ISBN 13: 978-1-935182-17-7
ISBN 10: 1-935182-17-X
57500
9 781935 182177

MANNING $75.00 / Can $78.99 [INCLUDING eBOOK]