QA 76
.73
.J38
A458
2012
Copy 1

OSGi IN DEPTH

Alexandre de Castro Alves

FOREWORD BY DAVID BOSSCHAERT



MANNING

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

OSGi in Depth

ALEXANDRE DE CASTRO ALVES



MANNING
SHELTER ISLAND

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

QA76
.73
.J38 A458
2012
Copy 1

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964
Email: orders@manning.com

©2012 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without elemental chlorine.

Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964

Development editor: Sebastian Stirling
Copyeditor: Linda Recktenwald
Poofreader: Andy Carroll
Typesetter: Marija Tudor
Cover designer: Marija Tudor

ISBN: 9781935182177
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 17 16 15 14 13 12 11

Library of Congress
2011 284960

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

foreword

After many years in ClassPath-land, the Java industry at large is finally starting to see the need for modularity in software development and deployment. There are many reasons for this. Maybe you just want to get out of the *Jar Hell*, where you're not quite sure anymore where a particular class comes from in your deployment. Other use cases for modularity center around providing isolation in a multi-tenancy context.

But there are more reasons. A modular approach makes developing software more scalable as modules tend to be highly focused with a clear demarcation of responsibility. A module's internals are inaccessible to the outside, which means that a modular approach tends to lead to well-defined APIs that better allow for concurrent development. It enables structuring of development teams such that parallel development of modules, which together form the application, is achievable.

The OSGi specifications provide a very mature, stable, and comprehensive modularity solution. The OSGi Core specification defines the OSGi framework, addressing modularity, lifecycle, services, and security aspects. Together these enable a dynamic system where *bundles*, the OSGi name for modules, are often remarkably reusable, properly encapsulated, and loosely coupled. The OSGi service registry enables an elegant plug-in model where consumers don't need to be preconfigured with any expected service provider implementation. The OSGi service model allows for services to be changed dynamically at runtime without the need to modify their consumers.

In this age of cloud computing, a system needs to be dynamically adaptable, highly manageable, and easily maintainable. OSGi technology facilitates all this alongside what is generally an extremely light infrastructure footprint. Runtime metadata and

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

framework management is available through a standard API or via JMX, if enabled, and because bundles generally have clearly defined purposes, maintenance is easier and more localized—there's no more *big ball of mud* that needs to be dealt with.

OSGi is the only standards-based solution to modularity today, and given that it has been around since the late 1990s, it's a well-matured and very stable technology. A number of highly popular open source projects provide OSGi framework implementations today, and a number of commercial implementations are also available. OSGi is being used in contexts from embedded, residential, and mobile devices to highly scalable and performant server systems. Additionally, it is the infrastructure behind many rich client applications, of which the Eclipse IDE is probably the most well-known.

In this book, Alex Alves looks in detail at many of the OSGi Core concepts while also elaborating on a number of vital technologies from the OSGi Compendium and Enterprise specifications. As cochair of the Enterprise Expert Group, I'm particularly pleased to see a number of Enterprise OSGi technologies covered. You'll find chapters about OSGi remote services (note a very interesting cloud computing section in this chapter), JDBC and JPA, transactions, JNDI integration, and JMX support, while the Configuration Admin and Event Admin services are also covered. Last but not least, you'll find coverage of OSGi Blueprint, a specification inspired by the Spring Framework aimed at using and creating OSGi services in a simple and user-friendly way.

This is a book that both covers the high-level big-picture architecture topics as well as the details involved in getting things working on a practical level. It will give you a deep understanding of OSGi and will provide you with the knowledge you need to utilize OSGi to the full.

DAVID BOSSCHAERT
PRINCIPAL SOFTWARE ENGINEER, JBOSS BY RED HAT
OSGi ENTERPRISE EXPERT GROUP COCHAIR

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

about this book

In this book, I show how the OSGi technology can be used to write better software, and in particular, how OSGi can be used to write better platforms for the development of better software.

Most importantly, I focus on real problems and on how OSGi can be used to solve them. Instead of just explaining OSGi's API for modularization, I first show you the problems that arise due to the lack of modularization. Instead of simply giving you OSGi's transaction API, I show you why and when you need to use transactions, and what the implications are of using transactions in a modularized service-oriented fashion in OSGi. As you read this book, you'll acquire in-depth knowledge of OSGi, and learn how to create containers that can manage transactions and persistence for applications!

Several years ago, it was inconceivable for a developer to write their own enterprise-grade development container or platform. OSGi has drastically changed this; it allows you to create your own domain-specific platform. OSGi is to domain-specific platforms what yacc is to domain-specific languages (DSLs).

Finally, in this book, I tackle some basic problems, but I don't shy away from the complex ones. You'll learn OSGi *in depth*. You won't just be looking at OSGi's API, but rather at the reason why the API is what it is, how you can best use it, and when not to use it. You'll even learn about open OSGi issues and what can be expected to change.

OSGi is an extraordinary technology. More than that, it's a game changer in the way large software is developed. To fully understand the reason for this, you need to understand OSGi *in depth*.

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

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

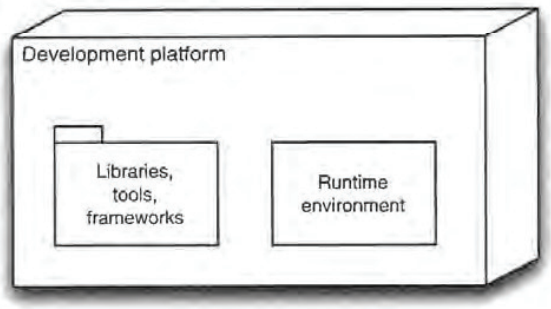


Figure 1.1 A development platform consists of a software framework and its supporting runtime environment.

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

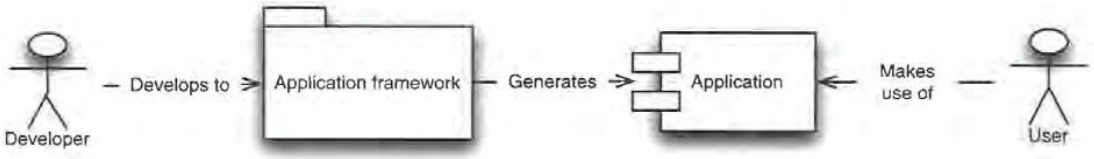


Figure 1.2 A developer creates an application, which is used by a user, and develops to a framework.

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.

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

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

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

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

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

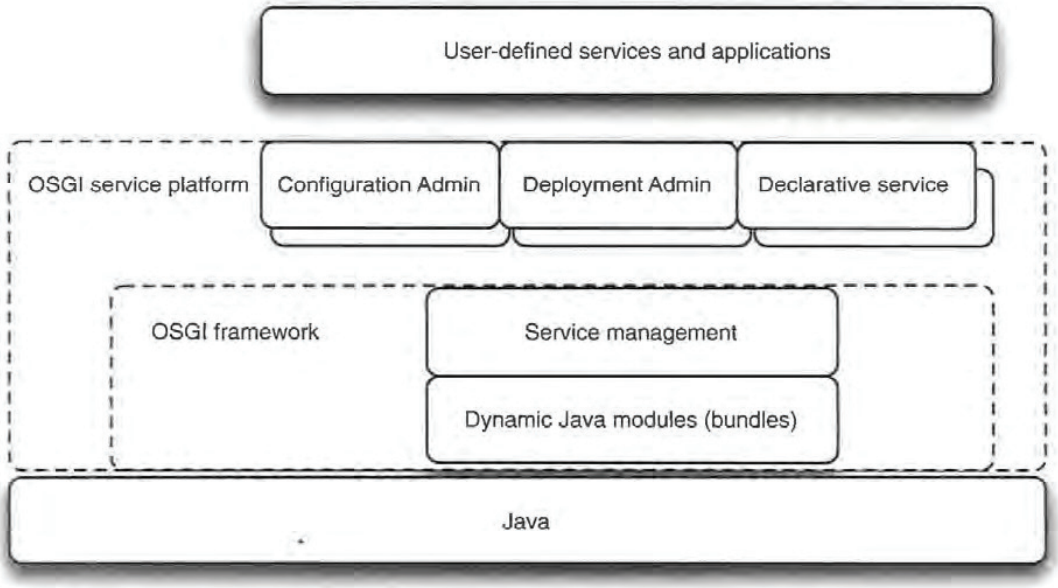


Figure 1.5 The OSGi Service Platform comprises the OSGi framework and the OSGI services.

- 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.

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

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.

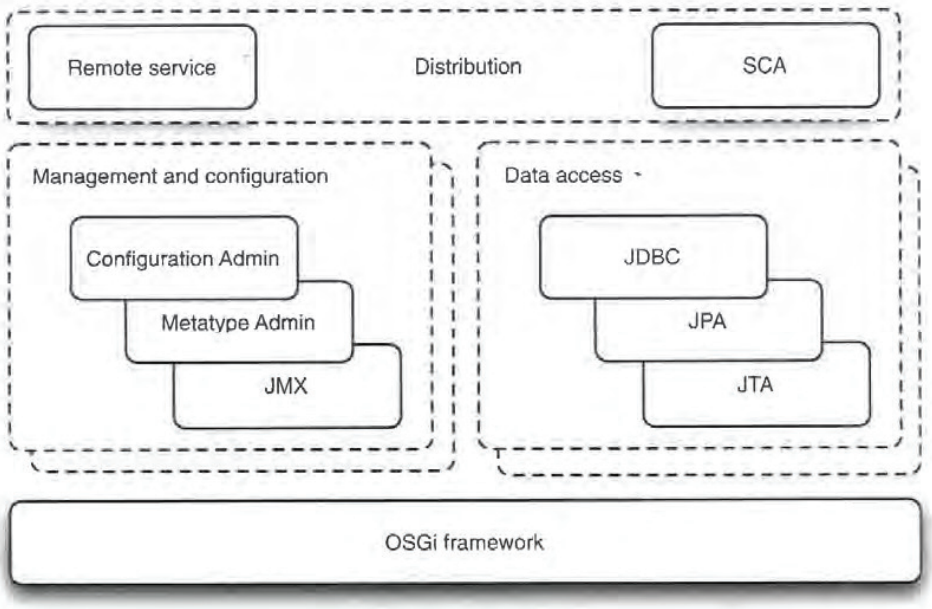


Figure 1.6 Enterprise OSGi consists of the OSGi framework and several OSGi services; together they provide enterprise features.

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

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.

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

But there are cases when this doesn't apply. For example, consider the case where a single bundle wants to open different Telnet sessions. This isn't possible today, because all the `TelnetService` services retrieved by this bundle would point to a single `TelnetImpl` object instance. In my opinion, this behavior should be made configurable, perhaps through the specification of some new predefined service property, so as to follow the same pattern initiated by `SERVICE_RANKING` and the other predefined properties. In the end, this is a result of overloading the function `getService()` to both indicate the retrieval of a service as well as the creation of a service when using a `ServiceFactory`. Therefore, perhaps another option would have been to define a `createService()` method in the `BundleContext`.

We've looked into several advanced features of OSGi services that help us improve decoupling. Next, you'll learn how to test your applications.

4.7 Improve robustness by testing your applications

In the previous sections, we went through several issues pertaining to developing OSGi bundles and services. One final issue related to the process of developing OSGi applications remains: how do you test OSGi bundles effectively?

4.7.1 Unit tests

It's a good idea to start by testing a bundle at the unit level. You want to make sure that the bundle, which generally is defined by a well-established contract or interfaces, works in isolation. For a standard library or module, this is generally done by creating JUnit test cases. The good news is that you can continue using JUnit to test your OSGi bundle. This is so because OSGi avoids imposing any of its technology-specific interfaces on the application. This allows developers to test their classes as they normally would.

Let's look into an actual example. Recall our auction framework from chapter 3? How could we test the `SealedFirstPriceAuction` class of the `auction.auctioneer.sealed` bundle, which is where the most complex application logic resides in the case of the auction application?

Because we were careful to isolate the OSGi-related code to a single class called `SealedFirstPriceAuctioneerActivator`, you can instantiate the class `SealedFirstPriceAuction` in isolation within a JUnit `Testcase` class without having to worry about the OSGi environment, as shown in the following listing.

Listing 4.9 A unit test for the `SealedFirstPriceAuction` class

```
public class TestSealedFirstPriceAuction extends TestCase {

    static class MockParticipant implements Participant {

        int numberOfAcceptances = 0;

        public String getName() {
            return "participant";
        }
```

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

```
        public void onAccepted(Auction auction, String item, Float price) {
            numberOfAcceptances+=1;                          <-- 1 Count acceptances
        }

        public void onRejected(Auction auction, String item, Float bestBid)
        {
            fail("Rejected");
        }
    }

    public void testSingleBidAsk() throws InvalidOfferException {

        SealedFirstPriceAuction auction =                    <-- 2 Class to be tested
            new SealedFirstPriceAuction(1);

        MockParticipant participant =                        <-- 3 Mock class
            new MockParticipant();

        auction.ask("book", new Float(50.0), participant);
        auction.bid("book", new Float(50.0), participant);

        assertEquals(2, participant.numberOfAcceptances);    <-- 4 Check results
    }
}
```

In ❷, you instantiate the test subject as a regular Java class, which in fact it is. You create a mock participant ❸, which only counts the number of accepted callbacks that were invoked ❶. Finally, you test the auction by sending a single bid and ask that match, and you assert the results ❹.

To run this test, which you can do through the command line or in some IDE such as Eclipse, you need to make sure the following JARs are in the class path: the JUnit library, the auction bundle, and the OSGi framework API (in our case, felix.jar will do). The latter is needed only because of the `SealedFirstPriceAuctioneerActivator` class. Remember that our test is a JUnit test case, so the easiest option is to run it using a JUnit runner, which is provided in the JUnit library itself.

What if we hadn't been so careful and had polluted our application logic with OSGi-related interfaces? Or what if you just wanted to unit test the bundle activator class itself?

In this case, there are two options: you could either run the unit test within the OSGi platform itself, or you could emulate (that is, mock) an OSGi environment for the unit test. Fortunately, the Spring Framework provides an OSGi mock library, making the latter approach an attractive one. Continuing with our example, let's unit test the `SealedFirstPriceAuctioneerActivator` class, as shown in the following listing.

Listing 4.10 Unit test for `SealedFirstPriceAuctioneerActivator` class

```
public void testAuctionnerActivator() throws Exception {

    MockBundleContext mockContext = new                      <-- 1 Mock BundleContext
        MockBundleContext();
```

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

```
SealedFirstPriceAuctioneerActivator activator =
    new SealedFirstPriceAuctioneerActivator();

activator.start(mockContext);

ServiceReference reference =
    mockContext.getServiceReference(Auctioneer.class.getName());

assertNotNull(reference);
}
```

❷ Activator to be tested

❸ Start using mock context

❹ Validate activator

In ❷, you create a bundle activator instance outside the OSGi environment. Then you create Spring's mock bundle context ❶ and use it to start the activator ❸, as if the activator was being started by the OSGi platform. Then you validate the activator by making sure that it has registered a service reference for the auction service ❹.

The Spring Framework

The Spring Framework is known for the popular dependency-injection library provided by the company SpringSource, among other things. SpringSource has adopted the OSGi platform and is providing several tools, libraries, and its own OSGi application server!

You can also run this test in isolation similarly to the previous case, but you must remember to add the Spring OSGi mock library to the class path.

NOTE Spring OSGi libraries, also called Spring DM, can be found at http://www.springsource.org/osgi.

After having unit tested all of our bundles, the next step is to make sure the bundles are able to talk to each other within the OSGi platform itself. This process is called *integration testing*.

4.7.2 Integration tests

To execute integration tests for our bundles, you'll need to run the bundles and the tests within an actual OSGi platform. And obviously, because you want to automate the execution of the tests, you'll want to do this through scripts or the command line. In essence, you need to do the following:

1. Start the OSGi platform.
2. Install all the bundles that need to be tested and their dependencies.
3. Install the test itself; this can be in the form of another bundle.
4. Trigger the execution of the test by invoking an MBean from a script or by having the test bundle include a bundle activator that starts the execution of the test.
5. Store the verdict (that is, pass or fail) of the test; writing the result to a file can do this.
6. Shut down the OSGi platform, and repeat the process if other tests are needed.

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

Fortunately, Spring DM also provides a facility for helping you doing this. Simply have your test extend the JUnit abstract class `AbstractConfigurableBundleCreatorTests`. This base class will take care of starting the OSGi platform, creating a test bundle, executing it, and shutting down the platform, as shown in the following listing.

Listing 4.11 Integration test for the auction framework

```
import org.osgi.framework.ServiceReference;
import org.springframework.osgi.test.AbstractConfigurableBundleCreatorTests;
import org.springframework.osgi.test.platform.Platforms;

import manning.osgi.auction.spi.Auctioneer;

public class IntegrationTestAuction extends
        AbstractConfigurableBundleCreatorTests {

    public void testSealedFirstPriceAuction() {
        ServiceReference reference =
            bundleContext.getServiceReference(
Auctioneer.class.getName());

        Auction auction =
            (Auction) bundleContext.getService(reference);

        // Place bids, asks...
    }

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

❶ Extend test framework class

❷ Use service as usual

❸ Specify OSGi framework

To test the auction framework, you create a JUnit class `IntegrationTestAuction`, which extends the Spring DM helper class `AbstractConfigurableBundleCreatorTests` ❶. This class is run as a bundle, so it has access to a `bundleContext`. With the `bundleContext`, you retrieve the Auction service ❷ and perform the actual test by placing bids and asks as you did previously, but now the whole interaction is done through the OSGi framework.

You can test the JUnit test case using a JUnit runner. But you need to include several JARs in its class path this time: the JUnit library, the Spring DM libraries and their dependencies, and the OSGi platform libraries, which in our case is felix.jar. You should run the test from the Felix home directory, so that the test helper class is able to start Felix correctly. You also need to tell the helper class that you want to use Felix and not some other OSGi platform, such as Equinox. This is done at ❸. And finally, don't forget to install all the necessary bundles, such as all of the auction application bundles, in Felix before running the test; otherwise they won't be found. It still seems a fair amount of work, but it's simpler than doing all these tasks manually.

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

4.8 Summary

In this chapter, we explored several advanced OSGi bundle and service features that allow you to develop flexible, extensible, and robust OSGi applications.

The OSGi framework provides several options for minimizing bundle dependencies. You can restrict a resource to be within a private package and thus not visible to any other bundle. You can also choose to export only a subset of the classes residing in a package, thus reducing the number of classes that make up the exported package contract. Finally, you learned that you should avoid creating bundle dependencies using the `Bundle-Required` header, because it can cause split packages and cause other problems.

We looked into three OSGi features that facilitate the creation of generic bundles. Dynamic imports provide a way out of the explicit import and export package contract, because they allow classes to be searched as needed. Optional packages are regular import packages that aren't considered an error if they're not resolved. And finally, fragment bundles are degenerated bundles that attach and merge with a host bundle, possibly extending its class path.

When a bundle loads a resource, this resource is searched through the bundle's class space, starting at the boot class loader, then at any `Import-Package` header, next at the bundle's private package, and finally at any fragment bundles attached to the bundle.

You learned how to specify complex service reference filters. OSGi service filters follow a prefix form, which can be non-intuitive but allows several operations to be grouped together on a single conjunction or disjunction.

You can use the predefined `SERVICE_RANKING` property to prioritize which service is used if multiple services meet a criterion when being retrieved from the OSGi service registry. This is useful when you need to specify which service among many should be the default one.

The predefined `SERVICE_PID` property is used to uniquely identify a service. This is useful when a service represents a real-world entity. No two services should be registered with the same `SERVICE_PID`, and the `SERVICE_PID` must be persisted; that is, the same value must be used when a service is restarted.

Generally, a service reference maps to a single service implementation object instance. But if each service must have its own state, then multiple service implementation object instances must be created for the same service reference. To do this, a service implementation object must implement the interface `ServiceFactory`.

The OSGi framework imposes very few vendor interfaces into the application classes, so it's generally easy to unit test a bundle by treating it as a regular Java class. Spring DM provides several mock classes for the OSGi framework API. Spring DM also allows you to perform integration tests. It has helper classes that start an OSGi platform, install bundles, and allow you to test their integration.

In the next chapter, we'll look at how to configure these decoupled bundles in a way that's consistent for the entire application.

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

10 Remote services and the cloud

This chapter covers

- Importing and exporting a remote OSGi service
- Understanding distribution providers and their properties
- Selecting a specific service endpoint and negotiating endpoint policies
- Learning the semantics of dealing with remote entities
- Understanding cloud computing and how it relates to OSGi

In today's world, it's almost guaranteed that your machine is networked and the services you use are provided through the internet. Remote communication is essential, and OSGi supports it by means of the OSGi Remote Service specification, which is the subject of this chapter.

We'll start by examining how to export a service for remote access and conversely how to import a remote service for local consumption. Next, we'll investigate how

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

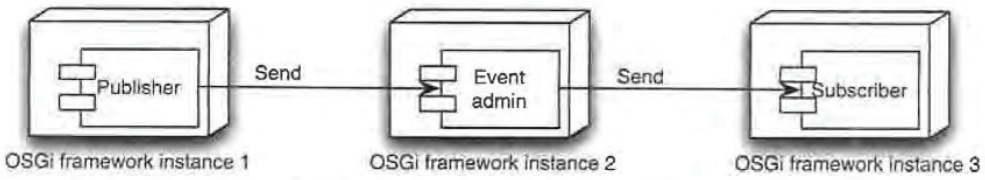


Figure 10.1 The publisher, subscriber, and Event Admin communicate remotely.

the actual transport is realized through the use of distribution providers, highlighting the fact that OSGi doesn't implement its own transport but rather allows existing transports to be plugged in. Finally, we'll look into how to negotiate common policies between the exporter and the importer of the remote services, for instance, to make sure that both sides agree on a common set of requirements, such as encryption.

Remote services opens up OSGi to be run in a remote server or, as is the tendency, in the cloud. We'll briefly discuss cloud computing and look at how the OSGi platform is an ideal platform for cloud computing. But before getting into cloud computing, we'll start the chapter by discussing remote service invocation.

10.1 Remote invocation

Recall from chapter 6 that one of the shortcomings of the Event Admin service is that it doesn't handle remote clients. Ideally, subscribers that have registered the `EventHandler` interface should be invoked regardless of their location.

For example, you could have a scenario where the publisher, the actual Event Admin implementation, and the subscriber are hosted in three different OSGi framework instances, as depicted in figure 10.1.

This distributed configuration can be achieved using OSGi's Remote Service. In this setup, distribution providers are responsible for creating proxies that represent remote services, therefore achieving a distributed system. Let's investigate the pieces, starting with a remote `EventHandler` subscriber in the next section.

10.1.1 Exporting a remote service

A service is specified as remote by tagging it with the following service properties:

```
service.exported.interfaces
service.exported.configs
```

The former specifies which service interfaces are to be exported in a remote fashion, and the latter specifies how those interfaces are to be made remote. The following listing provides an example for this remote subscriber scenario.

Listing 10.1 Remote `EventHandler` service

```
Dictionary props = new Hashtable();

props.put("service.exported.interfaces", "*");   <-- (1) Export EventHandler interface as remote
props.put("service.exported.configs",
```

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

invocation might unexpectedly fail or yield a high latency because of problems with the underlying transport.

Is there a way of knowing if a service is remote? Yes, all remote services have the following service property:

```
service.imported
```

This property is useful for filtering. For example, the following expression filters out all remote services, making sure that only local services are used:

```
(!(service.imported=*)
```

Imported services are also set with the service property `service.imported.configs`, which specifies the configuration type used for the particular endpoint of the imported service. For example, in our case it would be set to `org.apache.cxf.ws`. We'll discuss configuration types in depth later.

Finally, keep in mind that imported services are *not* set with the `service.exported.*` properties, such as `service.exported.interfaces`.

Next, let's take a quick look at how things actually work under the covers.

10.2 Distribution providers

It's the role of distribution providers to implement the actual transport between the bundle that's exporting a remote service in one OSGi framework instance and the bundle that's importing the service in another framework instance. A distribution provider can support different transports, such as RMI (Remote Method Invocation), SOAP (Simple Object Access Protocol), and REST (Representational State Transfer). The transport to be used is determined by the configuration type specified in the `service.exported.configs` service property. An endpoint description is created for each remote service and a particular configuration type. Consuming bundles use the endpoint description to establish a connection to the remote service.

> **NOTE** The OSGi Remote Service specification is a framework for remote communication that supports different transports, such as RMI and SOAP. The specification itself doesn't specify which transports are supported. Further, it doesn't define any new transport protocols.

A remote service may have multiple endpoints. For example, a remote service may be exported with both an RMI and SOAP endpoint descriptions. Clients can then select the proper endpoint for invocation. This approach allows the support of different transports, in a manner agnostic to the remote service and its consumers. Figure 10.2 describes the interaction between distribution providers, their supported configuration types, and the endpoints created by each configuration type for each remote service.

As you can infer, a distribution provider is needed for both the consuming and the providing sides, as shown in figure 10.3. On the consuming side, the distribution provider takes care of importing the remote service by establishing a local proxy endpoint

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

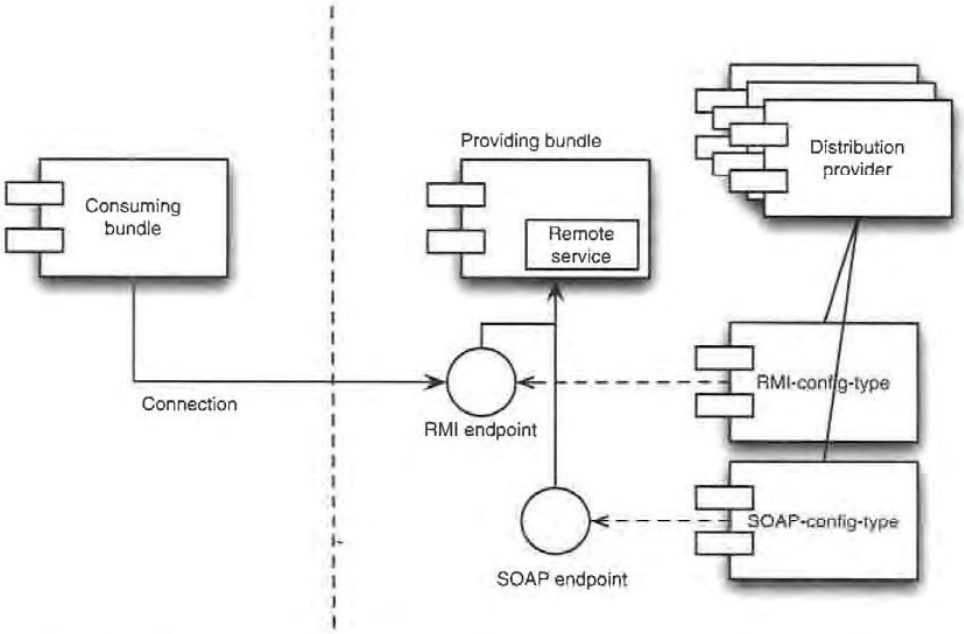


Figure 10.2 Distribution providers support different configuration types, which create separate endpoints for each remote service. In this example, the provider supports RMI and SOAP transports. Each transport endpoint is associated with the remote service and is tied to the client when a connection is established.

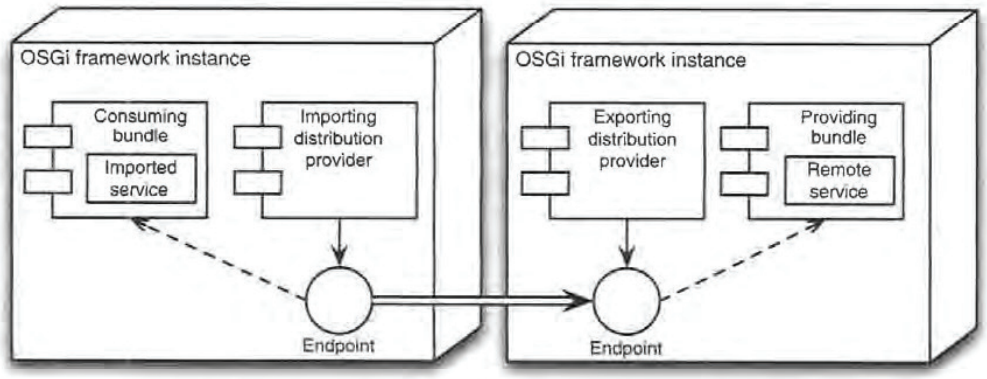


Figure 10.3 There's a distribution provider in each distributed framework instance, each responsible for its local endpoint and connecting to a remote endpoint.

and connecting it to the remote endpoint. We'll call it the *importing distribution provider*. On the providing side, the *exporting distribution provider* is responsible for establishing an endpoint that accepts connections and delegates to the proper service.

The distribution providers are also responsible for discovering and proxying the remote services as needed. That's why you don't need to explicitly mention the address of a remote service in the consuming bundle.

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

You've learned how a distribution provider works. In the next section, we'll look at how to handle multiple distribution providers and transports.

10.2.1 Selecting the proper endpoint

It's not uncommon for a system to support multiple transports. For example, RMI for general application communication, SOAP for web development, and perhaps some proprietary socket implementation (perhaps running on InfiniBand) when performance is essential.

When exporting a remote service, you should generally be as flexible as possible and therefore specify several or the entire set of supported configuration types. For example, we could export the `EventHandler` setting the `service.exported.configs` service property to `net.rmi, net.soap, net.socket`. If we did so, and the distribution provider supported all the aforementioned transports, then three distinct endpoints would be created, one for each transport.

But on the consuming side, we want to import the service using a single specific endpoint, as the situation demands. For example, if we wanted to use the event handler interface from a web application, then it's likely that we'd prefer to import the `EventHandler` service using the SOAP endpoint. This can be done using the `service.imported.configs` service property. The following code fragment retrieves an `EventAdmin` service that's tied to the SOAP endpoint:

```
ServiceReference ref =
   context.getServiceReference(EventAdmin.class.getName(),
      "(&(service.imported.configs=net.soap))");
```

Unlike the `service.exported.configs` property, the `service.imported.configs` property contains the single configuration type that's being used by the supporting endpoint, as shown in figure 10.4.

If the importing distribution provider in the OSGi framework supports all three transports we specified when exporting the remote service, then three endpoints are created and likewise three services are imported, each containing a different configuration type for its `service.imported.configs` property.

> **NOTE** The fact that endpoints for all configuration types are created may seem wasteful at first glance, but in reality a distribution provider implementation may do this on demand by using the OSGi service hooks.

But what happens if the importing distribution provider doesn't support the transport we specified? In other words, in the preceding example, how would we know there's a SOAP endpoint for us to depend upon? We can find the supported configuration types of a distribution provider by using the service property `remote.configs.supported`. The distribution providers register some anonymous service, setting this property to the list of supported configuration properties. In our case, should the importing distribution provider only support the configuration types of `net.rmi` and `net.soap`, then it would register a service whose `remote.configs.supported` service property is set to `net.rmi, net.soap`.

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

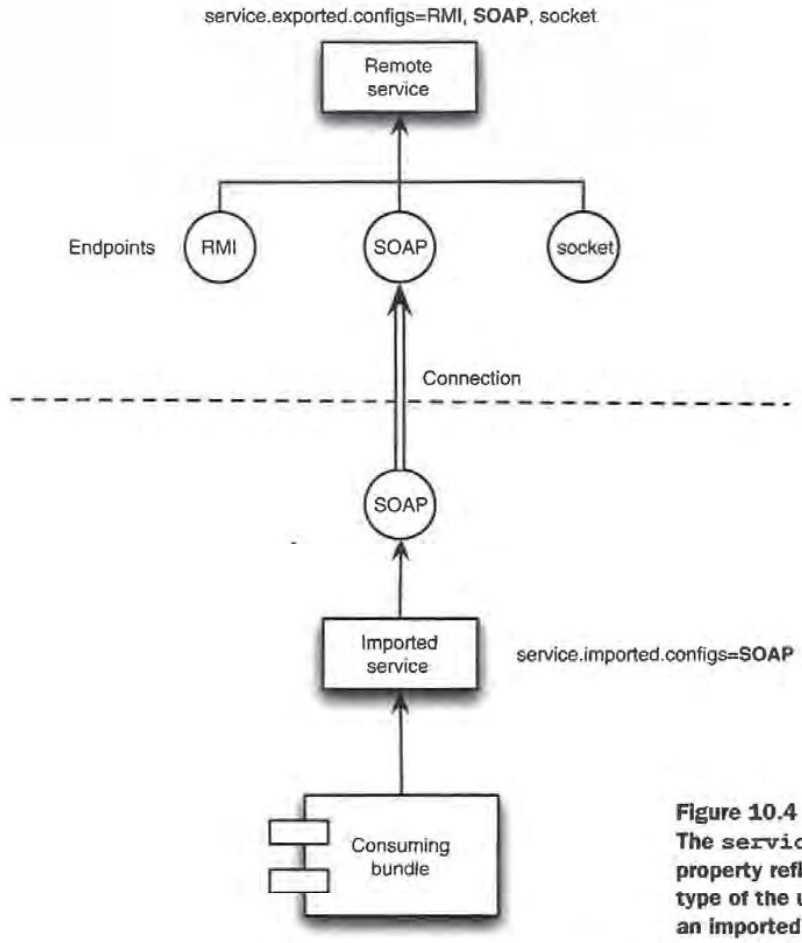


Figure 10.4
The `service.imported.configs` property reflects the configuration type of the underlying endpoint for an imported service.

Knowing this, our consuming bundle should first find out if the `net.soap` configuration type is supported and only then retrieve the SOAP endpoint for the `EventHandler` service. We can find out if the `net.soap` configuration type is supported as follows:

```
ServiceReference ref =
   context.getServiceReference(null,
      "(&(remote.configs.supported=net.soap))");
```

Finding out the supported configuration types is useful, but sometimes you may need more information, such as whether a particular policy is supported. This is the subject of the next section.

10.2.2 *Negotiating policies*

There are cases when not only do you need a particular endpoint but you also need to certify that a policy from the remote service or distribution provider is being supported. It's sometimes even necessary to negotiate policies between the consuming and the providing systems and make sure that both sides of the equation agree on common vocabularies.

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

`setAddress()` method from the `Customer` object, which mostly becomes a structure of values. Because the consuming bundle generally wouldn't know it's dealing with a remote service, the onus of the interface design falls largely on the provider.

The next problem we'll look at is that a communication transport may go down for a variety of reasons at any arbitrary time. There isn't much we can do to deal with this in the provider, and instead we need to design for this possibility in the client itself. For instance, just before we invoke the `updateCustomerAddress()`, our wired or wireless network connection may be lost, and the importing distribution provider would lose connectivity to the exporting distribution provider. In this case, the imported service should raise the runtime exception `ServiceException`, and a consuming bundle should be ready to handle this exception.

A naïve exception-handling implementation for a `ServiceException` might wait a few seconds in case the network has simply run into some transient problem and try again, as demonstrated in the following code:

```
ServiceReference servRef =
    bundleContext.getServiceReference(RemoteCustomerRegistry.class.toString());

if (servRef != null) {
    RemoteCustomerRegistry registry =
        (RemoteCustomerRegistry) bundleContext.getService(servRef);

    try {
        registry.updateCustomerAddress("Alex", "Updated Address!");
    } catch (ServiceException e) {
        Thread.sleep(2000); // wait for two seconds and try again

        registry.updateCustomerAddress("Alex", "Updated Address!");
    }
}
```

One final issue to consider is the underlying communication stack being used. The different transports have different characteristics and limitations. For example, if CORBA is being used, there are restrictions on the Java types that can be referenced in the remote services. This is because CORBA IDL (Interface Definition Language) doesn't support the entire spectrum of the Java types.

Remote services, sometimes called Distributed OSGi, are a major foundation for the emerging cloud technology, which we'll investigate in the next section.

10.4 Elasticity at the cloud

Needless to say, cloud computing is a vast subject, and it's beyond the scope of this book. But as you'll see, OSGi has become an interesting technology for supporting the concept of Platform as a Service (PaaS). In the following sections, we'll make some assumptions and simplifications as we discuss why and how that is.

Let's begin by exploring a use case. Suppose we'd like to make the auction application, which was explained in chapter 3, a public service for a website. Somewhat similar to eBay, the goal is to support web clients making offers of items to sell and bidding

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

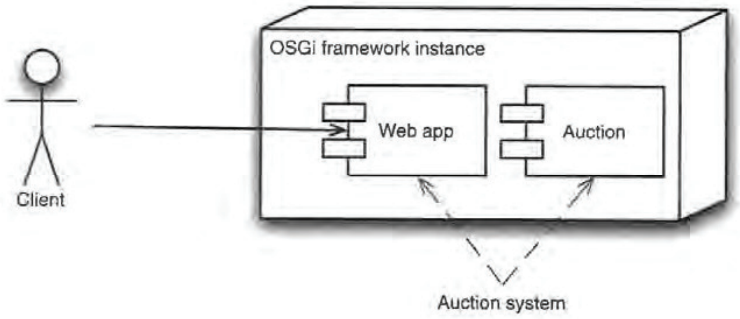


Figure 10.8 A single OSGi framework instance running the auction system

for these items. The first step toward achieving this is to create a web application that exposes a proper web page, where a customer makes offers and bids.

Simply put, the web application internals implement the `Participant` class and invoke the `Auction` class. And we are, in a naïve sense, finished. The web application is implemented as an OSGi bundle, and together with the auction application bundles and the bundle containing the web server (for example, Jetty), they're all installed in a single OSGi framework instance and hosted in a machine that's made accessible via the net. We'll refer to this whole environment as the *auction system*. That is, the auction system includes all the software components needed to run the auction application, such as the web server and any other supporting components, as shown in figure 10.8.

Life goes on; the auction system is running successfully and in fact is a major success—so much so that we get more customers than we expected, and because of that, performance degrades significantly. Whereas before buyers and sellers were able to bid and make offers instantly, it now takes several seconds, sometimes minutes if the item is very popular. The solution is to expand and acquire another machine. This trend continues, and we reach the point of having to manage a network of eight machines. This setup has become extremely expensive; not only do we need to buy several fully configured machines, but we also need to manage them. We decide to do some profiling to look for the bottleneckand find out what's causing the performance problems.

10.4.1 Designing for the cloud

As it turns out, the bottleneck occurs when correlating the offers and the bids in the `Auction` bundle implementation. Although we could try to improve the algorithm, this already tells us that the bottleneck isn't I/O bound and likewise isn't in the web application.

We decide to change our architecture. Rather then having eight instances of the OSGi framework each running the auction system, which consists of the web application, the auction application bundles, and the supporting infrastructure bundles, we change it to a two-tier architecture, as shown in figure 10.9. In our two-tier architecture, the first tier consists of the web application and the web server, which provides the client interface to the customers and then forwards the requests to the second tier,

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

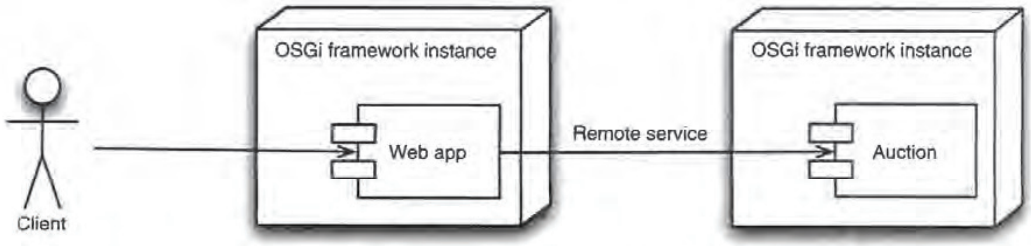


Figure 10.9 An auction system partitioned into two separate OSGi framework instances, communicating through an OSGi remote service

called the business tier. The business tier contains the auction bundles, because they perform the actual business-related code.

In this new setup, we host the web applications in only two machines, which communicate via REST with four other machines containing the auction application bundles. Because of the performance improvements, we're able to serve the same number of clients with only six machines, two fewer than before.

> **NOTE** In reality, we should have a three-tier architecture. The auction application would store its state in the third tier, the data tier. You'll see how to change the auction application to persist its data by using JPA in a future chapter. For the time being, we'll ignore this issue and focus on the communication aspect of the system rather than the persistence issues. But keep in mind that to function properly in the absence of a data tier, we'd need to make sure that all requests pertaining to an item are routed to a single auction bundle instance.

By doing this, we gain on several fronts:

- The machines are easier to manage. When a web server security patch is released, we only need to update the two machines of the client tier. Likewise, we only need to set up the firewall in these two machines. Further, had we been persisting the data, we'd only need to install the persistence storage, such as the RDBMS, in the machines of the data tier.
- The overall environment is less expensive, because we can customize the machines for their specific roles. For example, the machines in the client tier need better I/O but don't need particularly faster CPUs or disks. But the machines in the business tier should have fast CPUs, because we know from the profile that the auction application is CPU bound. By segregating the system into separate nodes, we can avoid having to buy a machine that has everything, such as fast I/O, fast CPUs, and fast disks. Instead, we can be smart and use cheaper machines without impacting the result.
- Performance improves, because we allow the machines that had the bottleneck to focus on just one thing, correlating the bids and offers, rather than also spending CPU cycles on the web application.

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

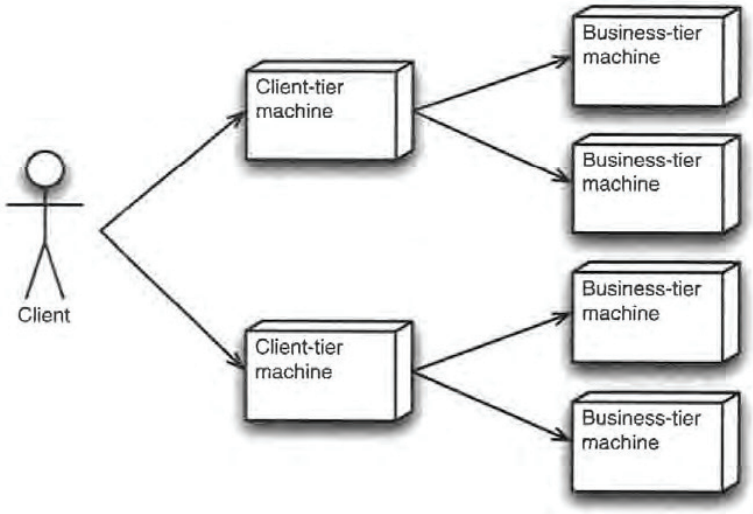


Figure 10.10 Two-tier architecture for the auction system

- The system is overall more scalable, because we have a flexible setup, where we understand which components can be replicated and how best to partition the requests.

Surprisingly, it's quite easy to change our existing implementation to the new two-tier architecture. Instead of collocating all the bundles of the auction system in a single OSGi framework instance, we partition the system into several separate OSGi framework instances, placing the web application bundle in the OSGi framework instances hosted in the client tier machines and placing the auction application bundles in the business tier machines, as shown in figure 10.10. The main effort is in making sure that the web application bundle is able to locate and use the remote Auction service. This, as you've seen in this chapter, is done by using distribution providers and setting the proper service properties, such as `service.exported.configs`.

During this exercise, you've learned an important albeit subtle lesson. The reason we've been able to easily move from a local environment to a multiple-machine environment and finally to a two-tier multiple-machine environment is that our auction system is modular. Had we designed the whole system as a single monolithic (web) application, we wouldn't have been able to partition it when needed. As you'll see next, being able to partition our system is a prerequisite to running in the cloud.

Let's continue our story. We've been running our auction system successfully for several months now. During this period, we've noticed that business is very slow on some dates, like at the end of January, and business is very good in other cases, like near Valentine's Day. This burst-like behavior is difficult to manage and eventually becomes a problem. If the business continues doing well, we may have to double the number of machines we have for Christmas, only to have them sit idle in the months that follow. We need to be able to dynamically cope with this variation of demand. Coping with this dynamism is the exact proposition of cloud computing, as you'll see next.

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

10.4.2 Cloud computing

The National Institute of Standards and Technology (NIST) defines cloud computing as follows:

> **DEFINITION** Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

To utilize cloud computing, we need to move our auction system from being hosted on our own machines to being hosted in machines managed by a cloud provider, as shown in figure 10.11. In fact, we don't even need to know the physical layout and characteristics of these machines; they're simply computing nodes performing their work. To do this, we create a virtual machine (VM) containing all of our software components—the operating system (OS), generally Linux, the OSGi framework distribution, and all of the application bundles and supporting libraries, like the Jetty web server. Our VM not only contains the binaries but also knows how to start them appropriately. For example, the VM should bootstrap the OSGi framework instance and make sure that our bundles have been installed and are activated. We can do this by setting up a shell script as an OS service. OS services are automatically started when the VM and thus the OS is booted.

Next, we give the VM to the cloud provider and tell the provider to assign six computing nodes to the VM. That is, our VM is deployed in six nodes somewhere in the cloud. As it gets close to Christmas, we ask the cloud provider to assign four additional

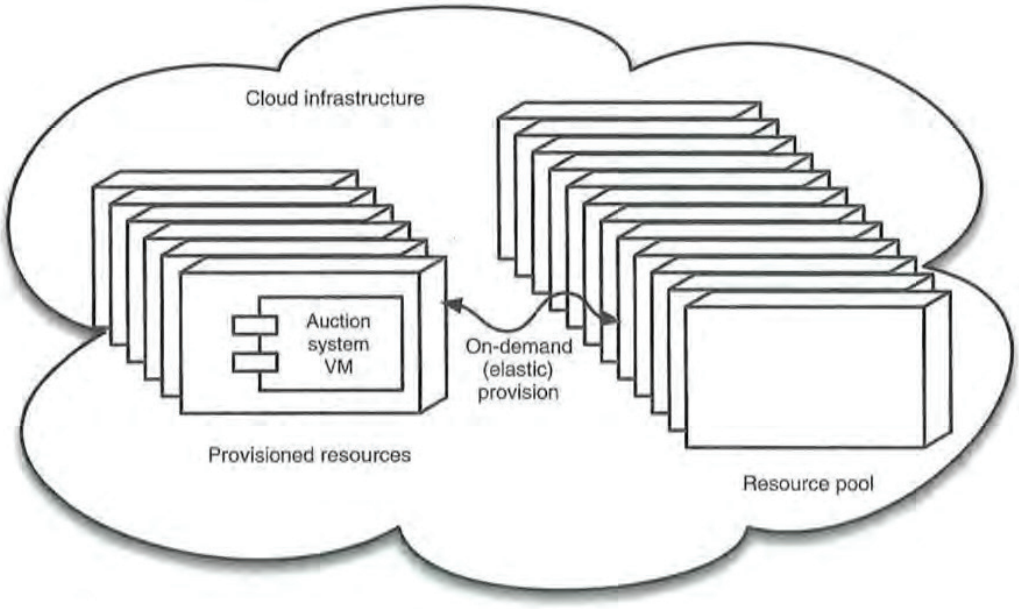


Figure 10.11 Auction system running in the cloud

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

nodes to our VM. After the holiday season ends, we ask the cloud provider to remove the four additional nodes; in fact, we ask it to keep only four nodes total running our VM. The cloud provider has a pool of resources, these being computing nodes, disks, and the like, which can be dynamically provisioned to run the customer's software.

> **DEFINITION** The cloud is made up of a pool of resources that are provisioned dynamically on behalf of the customers.

In our particular case, we're assigning the nodes per calendar date, but we could even be more ambitious and monitor the utilization of the resources. For example, we could set up provisioning rules, as follows:

- If the CPU crosses an 80% utilization threshold, then automatically provision a new node for our VM.
- If the CPU utilization falls to below 20% for more than two days, then automatically shut down the node.

As you can see, cloud computing allows us to dynamically and efficiently change our environment based on demand. It also has other advantages, just like any other hosted service, such as us not having to manage our own machines and networks. But keep in mind that the main proposition for cloud computing is indeed its elasticity.

> **WARNING** Cloud computing also has its drawbacks, one of the most serious being security. Because services from different customers are all sharing resources, there's always a higher likelihood of security breaches. Another disadvantage of cloud computing is the increased latency because services may hop several times between computing nodes before being fully served.

The question that remains to be answered is how OSGi relates to cloud computing. We'll address this question in the next section.

10.4.3 OSGi as the cloud platform

Let's say that to keep our business competitive, we need to fine-tune our auction algorithm frequently. Updating our auction system in the cloud isn't a trivial task; we need to change the auction bundle, re-create a new VM image, upload it to the cloud, shut down the VM instances, and finally replace them with the new image.

Generating VM images and the whole workflow process just described can be quite costly, especially if it must be done on a daily basis. Moreover, keep in mind that when generating a new VM image, we need to test all the components, and the time used for testing is proportional to the number of components being tested. By using OSGi, we can modularize our systems and therefore partition adequately, creating simpler and smaller VMs. You already saw an example of this when we partitioned the auction system into two tiers. The result is that we can have two separate VM images, one containing the components of the client tier and the second containing the components of the business tier. Managing smaller VMs is obviously better. For example, if we're updating just the auction bundle, it would prevent us from having to repackage and

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

retest the web application and server. Potentially, we could also rent cheaper computing nodes for the client tier VM, which contains the less-demanding web application bundles.

> **WARNING** It's a common misconception that you don't need to design your system to run in the cloud. To fully utilize cloud computing, the system must be able to scale. Modular systems scale better; therefore, OSGi is a clear infrastructure choice, because it emphasizes and helps in the creation of modular applications.

That's not the complete story though. Suppose that instead of uploading and managing VMs in the cloud, we were allowed to manage OSGi framework instances and their bundles. If this were the case, we'd just need to inform the cloud provider to update our auction bundle. The auction application, being an OSGi application, already has to deal with the fact that dynamic changes and updates may happen, so the fact that it's happening when it's running in the cloud or running as a standalone local application is mostly immaterial. Thus, if our supposition were true, OSGi would give us two important advantages:

- OSGi would let us update our system quite easily and efficiently, without us having to get into the business of generating full VM images.
- It avoids lock-ins, because we can design and validate our applications using any OSGi implementation in a standalone environment before moving to the cloud.

We could even take it a step further. The cloud provider could give us access to an OSGi bundle repository (OBR), where we could install, update, and uninstall bundles as needed. Then, after having updated the auction bundle, we could select an OSGi framework instance and tell it to move forward and take the new versions of the bundles from the OBR. If there are any problems, we could always tell it to revert to the previous versions.

Let's take a step back and revise. We have several OSGi bundles that have been uploaded to the cloud's OBR. In addition, we've provisioned OSGi framework instances to computing nodes, as shown in figure 10.12. For example, we have two OSGi framework instances in the client tier; therefore, we could assign each to a different computing node. Next, we have the remaining eight OSGi instances, which we could likewise assign to a different computing node. Finally, we could install the web application in the OSGi instances of the client tier and the auction application bundles in the OSGi instances of the business tier. And, of course, we would need to make sure that the web application bundle could invoke remote services residing in the bundles of the business tier. In this environment, we can easily manage updates to our system by leveraging the OSGi framework. But there's still one problem to consider: we're having to explicitly provision the OSGi framework instances and target the bundles to specific framework instances.

A better scenario would be the following: we upload the bundles to the OBR in the cloud, and each bundle tells the cloud its requirements, so that the infrastructure can

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

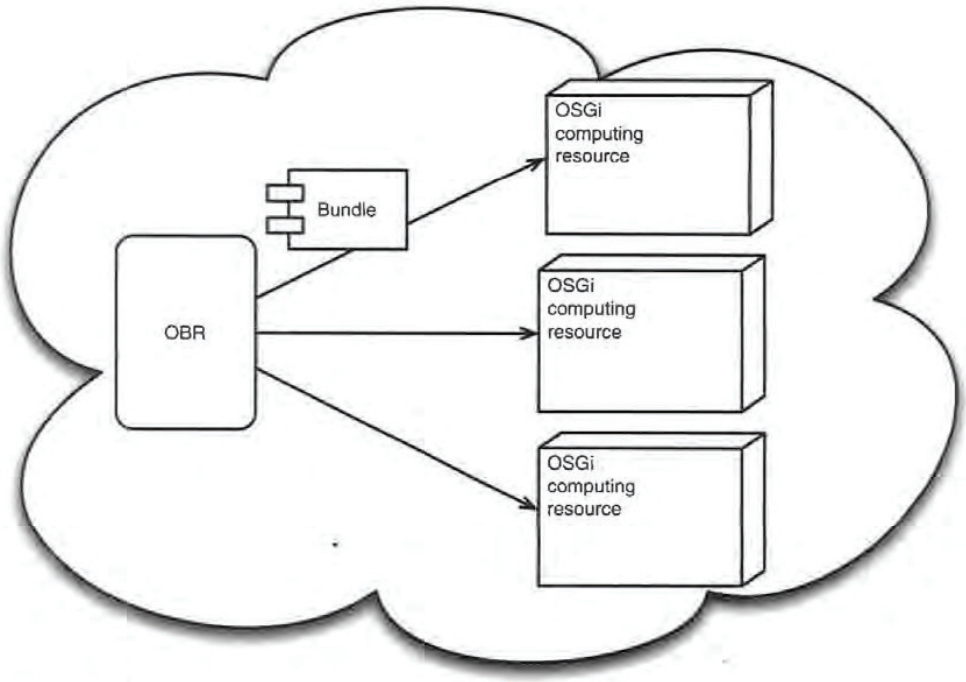


Figure 10.12 OBR and OSGi framework as a platform for cloud computing

dynamically provision the system. For example, the web application bundle could state that it depends on a web server and a cloud resource that has fast public access to the internet. Likewise, the auction bundles could state that they depend on a cloud resource where they get at least 80% CPU utilization.

The cloud infrastructure would consider all these requirements and assign the bundles to the correct OSGi framework instances that it provisions to its computing nodes. For example, the cloud infrastructure would spawn a new computing node that has internet access and install the web application bundle into it.

REQUIREMENTS AND CAPABILITIES

Does it seem far-fetched? Perhaps, but we already have most of the pieces. Dependencies between bundles can already be specified in OSGi using the regular `import-package` mechanism. Furthermore, starting in version 4.3 of the OSGi Service Platform, there's a generic mechanism for the specification of general requirements and capabilities. You can think of the `import-package` and `export-package` headers respectively as a particular case of a requirement and a capability specification.

For example, consider a bundle that specifies the following OSGi 4.3 `Require-Capability` MANIFEST header entry:

```
Require-Capability:
    com.cloudprovider;     filter:="(&(web-access)(cpu<90))"
```

This header is placed in the namespace of `com.cloudprovider`, which means that its interpretation is intended for some particular cloud provider. It states that this bundle

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

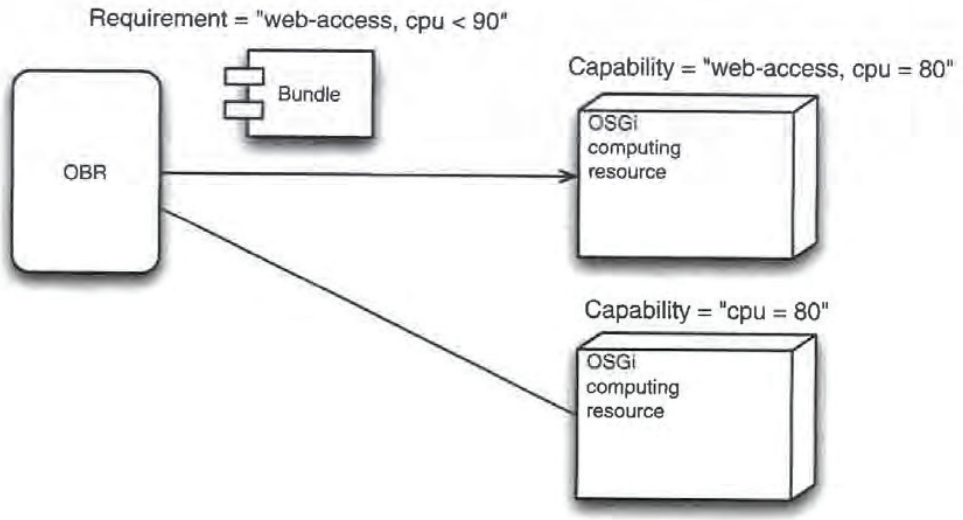


Figure 10.13 The cloud infrastructure matches a bundle's requirements with a resource node that's able to fulfill them.

requires the attribute `web-access` to be true and that the attribute `cpu`, which is a numeric value, must be less than 90 (percent).

On the opposite side, the cloud provider has to find an OSGi framework instance whose System bundle has been launched with the following property:

```
org.osgi.framework.system.capabilities.extra=
        "com.cloudprovider; web-access:Boolean=true; cpu:Long=80"
```

Essentially, the framework instance is telling us that it's capable of providing web access and it will monitor the CPU to make sure that it's never more than 80% busy. This is shown in figure 10.13.

There's still much to be collaborated before cloud computing is considered a mature technology. But the OSGi platform has three intrinsic qualities that make it ideal for supporting elasticity in the cloud:

- Application behavior dynamism provided by the OSGi framework API
- Transport abstraction through OSGi Remote Services
- Dependency management using the OSGi requirement/capability framework

In other words, cloud providers could expose the OSGi service platform as a service to their clients, so that they can manage their applications. In this environment, an OSGi bundle becomes the cloud's deployment unit, and OSGi the ideal Platform as a Service.

10.5 Summary

The OSGi Remote Service specification allows a consuming bundle to invoke a service that's provided by a bundle in a remote OSGi framework instance. You register a service using the service properties `service.exported.interfaces` and `service.exported.configs` to mark it as remote.

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

Distribution providers are responsible for transparently discovering and handling the remote services for the consuming bundles, and they support different transport protocols, as dictated by the `service.exported.configs` property. A remote service may be exported through different endpoints. A consuming bundle may optionally select a particular endpoint by filtering on the service property `service.imported.configs`.

It's not uncommon for the consumer and the provider of the remote services to negotiate policies, such as if encryption is necessary or based on the QoS of the transport itself. For this purpose, the provider can specify its requirements in the `service.exported.intents` property. The distribution providers must be able to fulfill these requirements, which are seen as capabilities in the consuming side in the service property `service.intents`.

Cloud computing is all about being able to efficiently meet the varying demand of customers by dynamically provisioning resources from a resource pool. The OSGi platform is an ideal Platform as a Service, because it has native support for the dynamic behavior of applications, remote services, and a generic requirements/capability framework for dependency resolution.

In the next chapter, we'll look into the crucial issue of how to manage this complex environment. Managing the OSGi framework and its applications is the subject of the next chapter.

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC

JAVA

LC CIP
0 028 873 659 5

OSGi IN DEPTH

Alexandre de Castro Alves

Free eBook
SEE INSERT

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

"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]

Exhibit 1008
Microsoft Corporation v. Edge Networking Systems, LLC