# Enterprise OSGi IN ACTION

Holly Cummins
Timothy Ward

SAMPLE CHAPTER

MANNING

# *Enterprise OSGi in Action*

by Holly Cummins
and Timothy Ward

**Chapter 10**

# *brief contents*

# 10 Hooking up remote systems with distributed OSGi

***This chapter covers***

- The Remote Services Specification
- How remote services differ from local services
- Exposing an OSGi service as an external endpoint
- Transparently calling a remote endpoint as an OSGi service
- SCA as an alternative distribution mechanism

In previous chapters, you've spent a great deal of effort building a scalable, extensible online superstore application. Although it's unlikely that this exact application will ever be used in a production system, it does demonstrate the sorts of patterns that real enterprise applications use, such as that the application can be added to over time.

Given the small loads and small amounts of data that you've been using in the Fancy Foods application, you aren't stretching the limits of a typical development laptop, let alone a production server, but it's easy to see that if you were to increase

the number of departments, increase the user load, and increase the volume of data involved, then things might be a little more constrained. These sorts of scaling requirements mean that many enterprise applications can't run on a single server because the volume of data and users would overwhelm the system. At this point, you have no choice but to distribute the application over more than one machine. This distribution process can be extremely painful if the application hasn't been designed for it. Fortunately, well-written OSGi code lends itself well to distributed systems, and the OSGi Enterprise Specification offers us a standard mechanism for expanding OSGi beyond a single framework using remote services.

Before we dive straight into the details of authoring a remote service, let's first make sure you understand some of the reasons why you might want to access a remote service. It's also important that you know about the drawbacks of using remote calls instead of local ones, particularly that you understand the limitations that being remotely available places on the design of the service you wish to access. It would be embarrassing to author a remote service only to find that you can't use it remotely!

## 10.1 The principles of remoting

Enterprise applications have to cope with huge scales, both in terms of the application and the user base. As an application and the demands upon it get bigger, it becomes impractical, or even impossible, to host the whole application on a single machine. Accessing services hosted on another machine is therefore a vitally important part of enterprise programming.

### 10.1.1 The benefits of remoting

The primary benefit of remoting is obvious: by allowing applications to access services and/or resources on another machine, you can dramatically reduce the computational load on a single machine. Spreading computationally difficult and memory-intensive tasks out to remote machines ensures that the server hosting your frontend has ample resources to accept new HTTP connections from other clients.

Remoting isn't the only way to scale out applications. As you've seen from the example applications in this book, it's perfectly feasible to write an enterprise application that can live comfortably on a single machine. But if this application needs to support a large number of concurrent users, then that single machine may not be capable of servicing every request without overloading its CPU or network adapter. Often applications like these are scaled out *horizontally* by adding more machines running the same application. Using a simple router fronting, these machines' clients can be transparently redirected to an instance of the application. Most routers are even able to remember which machine a client has previously been redirected to and ensure subsequent requests are sent to the same machine (see figure 10.1). Given that this sort of design is possible, why would you bother physically separating parts of your application with remoting?

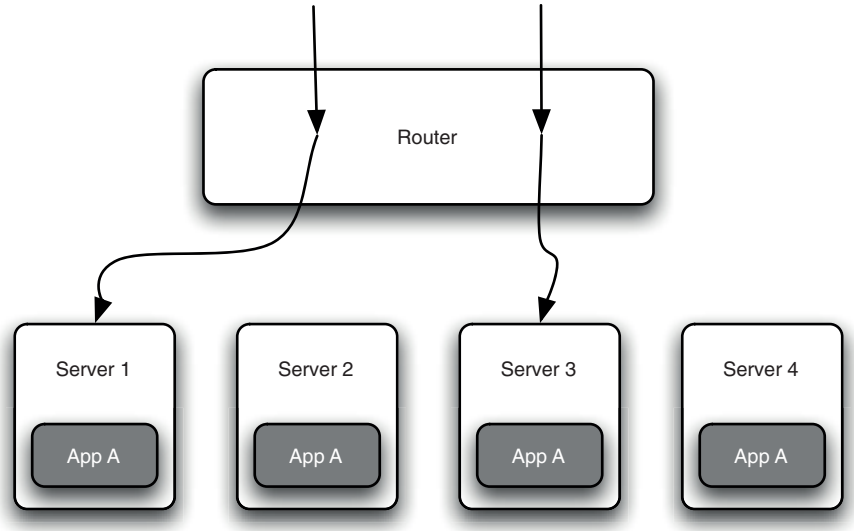


**Figure 10.1** **Horizontal scaling allows client requests to be transparently distributed across a number of servers to increase throughput.**

### ALLOWING SPECIALIZATION

As with many things, not all computers are created equal. Laptops are small, light, use comparatively little power, and, as a result, produce little heat. Laptops also have reduced processing speed, less memory, and slow disks. Mainframe computers are large, require specialized power supplies and cooling, and you certainly would struggle to lift one; on the other hand, they have vast quantities of memory, a number of fast processors, and usually fast disks.

Laptops and mainframes represent two ends of a large scale of computing equipment; one where processing power, reliability, size, and cost vary hugely. In terms of cost effectiveness, it's usually best to pick a system that suits the type of work that you're going to give it. Running your corporate directory database on a laptop is inadvisable (and probably impossible), but running a small internal web portal on a mainframe is also a waste (unless you also have that mainframe doing a number of other things at the same time). What remoting allows you to do is to put services on machines that fit the type of work that they perform.

If you look at the Fancy Foods superstore as an example, there's an obvious way in which it will fail to scale. A typical machine for hosting a web application like the superstore would be a rack-mounted blade server. These machines have reasonably fast processors and network adapters, but only moderate memory and disk performance. This makes blades excellently suited for handling web traffic and running servlets, but if your database runs on the same system, this is likely to have a significant negative impact on performance. As the number of clients increases, the amount of

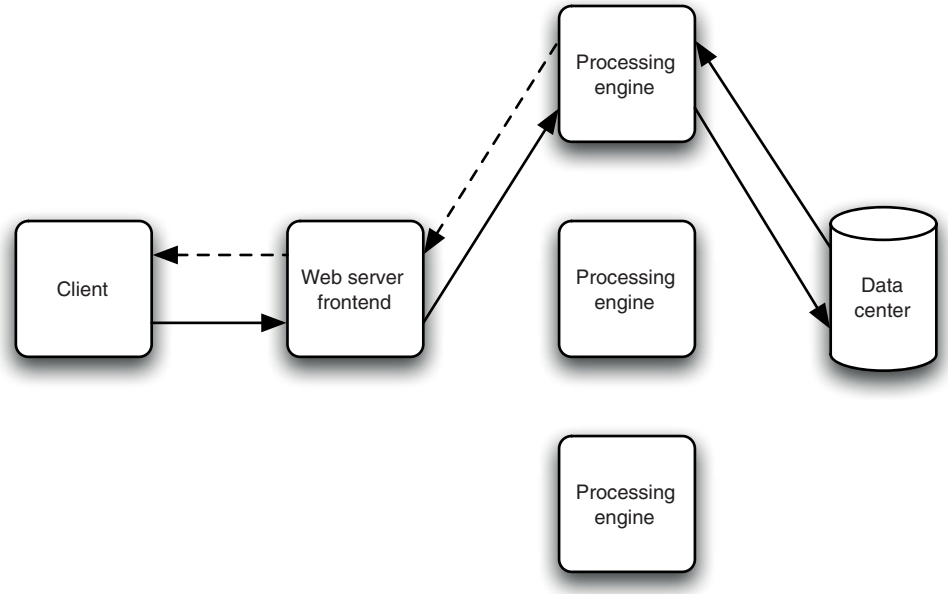


**Figure 10.2 A simple web server can forward complex processing tasks out to dedicated machines for execution, returning results when the task is complete.**

resource available to run the database drops significantly. By moving the database out to another machine, you can dramatically improve the performance of your application (see figure 10.2).

This is a common model, particularly for databases, but it also applies to other types of work. If, for example, you wanted to batch update 100,000,000 files, or run a vector-calculus-based fluid dynamics simulation, then it's unlikely that your local machine, or a central access point, are the best tools for the job.

Allowing for specialized workloads isn't the only reason that you might want to use remoting; probably the most common requirement has nothing to do with performance—it's all about reliability.

### INCREASING AVAILABILITY

If you have a single system responsible for performing a large number of tasks, then in some senses you're doing well. The utilization of the system is likely to be high, and the management costs are lower because there's only one machine. On the other hand, the more services you cram onto a single machine, the bigger the risk you're taking. Even if the machine is able to cope with the work load, what happens when it needs to be maintained or restarted? What if one of the services needs to be updated? If all the services need to be accessed at once, this is less of a problem. If the services are used by different groups of users, then things are rather more difficult to manage. Can user X afford to take a total production outage, so that user Y can get a fix for the intermittent problem they're seeing?

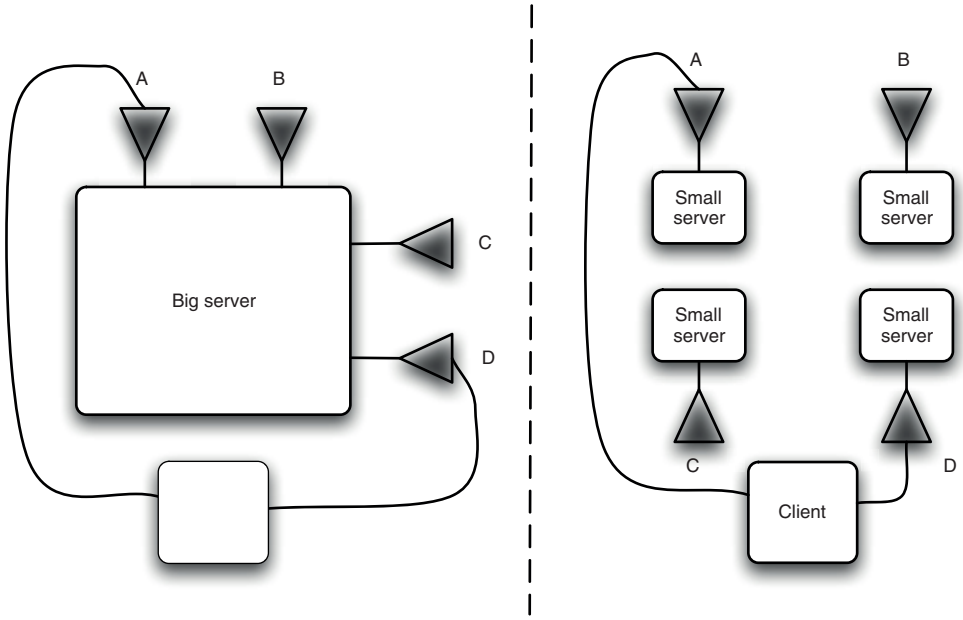


**Figure 10.3** **If all your services exist on one big server, then maintenance to any of them can break a client. In this case, the client only needs to access services A and D, but suffers an unnecessary outage whenever B or C needs updating. If the services all exist on separate servers, then the client can continue to operate even if B and C are no longer available.**

Remoting allows you to separate services onto different machines so that they can be more independently maintainable; you can even think of this as being like modularity. If all of your services are tightly coupled (by being on the same machine), then you have problems making a change to any of them. Furthermore, having the services on different machines makes it easier to decouple them, meaning that horizontal scaling for performance or reliability in the future will be much easier (see figure 10.3).

Sometimes availability concerns cut both ways: not only is it necessary for a service to be accessible independently of another, but sometimes you may not want a service to be accessible from the outside world.

#### PROTECTING ACCESS AND ENCOURAGING CO-LOCATION

Computing systems are a large investment, and almost always contain sensitive data or provide a mission-critical function. As a result, they need to be protected from external influences, both malicious and poorly written ones. Firewalls are a vital piece of infrastructure in enterprise networks, preventing access from one area of a network to another, except for specific, configurable types of traffic. It's extremely rare for an enterprise system, such as a corporate database or server room, not to be protected by one or more firewalls.

Using firewalls is essential, but it does cause some problems. If the Fancy Foods superstore protects its database behind a firewall and users can't access it, then it won't be

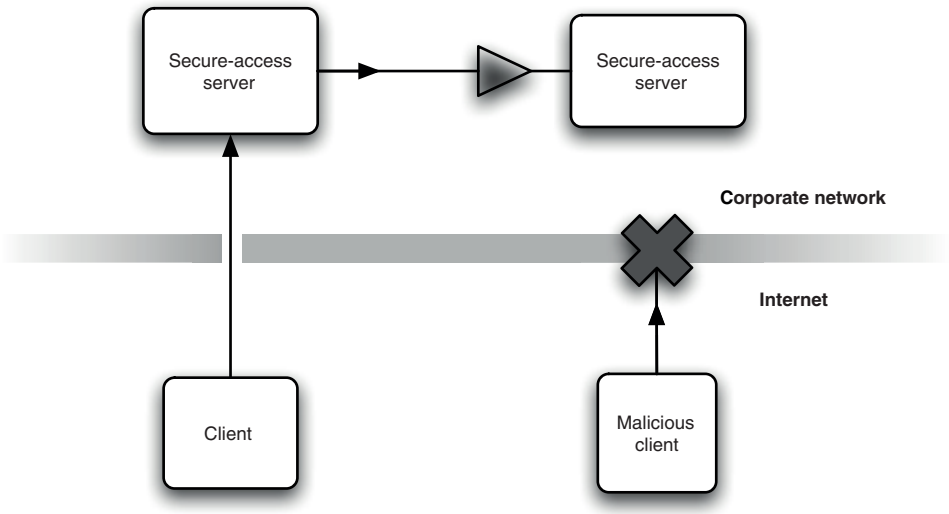


**Figure 10.4 In combination with a firewall, an intermediate server can be used to provide safe access to a secure system without allowing direct access to it.**

much use at all. On the other hand, you don't want users to have free access to the database machine, or probably even access to the network port exposing the database.

Remoting allows you to get around these sorts of problems by accessing a third party. If you have a reasonably secure intermediary system, then you can use that to access the backend service that you need. The intermediary is trusted to only access the database in a safe way, and so can be allowed through the firewall, whereas the client speaks to the intermediary and is shielded from the valuable content of the database (see figure 10.4).

This model isn't useful only for improving security; it also has a significant role to play in the performance of your system. It's best if services you access regularly can be accessed quickly. This is most likely if you're physically close (co-located) to the service you want to access. If performing a particular task requires you to access a service multiple times, then sending a single request to an intermediary, which then makes the rest of the calls and returns the result, can provide a significant performance boost.

Although remoting does have a number of advantages, it should be noted that there are some drawbacks when accessing remote systems.

### 10.1.2 The drawbacks of remoting

Although remoting is a critical part of enterprise programming, there are good reasons why it isn't used a lot of the time. Probably the single biggest reason that people avoid remoting is because of performance.

#### WHY REMOTING IS SO SLOW

One of the big lessons for most programmers is to avoid prematurely optimizing your code; still, it's worth thinking about trying to avoid things that have a negative impact

on performance. Typically, people learn that data stored in a processor cache can be accessed so quickly that it's effectively free; that data in memory can be accessed fast; that large data structures can begin to cause problems; and that data stored on disk is extremely slow. This advice boils down to "avoid I/O operations where you can."

Remoting, by its nature, requires some level of network communication, which is the key reason for its comparatively abominable performance. Even if you're making a method call that passes no arguments and has no return value, the speed of the network is a critical factor. In a typical modern system, a processor will have to wait for less than a nanosecond to load the next operation instruction from an internal cache, or up to a few tens of nanoseconds to load instructions or data from main memory. On a typical, small-scale network, fetching even a few bytes of data would be measured in hundreds of microseconds or possibly milliseconds; on a large network, or perhaps over the Atlantic, this latency can easily reach more than 100 milliseconds. As you can see from figure 10.5, any method invocation that you need to make over the local network is likely to be *a thousand times slower* than performing it locally.

Invocations over the internet can be *a million times* slower than local ones. In this measurement, we're ignoring the length of time it takes for the method to run. If the method takes a second to complete, then the remoting overhead is comparatively small. We're also assuming that the method parameters and return value are sufficiently small that transferring them will take negligible time. If the data to be transferred is large, say, 50 kilobytes, then this can be a much larger factor than the network latency (about a millisecond on a fast local network, or 10 to 15 seconds over a slow mobile phone network). Any way you slice it, making remote calls isn't a cheap thing to do.

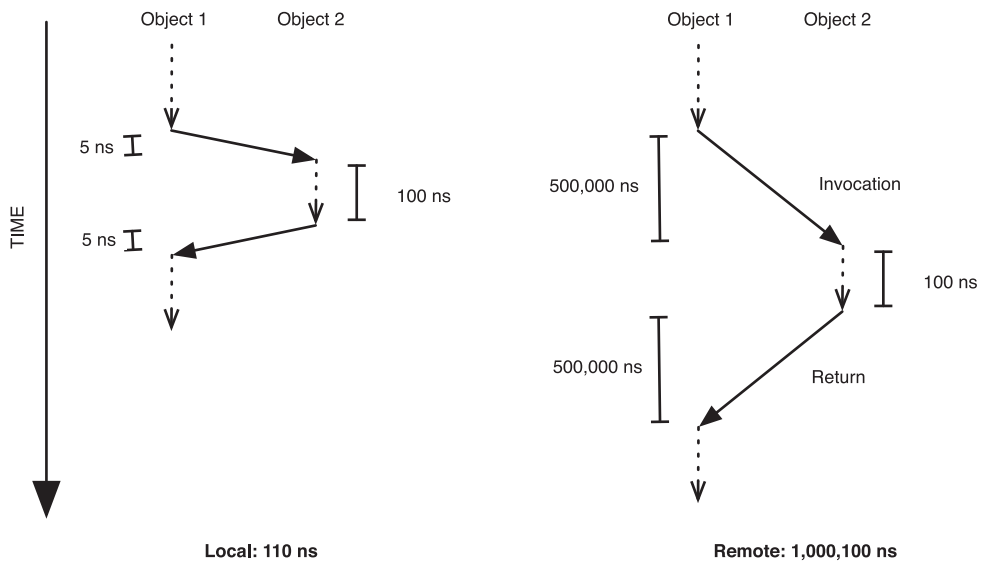


**Figure 10.5 The execution times for remote methods are dramatically increased by the latency of the network, even on a local network with a 500 microsecond latency.**

### REMOTE CALL SEMANTICS

One of the things that often surprises people about calling remote objects in Java is a subtle change in the semantics of the language. Java inherits a lot of its syntax and behavior from C. This includes its behavior when calling methods that use a pass-by-value mechanism for method parameters. This means that when you pass a value to a method (say, an `int`) the method gets a *copy* of the `int`, not the original. Therefore, if you modify the `int` in your method, the `int` the caller had will remain unchanged. This initially surprises many people, because they're used to the fact that if you pass an object as a method parameter, then the method operates on the same object, not a copy. There isn't a difference between objects and primitives passed as parameters, but you do need to know that what you pass to a method isn't the object itself, it's a *reference* to the object, and that's passed as a copy. If object references weren't passed as copies, then you would be able to re-assign your caller's object reference to some other object. This would be confusing and open to abuse; what if someone were allowed to replace the unmodifiable collection you gave them with a modifiable `HashSet` without you knowing!

Even though it doesn't always feel like it, Java uses pass-by-value semantics for local calls. You can, however, modify objects that are passed to you and expect those modifications to be visible to the person who called you. This is why you can write methods like `public void fillMap(Map<String,String> map)` to add entries to a map passed to a method. Remote Java calls are different. The local call in your `Map` filling method relies on the fact that the method has access to the same `Map` in memory that the caller has. If the method were running on a remote machine, then this wouldn't be the case. In a remote call, all of the method parameters have to be serialized over the network to the remote machine. This forces Java to copy not only the object reference, but the whole object as well. If your `Map` filling method were running on a remote server, then it would be operating on a copy of the `Map`, not the original reference, and you'd find that any `Map` you tried to fill would remain completely unchanged on the client.

The fact that remote calls behave differently from local ones isn't necessarily a disadvantage, and in many cases doesn't matter at all; however, it's a significant change to the normal flow of Java code. As such, you should be careful to ensure that any interface you write that you might want to expose remotely doesn't rely on changing the state of objects passed in.

### RELIABILITY AND REMOTING

Performance isn't the only thing that suffers when you make remote calls. When you make a local call, you can usually rely on an answer coming back. You can also rely on the fact that the data you need to execute will be available and not corrupted. Unfortunately, networks give you no such guarantee.

When you make a remote call, you usually need to wait for a reply indicating that the remote process has completed, often with a return value. You might receive your reply within a millisecond, or it might take several seconds, or even minutes. Crucially, you might *never* receive a reply from the remote server if, say, the server went down

after receiving your request, or if a network problem has cut off the route between you and them.

It's impossible to know if a response is about to arrive, or if it will never arrive. As a result, you need to be careful when making remote invocations. Error conditions aren't uncommon, and you must be careful to avoid ending up in an inconsistent state. To ensure this is the case, almost all enterprise applications that make use of remoting use transactions to coordinate any resources they access. Advanced remoting models assist the user here by allowing the transaction context to be propagated across to the remote machine. At this point, you're running in a *distributed* transaction, namely, one that's spread across more than one machine. The transaction is coordinated such that both systems can recover from a failure on either side of the remote link.

### 10.1.3 Good practices and the fallacies of remoting

The problems with remoting are well known, and a particularly good description of them is encapsulated in "Fallacies of Distributed Computing Explained" (Arnon Rotem-Gal-Oz; see http://www.rgoarchitects.com/files/fallacies.pdf). People make assumptions about distributed systems that are rarely, and often never, true:

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- There's one administrator
- Transport cost is zero
- The network is homogeneous

You may notice that we used a number of these pieces of information to explain why remoting isn't used all the time. In general, they indicate a few good practices for remoting:

- Make as few invocations across the remote link as you can.
- Try to keep parameter data and returned data small.
- Avoid hardcoding any locations for remote services.
- Be prepared to handle timeouts, badly behaved clients, and erroneous data.

Now that you know a little more about the principles of remoting, it's time for us to look at the enterprise OSGi standard for remoting, the Remote Services Specification.

## 10.2 The Remote Services Specification

One of the primary aims of the Remote Services Specification is that it should be as natural as possible to make use of a remote service from within an OSGi framework. One of the big drawbacks in many remoting models is that there are a number of

hoops you have to jump through to access a remote service. OSGi minimizes this problem by leveraging existing OSGi service mechanisms. In normal OSGi, bundles communicate via the OSGi Service Registry, which shields service clients from service providers. It turns out that not only is this a good thing for modularity within the local system, but that it provides excellent modularity for remote services as well!

The Remote Services Specification defines a mechanism by which a service can be exported from the Service Registry in one OSGi framework, making it appear in another OSGi framework. As far as the client is concerned, it can then consume both local and remote services in exactly the same way. From the provider's perspective, things are equally simple; they register a service into their local Service Registry, and then wait for incoming requests. These requests may be from local bundles, or from remote ones; the provider doesn't need to know or care.

To achieve this model, the Remote Services Specification defines two different roles for the remoting implementation, which is usually known as the *distribution provider*.

### 10.2.1 Exposing endpoints

When a local OSGi service is made available for remoting, it needs to be exposed as an endpoint. This endpoint could be anything from a web service to a proprietary bitstream format; the important thing is that the endpoint is accessible from outside the framework. This part of the process is relatively simple, and it's correspondingly easy to indicate to a distribution provider that a service should be made remotely accessible. To do this, you make use of the whiteboard pattern. Rather than looking up a remoting provider and providing it with a remotable object, you register that remotable object as a normal OSGi service. To indicate that your service should be made available remotely, you add the `service.exported.interfaces` property to your service. The value of this property defines the interface names that should be exposed remotely. This is a good example of section 5.2.4 in action (see figure 10.6).

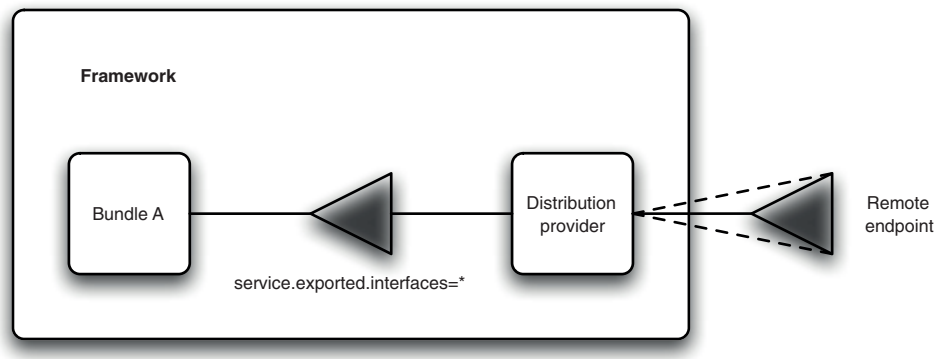


**Figure 10.6 A distribution provider uses the whiteboard pattern to locate remotable services and expose them outside the OSGi framework.**

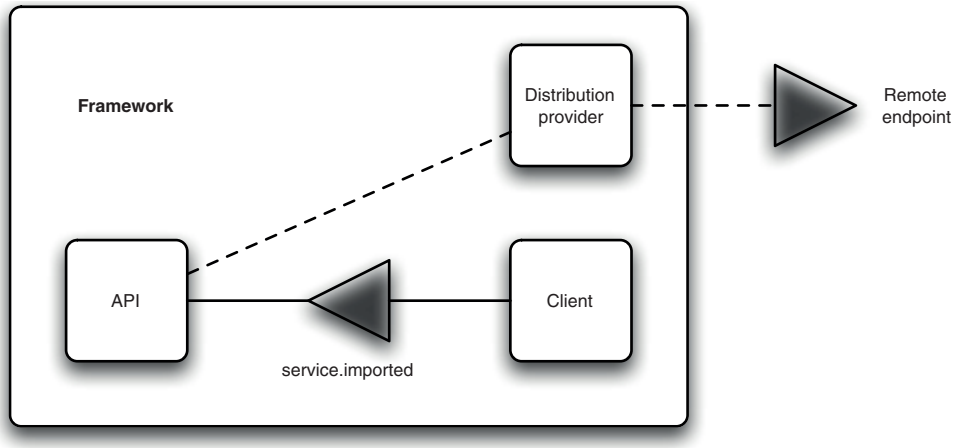


**Figure 10.7 A distribution provider can also create OSGi services within an OSGi framework that delegate to remote endpoints.**

### 10.2.2 Discovering endpoints

Both registering and consuming remotable services is easy, mostly because of the service discovery part of the distribution provider. This part of the distribution provider is responsible for detecting when new remote services are available, and they're for registering proxy services in the local framework. These proxy services mirror the backing services in the remote framework, including any changes in their service properties. The proxy services are also unregistered if the remote service becomes unavailable. Imported services (ones that proxy a service in a remote framework) also have a number of other properties registered by the distribution provider. Importantly, if the `service.imported` property is set to *any value*, then this service is a proxy to a remote service in another framework (see figure 10.7).

One further detail to note is that distribution providers are only required to support a limited set of parameter types when making remote invocations. These are primitive types, wrapper types for primitives, and Strings. Distribution providers must also cope with arrays or collections of these types.

Now that you understand the basics of the Remote Services Specification, let's try extending your superstore with a remote department.

## 10.3 Writing a remotable service

In chapter 3, you built an extensible facility for special offers. This is an excellent opportunity for you to add a remote special offer service. Writing a remote department for your superstore will be simple. It'll be the foreign foods department, and you'll base it heavily on the departments you've written before. Also, thanks to OSGi's modular, reusable bundles you can take the `fancyfoods.api`, `fancyfoods.persistence`, and `fancyfoods.datasource` bundles from chapter 3 without making any changes at all. Figure 10.8 shows the architecture of the distributed Fancy Foods system.

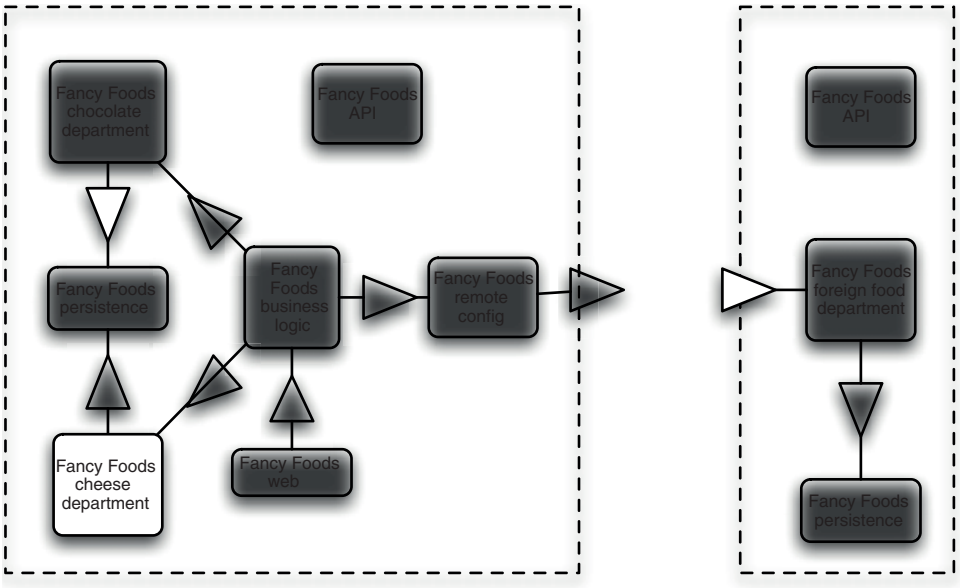


**Figure 10.8 The distributed Fancy Foods system includes two frameworks. The original framework includes a new `fancyfoods.remote.config` bundle; the remote framework reuses some Fancy Foods bundles and adds a new `fancyfoods.department.foreign` bundle.**

The first thing you need is an implementation for your remote service.

### 10.3.1 Coding a special offer service

To simplify things, your `SpecialOffer` implementation is effectively identical to the cheese department one, with a slightly different query, as shown in the following listing.

**Listing 10.1 A foreign foods special offer service**

```
public class ForeignFoodOffer implements SpecialOffer {

    private Inventory inventory;

    public void setInventory(Inventory inventory) {
        this.inventory = inventory;
    }

    @Override
    public String getDescription() {
        return "Something exotic to spice up your weekend.";
    }

    @Override
    public Food getOfferFood() {
        List<Food> foods =
            inventory.getFoodsWhoseNameContains("Foreign", 1);
        Food mostStocked = foods.get(0);
        return mostStocked;
    }
}
```

You'll also reuse the trick for populating the database, as follows.

**Listing 10.2 Populating the foreign food database**

```
public class InventoryPopulater {

    private Inventory inventory;

    public void setInventory(Inventory inventory) {
        this.inventory = inventory;
    }

    public void populate() {
        boolean isInventoryPopulated = (inventory.getFoodCount() > 0);

        if (!isInventoryPopulated) {
            inventory.createFood("Foreign Sushi", 3.45, 10);
            inventory.createFood("Foreign Borscht", 1.81, 15);
        }
    }
}
```

Once again, you can use Blueprint to glue everything together, as shown in the following listing.

**Listing 10.3 Exposing your remote service**

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint
    xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

    <service interface="fancyfoods.offers.SpecialOffer">

        <service-properties>

            <entry key="service.exported.interfaces"
                   value="fancyfoods.offers.SpecialOffer"/>

        </service-properties>

        <bean class="fancyfoods.department.foreign.ForeignFoodOffer">

            <property name="inventory" ref="inventory"/>

        </bean>
    </service>

    <reference id="inventory" interface="fancyfoods.food.Inventory"/>

    <bean class="fancyfoods.department.foreign.InventoryPopulater"
          activation="eager" init-method="populate">
        <property name="inventory" ref="inventory"/>
    </bean>
</blueprint>
```

Mark your service remotable

Eagerly populate database

Having built your bundle, you can deploy it into your test environment along with the API, datasource, and persistence bundles. You can also see your remote service registered in the Service Registry (see figure 10.9).

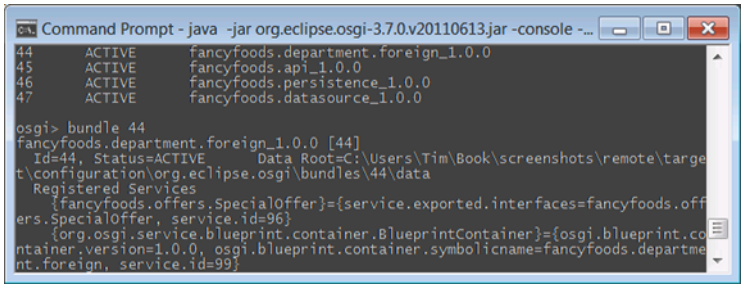


**Figure 10.9 The local view of your remote service**

Your foreign foods department can't do much at the moment; it's not even available remotely. For that you need a remote services implementation.

## 10.4 Adding in your remote service using Apache CXF

Apache CXF hosts the Distributed OSGi project, which is the reference implementation of the Remote Services Specification. (Other implementations, such as Eclipse ECF and Paremus Nimble, are also available.) Distributed OSGi makes use of CXF's web services runtime to expose OSGi services as web services. Distributed OSGi then uses CXF's web service client code to implement local OSGi service proxies to remote web services. By gluing these two pieces together, you get a complete remote services implementation.

Distributed OSGi is easy to use and will allow you to quickly and simply expose your foreign foods special offers to the outside world.

### 10.4.1 Making your service available

Distributed OSGi is available in two forms: as a set of OSGi bundles, or as a single bundle that packages its dependencies (sometimes known as an *uber bundle*). Both forms are reasonably common ways to release OSGi services; the former gives good modularity and fine control, whereas the latter is useful for getting up and running quickly. Because you want to start selling your foreign food as soon as possible, you'll be using the single bundle distribution of Distributed OSGi 1.2, which is available at http://cxf.apache.org/dosgi-releases.html.

You need to get the Distributed OSGi implementation into your framework, which can easily be achieved by dropping it into the load directory of your running framework. Adding this bundle will cause a huge flurry of activity, but importantly it will create a web service for any service that declares a `service.exported.interface`!

#### CONFIGURING YOUR DISTRIBUTION PROVIDER

It isn't immediately clear where Distributed OSGi has registered your web service, and it also isn't clear how you could get the service to be registered somewhere else. At this point, it's worth knowing that the Remote Services Specification defines a configuration property, `service.exported.configs`. This property can be used to define *alternatives* (different places where the same service should be registered) and *synonyms* (different configurations for the same service endpoint). If more than one configuration is required, then the service property can be an array of configurations.

In Distributed OSGi, `service.exported.configs` can be used to define the URL at which your web service is registered. To start, set the `service.exported.configs` property value to `org.apache.cxf.ws`. When Distributed OSGi sees this value for the `service.exported.configs` property, it begins looking for other, implementation-specific properties. The property for setting the web service URL is `org.apache.cxf.ws.address`. By default, Distributed OSGi URLs are of the form http://localhost:9000/fully/qualified/ClassName.

Distributed OSGi offers a number of other mechanisms for exposing your service, such as JAX-RS (all of this is implementation-specific). The Remote Services Specification requires implementations to support services that don't specify any implementation-specific configuration. As such, you'll keep your service clean and vendor-neutral by not specifying any configuration.

### VIEWING YOUR REMOTE SERVICE

Although you haven't imported your service back into an OSGi framework, you shouldn't underestimate the power of what you've achieved. What you've got here is a remotely available OSGi-based web service. You can even get hold of the WSDL (Web Services Description Language) for your service by going to http://localhost:9000/fancyfoods/offers/SpecialOffer?wsdl. For those of you who know a little about web services, this is exciting stuff: you can see a complete description of your OSGi service (see figure 10.10).

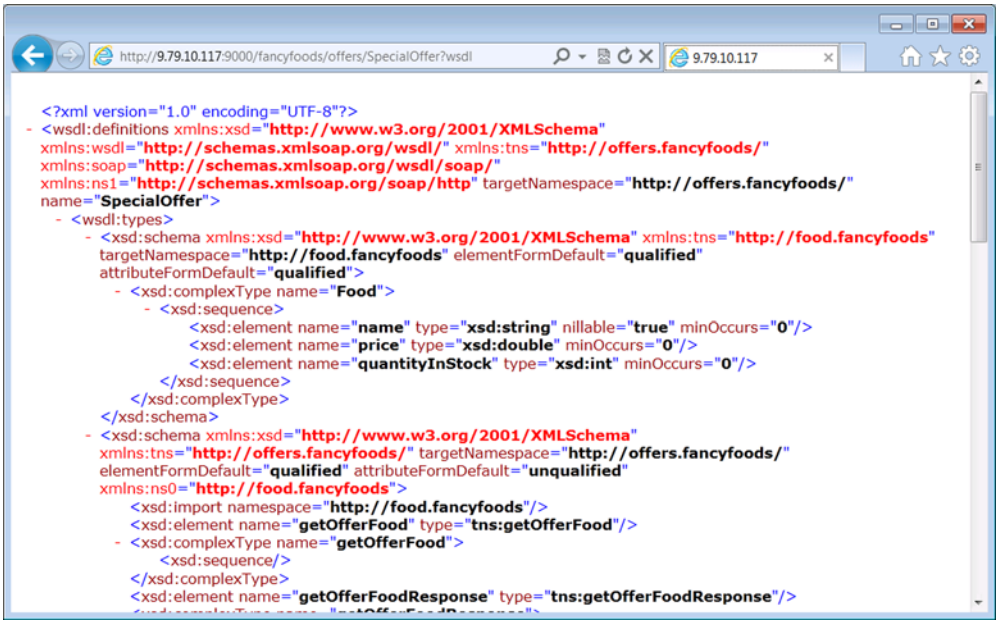


**Figure 10.10** **The WSDL for your remote service**

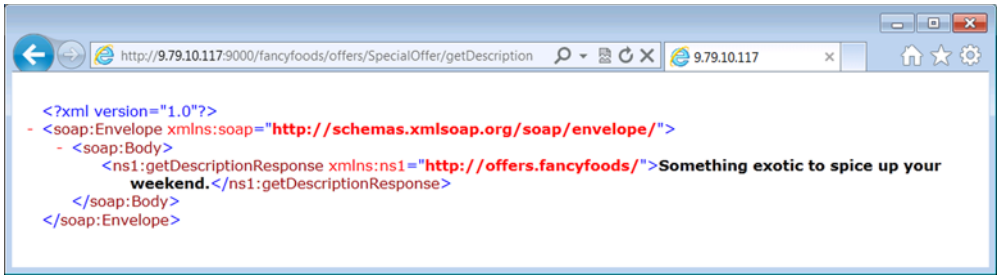


**Figure 10.11** **Invoking your remote service**

> **WARNING: WHO'S THAT URL FOR?** In some cases, Distributed OSGi tries to be too helpful in determining the default URL that it should use to register services. Rather than using localhost, the default becomes the current external IP address of your machine. If you find that your WSDL isn't visible on localhost then you should try using your current IP address instead. Unfortunately, you'll have to substitute this IP for localhost throughout the rest of this example.

We do understand that many of you won't get as excited by WSDL as we do, so as a treat why don't you try visiting http://localhost:9000/fancyfoods/offers/SpecialOffer/getDescription? We think you'll be rather pleased by what you find. See figure 10.11.

You now have conclusive proof that CXF and Distributed OSGi have successfully exposed your OSGi service as a publicly accessible web service. Now it's time to look at how you can get that service into another running OSGi framework.

### 10.4.2 Discovering remote services from your superstore

We're sure you'll agree that exposing your OSGi service was extremely easy. Happily for us, Distributed OSGi makes consuming the service back into another OSGi framework just as easy.

To start with, you need to get your application from chapter 3 up and running in a different OSGi framework. At this point, you can use the application happily, but you won't see any offers for foreign foods (see figure 10.12).

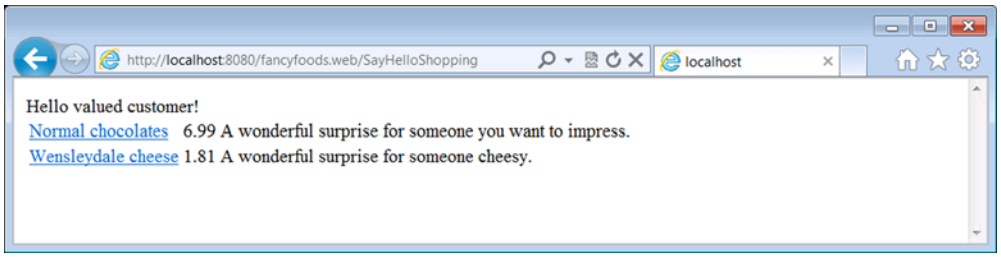


**Figure 10.12** **Without your remote service**

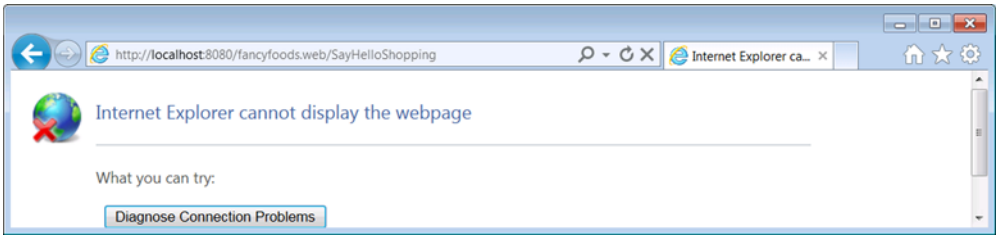


**Figure 10.13 After adding Distributed OSGi**

To get a remote service imported into your framework, you need to install the Distributed OSGi bundle, which again you can do by dropping it into the load directory of your framework. Sadly, this is enough to cause a problem. As you can see from figure 10.13, adding Distributed OSGi has caused something to go wrong with your application.

Although the problem may look severe, it's not as bad as it looks. Distributed OSGi makes use of the Jetty Web Container project to provide its HTTP web service hosting. This is exactly the same web container that your basic runtime uses, and when the Distributed OSGi bundle starts up, it ends up accidentally reconfiguring your runtime's Jetty instance as well as its own!

Unfortunately, the reconfiguration that happens moves Jetty onto a randomly assigned free port. Luckily, it's easy for you to find out what port Jetty was reassigned to. If you look at your runtime's `pax-web-jetty-bundle` in the OSGi console, you see something like figure 10.14.

If you look closely at figure 10.14, you can see that `pax` has registered an OSGi HTTP Service implementation in the Service Registry, and that one of the service properties for this service is called `org.osgi.service.http.port`, and it has a numeric value. This value is the port that you can access your application on. If you change your URL to use this port instead, then things look much rosier again (see figure 10.15).

Although your application is still working happily, notice that your foreign foods special offer is nowhere to be seen. This is because you haven't given Distributed OSGi any information about the remote endpoints you want it to connect to. If you want to go any further, you'll need to learn more configuration.

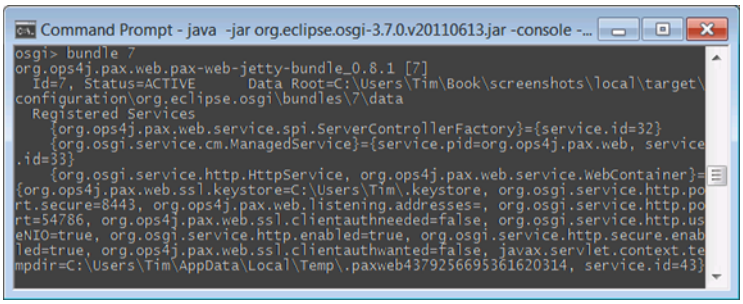


**Figure 10.14 Your Jetty service properties**

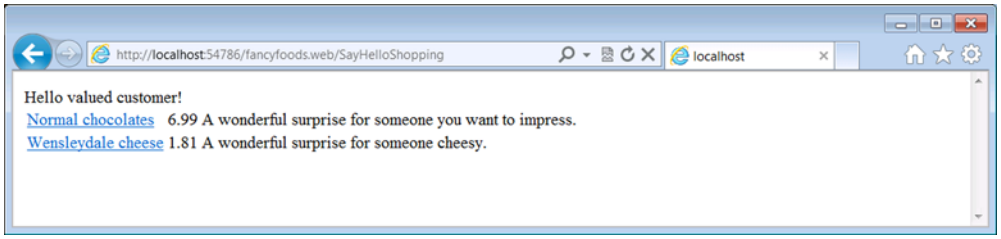


**Figure 10.15 Using your new Jetty port**

### CONFIGURING ACCESS TO A REMOTE ENDPOINT

Before describing the details of configuring remote endpoints in Distributed OSGi, we feel that it's important to say that configuration isn't always required. A number of *discovery* services can be used to automatically locate endpoints, but these aren't part of the OSGi standards. What you'll do is use part of an OSGi standard called the Remote Service Admin Service Specification. This specification has an odd-looking name, but effectively it's a standard describing a common configuration and management model for different Remote Services Specification implementations.

You'll provide your configuration using the Endpoint Description Extender Format. This is a standard way of providing remote services endpoint configuration inside a bundle using the Remote Services Admin Service, and an alternative to dynamic discovery. To start with, you'll need a basic manifest, as shown in the following listing.

**Listing 10.4 A remote services endpoint configuration bundle manifest**

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName: fancyfoods.remote.config
Bundle-Version: 1.0.0
Remote-Service: OSGI-INF/remote-service/*.xml
```

The new and interesting entry in your manifest is the `Remote-Service` header. This header signals to the extender that this bundle should be processed for remote service endpoint descriptions. The value of the header is a comma-separated list of XML endpoint descriptions, potentially including wildcards.

Other than its manifest, your bundle will only contain one file, OSGI-INF/remote-service/remote-services.xml. This file matches the wildcard description described in the `Remote-Service` header of the bundle's manifest. This file uses a standard XML namespace to describe one or more endpoints that should be exposed by the remote services implementation. As a result, this file could be used with any remote services implementation, not just Distributed OSGi, as follows.

**Listing 10.5 A remote services endpoint configuration file**

```
<endpoint-descriptions xmlns="http://www.osgi.org/xmlns/rsa/v1.0.0">
  <endpoint-description>
```

```
    <property name="objectClass">
      <array>
<value>fancyfoods.offers.SpecialOffer</value>
      </array>
    </property>

<property name="endpoint.id"
      value="http://localhost:9000/fancyfoods/offers/SpecialOffer"/>

<property name="service.imported.configs"
      value="org.apache.cxf.ws"/>
  </endpoint-description>
</endpoint-descriptions>
```

**Service interface(s) exposed**

**Your endpoint location**

**Type of endpoint to import**

As you can see in listing 10.5, you provide the URL for your endpoint, as well as its interface type and some information about how the service was exported, in this case as a CXF-generated web service. This is rather brittle, unless you have a known proxy server that can get the routing right for you. For now a fixed URL is fine. If you build your manifest and XML configuration into a bundle and drop it into the load directory of your client system, then something rather magical happens when you refresh the front page of your superstore.

As you can see in figure 10.16, your special offers now include some foreign food from your remote system! Note that you didn't make any changes to your API or to the client application, and that you wrote a total of two manifests, two classes, and two XML files. This is how easy life can be when you have a modular OSGi application. We hope it's as exciting for you as it is for us!

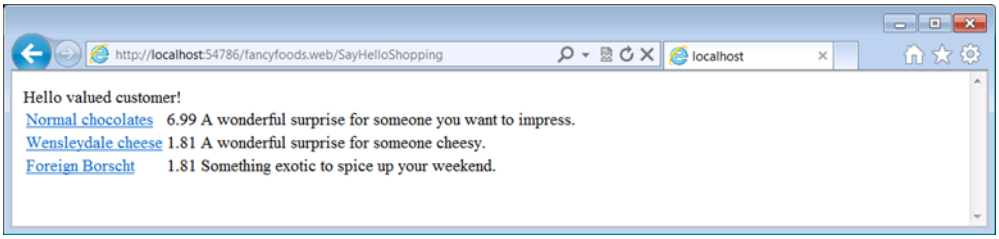


**Figure 10.16** **Your remote-enabled superstore**

## 10.5 Using your remote application

You could stop adapting your application here, now that you know how to expose and consume a remote OSGi service in a transparent way using only standard configuration. On the other hand, there was more to your application than the special offers on the front page. What happens if you try to buy some of the foreign food that you've added? Clicking on the link to your special offer yields the page shown in figure 10.17.

The page in figure 10.17 is promising; everything looks about right, but what happens if you attempt to complete a purchase? See figure 10.18.

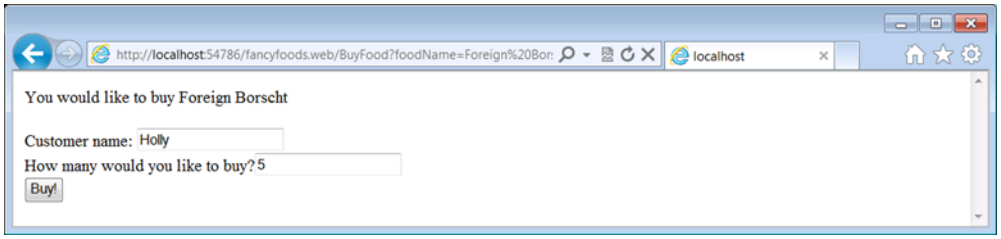


**Figure 10.17** **Trying to buy some foreign food—part 1**

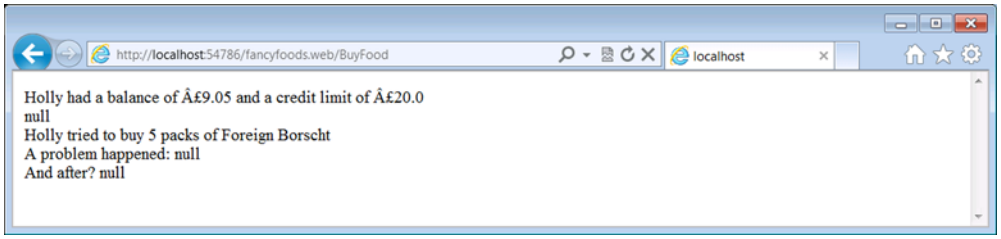


**Figure 10.18** **Trying to buy some foreign food—part 2**

Oh dear. Clearly there's something wrong here; your purchase should be able to proceed, but it fails. The reason for this is simple, and it has to do with your database. You may remember that you've been using a local, in-memory database. Hopefully, you'll remember that you copied your datasource bundle into both the local and the remote framework. Doing this did let you get up and running more quickly, but it also created a second copy of your database, which definitely isn't good for your application!

Fortunately, the fix for this situation is simple. You need to build a new version of your datasource bundle that speaks to a remote database, rather than creating a local one.

### 10.5.1 Setting up your remote database connections

A huge number of powerful, scalable, network-enabled database implementations exist, any one of which would be more than capable of handling your new requirements. For simplicity you'll stick with the Apache Derby database, but you should feel free to use another if you fancy an extra challenge.

Setting up datasources for a remote Derby instance will require some changes to the Blueprint for your datasource bundle. Because there's no code in your datasource bundle, it's nontrivial to say what the new version of your datasource bundle should be. Our view is that moving the database to a remote machine, making it possible for a number of clients to concurrently share access, and making the database persistent is a major change, and therefore warrants an increment to the major version of the bundle.

The changes to your Blueprint are simple. You can leave the `service` elements untouched; what you need to do is to make some changes to the beans that they reference. In listing 10.6, you can see that all you need to do is to change the implementation class of your beans and supply some additional properties to indicate where your Derby

server will be running. After you've made those changes, all you need to do is to increase the version of your bundle and package it up again.

**Listing 10.6 Exposing your remote service**

```
<bean id="derbyDataSource"
    class="org.apache.derby.jdbc.ClientDataSource">
    <property name="createDatabase" value="create"/>
    <property name="databaseName" value="fancyfoodsDB"/>
    <property name="portNumber" value="1527"/>
    <property name="serverName" value="localhost"/>
</bean>

<bean id="derbyXADataSource"
    class="org.apache.derby.jdbc.ClientXADataSource">
    <property name="databaseName" value="fancyfoodsDB" />
    <property name="createDatabase" value="create" />
    <property name="serverName" value="localhost"/>
    <property name="portNumber" value="1527"/>
</bean>
```

Network Derby driver

No longer an in-memory database

Remote Derby location

### STARTING YOUR DERBY SERVER

It's easy to get your Derby server running. First you need to download a copy of Derby, because the one in your basic runtime doesn't have any network Derby support. Copies of Derby can be obtained from http://db.apache.org/derby/derby_downloads.html.

When you have a copy of Derby, you need to start the server running. This can be done by unzipping the Derby zip and issuing a single command from the Derby lib directory:

```
java -jar derbyrun.jar server start
```

Now that your Derby instance is running, you need to alter your basic runtime to support remote client connections to Derby. To do this, you need to replace the existing Derby JAR, which will be called something like `derby_10.5.3.0_1.jar`, with the `derbyclient.jar` from the Derby you downloaded. To get this bundle picked up in your basic runtime, you'll also have to edit the configuration for your framework. This configuration file is called configuration/config.ini.

**WARNING: DERBY'S SPLIT PACKAGE** The version of Derby in your basic framework only provides support for embedded databases. As a result, it doesn't include the remote client drivers. Unfortunately, the remote client drivers and embedded drivers are in the same package. This means that you don't get a resolve-time error about the missing class. If you don't update your basic framework to replace the embedded Derby package, then you'll see a `ClassNotFoundException` from the Blueprint container trying to create your datasource.

After you replace the Derby bundle with the remote client, you may find that you start seeing a `ClassNotFoundException` from another datasource bundle. You aren't using this bundle, so it's nothing to worry about. You can remove the bundle from your basic runtime if the exception makes things difficult to follow.

When you open configuration/config.ini, you'll see a large number of framework properties used to determine the bundles that get started in your basic runtime. You're looking for the Derby entry that's something like the following:

```
derby-10.5.3.0_1.jar@start,\
```

You need to change the JAR name in this entry to `derbyclient.jar` so that the Derby network client code is available, as follows:

```
derbyclient.jar@start,\
```

After you've done this in both of your frameworks (the service provider and consumer frameworks), it's time to start up your application again. Initially there should be no difference at all; after all, your previous database had perfectly good read-only behavior (see figure 10.19).

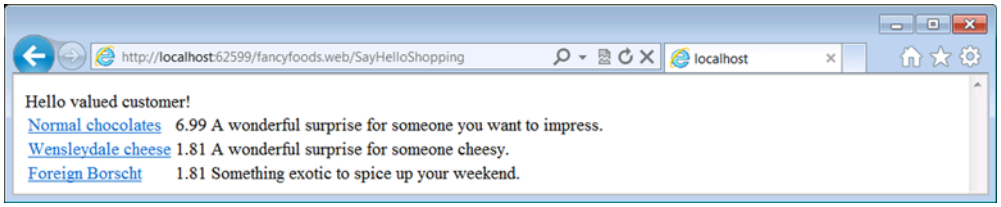


**Figure 10.19 Running with a remote database**

If you attempt to buy some of your foreign food, then, as before, you're allowed to put some values into the form (see figure 10.20).

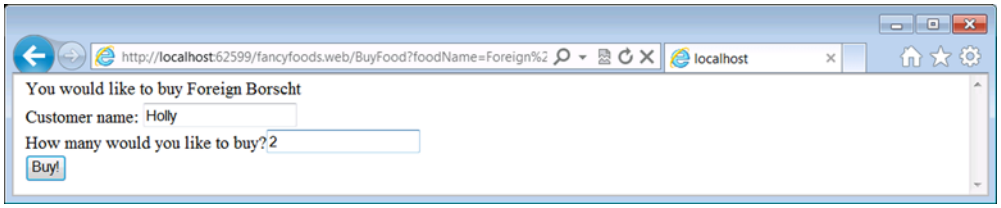


**Figure 10.20 A second attempt to buy from your foreign foods department**

This time, however, the result is a lot more pleasant! See figure 10.21.

Now that you have a single backing database, you're able to interact with the same data that your remote system is. As a result, your superstore can be distributed across

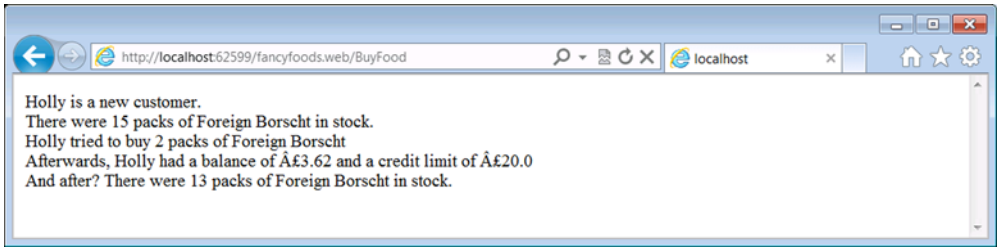


**Figure 10.21 A successful purchase**

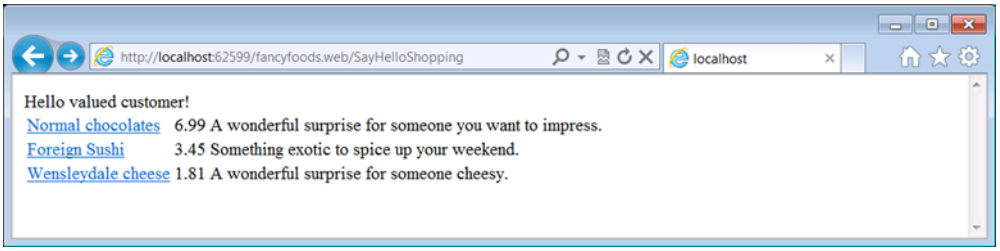


**Figure 10.22 A new special offer!**

as many machines as you want. As a final proof that the remote service is behaving properly, try making a bulk order for some more of your foreign food. This will cause your stock to drop, and if you buy enough, should cause your foreign foods department to select a new special offer (see figure 10.22).

We hope that you're now convinced that your remote service is not only live, but operating on the same data as the consuming framework. One thing you haven't done is propagate distributed transactions. This isn't always a problem—for example, your special offer service is read-only, and it isn't a disaster if you advertise an offer that's been changed by another user's large order. Sometimes, however, it's vitally important that both ends of the remote call run under the same logical transaction, and this isn't an area where CXF is strong. Although Apache CXF is probably the simplest way to take advantage of OSGi remote services, it's not the only one.

## 10.6 Using SCA for remoting

The Apache Tuscany project provides an alternative remote services implementation based on the Service Component Architecture (SCA). SCA is designed to enable component-oriented programming, in which discrete components are wired together through well-defined endpoints. These components may be co-located on the same system or—more usefully for remote services—distributed across a network.

### 10.6.1 Apache Tuscany

The Apache Tuscany project is an open source implementation of the SCA specification based on a Java-language runtime. As well as the core SCA runtime, Tuscany includes a number of SCA component implementation types. Importantly for OSGi, the Tuscany project hosts the reference implementation of the OSGi SCA component type, known as `<implementation.osgi/>`.

Because service component architecture is all about services, and OSGi services are one of the great features of the OSGi programming model, you may be guessing that these two technologies are a good fit. Unsurprisingly, SCA's OSGi component type makes use of the OSGi Service Registry to provide and consume SCA services. An OSGi SCA descriptor (following the typical SCA syntax) can be used to select particular OSGi services from the Service Registry using service properties to identify particular implementations. These services are then exposed as endpoints through the SCA runtime.

In addition to exposing existing OSGi services to SCA applications, the OSGi SCA component type also allows SCA endpoints to be advertised and consumed as services in the OSGi Service Registry. The properties and interface with which the SCA service is registered are also defined in the SCA descriptor, but OSGi bundles can then consume the SCA service as if it were any other OSGi service.

#### USING TUSCANY AND ARIES TOGETHER

Much like CXF, you can integrate Tuscany into your little Aries runtime by downloading a Tuscany distribution and copying all the bundles into the load directory. Alternatively, you can merge the config.ini configuration files for the Aries and Tuscany runtimes. Tuscany can be downloaded from http://tuscany.apache.org/sca-java-releases.html.

> **WARNING: GETTING TUSCANY WORKING WITH ARIES** You may find that combining Tuscany and Aries isn't as straightforward as it was for CXF and Aries. When we tried it, we hit several issues and we needed to do some tweaking to get everything working together. At a minimum, you'll need a Tuscany release more recent than 2.0-Beta3, because some OSGi bundles didn't resolve in that release. Some of Tuscany's dependencies required Declarative Services, but no implementation was included in the Tuscany distribution we tried. You'll need to make sure a Declarative Services implementation like Felix SCR is present in your runtime. Finally, Tuscany includes a bundle for the JPA API that's incorrectly labeled version 3.0, and this needs to be removed.

If getting a combined Tuscany-Aries runtime up and running is sounding like too much hard work, you might consider one of the prebuilt server stacks that have already done the integration for you. Remember that Apache Aries isn't intended to be an application server—it's a set of components that can be included in a server. IBM's WebSphere Application Server includes both an SCA runtime, based on Apache Tuscany, and an enterprise OSGi runtime, based on Apache Aries. We'll discuss other options for an enterprise OSGi stack more in chapter 13.

### 10.6.2 Importing a remote service

After you've got your Tuscany runtime up and running, you'll find that remote services with Tuscany looks a lot like remote services with CXF. Because you started off exporting a service with CXF, this time you'll go in the opposite order and start by importing a service using Tuscany. As with CXF, you'll need a remote services configuration file, as shown in the following listing.

**Listing 10.7 The remote services endpoint configuration file for SCA**

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoint-descriptions
    xmlns="http://www.osgi.org/xmlns/rsa/v1.0.0"
    xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"
```

```
    xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.1">
    <endpoint-description>
        <property
            name="objectClass"
            value="fancyfoods.offers.SpecialOffer" />

        <property
            name="remote.configs.supported"
            value="org.osgi.sca" />
        <property
            name="service.imported.configs"
            value="org.osgi.sca" />
        <property
            name="org.osgi.sca.bindings">
            <list>
                <value>{http://fancyfoods}ForeignFood</value>
            </list>
        </property>
    </endpoint-description>
</endpoint-descriptions>
```

**Type of endpoint to import**

**Pointer to endpoint configuration**

This is similar to the remote services configuration file you defined in listing 10.5. But the details of the configuration are different, because you're using a different remote services implementation. You set the `service.imported.configs` property to `org.osgi.sca`. Instead of providing lookup information for the service directly, as you did in listing 10.5, you provide a reference to an SCA *binding* by specifying a `org.osgi.sca.bindings` property.

The SCA bindings are set in a small SCA configuration file, as shown in the following listing.

**Listing 10.8 The SCA configuration for a WSDL binding of a remote service**

```
<scact:sca-config
    targetNamespace="http://fancyfoods"
    xmlns:scact="http://www.osgi.org/xmlns/scact/v1.0.0"
    xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912">
    <sca:binding.ws
        name="ForeignFood"
        uri="http://localhost:8081/foreignfood" />
</scact:sca-config>
```

Finally, you'll need to add two extra headers to your manifest, so that SCA can find the remote services and SCA configuration files:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName: fancyfoods.remote.config.sca
Bundle-Version: 1.0.0
Import-Package: fancyfoods.food;version="[1.0.0,2.0.0)",
 fancyfoods.offers;version="[1.0.0,2.0.0)"
Remote-Service: OSGI-INF/remote-service/*.xml
SCA-Configuration: OSGI-INF/sca-config/food-config.xml
```

### TRYING OUT THE SCA IMPORTED SERVICE

Let's have a look at how the remote services behave in your SCA runtime. Build the `fancyfoods.remote.config.sca` bundle and drop it into the load directory. You should see the bundle appear, but more interestingly, you should also see some new services appear. List the services that implement the `SpecialOffer` as follows:

```
osgi> services (objectClass=fancyfoods.offers.SpecialOffer)
```

As well as the familiar cheese and chocolate offers, there's a new offer, with lots of extra SCA-specific service properties. You may expect the bundle that registers the service would be the `fancyfoods.remote.config.sca` bundle. But it's the bundle that *consumes* which registers its remote representation. If no bundles consume the remote service, it won't even be registered. (And if you stop the `fancyfoods.remote.config.sca` bundle, the `SpecialOffer` service will be unregistered.)

At this point, you may be slightly suspicious. You haven't written the SCA foreign food service implementation, much less deployed and started it. How can a service be available when there's no backend?

If you try to access the Fancy Foods front page, you'll discover that the backing remote service is definitely missing, which makes a mess (see figure 10.23).

The SCA runtime registers services based on what's in the bundle metadata, but it doesn't require the backing services to be available until they're used. This may seem error prone, and in a sense it is—but it's merely reflecting the fact that remote invocations are naturally subject to service interruptions. Networks are inherently unreliable, and remote services may come and go, so even if the remote service is available at the time of registration, it might have vanished by the time it's used. This volatility is something to be aware of when using services that could be remote; more error handling

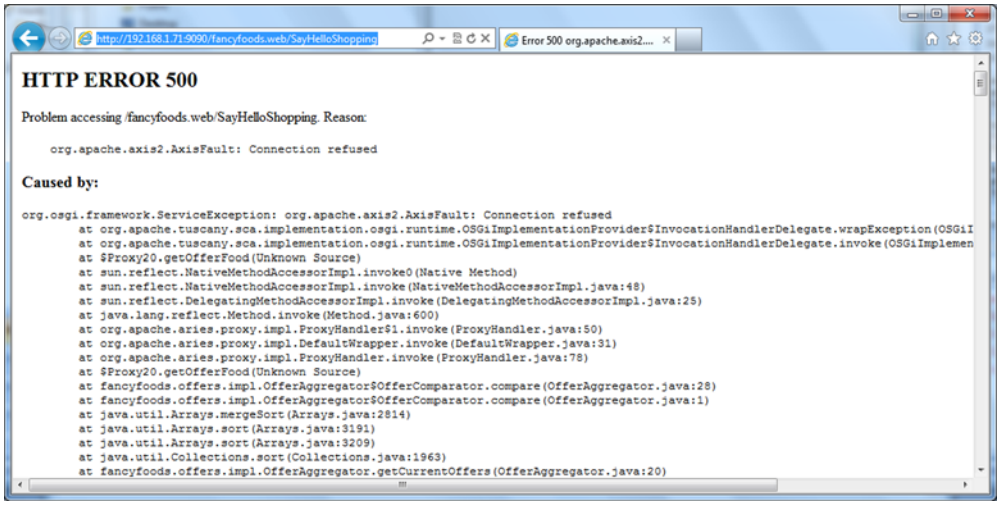


**Figure 10.23 If the remote implementation of a remote service is missing, the service will still be available, but using it will cause errors.**

may be required than you're used to. Remember that local proxies of remote services will have the `service.imported` property set.

### 10.6.3 Exporting remote services

We're sure you know how to write code for handling errors, so rather than demonstrating that, we'll skip straight on to ensuring the remote service is available, which is much more fun.

You'll use a bundle similar to the foreign food department bundle from section 10.3. The Blueprint for the foreign food service is nearly identical to listing 10.3, but you'll need to add some extra service properties to tie together the exported service and some extra SCA configuration:

```
<service interface="fancyfoods.offers.SpecialOffer">
        <service-properties>
            <entry key="service.exported.interfaces" value="*"/>
            <entry key="service.exported.configs" value="org.osgi.sca"/>
            <entry key="org.osgi.sca.bindings" value="ForeignFood"/>
        </service-properties>

        <bean class="fancyfoods.department.foreign.ForeignFoodOffer">
            <property name="inventory" ref="inventory"/>
        </bean>
    </service>
```

The extra SCA configuration file is exactly the same as that of the client-side bundle (listing 10.8), and is configured with a similar `SCA-Configuration:` reference in the bundle's manifest.

With the new `fancyfoods.department.foreign.sca` installed in the server-side framework, if you access http://localhost:8081/foreignfood?wsdl, you'll be able to see the generated WSDL description for the foreign food service.

### 10.6.4 Interfaces and services

The process of remoting services using SCA is similar to that of remoting them using CXF, although SCA does need some extra configuration files. But when you link up the remote service client to the remote services provider, an important difference between SCA and CXF becomes clear (see figure 10.24).

Remember that in section 10.2, we mentioned that distribution providers are only required to support distributed invocations with simple parameter types. Although CXF allowed you to use interfaces with much more complex parameters, such as `Food`, Tuscany's WSDL binding doesn't. This means the `SpecialOffer` interface can't easily be used for Tuscany remote services.

Remoting complex types is often handled by writing an `XMLAdapter` class to convert between complex and simpler types, and annotating method declarations or classes with the `@XmlJavaTypeAdapter` annotation. In the case of Fancy Foods, you'd add an annotation to the `SpecialOffer` class, as follows:

```
@XmlJavaTypeAdapter(value=fancyfoods.FoodAdapter.class,
  type=fancyfoods.foods.Food.class)
```

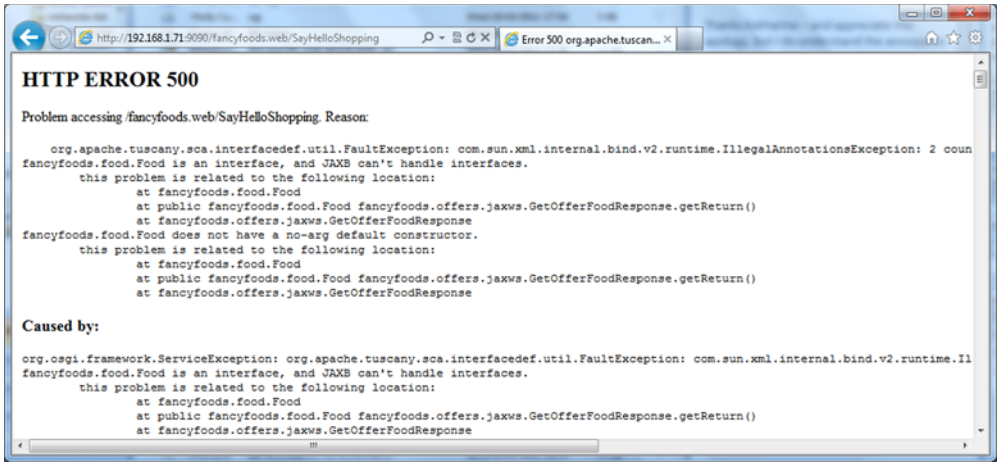


**Figure 10.24 As this server error demonstrates, SCA's WSDL binding only works well for services whose interfaces are restricted to using primitive types.**

The difficulty comes when you try to implement a `FoodAdapter`. By necessity, it will depend on a `FoodImpl` implementation class, which leaves the `SpecialOffer` interface with a dependency on an implementation class. If you're getting into the OSGi spirit of things, this should leave you feeling like you haven't brushed your teeth in a week.

A much more OSGi-centric solution to the problem is to wrapper the complex `SpecialOffer` service with a simpler `PrimitiveOffer` service for the purpose of distribution. A Blueprint reference listener can be used to track `PrimitiveOffer` services. The listener would automatically detect and consume each bound `PrimitiveOffer` service, and use it to generate and register a wrapper `SpecialOffer` service. But this significantly increases the complexity of using SCA for service remoting.

Does this mean you shouldn't use SCA for remoting your OSGi services? Not necessarily. Although we haven't provided a detailed example, SCA does provide support for distributed transactions, which may be critical to your application. Furthermore, as you'll see in chapter 11, SCA offers a lot more than remote services. You may find SCA remote services fit neatly into a broader SCA-based architecture. If your application server includes SCA, you can bypass the hand-assembly of your stack that we described earlier. And if your service interfaces are already based around primitives and Strings, you won't need to worry about rewriting services to avoid the use of complex types.

## 10.7 Summary

Remoting is generally considered to be one of the more difficult parts of enterprise programming. We hope that you can now see how the intrinsic dynamism and modularity of an OSGi framework dramatically simplifies this situation. Not only does OSGi lend itself to a remote model, but it also provides simple, easily reusable remoting

specifications. Conveniently, several open source implementations of this specification are available.

With a small amount of effort, and no changes to your core application at all, you've seen that you can successfully add a remote service to your application. At no point in this chapter did you have to deviate from a standard, and the only piece of customization you did was to specify CXF and SCA configuration types in your endpoint definitions.

Although the Remote Services Specification is a straightforward way of remoting OSGi services, its scope is restricted to relatively homogeneous OSGi-only systems. In our experience, heterogeneous environments, with a mix of new and old components, and OSGi components and non-OSGi components, are more common. Although it works as a remote services implementation, one of SCA's real strengths is the flexibility of its bindings and component types, which allow a wide range of different types of systems to be knit together into a harmonious unified architecture. We'll discuss strategies for integrating heterogeneous systems, as well as how to migrate legacy applications to an OSGi-based platform, in the next chapter.

JAVA

# Enterprise OSGi IN ACTION

Cummins • Ward


Enterprise OSGi is a set of standards for building modular Java applications which integrate seamlessly with existing Java EE technologies. It extends the OSGi component framework to distributed systems.

**Enterprise OSGi in Action** is a hands-on guide for developers using OSGi to build enterprise Java applications. Many examples and case studies show you how to build, test, and deploy modular web applications. The book explains how to take advantage of dynamism, distribution, and automatic dependency provisioning, while still integrating with existing Java EE applications.

## What's Inside

- Build modular applications using servlets, JSPs, WARs, and JPA
- Better component reuse and robustness
- Expert tips for Apache Aries

This book is written for Java EE developers. No prior experience with OSGi is required.

**Holly Cummins** and **Tim Ward** are lead engineers and frequent speakers at conferences. Tim has written standards in both the OSGi Core and Enterprise Specifications, and both authors are active Apache Aries committers.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/EnterpriseOSGiinAction

"The definitive guide to using Enterprise OSGi in the real world. Highly recommended."
—Felix Meschberger
Adobe Systems Inc

"Comprehensive, hands-on."
—Pierre De Rop
Apache Felix committer

"Gets you up to speed on Enterprise OSGi."
—David Bosschaert, Red Hat

"Explains not just the shiny details but also the problems ... and ways to work around them. Great book."
—Teemu Kanstrén, VTT

ISBN 13: 978-1-617290-13-8
ISBN 10: 1-617290-13-0
57000
9 781617 290138

MANNING $70.00 / Can $80.99 [INCLUDING eBOOK]