```
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

    <persistence-unit name="Lesson33">
      <class>Employee</class>
    </persistence-unit>

  </persistence>
```



Please select Lesson 33 on the DVD with the print book, or watch online at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.

O'REILLY

Chapter 34

Working with RESTful Web Services

In the nineties the Web became widely used, and newly created web applications were consumed by millions of people around the world. At the same time lots of legacy applications were available for use only within corporate walls. They were written in a variety of programming languages and deployed on a plethora of types of hardware. There was a need to expose corporate data to wider audiences, which resulted in the creation of the standard interface for consuming data over the Web.

The SOAP Web Services

The first standard for publishing and consuming web services was the XML-based Simple Object Access Protocol (SOAP). Web clients would form HTTP requests and receive responses using the SOAP syntax.

The difference between traditional JSP/Servlet/JSF web applications and web services is that the latter offer just the data and have no interest in what the client's UI will look like. For example, an insurance company could offer information about its products, or a mutual fund could expose its data as a web service returning XML documents. Clients didn't need to know that this insurance company was running its applications using a server from Sun Microsystems or that the mutual fund was running its on mainframe computers from IBM. The clients needed to know the directory of services available from this

particular organization and the address of the **endpoint** to connect to in order to consume this service.

The directory of services could be published with the XML-based Web Services Description Language (WSDL), which is pretty verbose. In the Java world, SOAP messages could be processed by means of JAX-WS without the need for a directory of services.

Even though SOAP web services are verbose, they are still widely used as a simple means of integration with the software produced by third parties. Some SOAP services are publicly available. For example, the web page **www.webservicex.net** offers descriptions and WSDL locations of such information and services as weather forecasts, US address verification, currency converters, and stock quotes. You can integrate these into your application, but providing a user interface for them remains your responsibility.

The RESTful Web Services

As opposed to SOAP, REST is not a protocol, but a lighter-than-SOAP means of building web services, or, to be precise, REST is an architectural style Dr. Roy Fielding identified the REST principles in his PhD dissertation. A web service built on REST principles is called a RESTful web service.

Java EE 6 specification includes the JAX-RS API for creating RESTful web services. Several Java application servers already implement JAX-RS. I'll continue using GlassFish v3, which comes with the JAX-RS implementation known as Jersey, and you won't need to download any additional libraries to run the sample code from this lesson. Other popular open-source JAX-RS implementations are Apache Wink and JBoss RESTEasy.

REST stands for representational state of transfer. REST defines a set of constraints that an application must comply to, and web resources

that the user may need to work with.

A **resource** is anything that you can access with a hyperlink. Each resource has a uniform resource identifier (URI), such as http://localhost:8080/StockServer or www.dice.com/yakovsresume.pdf. These URIs represent the state of a stock server and Yakov's résumé, respectively.

REST resources have to support standard stateless requests. But if a typical web application uses only the HTTP methods `GET` for reading the resource and `POST` for updating it, in the RESTful world you can also use `PUT` for resource creation or updates and `DELETE` for resource removal. Web application developers often freely use both `GET` and `POST` for reading data, but REST is stricter about this.

The RESTful Stock Server

In this section I'll redesign the familiar stock server example using REST. The representation of the resources (in this case stocks) can vary and is determined by media type.

With Java EE 6, creating a RESTful application is pretty straightforward — create a POJO and annotate it. For example, annotating a Java bean with `@XmlRootElement` can bring into action the JAXB framework for processing XML, which will turn the bean into an XML document before sending it to the web client. **Listing 34-1** shows a Java bean called `Stock` annotated with `@XmlRootElement`.



Listing 34-1: Root class Stock

```
import javax.xml.bind.annotation.XmlRootElement;
```

```
@XmlRootElement
public class Stock {
    private String symbol;
    private Double price;
    private String currency;
    private String country;

    public Stock() {
    }

    public Stock(String symbol,Double price, String currency, String country) {
        this.symbol = symbol;
        this.price = price;
        this.currency = currency;
        this.country = country;
    }

    public String getSymbol() {
        return symbol;
    }

    public void setSymbol(String symbol) {
        this.symbol = symbol;
    }

    public Double getPrice() {
        return price;
    }

    public void setPrice(Double price) {
        this.price = price;
    }

    public String getCurrency() {
        return currency;
    }

    public void setCurrency(String currency) {
        this.currency = currency;
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
```

```
    }
}
```

Here's an example of how a RESTful service (**Listing 34-2**) may return the representation of a particular stock that's identified by the URI **http://localhost:8080/Lesson34/resources/stock/IBM**:

```
<stock>
  <country>US</country>
  <currency>USD</currency>
  <price>43.12</price>
  <symbol>IBM</symbol>
</stock>
```

From the standpoint of RESTful Web service, the preceding XML code fragment is more important than the Java code in **Listing 34-1**. The latter is an implementation detail. It's the actual representation expressed here in XML that really matters.

Just to reiterate the concept of resources and URIs, the stock MSFT is considered another resource and can be represented by the URI **http://localhost:8080/Lesson34/resources/stock/MSFT**. The class `StockResource` is a POJO, but if you'd like to turn it into an EJB, just annotate it with the `@Stateless` annotation.

The class `StockResource` is heavily sprinkled with annotations. First comes the annotation `@Path`, which can be used with either a class or a method. JAX-RS maps client's requests to class methods. If more than one annotation `$Path` are used in a class, their values are going to be concatenated for finding the matching method.

In our example in **Listing 34-2**, during the client's request the class-level `@Path("/stock")` results in the routing of each request containing

`/stock` in its URI to the class `StockResource`. Then the matching process continues, and if there is a stock symbol after the word `stock`, (e.g., `/stock/MSFT`), the method-level `@Path("{symbol}")`will map the `getStock()` method for processing. The `@PathParam("symbol")` will inject the stock symbol included in the URI into the `symb` argument of the method `getStock()`.

The `addStock()` method in **Listing 34-2** is annotated to consume the form data and is called when the Add button in the web client (see **Figure 34-1**) is clicked. The `@FormParam` annotation injects the values entered in the HTML form into the method `addStock()`. If the web client isn't using a form but instead is using an HTTP `GET` request with parameters, you have to use the annotation `@QueryParam` instead of `@FormParam`.



Listing 34-2: REST resource StockResource

```
import javax.ws.rs.Consumes;
import javax.ws.rs.FormParam;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Response;

@Path("/stock")
public class StockResource {

    @Produces({"application/xml", "application/json"})
    @Path("{symbol}")
    @GET
    public Stock getStock(@PathParam("symbol") String symb) {

        Stock stock = StockService.getStock(symb);

        if (stock == null) {
            return new Stock("NOT FOUND", 0.0, "--", "--");
        }
```

```
        return stock;
    }

    @POST
    @Consumes("application/x-www-form-urlencoded")
    public Response addStock(@FormParam("symbol") String symb,
                             @FormParam("currency") String currency,
                             @FormParam("price") String price,
                             @FormParam("country") String country) {

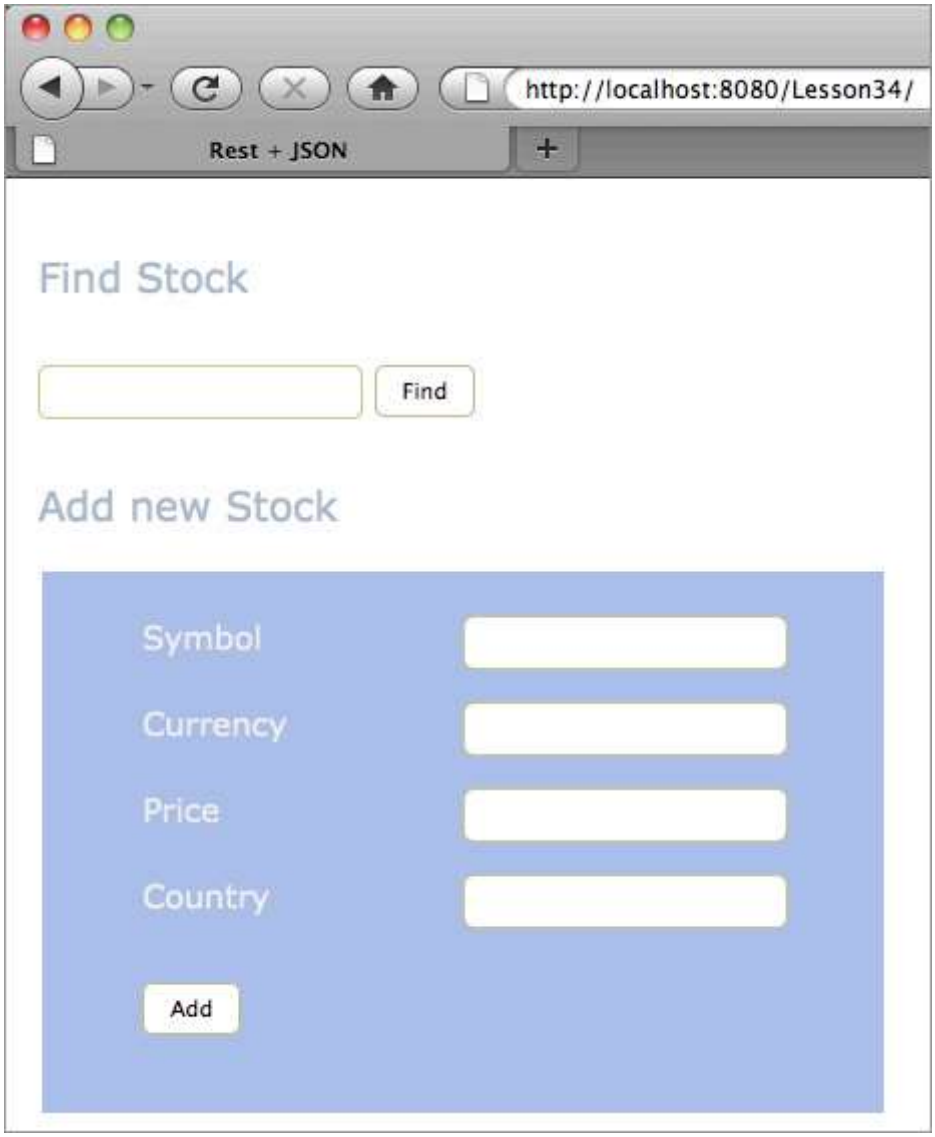
        if (StockService.getStock(symb) != null)
            return Response.status(Response.Status.BAD_REQUEST).
                    entity("Stock " + symb +
                    " already exists").type("text/plain").build();

        double priceToUse;
        try {
            priceToUse = new Double(price);
        }
        catch (NumberFormatException e) {
            priceToUse = 0.0;
        }

        StockService.addStock(new Stock(symb, priceToUse,
                                                currency, country));

        return Response.ok().build();
    }
}
```

Figure 34-1



One of the methods in `StockResource` is marked with a `@GET` annotation and the other one with `@POST`. You've guessed it right — they will automatically be invoked to process the corresponding HTTP requests. The MIME types specified in the annotation `@Produces` mean that the method `getStock()` can produce the data in either XML format or JSON format. In REST terminology, you can say that the `StockResource` resource supports two representations — JSON and XML. JSON stands for JavaScript Object Notation; it can be used as a lighter alternative to XML.

JSON is popular with web clients written in JavaScript (see **Listing 34-5** for the sample code). There are JSON parsers for programming languages other than JavaScript languages, including Java. Here's an ex-

ample of the data in JSON format, which clearly requires less overhead to represent the object `Stock` than XML does:

```
"stock": {
  "country": "US",
  "currency": "USD",
  "price": 43.12,
  "symbol": "IBM"
}
```



The public community domain **www.geonames.org** enables you to search for geographical and statistical information about countries, cities, et cetera. This website uses RESTful web services for the most part. To compare how the same data is represented in XML and in JSON, visit this web page: **www.geonames.org/export/ws-overview.html**.

The class `StockResource` is using the helper class `StockService`, shown in **Listing 34-3**. For simplicity, this class has two hard-coded stocks, but in the real world it would be connected to one of the financial data feeds. This class uses static initializer, which calls the method `generateStocks()` on the first reference to `StockService`.



Listing 34-3: StockService class

```
import java.util.HashMap;
import java.util.Map;
```

```
public class StockService {
    public static void addStock(Stock stock) {
        stocks.put(stock.getSymbol(), stock);
    }

    public static void removeStock(String symbol) {
        stocks.remove(symbol);
    }

    public static Stock getStock(String symbol) {
        return stocks.get(symbol);
    }

    private static Map<String, Stock> stocks = new HashMap<String, Stock>();

    static {
        generateStocks();
    }

    private static void generateStocks() {
        addStock(new Stock("IBM", 43.12, "USD", "US"));
        addStock(new Stock("APPL", 320.0, "USD", "US"));
     }
}
```

The `web.xml` file from **Listing 34-4** is configured to redirect all HTTP requests that contain the pattern `/resource` to `com.sun.jersey.spi.container.servlet.ServletContainer`, the Jersey implementation class.



Listing 34-4: web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">
```

```
    <servlet>
        <servlet-name>Rest</servlet-name>
        <servlet-class>
           com.sun.jersey.spi.container.servlet.ServletContainer
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Rest</servlet-name>
        <url-pattern>/resources/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Next comes the thin web client written in HTML and JavaScript, which is not covered in this book. I'll just highlight the relevant code fragments to explain the client/server communication in the stock server example. **Listing 34-5** shows the `index.html` file, which includes the popular JavaScript library JQuery (see the reference in the `<head>` section).



Listing 34-5: Thin web client index.html

```
<!DOCTYPE html>
<html>

<head>
    <title>Rest + JSON</title>
    <link rel="stylesheet" type="text/css" href="css/main.css"/>

    <script src="http://code.jquery.com/jquery-1.4.4.js">
    </script>
</head>

<body>

<p class="title">Find Stock</p>

<form id="get-stock" action="javascript:alert('submit')">
    <input id="get-stock-symbol" type="text"/>
```

```
    <input class="button" type="submit" value="Find"/>
</form>

<div class="result">
<table>
    <tr>
        <td>Symbol</td>
        <td id="symbol"></td>
    </tr>
    <tr>
        <td>Price</td>
        <td id="price"></td>
    </tr>
    <tr>
        <td>Currency</td>
        <td id="currency"></td>
    </tr>
    <tr>
        <td>Country</td>
        <td id="country"></td>
    </tr>
</table>
</div>

<p class="title">Add new Stock</p>

<div id="add-stock-flash-message" class="flash-message">
</div>

<form id="add-stock" action="javascript:alert('submit')">
    <fieldset>
        <ol>
            <li>
                <label for="add-stock-symbol">Symbol</label>
                <input id="add-stock-symbol" name="symbol" type="text"/>
            </li>
            <li>
                <label for="add-stock-currency">Currency</label>
                <input id="add-stock-currency" name="currency" type="text"/>
            </li>
            <li>
                <label for="add-stock-price">Price</label>
                <input id="add-stock-price" name="price" type="text"/>
            </li>
            <li>
                <label for="add-stock-country">Country</label>
                <input id="add-stock-country" name="country" type="text"/>
            </li>
            <li>
```

```
                <input class="button" type="submit" value="Add"/>
            </li>
        </ol>
    </fieldset>
</form>

<script>
    $('.result').hide();
    $('#add-stock-flash-message').hide();

    $("#get-stock").submit(function() {
        $.getJSON('resources/stock/' + $("#get-stock-symbol").val(),
      function(data) {

            $('.result').fadeOut(500, function(){
                $('#symbol').html(data.symbol);
                $('#price').html(data.price);
                $('#currency').html(data.currency);
                $('#country').html(data.country);

                $('.result').fadeIn(500)
            });
        });

        return false;
    });

    $("#add-stock").submit(function() {

        $.ajax({
            type: "POST",
            url: 'resources/stock',
            data: $("#add-stock").serialize(),

            success: function() {
                $("#add-stock-flash-message").show().html(
                          "Stock was added").fadeOut(5000);
            },

            error: function(request, textStatus, errorThrown) {
                if (textStatus == 'error') {
                    $("#add-stock-flash-message")
                      .show().html(request.responseText).fadeOut(5000);
                }
                else {
                    $("#add-stock-flash-message")
                      .show().html("Server error").fadeOut(5000);
                }
            }
```

```
        });

        return false;
    });
</script>

</body>
</html>
```

This HTML/JavaScript client can be invoked in such a way as to request a resource represented either as JSON or as XML. To get the data in JSON format, simply direct your web browser to **http://localhost:8080/Lesson34/index.html**. When you press the Find button on the form shown in **Figure 34-1**, the code in `index.html` calls the function `getJSON()`, which creates the pattern `resources/stock` , concatenates the entered stock symbol, and sends the request to the server. In the following JQuery code fragment `get-stock` is the ID of the top HTML form being submitted to the server:

```
$("#get-stock").submit(function() {
        $.getJSON('resources/stock/' + $("#get-stock-symbol").val(),
      function(data) {
...
});
```

The function `getJSON()` has requested the resource `Stock` in JSON format. Even though the Java bean `Stock` has been annotated with `@XmlRootElement`, JAXB can represent it not only as XML, but also as JSON.

To test the XML branch, enter a URL containing the pattern `/resources` that was used in the `web.xml` file from **Listing 34-4**. For example, the URL **http://localhost:8080/Lesson34/resources/stock/IBM** will return this REST resource as XML, as shown in **Figure 34-2**.

In this use case the code from the file `index.html` is not being used, and the request is redirected to the function `getStock()` of `StockResource`. The function `getStock()` is capable of producing both JSON and XML, but this time it produces plain web browser's code that knows how to consume XML only, and so JAX-RS picked up the XML format as a common denominator.

Figure 34-2



Try It

Your assignment is to make the code shown in the listings work. You'll need to create an Eclipse project and copy to the proper places the code samples from the accompanying DVD (Listings 34-1 to 34-5). Then you'll need to deploy this project in GlassFish and run it.

Lesson Requirements

You should have Java and GlassFish v3 server installed.



You can download the code and resources for this Try It from the book's web page at **www.wrox.com**. You can find them in the Lesson34 folder in the download.

Hints

Eclipse IDE for Java EE Developers has a template for the creation of REST resources. Right-click the project name, select New, and then select RESTful Web Service From Pattern (Java EE 6). Eclipse will create for you an annotated class similar to the one from **Listing 34-2**, which you can edit to meet your application's requirements.

Step-by-Step

1. Create a dynamic web project called Lesson34 in Eclipse. Then create a package called `com.practicaljava.lesson34`.

2. Copy the classes `Stock`, `StockResource`, and `StockService` from the DVD into this package.

3. Copy the file `web.xml` shown in **Listing 34-4** to the directory `WebContent/WEB-INF`.

4. Create the folder `WebContent/css` and copy to it the file `main.css` from the DVD — this will add some colors to the client's UI.

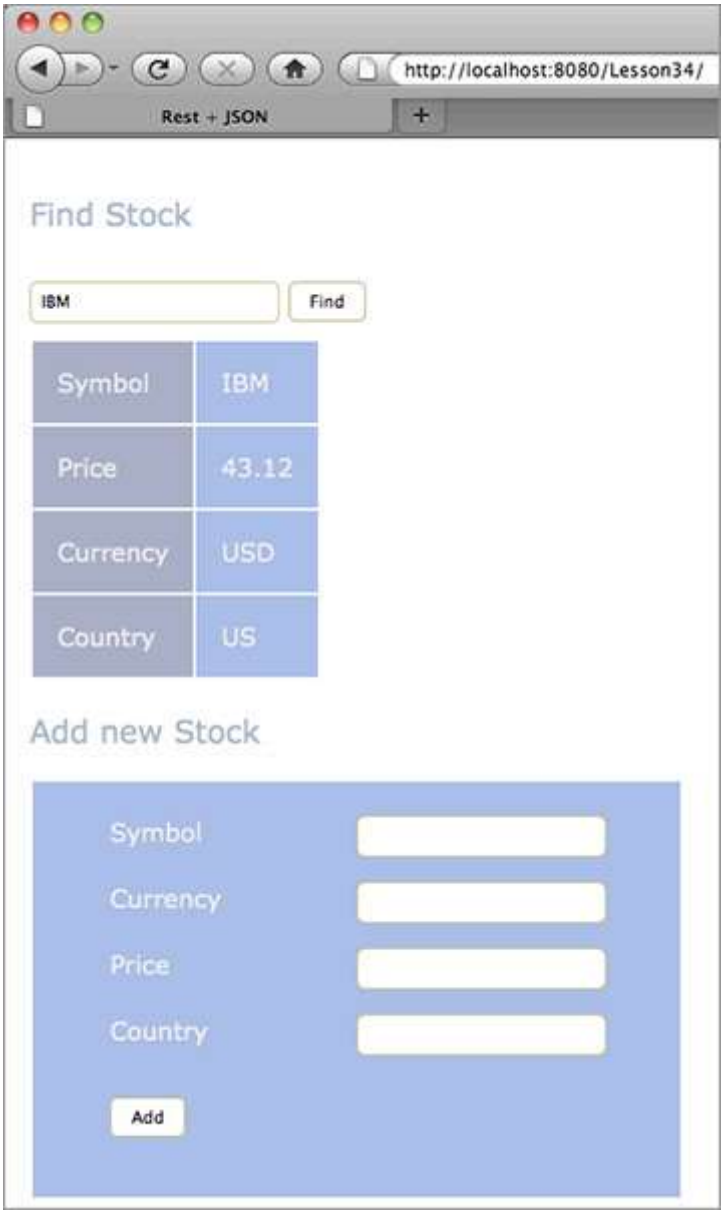
5. Deploy the project Lesson34 under GlassFish — right-click the server name, select Add and Remove, and in the pop-up window move Lesson34 from the left box to the right one by clicking Add. Then click Finish.

6. Open a web browser and go to **http://localhost:8080/Lesson34/**, which will open the file `index.html` from **Listing 34-5**. You'll see something similar to **Figure 34-1**.

7. Enter `IBM` in the search field and click Find. The pricing information (hard-coded) about this stock will be displayed as in **Figure 34-3**.

8. Test the Add New Stock functionality by filling out the form shown in **Figure 34-3** and clicking Add. This will make a `POST` request to the RESTful service, which will be mapped to the method `addStock()` of `StockResource` from **Listing 34-2**. After the new stock has been added you can search for it by entering its name in the search field and clicking Find.

Figure 34-3



Please select Lesson 34 on the DVD with the print book, or watch online at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.

Chapter 35

Introduction to Spring MVC Framework

At the beginning of the century Java EE was called J2EE, and it was a heavy and over-engineered standard. Programming EJBs was difficult and required writing a lot of boilerplate code. Java annotations had not been invented yet. Having a Java container that would take care of multi-threading, transactions, persistence, and security was nice, but developers needed a lighter container and a simpler programming model.

Rod Johnson, a talented software engineer, wrote a book titled Expert One-on-One J2EE Design and Development, which was published by Wrox at the end of 2002. A year later the book's code samples and development recommendations were released as the Spring Framework.

The Spring Framework is a Java container that provides a living space for an application's objects, wired to each other in a loosely coupled way in XML configuration. This loose coupling comes at a price, though — configuration files are quite large in size.

Spring Framework was built based on the **inversion of control** (IoC) principle: The container is responsible for the instantiation and location of the objects. IoC is also known as the Hollywood principle: "Don't call us, we'll call you."

Instead of having a class called `Customer` reach out to some container's registry trying to find an `Order`, under the Spring Framework the container would use a system of callbacks to instantiate the `Order` and inject it into the instance of the `Customer` object. This is an example of the **dependency injection** pattern — the `Customer` gets an injection of required resources. You saw resource injection in action in the Java EE lessons.

Don't forget that even five years ago the injection of resources was not implemented in Java EE application servers. Today not only is resource injection implemented, but Java EE 6 has yet another injection mechanism known as **contexts and dependency injection** (not covered in this book), which will make future Java EE applications even lighter. But don't forget that some ideas implemented in the current Java EE specification were originally introduced by the Spring Framework.

Over the last several years the Spring Framework has served as an alternative solution for developing enterprise Java applications. It has provided an infrastructure that allows you to easily plug in third-party frameworks and components. For example, the combination of Spring and the persistence framework Hibernate (see Lesson 36) is a popular architecture for lots of production-grade applications.

Serious improvements in the Java EE 6 specification may reduce the number of applications developed with Spring/Hibernate, but as of today this duo certainly remains a nice infrastructure for developing enterprise Java applications.

Bean Wiring

In the Spring Framework, beans are configured in XML files, and if a class called `Customer` needs an `Order` bean you can either look for this bean by issuing `getBean("Order")` on an instance of the main Spring factory, `ApplicationContext`, or, better yet, configure injection of the `Order` into `Customer`, as in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <bean id="customerBean" class="com.practicaljava.lesson35.Customer">
   <property name="order" ref="ord" />
   </bean>

   <bean id="ord" class="com.practicaljava.lesson35.Order"/>
</beans>
```

This is an example of configuring injection of one object into the field of another one in a loosely coupled way. The class `Customer` knows that someone will provide an `Order` when the time is right. This makes it easy to test the `Customer`'s functionality alone by wiring orders with hard-coded values — just change the value in the preceding XML from `com.practicaljava.lesson35.Order` to `com.practicaljava.lesson35.FakeOrder`. When the real data is available, change it back. In simple cases you can inject simpler values into Spring bean properties via setters or constructor arguments. You can provide the name of the file with configured beans to the constructor of `ApplicationContext` during instantiation.

Overview of Spring MVC

The Spring Framework consists of a number of modules that can be categorized as **web**, **core**, and **data access**. And the good part is that they can be used à la carte as you need them. The web module simplifies development of web applications. The core module provides the infrastructure and wiring for Java components living in the Spring container. Data access enables you to create an abstraction layer for data access so that no Java code needs to be changed if you decide to use a different JDBC driver or DBMS.

In this lesson you'll get familiar with the Spring MVC module that was created for the development of web applications. Like Java application servers, Spring manages your Java classes, so I'll be using the term **Spring container**. The classes that are going to be managed by the Spring container can be referred to as **Spring beans**.

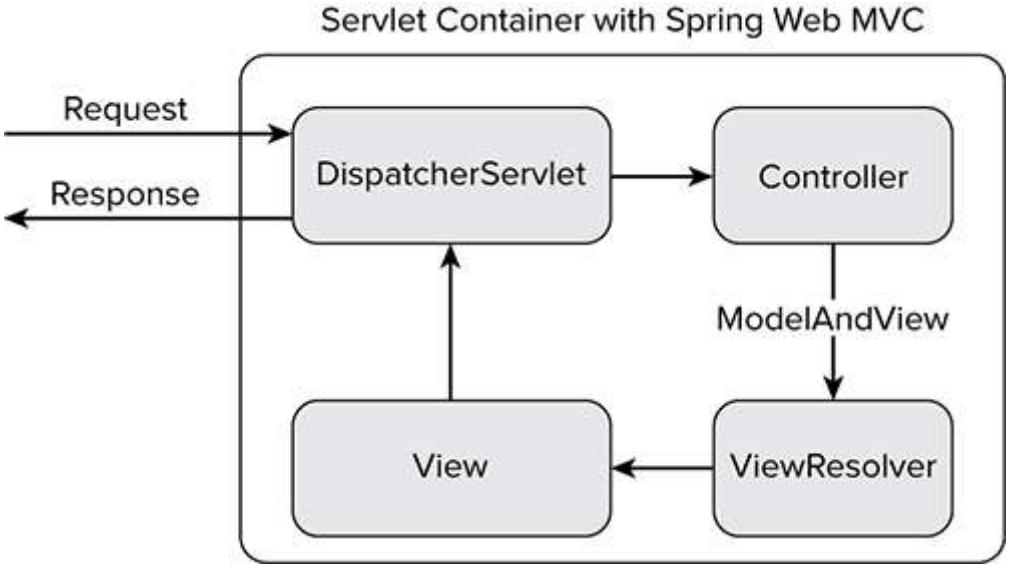
The Spring Framework offers a solution for building web applications. It includes a number of modules: Spring Faces, Spring Web Flow, Spring JavaScript, Spring Security, and Spring BlaseDS. And you don't have to use all these modules to build a web application — just use what you need.

There are two main models for building web applications with HTML clients:

- The client connects to a server-side presentation object (such as JSP) that is responsible for further routing and processing the request and for forming the next view to be returned to the client.
- The client connects to a server-side controller object (has no UI code) that performs the routing and processing of the request and for selection of the proper view to be returned to the client.

Building a web application in the Spring Framework represents the second model, where `DispatcherServlet` intercepts client requests and passes them to the appropriate Java controller to handle the request. The controller creates a `ModelAndView` object and gives it to a view resolver, which finds the view component to return to the client. The entire dataflow is shown in **Figure 35-1**.

Figure 35-1



This architecture is supported by the easily extendable Spring MVC layer. Other frameworks can be plugged into the Spring infrastructure. It's popular to integrate Spring and Hibernate for data persistence, but there are other examples of framework integration. For example, Adobe's BlaseDS framework or the Apache Struts legacy framework can be plugged into Spring.

HTML forms with data validation and error processing are also supported, and you'll see how to develop such applications in the next section.

Processing HTML with Spring

In this section I'll show you a simple web application using the Spring Web MVC framework. You'll see how to do the following:

- Configure the controller `ServletDispatcher`.
- Configure the `UrlBasedViewResolver` bean.
- Create a simple controller called `HelloWorldController` to produce data for presentation by a JSP.
- Handle the request of a "contact us" HTML form with the Spring JSP tag library.

Understanding the Spring Web MVC Workflow

To start, create an Eclipse Dynamic Web Project named Lesson35. Now you'll need to download several jar files from the Spring framework and copy them to the `WebContent/WEB-INF/lib` folder. Register at the SpringSource website **www.springsource.com/download/community** and download the latest zip file of the Spring Framework. At the time of this writing it's `spring-framework-3.0.5.RELEASE.zip`.

Unzip the archive and add the following jars to the build path of your Eclipse project. Also copy these jars to the `WebContent/lib` directory so they are visible at the deployment time.

- `org.springframework.asm-3.0.5.RELEASE.jar`
- `org.springframework.beans-3.0.5.RELEASE.jar`
- `org.springframework.context-3.0.5.RELEASE.jar`
- `org.springframework.core-3.0.5.RELEASE.jar`
- `org.springframework.expression-3.0.5.RELEASE.jar`
- `org.springframework.web.servlet-3.0.5.RELEASE.jar`
- `org.springframework.web-3.0.5.RELEASE.jar`

For logging support you'll also need to download and add to the `lib` folder the Commons Logging API `commons-logging-api-1.1.1.jar`, which you can download from the same website — it's in `org.apache.commons` in the archive with dependencies.

Now you need to deploy the Spring Framework as a web application under GlassFish. Create the file `web.xml` with the content shown in **Listing 35-1** and add it to the `WEB-INF` folder. This file instructs GlassFish to load the Spring's `DispatcherServlet` on startup and to redirect any `*.html` requests to it.



Listing 35-1: web.xml with DispatcherServlet

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns=http://java.sun.com/xml/ns/javaee
 xmlns:web=http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd
 xsi:schemaLocation=
"http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
id="WebApp_ID" version="2.5">

  <display-name>Lesson35 Sample</display-name>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

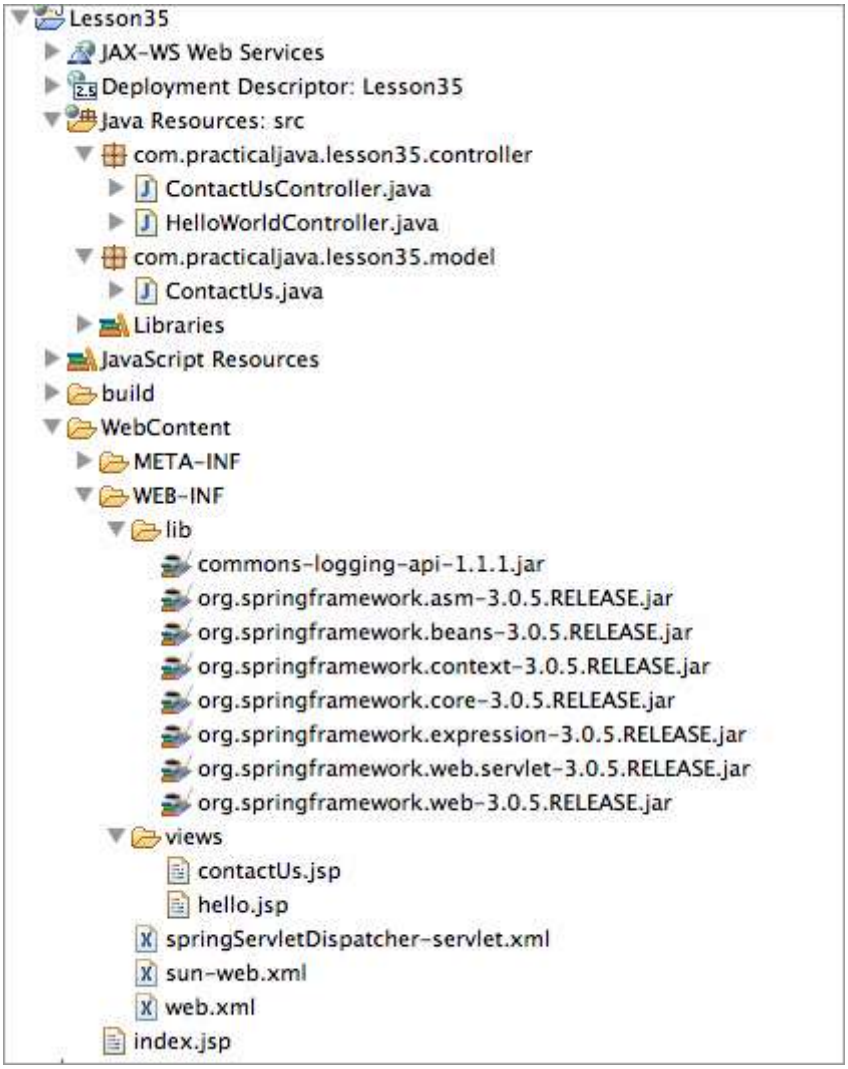
  <servlet>
        <servlet-name>springServletDispatcher</servlet-name>
        <servlet-class>
          org.springframework.web.servlet.DispatcherServlet
       </servlet-class>
        <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
        <servlet-name>springServletDispatcher</servlet-name>
        <url-pattern>*.html</url-pattern>
  </servlet-mapping>
</web-app>
```

In our example the MVC pattern will be represented by the following: for the model a Java class called `ContactUs`; for the view two JSP pages, `hello` and `contactUs`; and for the classes two Java controller classes, `HelloWorldController` and `ContactUsController`. **Figure 35-2** shows a snapshot of my Eclipse project to give you an idea of where the files are located.

`HelloWorldController` is shown in **Listing 35-2**. It includes the `@RequestMapping` annotation that instructs the controller to call the method `sayHello()` for any requests with the `/hello` pattern.

Figure 35-2





Listing 35-2: HelloWorldController

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class HelloWorldController {

    @RequestMapping("/hello")
    public ModelAndView sayHello(){

        String greetings = "Hello people";
```

```
        return new ModelAndView("hello", "greetings", greetings);
    }
}
```

The view resolver of our application is an XML configuration file named `springServletDispatcher-servlet.xml`, shown in **Listing 35-3**. Note that the prefix of this file name matches the value in the tag `<servlet-name>` in **Listing 35-1**. This application uses one of the view resolvers provided by Spring, `UrlBasedViewResolver`, which creates the appropriate view based on the URL.



Listing 35-3: springServletDispatcher-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
 xmlns:p="http://www.springframework.org/schema/p"
      xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context-3.0.xsd">

      <context:component-scan
            base-package="com.practicaljava.lesson35.controller" />

   <bean id="viewResolver" class=
"org.springframework.web.servlet.view.UrlBasedViewResolver">
       <property name="viewClass"
         value="org.springframework.web.servlet.view.JstlView"/>
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
   </bean>
</beans>
```

The preceding file is configured to add the prefix `WEB-INF/views` and the suffix `.jsp` to the value specified in the `@RequestMapping` of the controller. As an example, for `HelloWorldController` the URL of the view given to the response object becomes `WEB-INF/views/hello.jsp`. The source code for `hello.jsp` is shown in **Listing 35-4**.



Listing 35-4: hello.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
 ${greetings}
</body>
</html>
```

What does `${greetings}` mean? Look at the code of `HelloWorldController` in **Listing 35-2**. It contains the following lines:

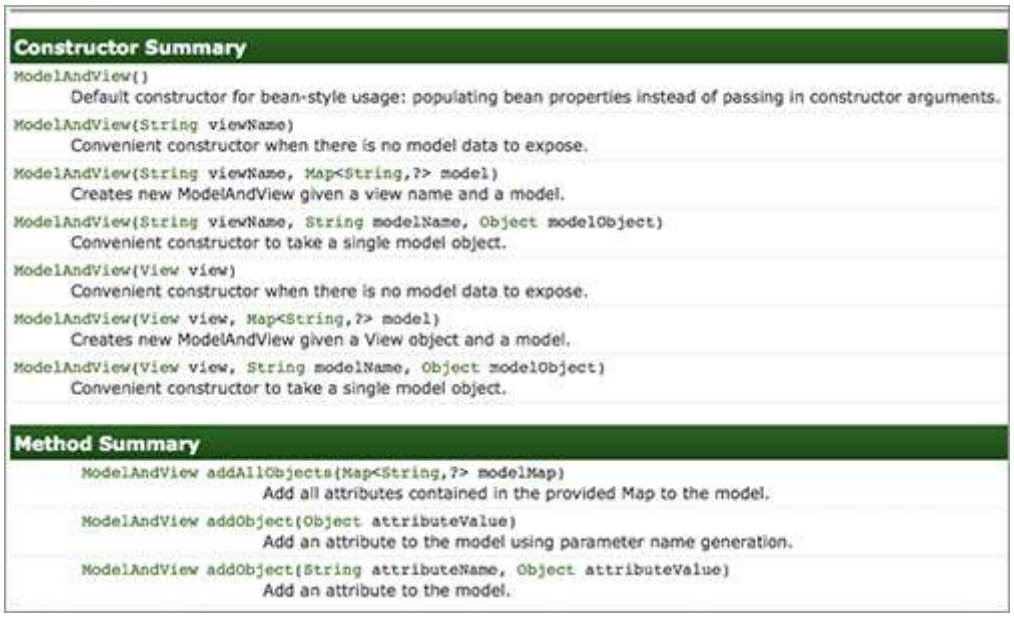
```
String greetings = "Hello people";
return new ModelAndView("hello", "greetings", greetings);
```

This is how the Spring controller tells the view resolver what to do with the view before it is returned to the client. The first argument of the `ModelAndView` constructor contains the name of the view, and the sec-

ond one contains the name of the JSP element to be replaced with the content of the third argument. Basically, the name of the class reflects the fact that it contains elements of both the model and the view.

The class `ModelAndView` has a number of overloaded constructors (see **Figure 35-3**) that enable you to pass multiple values to the view object. You can also populate `ModelAndView` with data by using the method `addObject()`.

Figure 35-3



Constructor Summary

`ModelAndView()`
Default constructor for bean-style usage: populating bean properties instead of passing in constructor arguments.

`ModelAndView(String viewName)`
Convenient constructor when there is no model data to expose.

`ModelAndView(String viewName, Map<String,?> model)`
Creates new ModelAndView given a view name and a model.

`ModelAndView(String viewName, String modelName, Object modelObject)`
Convenient constructor to take a single model object.

`ModelAndView(View view)`
Convenient constructor when there is no model data to expose.

`ModelAndView(View view, Map<String,?> model)`
Creates new ModelAndView given a View object and a model.

`ModelAndView(View view, String modelName, Object modelObject)`
Convenient constructor to take a single model object.

Method Summary

`ModelAndView addAllObjects(Map<String,?> modelMap)`
Add all attributes contained in the provided Map to the model.

`ModelAndView addObject(Object attributeValue)`
Add an attribute to the model using parameter name generation.

`ModelAndView addObject(String attributeName, Object attributeValue)`
Add an attribute to the model.

Add the project Lesson35 to GlassFish in the Eclipse Servers view and start the server. Direct your web browser to **http://localhost:8080/Lesson35/hello.html** and you'll see something like what's shown in **Figure 35-4**. Just to recap, because of the `*.html` mapping in `web.xml`, Spring's `ServletDispatcher` was invoked. The `.html` part was stripped off, the word `hello` was identified, and the `HelloWorldController` was instantiated and its method `sayHello()`invoked. The view resolver found `hello.jsp`, and its variable `greetings` was replaced with the text "Hello People" as per the content of the `ModelAndView` instance.

Figure 35-4



Processing HTML Forms

Processing HTML forms with Spring is just a little bit more complex than the "Hello People" example. The Spring Framework comes with integrated tag libraries with a tag corresponding to each HTML element. In particular, there is the `form` tag library for processing HTML forms. **Listing 35-5** shows `ContactUs.jsp`, which includes the appropriate `<%taglib>` directive.



Listing 35-5: ContactUs.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
       <%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<h2>Contact Manager</h2>

<form:form method="post" action="addQuestion.html">

      <table>
      <tr>
            <td><form:label path="author">Who are you?</form:label></td>
            <td><form:input path="author" /></td>
```

```
		</tr>
		<tr>
			<td><form:label path="subject">Subject</form:label></td>
			<td><form:input path="subject" /></td>
		</tr>
		<tr>
			<td><form:label path="message">You Question</form:label></td>
			<td><form:textarea path="message" /></td>
		</tr>
		<tr>
			<td colspan="2">
				<input type="submit" value="Submit your question"/>
			</td>
		</tr>
	</table>

	</form:form>
	</body>
	</html>
```

When the user clicks Submit, the request is posted to the server with `addQuestion.html` in the URL. Then `DispatcherServlet` maps `addQuestion` with the method `addUserQuestion()` in the controller shown in **Listing 35-6**.



Listing 35-6: ContactUsController.java

```
package com.practicaljava.lesson35.controller;

import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.servlet.ModelAndView;

import com.practicaljava.lesson35.model.ContactUs;

@Controller
```

```
@SessionAttributes
public class ContactUsController {

    @RequestMapping(value="/addQuestion", method=RequestMethod.POST)
    public String addUserQuestion(@ModelAttribute("contactUs") ContactUs
                                        contactUs, BindingResult result){

       System.out.println("New message:\nSubject:"
                                            +contactUs.getSubject());
       System.out.println("Author " + contactUs.getAuthor());
       System.out.println("Message "+ contactUs.getMessage());

       return "redirect:contactUs.html";
    }

    @RequestMapping("/contactUs")
    public ModelAndView contactUs(){
        return new ModelAndView("contactUs", "command", new ContactUs());
    }
}
```

`@ModelAttribute` binds the method parameter to the named model attribute (see the model class in **Listing 35-7**). The Spring Framework maps the attributes from the model to corresponding attributes in `ContactUs.jsp`, shown in **Listing 35-5**.



Listing 35-7: ContactUs.java

```
public class ContactUs {
    private String subject;
    private String message;
    private String author;

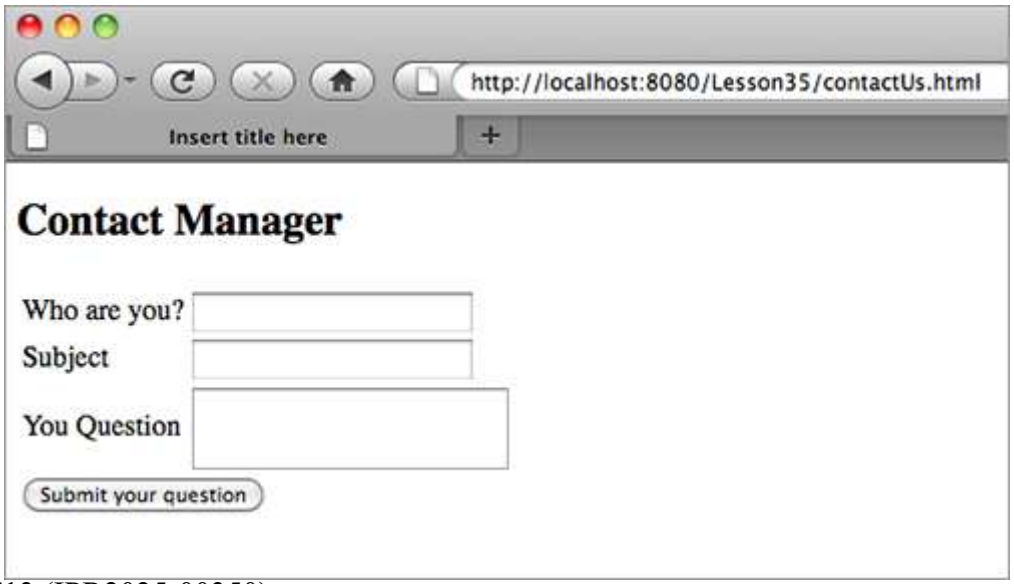
    public String getSubject() {
        return subject;
    }
    public void setSubject(String subject) {
        this.subject = subject;
```

```
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
    public String getAuthor() {
        return author;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
}
```

Open your web browser and enter the following URL: **http://localhost:8080/Lesson35/contactUs.html**. The `@RequestMapping("/contactUs")` annotation in **Listing 35-6** responds to the client with `ContactUs.jsp`, which looks as in **Figure 35-5**.

Enter some values in the form and click Submit. This time the `/addQuestion` action will be matched with the method `addUserQuestion()` of `ContactUsController`. Received values from the client will be printed in the log file of GlassFish Server, located in `logs/server.log` under your domain directory.

Figure 35-5



Try It

Modify the code of `hello.jsp` to accept more than one variable. Change the code of `ModelAndView` so that it carries a `Map` of values.

Lesson Requirements

You should have Java and GlassFish v3 installed.



You can download the code and resources for this Try It from the book's web page at www.wrox.com. You can find them in the Lesson35 folder in the download.

Step-by-Step

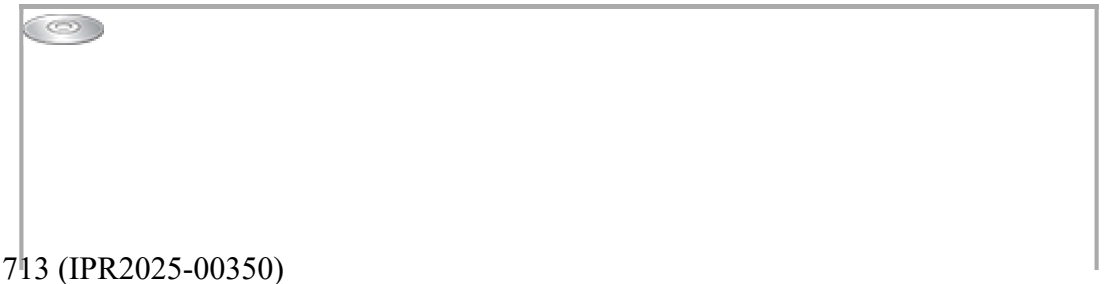
1. Modify `hello.jsp` to add one more variable in addition to `$greetings`:

2. Modify `HelloWorldController` to create a `Map` with two Strings:

```
Map helloModel = new HashMap();
helloModel.put("Hello Mary");
helloModel.put("My name is John");
```

3. Use the two-argument constructor of `ModelAndView` and pass the `Map` as the second argument.

4. Direct your web browser to http://localhost:8080/Lesson35/hello.html and observe that `hello.jsp` shows the values from the `Map`.



Please select Lesson 35 on the DVD with the print book, or watch online at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.

Chapter 36

Introduction to Hibernate Framework

In Lesson 33 you got familiar with the Java Persistence API, which is part of Java EE 6. But JPA wasn't created from scratch. It took many ideas from Hibernate — an open-source object-relational framework started by Gavin King in 2001. Hibernate version 3.5 and above support the JPA 2.0 specification.

The official website of Hibernate is **www.hibernate.org**, where you can find documentation, code samples, and tools. Plenty of books and online articles are published about this popular framework. I prefer using SQL and JDBC for all data-related processing, but these days knowledge of Hibernate is required in many job postings. That's why I decided to include this brief overview of Hibernate even though it's not a part of the Java specification but a third-party framework.

If you are going to use Hibernate for work, consider using the Eclipse plug-ins available at **www.hibernate.org/subprojects/tools.html**.

The Big Picture

The main idea behind any ORM framework is to replace writing SQL with manipulating the objects. In the Java world, this means putting a layer on top of JDBC that will enable you to simply create a POJO, as-

sign values to its fields, and tell it "Persist yourself in the database." This is what Hibernate is for.

Proponents of this approach argue that removing the dependency on vendor-specific SQL syntax provides easy switching from one DBMS to another without the need to modify the Java code responsible for querying and persisting data. To me this doesn't sound like a compelling reason to move away from SQL, because it's very seldom that you need to switch from one DBMS to another in real-world projects. Besides, if you'll be working on projects that require swift query execution, the ability to tune SQL manually and use tricks specific to DBMSes can be helpful.

On the other hand, Hibernate offers such goodies as **lazy loading** and the ability to easily configure the in-memory cache to greatly minimize disk access, which is a lot slower than memory access.

Lazy loading is easy to explain with an example. Suppose the user is presented with a list of customers, each of which may have multiple orders. With lazy loading only the list of customers is initially loaded, and only when the user selects a customer will the query to retrieve the corresponding orders be sent to data storage.

The process of retrieving data with Hibernate is very similar to what you saw in Lesson 33. You create the mapping of entities to database tables, then instantiate the `Configuration` object needed to create a `SessionFactory` and `Session`. After this is done you execute queries on these entities using Hibernate Query Language (HQL), Criteria queries, or native SQL.

The rest of this lesson is a step-by-step walkthrough that demonstrates how to write a simple application that will get the data from a DBMS using Hibernate.

Installing and Configuring Hibernate

In this section you'll download required jars and create the configuration file `hibernate.cfg.xml`. I'll continue using Derby, which is included with GlassFish, as our sample DBMS, but the application server GlassFish is not needed here. The idea is to look at Hibernate as an alternative persistence solution that doesn't require the Java EE application server.

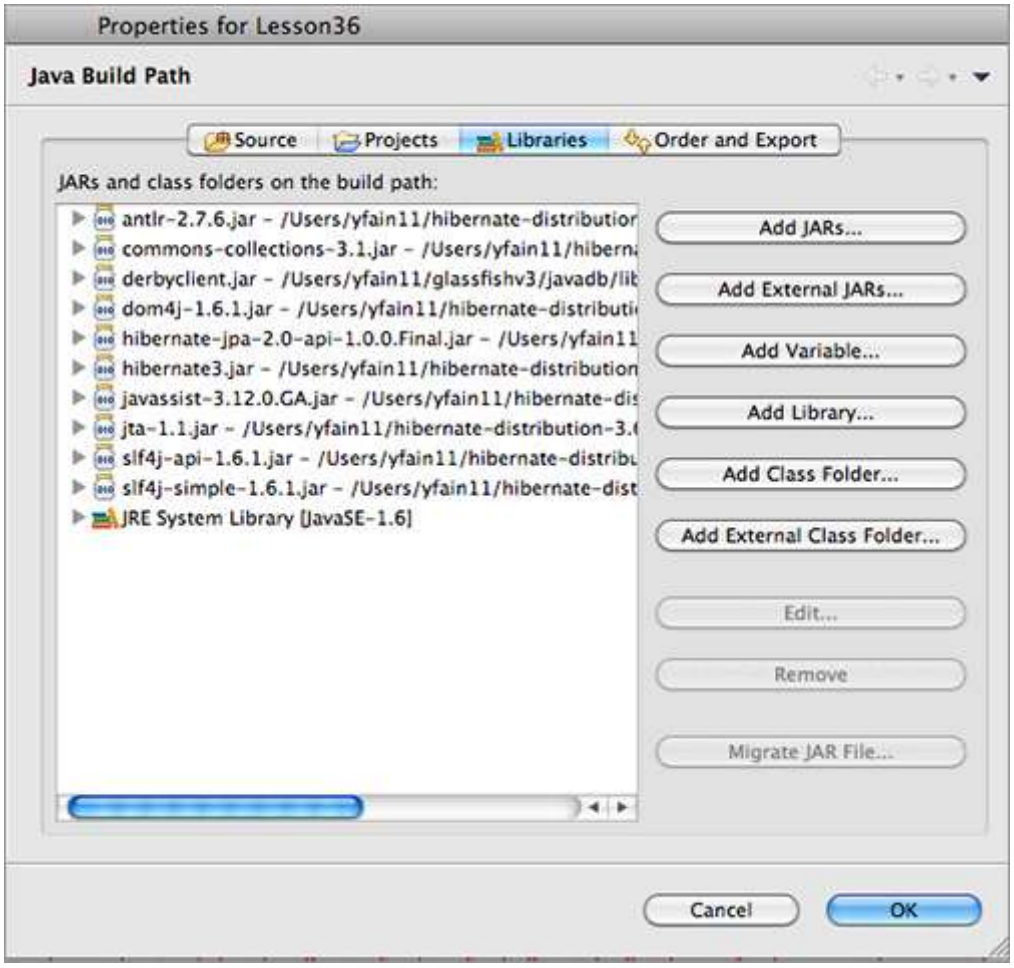
Hibernate is an open-source framework hosted on the web at SourceForge (**http://sourceforge.net/projects/hibernate/files/hibernate3/**). At the time of this writing, the latest release of Hibernate is 3.6; you need to download the file **hibernate-distribution-3.6.0.Final-dist.zip**. This distribution includes required jar files, a well-written reference manual, and a folder with libraries that might be used to support memory caching.

Adding Hibernate Jars to an Eclipse Project

Now create a new Eclipse project called Lesson36 by selecting File⇒⇒ New⇒⇒ Other⇒⇒ Java Project. Note that this is a Java SE project — I'm not going to use any Java EE application server here. Make sure your Eclipse Java project includes `hibernate3.jar`, all the jars from the `required` folder, and the jar from the `jpa` folder. There is one more logging-related file, `slf4j-1.6.1.zip`, that for some licensing reason has to be downloaded from **www.slf4j.org/download.html**. You need to add to the build path just one file from there: `slf4j-simple-1.6.1.jar`.

You can add these external jars in the tab Libraries of the Java Setting window while you're creating the project, or you can add them after the project is created via Properties⇒⇒ Java Build Path⇒⇒ Libraries⇒⇒ Add External Jars. **Figure 36-1** shows which libraries are included in the build path of my Eclipse project.

Figure 36-1



Testing the Database Connection

In this lesson I'll keep working with the database that was initially created in Lesson 22 and then reused in Lesson 33. It contains a table called Employee and the connection is already configured (see Figure 33-4). This connection uses the Derby JDBC driver located in `derbyclient.jar`. Add this jar, located in `glassfish/javadb/lib`, to the build path of the new Eclipse project.

Now right-click the database connection Lesson22 in Eclipse and select Properties⇒⇒ JDBC Connection Properties. Keep the user ID as `user` and change the password to `password123`. Now copy the connection URL (refer to Figure 33-3) to a text editor — you'll need the connection parameters in the next section. You can also check the properties of the selected DerbyDB driver by clicking the small button with an asterisk on it — copy the name of the JDBC driver class too.

Configuring Hibernate with a DBMS

Now it's time to add the DBMS-related entries to the file `hibernate.cfg.xml`. You don't need to create this file from scratch; copy to the `src` folder of your Eclipse project one of the files that come with Hibernate, such as the one located in the folder `documentation/quickstart/tutorials/basic/src/test/resources`. Modify the driver class and connection URL with the values from the previous section. Make sure that this file doesn't have a setting to drop and re-create the database schema on startup. You won't need any mapping files because our entity will use annotations.

Every DBMS implements standard SQL plus some additional features meant to give it a competitive edge. To deal with these extras Hibernate includes configurable `dialects` that hide the differences from the application developer. That's why the configuration file has to specify the dialect that the current application has to support. Each dialect is a class located in the package `org.hibernate.dialect` inside `hibernate3.jar`, and you can find the name of the dialect that matches our database, which is `DerbyDialect`.

You might remember that the table Employee was originally created in the database schema APP. Because I specified that the connection ID is `user`, by default the queries will try to find the referenced database objects in the schema USER, which doesn't exist. To fix this situation let's specify in the configuration file that the default schema is APP.

Finally, we want Hibernate to print in the Eclipse console the underlying SQL statements that it'll use in communication with the database.

After you make all these changes your `hibernate.cfg.xml` file should look like **Listing 36-1**.



Listing 36-1: Hibernate.cfg.xml

```
<?xml version='1.0' ?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->
        <property name="connection.driver_class">
            org.apache.derby.jdbc.ClientDriver
        </property>

        <property name="connection.url">
            jdbc:derby://localhost:1527/Lesson22;create=true
        </property>

        <property name="connection.username">user</property>
        <property name="connection.password">password123</property>

        <!-- Set the default schema to be APP, where table Employee was
                                                              created-->
        <property name="hibernate.default_schema">APP</property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.DerbyDialect</property>

        <!--  Enable Hibernate's current session context -->
        <property name="current_session_context_class">thread</property>

        <!-- Disable the second-level cache  -->
        <property name="cache.provider_class">
                 org.hibernate.cache.NoCacheProvider
        </property>

        <!-- Echo all executed SQL to stdout -->
```

```
        <property name="show_sql">true</property>

    </session-factory>

</hibernate-configuration>
```

Retrieving Data with Hibernate

Now we need to create a Java `Employee` entity, which in this simple example corresponds to the database table Employee. The good news is that because Hibernate supports JPA 2, we can simply reuse the `Employee` entity from Lesson 33. **Listing 36-2** shows this entity after we copy it into the package `com.practicaljava.lesson36`.



Listing 36-2: Employee entity

```
package com.practicaljava.lesson36;

import java.io.Serializable;
import javax.persistence.*;

@Entity
public class Employee implements Serializable {

  private static final long serialVersionUID = 1L;

  @Id
  private int empno;

  private String ename;

  @Column(name="JOB_TITLE")
  private String jobTitle;

  public Employee() {}

  public int getEmpno() {
```

```
    return this.empno;
  }

  public void setEmpno(int empno) {
    this.empno = empno;
  }

  public String getEname() {
   return this.ename;
  }

  public void setEname(String ename) {
   this.ename = ename;
  }

  public String getJobTitle() {
   return this.jobTitle;
  }

  public void setJobTitle(String jobTitle) {
   this.jobTitle = jobTitle;
  }
}
```

The code to retrieve employees comes next. The program `HRManager` shown in **Listing 36-3** will do this job. It starts by loading the configuration file `hibernate.cfg.xml` and creating a Hibernate session. We didn't provide any arguments to the method `configure()` — by default this method assumes that the configuration file is called `hibernate.cfg.xml`.

Then `SessionFactory` creates an instance of the `Session`. The transaction is opened, and `createQuery("from Employee")` means that we're interested in `Employee` entities. In this sample, the name of the entity and the name of the database table are the same. If this weren't the case we'd have had to specify the table name in the `@Entity` annotation in **Listing 36-2**.



Listing 36-3: Retrieving Employee entities

```
import java.util.List;
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import com.practicaljava.lesson36.Employee;

public class HRManager {

public static void main(String[] args){

  Configuration config = new Configuration().configure();
  config.addAnnotatedClass(Employee.class);

  SessionFactory sf = config.buildSessionFactory();
  Session session = sf.getCurrentSession();

  session.beginTransaction();

  List<Employee> employee = session.createQuery("from Employee").list();

  session.getTransaction().commit();

  for (Employee emp: employee){

       System.out.println("Got " + emp.getEmpno() + ", "
                           + emp.getEname() + ", "+ emp.getJobTitle());
  }
 }
}
```

The method `addAnnotatedClass(Employee.class)` tells Hibernate that there is an entity called `Employee` configured using annotations. After running, `HRManager` prints the following in Eclipse's Console view:

```
Hibernate: select employee0_.empno as empno0_, employee0_.ename as ename0_,
 employee0_.JOB_TITLE as JOB3_0_ from APP.Employee employee0_

Got 7369, John Smith, Clerk
Got 7499, Joe Allen, Salesman
Got 7521, Mary Lou, Director
```

Retrieving data with Hibernate is pretty trivial. Persisting the data in the database is not rocket science either. You'll have a chance to see such persisting for yourself when you work on the assignment in the "Try It" section.

Try It

This lesson shows you how Hibernate creates database objects from Java entities. You will have to declare the Java entity `Address` and have Hibernate create and populate the corresponding table in the database.

Lesson Requirements

You should have Java and GlassFish v3 installed.



You can download the code and resources for this Try It from the book's web page at **www.wrox.com**. You can find them in the Lesson36 folder in the download.

Hints

In the toolbar of the Data Source Explorer view is an icon that opens SQL Scrapbook. It enables you to select the DBMS and the database and to execute SQL statements. For example, enter **Select * from APP.Employee**, then right-click and select Execute All. The SQL statement will be executed and you'll see the content of the table Employee.

Step-by-Step

1. Create the following Java entity, `Address`:

```
package com.practicaljava.lesson36;

import javax.persistence.*;

@Entity
public class Address {

   private long addressID;
   private String streetAddress;
   private String city;
   private String zip;

   @Id
   public long getAddressID() {
      return addressID;
   }

   public void setAddressID(long addressID) {
      this.addressID = addressID;
   }

   public String getStreetAddress() {
      return streetAddress;
   }

   public void setStreetAddress(String streetAddress) {
      this.streetAddress = streetAddress;
   }

   public String getCity() {
      return city;
   }

   public void setCity(String city) {
      this.city = city;
   }

   public String getZip() {
      return zip;
   }

   public void setZip(String zip) {
      this.zip = zip;
   }
}
```

2. Create a class called `AddressCreator`. It should look like this:

```
import java.util.List;

import org.hibernate.*;

import org.hibernate.cfg.Configuration;
import org.hibernate.tool.hbm2ddl.SchemaExport;

import com.practicaljava.lesson36.Address;

public class AddressCreator {

      public static void main(String[] args){
         Configuration config = new Configuration().configure();
         config.addAnnotatedClass(Address.class);

         //Create the Address table
         new SchemaExport(config).create(true,true);

         SessionFactory sf = config.buildSessionFactory();
         Session session = sf.getCurrentSession();

         session.beginTransaction();

         // Instantiate and Populate the entity Address
         Address addr1 = new Address();
         addr1.setStreetAddress("123 Main St.");
         addr1.setAddressID(1);
         addr1.setCity("New York");
         addr1.setZip("10001");

        session.save(addr1);

        // Save the new entity in the database
          session.getTransaction().commit();

      }
}
```

3. Run the program AddressCreator. Observe the output in the Eclipse console. You should see the following SQL statements generated by

Hibernate:

```
drop table APP.Address
create table APP.Address (addressID bigint not null, city varchar(255),
 streetAddress varchar(255),
zip varchar(255), primary key (addressID))

Hibernate: insert into APP.Address (city, streetAddress, zip, addressID)
values (?, ?, ?, ?)
```

4. Go to the Data Source Explorer view, reconnect to the Lesson22 database, and observe the table Address in the schema APP. Right-click the Address table and select Data⇒ Sample Contents — you'll see one row with the New York address specified in `AddressCreator`.



Please select Lesson 36 on the DVD with the print book, or watch on-line at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.

Chapter 37

Bringing JavaFX to the Mix

JavaFX was designed to serve as a tool for creating rich Internet applications (RIAs). It is meant to compete with such RIA tools as Adobe Flex and Microsoft Silverlight. The competing technologies use two languages for creating RIAs — one for declarative definition of the UI components, and another for the coding of processing logic. Adobe Flex uses MXML for UI components and ActionScript for processing logic. Microsoft Silverlight uses XAML for UI components and one of the .NET languages (such as C#) for processing.

JavaFX was originally designed to use JavaFX Script for UI components and Java for processing logic. But recently things changed, and up to the last minute I wasn't sure if this lesson should be included in the book. Oracle has announced that JavaFX will be undergoing a major redesign in 2011. JavaFX 1.3 includes the JavaFX Script language, but JavaFX 2.0 will be completely Java-based.

Currently, no pre-release version of JavaFX 2.0 is available, and there is a chance that whatever you read about the client portion of this lesson's application will be outdated. But no matter what changes in JavaFX 2.0, the deployed application will run in JVM with the installed JavaFX run time. There is also hope that the Java community will continue supporting the open-source version of JavaFX Script.

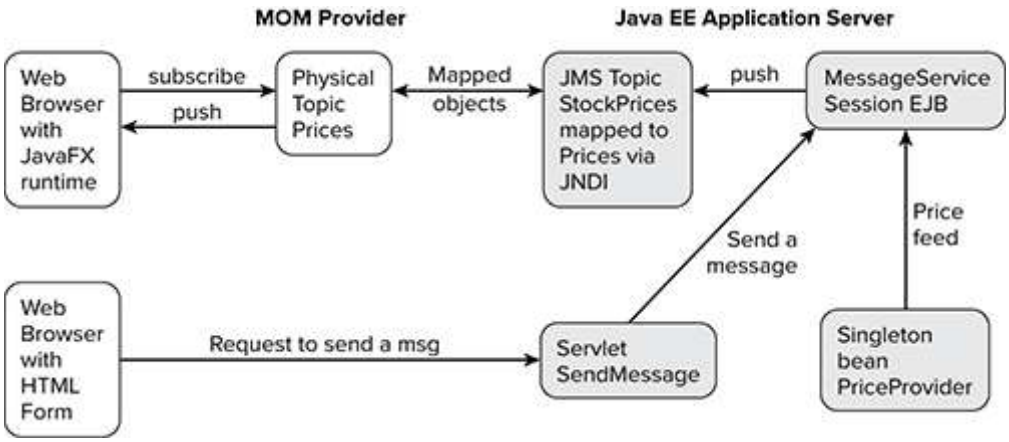
This lesson is not an introduction to JavaFX programming, but rather an overview of an application that uses multiple technologies you are

already familiar with, plus a client created with JavaFX 1.3.

Consuming Stock Quotes with JavaFX

In this lesson you'll see how a web client written in JavaFX can subscribe to price quotes published from the server side via messaging. Multiple technologies used in this version of the Stock Server example are depicted in **Figure 37-1**.

Figure 37-1



These are the players that provide the generation and processing of the price quote feed:

1. The singleton session bean `PriceProvider` (see the source code in **Listing 37-1**) knows how to generate random price quotes. It gets instantiated on startup of the GlassFish application server.

2. A stateless session bean called `MessageService` (**Listing 37-2**) gets the references to JMS-administered objects via resource injection and publishes the price quote to the JMS topic known as `jms/StockPrices`, which is mapped to the physical topic `Prices` in the Open MQ messaging server.

3. The Java client subscribes to the topic `Prices`. **Listing 37-3** shows the Java classes `Messages` and `JMSListener`, and the interface `MessageHandler`. These client classes do not have any presentation logic.

4. The presentation client `Main.fx` (**Listing 37-4**) is written in JavaFX Script. It wraps the Java client messaging classes and gets the price quotes from them. As soon as the new quote comes in, the old one is pushed up and off the screen with a special animation effect.

5. To illustrate another way of sending the message, this application has a simple HTTP client (**Listing 37-5**) that allows the user to enter any text, which is then routed via the servlet `SendMessage` (**Listing 37-6**) to the `MessageService` EJB. The rest of the data flow is as described in Steps 2 through 4.

Code Walkthrough

By now you should be familiar with each of the Java technologies used in this application, except JavaFX, so I'll provide only very short explanations for the code listings.

The `PriceProvider` singleton bean shown in **Listing 37-1** serves as an emulator of the data feed with stock quotes. Replace the random-number generator with a real-time Wall Street stock market feed, and this sample application will show the real data. I'd like to point out two annotations: `@Startup` and `@Schedule`. The first ensures that the Java EE application server automatically instantiates the `PriceProvider` bean whenever the server starts. The second is a calendar-based timer that will call the method `getPriceQuote()` every five seconds.



Listing 37-1: PriceProvider singleton bean

```
@Singleton
@Startup
public class PriceProvider {

    @EJB
    private MessageService messageService;
```

```
    // In this example the method getQuotes() simply generates
    // prices, but in the real world it will get the quote
    // from some external data feed

    @Schedule(second = "*/5", minute="*", hour="*")
    public void getPriceQuote() {

        String price = Double.toString(generator.nextDouble() * 100.0);
        String symbol = symbols[generator.nextInt(symbols.length)];

        messageService.sendMessage(symbol + " " + price);
    }

    private static final Random generator = new Random();

    private static final String[] symbols =
                    {"AAPL", "MSFT", "YHOO", "AMZN", "MOT"};
}
```

The stateless session bean `MessageService` publishes the text message given as an argument to `sendMessage()` to the JMS topic that has been bound to the JNDI tree of GlassFish under the name `jms/StockPrices`. Both the topic connection factory and the topic objects are injected into the bean by the EJB container. The JMS topic will be mapped to the physical topic `Prices` of the MOM provider Open MQ.



Listing 37-2: Stateless bean MessageService

```
@Stateless
public class MessageService {

    @Resource(mappedName = "jms/StockPrices")
    private Topic topic;

    @Resource(mappedName = "TopicConnectionFactory")
    private TopicConnectionFactory topicConnectionFactory;
```

```
    public void sendMessage(String text) {
        try {
            Connection connection = topicConnectionFactory.createConnection();
            Session session = connection.createSession(true, 0);
            MessageProducer producer = session.createProducer(topic);

            TextMessage textMessage = session.createTextMessage(text);
            producer.send(textMessage);

            connection.close();
        }
        catch (JMSException e) {
            throw new RuntimeException(e);
        }
    }
}
```

The web clients of this application are written in Java, JavaFX Script, and HTML. The Java class `Messages` is a part of the JavaFX applet that directly subscribes to the Open MQ topic `Prices`. Originally, I wanted the applet to use a JNDI call to GlassFish to get the topic `jms/StockPrices` from there. This would be a cleaner way of subscribing to the topic — no dependency on the particular MOM provider, and no client code changes needed should we decide to change the MOM provider or the application server.

Unfortunately GlassFish v3 doesn't offer a lightweight means of adding JNDI-related classes to the applet. It comes with the utility `appclient`, which simply packages 40 MB worth of Java classes into the one jar. Sending such a heavy extra weight over the network with the applet is not right, and that's why I decided to use a lighter solution: the applet's class `Messages` (**Listing 37-3**) doesn't perform JNDI lookup but directly connects to Open MQ. The Java class `Messages` uses an inner class called `JMSListener` and the interface `MessageHandler`.



Listing 37-3: The Java client subscriber

```
public class Messages {

    public static void startListening(MessageHandler messageHandler)
                                                        throws Exception {
       JMSListener jmsListener = new JMSListener(messageHandler);

        ConnectionFactory connectionFactory = new ConnectionFactory();
        connectionFactory.setProperty(ConnectionConfiguration.imqAddressList,
                                                         "mq://localhost:7676");

        connection = connectionFactory.createQueueConnection("admin","admin");

        Session session = connection.createSession(false,
                                                  Session.AUTO_ACKNOWLEDGE);

        Topic topic = session.createTopic("Prices");
        MessageConsumer consumer = session.createConsumer(topic);

        consumer.setMessageListener(jmsListener);
        connection.start();
   }

    public static void stopListening() {
        try {
            connection.stop();
        }
        catch (JMSException e) {
            throw new RuntimeException(e);
        }
    }

    private static Connection connection;

    private static class JMSListener implements MessageListener {
        private MessageHandler messageHandler;

        private JMSListener(MessageHandler messageHandler) {
            this.messageHandler = messageHandler;
        }
```

```
        @Override
        public void onMessage(Message message) {

            TextMessage textMessage = (TextMessage) message;

            try {
                System.out.println("Got message: " + textMessage.getText());

                messageHandler.handle(textMessage.getText());
            }
            catch (JMSException e) {
                throw new RuntimeException(e);
            }
        }
    }
}

public interface MessageHandler {
    void handle(String message);
}
```

Reading JavaFX Code

The JavaFX client shown in **Listing 37-4** is responsible for the UI. It's deployed as an applet and is played in the client by the Java plug-in. Starting from Java 6 Update 10, this plug-in runs not as a part of the web browser, but as an independent JVM.

This applet is deployed on the server and is described in a special text file with the extension `.jnlp` (see **Listing 37-7**). The client's machine has to have the JavaFX run time installed to play this applet. Because you have the latest Java SE installed on your computer, you won't have any problem playing JavaFX applets. But users with older versions of Java are not guaranteed to have the same results. This is one of the reasons JavaFX has difficulties competing with Adobe Flex or Microsoft Silverlight in a non-corporate uncontrolled environment.

JavaFX Script can access the Java API and in this sample the JavaFX code communicates with the preceding Java message listener. JavaFX Script files are compiled into Java classes and then run on JVM. JavaFX

developers need to have JavaFX SDK installed in addition to the run time.

The code in **Listing 37-4** shows you the syntax of JavaFX Script. Variable declarations start with the keyword `var`, and specifying the data type is optional — the compiler will try to figure out what type to use based on assigned values.

The keyword `bind` represents binding — the value of the variable on the right is assigned to the variable on the left, and all the changes to the right-side values will be immediately reflected in the variable on the left.

Functions in JavaFX are similar to methods in Java, but they don't have to be defined inside a class and can just appear anywhere in the script. Functions can be defined dynamically, which makes them much like Java anonymous inner classes, but simpler. To call a function asynchronously, much as you would call Java's `invokeLater()`, just give this function `FX.deferAction()` as an argument, where `FX` is a class with system utilities. This is done at the beginning of **Listing 37-4**.

The instance of `JavaFXMessageHandler` is given to the class `Messages` (**Listing 37-3**), and every time the new message is published to the topic the `addNewText()` method is invoked.



Listing 37-4: The JavaFX client Main.fx

```
class JavaFXMessageHandler extends MessageHandler {
    override public function handle(message:String):Void {
        FX.deferAction( function() { addNewText(message) } );
    }
}
```

```
Messages.startListening(new JavaFXMessageHandler());

var oldText = "";
var newText = "";

var oldTextOpacity = 0.0;
var oldTextYShift = -40.0;

var font = Font {
    name: "Verdana"
    size: 16
}

var oldTextLabel:Label = Label {
    font: font
    text: bind oldText
    textFill: Color.web("#808020")
    translateY: bind oldTextYShift
    opacity: bind oldTextOpacity
}

var newTextOpacity = 0.0;
var newTextYShift = 40.0;

var newTextLabel:Label = Label {
    textAlignment: TextAlignment.CENTER
    font: font
    textFill: Color.web("#808020")
    text: bind newText
    translateY: bind newTextYShift
    opacity: bind newTextOpacity;
}

var oldTextAnimation = Timeline {
    repeatCount: 1
    keyFrames: [
        at (0s) {
            oldTextOpacity => 1.0 tween Interpolator.LINEAR;
            oldTextYShift => 0.0 tween Interpolator.LINEAR;
        }

        at (1s) {
            oldTextOpacity => 0.0 tween Interpolator.LINEAR;
            oldTextYShift => -40.0 tween Interpolator.LINEAR;
        }
    ];
}

var newTextAnimation = Timeline {
```

```
    repeatCount: 1
    keyFrames: [
        at (0s) {
            newTextOpacity => 0.0 tween Interpolator.LINEAR;
            newTextYShift => 40.0 tween Interpolator.LINEAR;
        }

        at (.5s) {
            newTextOpacity => 1.0 tween Interpolator.LINEAR;
            newTextYShift => 0.0 tween Interpolator.LINEAR;
        }
    ];
}

function addNewText(text: String): Void {
    oldText = newText;
    newText = text;

    oldTextAnimation.playFromStart();
    newTextAnimation.playFromStart();
}

Stage {
    title: "Application title"

    scene: Scene {
        width: 600
        height: 100

        content: VBox {
            content: [
                Stack {
                    nodeHPos: HPos.CENTER
                    content: [
                        Rectangle {
                            width: 600
                            height: 80
                            opacity: 0.3
                            fill: LinearGradient {
                                startX: 0, startY: 0.0, endX: 0, endY: 80.0
                                proportional: false
                                stops: [
                                    Stop { offset: 0.0 color:
                                                Color.web("#303030") }
                                    Stop { offset: 1.0 color:
                                                Color.web("#FFFFFF") }
                                ]
                            }
                        },
```

```
                        oldTextLabel,
                        newTextLabel ]
                }]
            }
        }

        onClose : function() { Messages.stopListening(); }
    }
```

The price quotes are displayed on a gray gradient background (see **Figure 37-2**) and each new price quote pushes the previous one from the screen. Both old and new message texts have animation effects (`oldTextAnimation` and `newTextAnimation`) that make the view more engaging.

Figure 37-2



The JavaFX UI components are not created to live inside a specific top-level container. You add a group of components to a **scene**, which is added to the **stage**. At run time the stage will be linked to a top-level container according to the platform on which the application has to work. A mobile device and a desktop computer will use different top-level containers to display your stage. Each of the components that you add to the scene is an instance of a `Node` class.

The content of a `Scene` is listed in its `content` property. The code in **Listing 37-4** creates the `Stage` object and adds an instance of a `Scene`, which contains a `VBox` — a container with a vertical layout. Inside the `VBox` are three nodes: `Rectangle` with `LinearGradient` (a color effect changing from dark to light gray) and two `Label` nodes: `oldTextLabel` and `newTextLabel`.

Another interesting JavaFX component is `Timeline` — think of a reel of film in a projector. Spooling through frames rapidly produces the illusion of movement. If you need to repeat some actions in JavaFX you can create a `Timeline` with `keyFrames` that will occur at specific intervals.

The last line in the code of the `Stage` instance handles the `onClose` event. This event instructs the `Messages` class to stop listening to the price quotes.

The HTML Client

To provide another door to the messaging system you can use an HTML client that can send any text you want. This may be useful for managing the application. For example, an administrator may want to arrange it so that as soon as the client application receives the word `STOP` as a text message, it will stop listening to price quotes.

The simple HTML form in **Listing 37-5** has just one input field, and when the user presses the Enter key the text is posted to the server by the action `send`.



Listing 37-5: Index.html — the HTML client

```
<html>
<head>
    <title>JMS Demo</title>
</head>
<body>
    Click this link to open <a href="prices.html">
                  JavaFX client consuming price quotes</a>
    <p>Enter any text in the input field below and watch how JavaFx
client will receive it</p>
    <form action="send" method="POST">
        <label title="Text to send">
```

```
            <input value="" name="text"/>
        </label>
    </form>
</body>
</html>
```

The Java EE application server will route all HTTP requests with the URL pattern `send` to the servlet `SendMessage` from **Listing 37-6**, which will get the text entered by the user and invoke `sendMessage()` on the session bean `MessageService`. This text will be published to the topic `jms/StockPrices` as will any other messages that originate from the singleton `PriceProvider`.



Listing 37-6: SendMessage servlet

```
@WebServlet("/send")
public class SendMessage extends HttpServlet {

 @EJB
 private MessageService messageService;

 protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws IOException, ServletException {
  String text = request.getParameter("text");
  messageService.sendMessage(text);

  response.sendRedirect("index.html");
 }
}
```

When this JavaFX applet is ready for deployment on the server, you need to add resources that are needed on the user's machine. Our `Main.fx` client from **Listing 37-4** uses the JMS API to subscribe to the messages from the Open MQ MOM provider. Therefore the classes im-

plementing the JMS API have to be downloaded to the user's computer along with `Main.fx`.

The JNLP file shown in **Listing 37-7** has to be deployed on the server. It includes the `resources` section that lists what has to be downloaded to the client: the run time for JavaFX and the `javafx-ui.jar` file that includes our sample code and all related JMS supporting classes. The `<applet-desc>` tag includes the `main-class` parameter with a fully qualified name of the JavaFX applet itself with the name of the main class.



Listing 37-7: JNLP deployment file

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+">
    <information>
        <title>jms-send-receive</title>
        <vendor>practicaljava</vendor>
        <homepage href=""/>
        <description>jms send and receive</description>
        <offline-allowed/>
    </information>
    <resources>
        <j2se version="1.5+"/>
        <extension name="JavaFX Runtime"
 href="http://dl.javafx.com/1.3.1/javafx-rt.jnlp"/>

        <jar href="javafx-ui.jar" main="true"/>
    </resources>
    <applet-desc name="javafx-ui"
    main-class="com.sun.javafx.runtime.adapter.Applet"
width="200" height="80">
        <param name="MainJavaFXScript" value="com.practicaljava.lesson37.ui.Main"/>
    </applet-desc>
    <update check="always"/>
</jnlp>
```

The hyperlink in the main HTML page (**Listing 37-5**) downloads the HTML file `Prices.html` (**Listing 37-8**), which includes a special JavaScript function called `javafx` (its source code is available at **http://dl.javafx.com/1.3/dtfx.js**). This function is a part of the JavaFX deployment toolkit. It ensures that the client's computer has the right version of JRE and automatically generates the reference to the JNLP file shown in **Listing 37-7**.



Listing 37-8: Prices.html

```
<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
    <head>
        <title>Prices</title>
    </head>
    <body>
        <p>Real time price updates</p>

        <script src="http://dl.javafx.com/1.3/dtfx.js"
type="text/javascript" language="JavaScript"></script>
        <script type="text/javascript" language="JavaScript">
            javafx(
            {
                archive: "javafx/javafx-ui.jar",
                draggable: true,
                width: 600,
                height: 80,
                code: "com.practicaljava.lesson37.ui.Main",
                name: "Messages"
            }
        );
       A </script>
    </body>
</html>
```

Try It

This is the final hands-on exercise in this tutorial. The goal is to build the application and make it work on your computer. This time you'll be building and deploying the application and starting the servers from the command window without any help from Eclipse IDE.

Lesson Requirements

You should have Java and GlassFish v3 installed. You'll need to install Mercurial version control software to get the code for this example from the Google source control repository where all code samples used in this book are stored. You'll also need to install the build tool Maven.



You can download the code and resources for this Try It from the book's web page at **www.wrox.com**. You can find them in the Lesson37 folder in the download.

Hint

After this application is deployed from the command line, download the JavaFX plug-in for Eclipse IDE from **http://download.oracle.com/javafx/1.2/gettingstarted/eclipse-plugin** and try to create, deploy, and test this project inside Eclipse.

Step-by-Step

1. Download and install the distributed source control management tool Mercurial from **http://mercurial.selenic.com**. In the Terminal window enter **hg –version** and you should see the version of Mercurial installed.

2. Check out the code samples for the book into a folder called `java24hourtrainer` from **http://code.google.com/p/practicaljava**. You can do this

by entering the following command in the Terminal (command) window:

```
hg clone https://practicaljava.googlecode.com/hg/ java24hourtrainier
```

3. Download and install JavaFX 1.3 SDK, available at http://javafx.com/downloads/all.jsp?bundle=javafx_sdk. In Windows, JavaFX SDK will be installed in the default location `c:\Program Files\JavaFX`, and in Mac OS in `/Library/Frameworks/JavaFX.framework`.

4. Check to see if you have the Maven build tool already installed on your computer by entering **mvn –v** at the command prompt. If you don't have it, download and install it from http://maven.apache.org/download.html. All you have to do is unzip the downloaded file into a directory and add Maven's `bin` directory to the system `PATH` variable. In Windows you do this via Control Panel, and on Mac OS by entering the following command in the Terminal window (enter the exact name of your maven directory):

```
export PATH=~/development/apache-maven-3.0/bin/:$PATH
```

You'll be using the Maven build scripts that were created specifically for this lesson. Maven is out of the scope of this tutorial, but if you are curious to see where all the build logic is located, open up the three files called `pom.xml` in subdirectories `lesson37`, `webdemo`, and `javafx-ui` under `java24hourtrainer`.

5. To build the project for this lesson, in the Terminal or command window go to `java24hourtrainer/java-ee/lesson37` and run `mvn install`.

6. As a part of Maven's install, it'll create a directory called.`m2`. Create a plain text file in this directory, named `settings.xml`, with the following content:

```
<settings>
  <pluginGroups>
     <pluginGroup>com.practicaljava.maven.plugin</pluginGroup>
  </pluginGroups>
</settings>
```

7. Change to the directory `java24hourtrainer/maven-plugins` and run `mvn install`.

In several seconds Maven will have built two plug-ins — one for GlassFish and another for JavaFX.

8. Now go to the `/java-ee/lesson37/web-demo` directory and run the script to create a new domain in GlassFish; start it and deploy the web application there. In this example you are not going to run the instance of GlassFish that you've been using in previous lessons.

```
mvn gf:create-domain
mvn gf:start-domain
mvn gf:deploy
```

9. After the instance of GlassFish is running start Open MQ on port 7676 and its admin console, as described in the section "Administering Objects of Open MQ" in Lesson 30. You need to create a topic called `Prices` there.

10. Now start the admin console of GlassFish by entering the following URL in your web browser: **http://localhost:5000/**. Create there a `TopicConnectionFactory` and a topic `jms/StockPrices` mapped to `Prices`. This process was described in Lesson 31.

11. Finally, test this application from end to end. Make sure that the Open MQ and GlassFish servers are up and running, and point your web browser to **http://localhost:8080/javafx-web-demo/**. First you'll see a scary-looking message that an applet from dl.javafx.com is requesting access to your computer. Accept it and you'll see a simple web page with

a hyperlink and an input field, as per **Listing 37-5**. Click the hyperlink and you should see price quotes being pushed to your JavaFX client every five seconds, as shown in **Figure 37-2**.

12. Test the second way of publishing messages through the HTTP request to the servlet. To do this, open another web browser page and direct it to the same URL: **http://localhost:8080/javafx-web-demo/**. Arrange both browsers on your computer monitor so you can see them at the same time. This time do not click the link, but type any text in the input field and hit the Enter key on the keyboard. The text you entered should be received and displayed by the JavaFX client in the second web browser.



Please select Lesson 37 on the DVD with the print book, or watch online at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.

O'REILLY

Chapter 38

Java Technical Interviews

Technical job interviewing is a game with well-defined rules. I've worn the hats of both interviewer and interviewee many times over the last 12 years of my Java career. In this lesson I'll share with you my views on hiring Java developers and on preparing for technical interviews.

Regardless of whether the IT job market is hot or not, there are some rules and techniques that can increase your interview success rate. The process of getting a job consists of three separate tiers:

1. Getting the interview

2. Interviewing successfully

3. Considering the offer

I can't stress enough how important it is to work on achieving each of these goals separately, one step at a time! Your résumé is the most important thing in the first step. Adjust it for each position you are applying for (no, I'm not asking you to lie). Make sure it's short and to the point (not more than two pages long).

Getting the Interview

If you are applying for a Java developer's position, nobody needs to know the details of that Visual Basic project from 10 years ago. Always update your résumé based on the feedback you receive from recruiters or more experienced programmers.

There is a summary section on each résumé and many people just waste this space with some junk like "I'm looking for a challenging position that will let me improve my talents." What a waste! Use this summary line to show your relevant skills.

Say you've been specializing in Java messaging during the last two years, but this job posting requires web developers. Chances are that you've developed web applications before; highlight your web experience in the summary section of your résumé. The same day you may be responding to another ad, this one for people who know Java messaging. Modify your summary section accordingly and send a version of your résumé that emphasizes your experience with messaging.

Job requirements are more involved these days, and recruiting companies don't even want to submit your résumé to the client if you have "only" 8 out of 10 required skills. Read the requirements and highlight all the relevant skills you have. Do not be lazy; work with your résumé.

Networking is another great way to get an interview. This time I'm not talking about Java networking — I'm talking about meeting other IT people who may be hiring or know people who are hiring. Attend your local Java Users Group meeting, follow Java developers on Twitter, go to professional conferences and seminars.

Doing Well at the Interview

OK. The interview is scheduled, and now your main goal is to ace it. Remember, your interviewer has a difficult task: He needs to assess your technical skills within 30 to 60 minutes, so help him! Try to get as

many technical details about the job as possible from your recruiter. If the position you are applying for requires knowledge of Java sockets, research working with non-blocking sockets. If they're building Java EE applications, find out which application server they use and read up on it.

Do your homework and prepare a short talk about some interesting and challenging technical problems you might have experienced in one of your college or real-world projects. If you didn't have any super-complex projects, just pick up a topic from one of the multiple online programmers' forums and research it. Another great source of Java tips and tricks is Josh Bloch's book Effective Java, Second Edition, Addison Wesley, ISBN: 0-321-35668-3.

For example, if you have prepared a talk on concurrent collections, you're not allowed to leave the interview without talking about this. But what if the interviewer won't ask you about Java concurrency? It doesn't really matter. Find a way to switch the conversation to the prepared topic and do your best. The interviewer will be happy because he doesn't need to think about what to ask next, and you're happy because you've had a chance to talk about a well-prepared subject.

Will this technique work all the time? No. But it'll work most of the time. Obviously you need to research the selected subject well, or you'll get fried. You must talk about technical challenges you've resolved. You need to remember that your interviewers have a difficult time trying to figure out your technical abilities in just 30 minutes. So, you need to help them out. Be in charge.

If you're a junior developer, spend some time answering the multiple-choice computer tests that are usually required for certification exams. You don't need to get certified, but all these books and online mock tests will help you pass similar tests offered by some job agencies. Find some sample interview questions online.

A technical interview is a game with known rules, but in many cases the interviewers are not prepared to run the interviews. Sometimes they just go by a prepared list of questions. Some interviewees take advantage of this and just spend some time studying introductory courses, and then memorize questions and answers for technical interviews. Believe it or not, in many cases this works.

What does a good enterprise Java developer have to know in addition to understanding the difference between abstract classes and interfaces? Usually employers are looking for people with knowledge of the following: Java Servlets, JSP, Spring framework, EJB, JMS, any commercial message-oriented middleware, JDBC, JNDI, HTML, XML, Hibernate, Ant or Maven build tools, SQL, one of the major application servers, and a couple of relational database management systems.

Understanding why a particular Java EE component is being used in your current project is equally important. If the interviewer asks you, “Why did you use EJB in this project?” please do not answer, “That decision was made before I joined the project.” Have your own opinion and explain why you think it was a good or bad choice for this particular project.

Here’s another tip: Don’t critique the application architecture of your potential employer. You’ll have plenty of chances to provide technical advice if you’re hired, so just focus on getting an offer.

Be energetic during the interview and show your interest in the job. Even if you are a technical guru, don’t behave as if you’re doing the interviewer a favor just by showing up. Personality matters. People don’t like prima donnas.

Be prepared to write code on a whiteboard and practice that skill. The interviewer may not ask you to do this, but it may be a good idea to start illustrating your thoughts on the board if there is one in the room. If there is no board, use your own notepad. Often interviewers

are trying to judge how well you think and approach a problem more than how effectively you've memorized an algorithm, class name, or method arguments.

No need to rush. If you start writing a code fragment on a board, don't be afraid of making mistakes. Think aloud. It may be even beneficial for the interviewer to see how you can identify the wrong approach and then pick the right one. Practice explaining little code fragments to your friends or relatives; they don't need to know Java to participate in these role-playing games.

After the interview, as soon as you leave the building, take notes about what just happened. Don't postpone it until you get home; you may forget important details. Make a note to work on the questions you might have not answered correctly. These questions require your attention and research. Improve your technical skills after each interview.

Considering the Offer

You've got an offer! Now think hard about whether you want to accept it. Have I mentioned that you should look for a new job not when your employer decides to let you go or your contract ends, but when you have a stable job, the sky is blue, and the grass is green? This gives you a tremendous advantage: You can consider the offer without being under pressure from unpaid bills.

Don't accept an offer just because the new job pays an extra $5,000 a year, which comes to less than $300 a month after taxes. But do accept the offer that will give you a chance to work with interesting technologies or business applications even if it won't pay you an extra dime.

Other nonmonetary factors of an offer are health benefits, flexible work hours, ease of commute, or simply a kinship that you feel with the people working for the potential employer. Believe it or not, some

people won't accept offers from employers who require you to wear suit and tie. Just make a decision about what's more important for you — having an interesting job or being able to show off your latest tattoos.

No matter what your preferences are, take charge of your career and actively build it the way you want.

Interviewing Enterprise Developers

When the job market is healthy, major online job search engines show thousands of openings, and people are competing for these jobs. These days a seasoned developer has to know about 10 different tools or technologies to find a good job and feel relatively secure for a couple of years. Over the last several years I've interviewed lots of Java developers, and this is what I've noticed:

- People do not call themselves Java developers or programmer-analysts anymore — most of them prefer the title Java architect. Unfortunately, only some of them really understand how Java EE components operate and can suggest design solutions. Of course, knowledge of Java EE is not all that a Java architect should know.
- Job applicants are more senior, and I barely see any college graduates or junior programmers. Many of the junior positions are being outsourced and the number of graduates with computer science degrees has declined over the past several years.
- Having software certification does not make a résumé stand out. Actually, if a résumé starts with a list of certifications, most likely it's a beginner's. I'm not against certifications, as they help you learn a language or a tool, and show that you are willing to and can study. But a certificate doesn't make someone a skilled professional.
- With the introduction of middle-tier object-relational mapping frameworks such as Hibernate, many people don't even bother learning how database management systems work or how to write