IBM PRESS

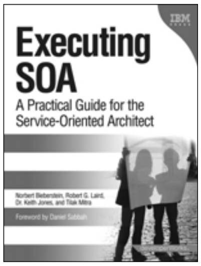
Modern Web Development with IBM WebSphere

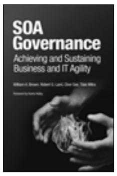
Developing, Deploying, and Managing Mobile and Multi-Platform Apps

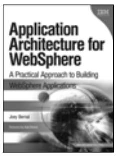
Kyle Brown ▪ Roland Barcia ▪ Karl Bishop ▪ Matthew Perrins

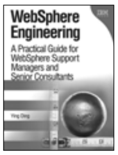
developerWorks® series

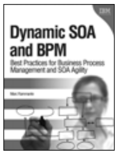
Related Books of Interest











This page intentionally left blank

Modern Web Development with IBM® WebSphere®

This page intentionally left blank

IBM PRESS

Modern Web Development with IBM® WebSphere®

Kyle Brown, Roland Barcia,
Karl Bishop, Matthew Perrins

IBM Press
Pearson plc
Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

ibmpressbooks.com

IBM Press Program Managers: Steven M. Stansel, Ellice Uffer
Cover design: IBM Corporation

Associate Publisher: Dave Dusthimer
Executive Editor: Mary Beth Ray
Marketing Manager: Stephane Nakib
Publicist: Heather Fox
Senior Development Editor: Christopher Cleveland
Technical Editors: David Artus, Gang Chen
Managing Editor: Kristy Hart
Cover Designer: Alan Clements
Senior Project Editor: Lori Lyons
Copy Editor: Krista Hansing Editorial Services, Inc.
Indexer: Publishing Works
Senior Compositor: Gloria Schurick
Proofreader: The Wordsmithery LLC
Manufacturing Buyer: Dan Uhrig

Published by Pearson plc
Publishing as IBM Press

Library of Congress Control Number: 2014935480

ISBN-13: 978-0-13-306703-3
ISBN-10: 0-13-306703-3

Text printed in the United States on recycled paper at Courier Westford in Westford, Massachusetts.
First printing June 2014

Kyle:
This one is dedicated to my wife, Ann, and my son, Nate,
who supported me and sometimes worried me
(yes, that was you, Nate!) during its writing.

Roland:
Thank you to God the Father and my Lord Jesus Christ.
Thank you, Blanca, for all your support.
Thank you to my children: Alyssa, Savannah, Amadeus, and Joseph.
Thank you, Kyle, for being a great mentor.
Thanks to my coauthors and to my family and friends.

Karl:
I'd like to thank my family—my wife, Cheryl, and my awesome kids, Matthew and Aubriana.
It still amazes me that you put up with me being so continually
distracted by work and this book for so long.
I would also like to thank IBM for being a great place to work and grow,
as well as all the IBM clients I've had the pleasure of working
with to provide high-quality solutions for your customers.

Matt:
To my family, Tania, Elyse, Fintan, Joe, and Tabitha—thanks for being patient.

Contents-at-a-Glance

Contents

Preface

"The only thing that is constant is change."
—Greek philosopher Heraclitus, circa 500 B. C.

Some days I can't believe it, but I've now been a consultant in the IT industry for more than 20 years. During that time, I've seen a number of technology shifts. When I started in the industry, the client/server wave was just beginning. What is now called the first generation of the web (or Web 1.0) came next, quickly followed by Web 2.0. Web 2.0 is now cresting, followed closely by the mobile wave.

The amazing part of all this is that each new technology wave appears to have little in common with the ones before them, but the lessons learned during one wave are actually foundations for the next wave. We developers learned the hard lessons of distributed computing during the client/server wave; that affected the way we went about building distributed systems in Web 1.0. In Web 1.0, we learned quickly that you can't easily separate the design of your website from the functionality of your website, and we saw that a good dynamic website is not the same as either a static website or a client/server system. These lessons about separation of concerns prepared us for the work of designing Web 2.0 websites that were more much responsive and easier to use. Likewise, the lessons we've learned in Web 2.0 about designing for different browsers and screen aspects have prepared us for the mobile wave.

Essentially, that's what this book is about. My co-authors and I have written it to help you understand how to apply all these different lessons we've learned over time in the context of a coherent strategy for building what we're terming "Modern" Web Applications (meaning ones suitable for use by mobile devices, or browser-based systems using Web 2.0 design techniques).

This book had its genesis in a previous book I wrote for IBM® press titled *Enterprise Java™ Programming for IBM WebSphere®*. In the two editions of those books, my coauthors and I concentrated on both providing soup-to-nuts coverage of the capabilities of IBM WebSphere Application Server and delivering that same kind of "here's the lessons you need to learn" approach. But the problem is that Enterprise Java, or JEE, has grown so large (as has WebSphere Application Server) that it's no longer feasible to cover all of it effectively in one book.

Instead, we've found that developers of Modern Web Applications have started to specialize in one of two areas: "Front-end" developers write the code that provides both the user interface and the API to the application. "Back-end" developers are more concerned with building infrastructure and dealing with issues of enterprise connectivity to mainframe systems, messaging systems, and so on. We feel that this forms a natural split between the issues involved, and the book you hold in your hand is our first of at least a two-part set.

Of course, just as in earlier books we had to show how developers build applications for IBM WebSphere Application Server using the Rational® VisualAge® for Java and then Rational Application Developer Toolset, now we have a whole new set of tools that developers need to understand. These include not only the tools that come in Eclipse for building Java applications with RESTful services, but also tools for building and deploying mobile applications, such as IBM Worklight®. We demonstrate to you in this book how the different parts of your team can use these tools to build Modern Web Applications more effectively.

Revamped IBM developerWorks Series

At the same time we were planning this book, my editors at IBM Press and I looked at the landscape of books about IBM products and noticed a gap in the coverage we needed to fill. *Enterprise Java Programming with IBM WebSphere* was one of the first titles in the IBM Press imprint. It was designed to provide practical, hands-on advice to teams adopting the then-new WebSphere Application Server. We've seen that, as new technologies such as mobile, cloud, and social computing technologies have developed, we've not provided that same level of practical, hands-on coverage. Thus, we also intend to meet that need with additional topics in the revamped IBM developerWorks® series—this book is the first example.

Just as IBM developerWorks has always provided the most up-to-the-minute information on topics of interest to developers, we want the books in this series to provide the best combination of in-depth instruction and links to new and updated material on the web so that the books will both inform our readers on the subjects that interest them and help readers follow along with exercises and examples even when the underlying technologies and products change.

So one of the key aspects of the books in this new series is that we not only provide links to information on developerWorks that is relevant to the topics in the text, but we also provide a "landing pad" about each book on developerWorks that links to constantly updated instructions for installing the tools, working through the examples, and helping developers understand what they need to do to be effective with the IBM products that the books are about.

You can find the landing page for this book at www.ibm.com/developerworks/dwbooks/modernwebdev/index.html. We hope you enjoy reading this book as much as we've enjoyed writing it.

—Kyle Brown, January 2014

How This Book Is Organized

This book gives you with a simple guide to the principal techniques and tools necessary to build Modern Web Applications with the IBM WebSphere Application Server and developer tools such as Eclipse and IBM Worklight. We alternate between more in-depth chapters focusing on our example and concept-focused chapters to help you gain an understanding of both the material and our recommended approach.

- **Chapter 1, "The Modern Web,"** defines what we mean by a Modern Web Application and introduces you to the landscape of technologies and tools that we use for developing Modern Web Applications. We also introduce you to the example we reference throughout the rest of the book.
- **Chapter 2, "WAS and the Liberty Profile,"** describes the first of our tools: the WebSphere Application Server Liberty profile for lightweight JEE application development. We provide a short "conventional" JEE application example to show you how applications are developed, deployed, and run with WebSphere Liberty.
- **Chapter 3, "Design,"** discusses the importance of an agile, user-centered design method and introduces Page-Oriented User Interface design. We also introduce the UI design patterns and elements for our example.
- **Chapter 4, "REST Web Services in WebSphere Liberty,"** demonstrates the advantages of using the WebSphere Application Server Liberty profile as a server for writing REST services. We introduce the JEE annotations used in building REST services and provide a progressively more complex example to show you how to construct useful REST services with WebSphere Liberty and Eclipse.
- **Chapter 5, "Application Architecture on the Glass,"** provides an overview of how to build an effective front-end application architecture. This includes lessons about building and using front-end frameworks, and it also covers the front-end JavaScript design of the example application.
- **Chapter 6, "Designing and Building RESTful Applications with Modern Java EE,"** introduces the techniques, technologies, and annotations to help you understand how to build more complex transactional RESTful services that interface with databases and other data sources. In this chapter, we revisit and complete our earlier samples to show you a full example of what constitutes a functional RESTful interface for a Modern Web Application.
- **Chapter 7, "Introduction to IBM Worklight,"** looks at the elements, architecture, and fundamental components of Worklight, the IBM platform for developing, deploying, and managing mobile and multiplatform applications.

- **Chapter 8, "Building a Worklight Hybrid App with Open Source Frameworks,"** covers how to build a Worklight application using the open source frameworks of jQuery Mobile, Backbone, Require.js, and Handlebars.
- **Chapter 9, "Testing and Debugging,"** introduces a number of techniques and tools for finding and fixing problems in cross-platform, multilanguage Modern Web Applications.
- **Chapter 10, "Advanced Topics,"** covers other topics relevant to building enterprise-scale Modern Web Applications. This includes scalability and caching, security, and services connectivity into the wider enterprise.
- **Chapter 11, "Key Takeaways for Modern Web Development,"** wraps up our coverage by discussing how Modern Web Development fits in with other emerging and dominant trends in the industry.
- **Appendix A, "Installation Instructions,"** covers how to locate, install, and configure all the software necessary to compile, run, and debug our examples, as well as how to obtain and download the sample code for the book. You can find its complete and most up-to-date form here:

  www.ibm.com/developerworks/dwbooks/modernwebdev/index.html.

Acknowledgments

We would first like to thank our wonderful technical reviewers, David Artus and Gang Chen, whose time, dedication, and effort made this book a much more valuable and insightful document. We would also like to thank Andrew Ferrier, Chris Mitchell, Cory Eden, Mohsen Yasarizare, Alberto Manjarrez, Leigh Williamson, Greg Truty, and others for additional insight.

We also want to thank the team at IBM Press for the opportunity to deliver this book to you. Thanks to Chris Cleveland for making this the best it can be. Thanks also to Executive Editor Mary Beth Ray for giving us the opportunity to create this book on a flexible schedule, despite several restarts and delays. Finally, we want to thank Steve Stansel, Editorial Program Manager for IBM Press, who helped us immensely through this whole process by cajoling us, lending a sympathetic ear, and keeping things moving along.

About the Authors

Kyle Brown is a Distinguished Engineer and CTO of Emerging Technologies with IBM Software Services and Support for WebSphere. He has 20 years of experience in designing and architecting large-scale systems. In his role as a DE, he is responsible for helping customers adopt emerging technologies, specifically cloud technologies and services-oriented approaches. He specializes in developing and promoting best practices approaches to designing large-scale systems using SOA, Java Enterprise Edition (JEE), and the IBM WebSphere product family. He is a best-selling author and regular conference speaker, as well as an internationally recognized expert in patterns, JEE, and object technology.

Roland Barcia is an IBM Distinguished Engineer and CTO for the Mobile and WebSphere Foundation for Software Group Lab Services. Roland is responsible for technical thought leadership and strategy, practice technical vitality, and technical enablement. He works with many enterprise clients on mobile strategy and implementations. He is the coauthor of four books and has published more than 50 articles and papers on topics such as mobile technologies, IBM MobileFirst, Java™ Persistence, Ajax, REST, JavaServer Faces, and messaging technologies. He frequently presents at conferences and to customers on various technologies. Roland has spent the past 16 years implementing middleware systems on various platforms, including Sockets, CORBA, Java EE, SOA, REST, web, and mobile platforms. He has a master's degree in computer science from the New Jersey Institute of Technology

Karl Bishop is a Product Manager with the IBM Worklight team. He currently works with the IBM Worklight Product Design team, focusing on developer experience. Before that, Karl spent many years working within the IBM Software Services for WebSphere group. His technical focus has been mobile app development, HTML5, Web 2.0, and JavaScript toolkits. Karl has worked for IBM for more than 16 years. He previously spent another dozen years honing his geek credentials at another computer company in California. Karl currently works out of his house, hidden away in the Sandhills near Pinehurst, North Carolina.

Matthew Perrins is an Executive IT Specialist and the lead architect for the BlueMix Mobile Backend as a Service Client SDK. He is the technical lead for IBM Software Services for Mobile for Europe, which is focused on delivering first-of-their-kind mobile solutions for IBM clients. He has worked for IBM since 1989 and has spent a significant amount of time designing and building Java-based enterprise solutions based on WebSphere.

This page intentionally left blank

CHAPTER 4

REST Web Services in WebSphere Liberty

What Is REST?

Earlier in Chapter 1, “The Modern Web,” and again in Chapter 3, “Design,” we discussed the REST approach to building web services. REST is about creating web services around a set of constraints. By sticking to the constraints that stem from the way the web was designed and built, you can take better advantage of the existing web infrastructure. Routers, caching proxies, and web servers are all optimized to deliver web content. Delivering your services as web content through these channels enables you to take advantage of existing optimizations in these channels.

The philosophy of RESTful services construction has two parts. One part centers on exposing resources, putting them into the hands of the masses and then allowing others in the community to create new types of applications by mixing and matching content from various places. In our context, you can view REST services as the data source for your model layer inside a Modern Web Application that runs in the browser or as a mobile app. REST services can provide data for reusable widgets that you can mix together to create mashups. REST services can also present data as feeds, notifying end users of content through the use of feed readers.

In general, almost any data—including business logic—can easily be expressed through REST. What’s more, it’s easy to build REST services that provide content in different forms (for instance, JSON and XML). For these reasons, REST services are extending into the API space and rapidly becoming accepted as the default standard for providing externally accessible web APIs. This services exposure for enabling reuse is a central part of the REST philosophy.

The second aspect of the REST philosophy focuses on using RESTful idioms to simplify access, increase orthogonality, and enable discovery. This is where the arguments tend to start. REST is based on a set of simple principles that derive from the HTTP specification and other web standards. Some developers tend to follow these principles very closely, whereas others are more lax in their compliance with the principles set forward in the original REST paper by Roy Fielding. In our examples, we tend toward a more strict interpretation of REST, but we do point out places where deviations might commonly occur.

The Pieces of a RESTFul Web Service

Creating a RESTful web service is like forming a sentence: You need a noun, a verb, and an adjective. In REST terms, nouns are the resources URLs point to. The verbs are the HTTP actions on resources. The adjectives (okay, this might be stretching the analogy) are the formats of data in which the resources are represented. We like to lay this out in tables similar to the way we broke down sentences in primary school. For example, Table 4.1 outlines how you might describe a set of services related to a prospect list application, such as the one in Chapter 2, "WAS and the Liberty Profile."

Table 4.1 Prospect URI Structure

Sentence (Resource Description)	Noun (URI)	Verb (Action)	Adjectives (Formats)
List all the prospects.	`.../prospects`	`GET`	JSON, XML
Get a specific prospect.	`.../prospects/{id}`	`GET`	JSON, XML
Add a contact.	`.../prospects`	`POST`	JSON, XML
Delete a specific contact.	`.../prospects/{id}`	`DELETE`	JSON, XML

- **Nouns/URIs:** URLs are the most identifiable part of the web and, as such, are a straightforward way of organizing your services. Organizing a unique URI for each resource avoids confusion and promotes scalability.
- **Verbs/actions:** In REST, you usually perform four HTTP operations against these URLs: `POST`, `GET`, `PUT`, and `DELETE`. (HTTP supports a few more actions, or officially *request-methods*, but these are the interesting ones.) Although having just four operations might seem constraining, the simplicity is somewhat liberating. These operations roughly map to Create, Read, Update, and Delete (CRUD). CRUD provides the foundational functions needed to interface with a relational database or other data store, so you can use these four methods in interesting and powerful ways.
- **Adjectives/data formats:** There are well known data types (the MIME types—text/html, image/jpeg) that HTTP servers and browsers natively support. Simple XML and JSON allow more custom data formats that are self-describing and can easily be parsed by the user. (When we say *parse*, we also mean "read with your eyes and parse with your brain.")

Using REST enables you to take advantage of many assumptions made by web infrastructure. Because you constrain the problem to only HTTP, you can make assumptions about items such as caching and HTTP-based security models. Because these technologies are ubiquitous, following this approach enables you to take advantage of existing solutions such as browser caches and web security proxies. By making your resources stateless, you can easily partition your resources across multiple servers, providing scalability opportunities. Another advantage is you can easily test HTTP-based services using a browser or a simple command-line tool such as cURL. By following RESTful idioms such as representing connections between resources by links in the data, you can enable runtime discovery of additional services. Finally, from the consumer perspective, services written to RESTful idioms have a regularity that enables you to benefit from examples and to practice reuse through cut-and-paste.

Building an effective REST architecture involves many aspects:

- Resources
- Resource types
- Query formats, headers, and status codes
- Content negotiation
- Linking
- Versioning
- Security
- Documentation
- Unit tests

We begin to cover these issues in this chapter, and we address more of these topics more fully in later chapters.

Introducing JAX-RS

Very soon after the REST model was described, it began to gain acceptance in the Java community. Early efforts focused on building REST services directly with Java Servlets, but an effort soon concentrated on creating a draft specification (JSR) for developing REST services in Java. The specification that resulted from that effort (JSR-033) became the JAX-RS standard. The authors of the JAX-RS standard set some specific goals for the JAX-RS approach:

- **POJO based:** The authors of the specification wanted to allow developers to build their services entirely with annotated POJOs—no special component model, such as earlier versions of EJB or web services standards, required.
- **HTTP-centric:** In keeping with the REST architectural approach, the JAX-RS standard assumes that HTTP is the only underlying network protocol. It does not attempt to be protocol independent—in fact, it provides simple mechanisms for exploiting the underlying HTTP protocol.

- **Format independent:** The developers of the standard also wanted to make the JAX-RS standard compatible with a number of different content types. Therefore, they focused on providing plugability so that additional content types could be added in a compliant way.

Basic Concepts: Resources and Applications

The most basic concept in the JAX-RS standard is the notion of a resource. A resource is a Java class that uses annotations to implement a RESTful service. If you consider a web resource to be a specific URI (or pattern of URIs) that represents an entity type, then the resource is the implementation of that entity. Resources are tied together logically by your `Application` subclass, which extends the `javax.ws.rs.core.Application` class provided by the JAX-RS runtime. To implement the simplest JAX-RS service in WebSphere Liberty profile, all you need are two classes and a bit of configuration.

A JAX-RS "Hello World" in WebSphere Liberty

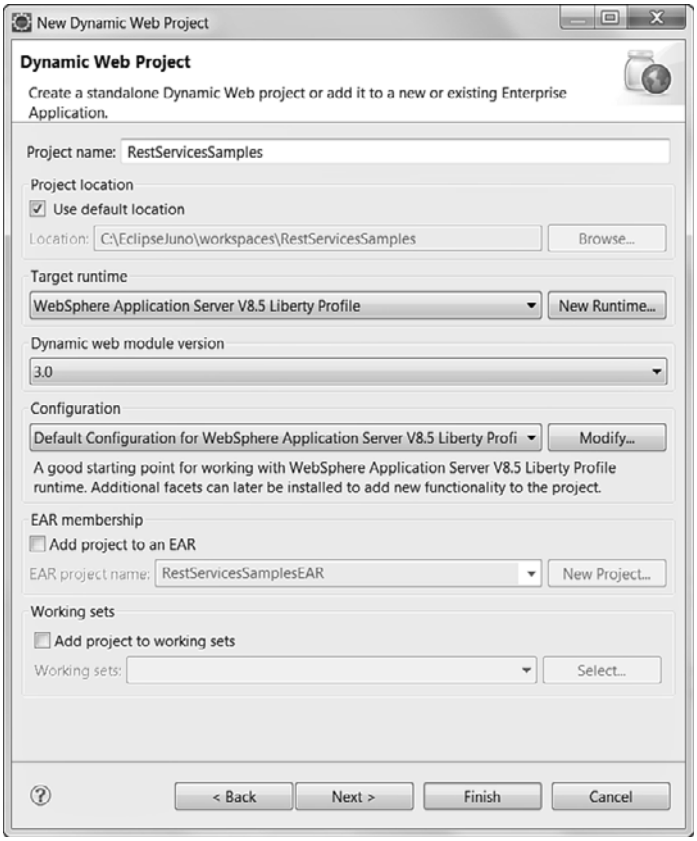
Given that we need to introduce several concepts with JAX-RS, we assume that you'll be developing your JAX-RS services inside Eclipse using the WebSphere Liberty Profile Test Server. For instructions on downloading and installing Eclipse, WebSphere Liberty, and the WebSphere Liberty tools for Eclipse, either see our website or take a look at Appendix A, "Installation Instructions." Note that the following instructions were specifically written and tested on Eclipse Juno for Java EE Developers Service Release 2 and the WAS 8.5.5 Liberty Profile. If you're using a different (later) version of Eclipse or WAS, you might see some differences, but they should remain nearly the same.

AUTHORS' NOTE

This chapter walks you through the process of creating a new web project and a number of new classes. If you'd rather not type in the code and you instead want to just run the completed examples, then follow the instructions in Appendix A and load the `Chapter4Examples.zip` file from the book's website into your Eclipse workspace.

You start the process by creating a new Eclipse Dynamic Web Project. The JAX-RS specification gives container providers some flexibility in how they can implement the specification, but they assume that artifacts will be deployed in a Servlet container such as WebSphere Liberty. In the Eclipse web development tool suite, a web project represents artifacts that are meant to be deployed to a Servlet container and packaged in a WAR file. If you're not familiar with development in Eclipse, you might want to first refer to any of the helpful tutorials on Java development with Eclipse.[1]

First, switch to a Java EE perspective in Eclipse. Then select **File > New**. The menu that pops up enables you to select a Dynamic Web Project. After you select that, the dialog box in Figure 4.1 appears.


New Dynamic Web Project

Dynamic Web Project

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: RestServicesSamples

Project location
☑ Use default location
Location: C:\EclipseJuno\workspaces\RestServicesSamples

Target runtime
WebSphere Application Server V8.5 Liberty Profile

Dynamic web module version
3.0

Configuration
Default Configuration for WebSphere Application Server V8.5 Liberty Profi
A good starting point for working with WebSphere Application Server V8.5 Liberty Profile runtime. Additional facets can later be installed to add new functionality to the project.

EAR membership
☐ Add project to an EAR
EAR project name: RestServicesSamplesEAR

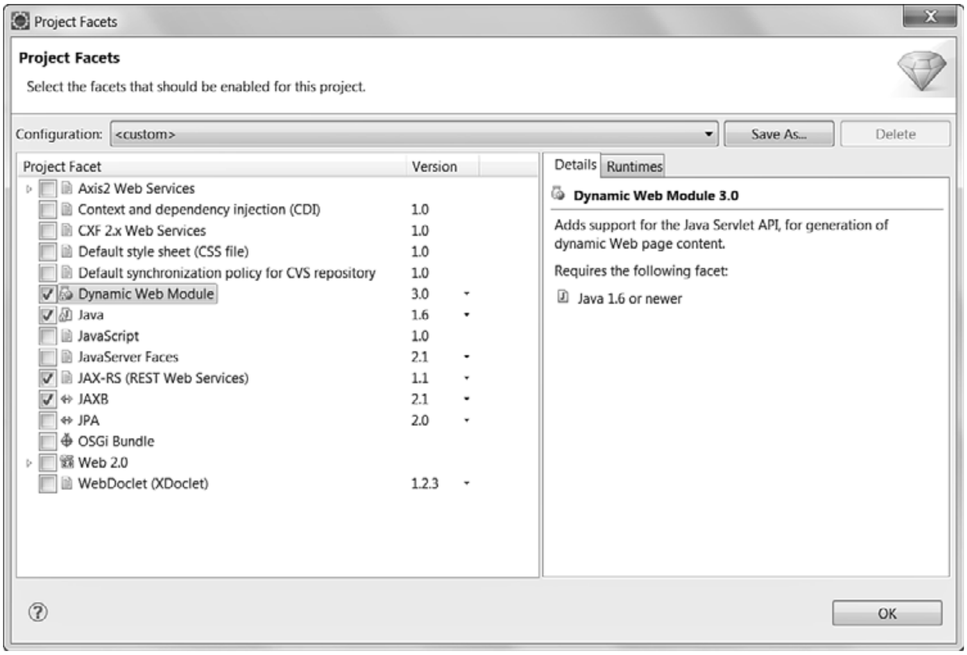
Working sets
☐ Add project to working sets
Working sets:

Figure 4.1 Creating a new web project

Name your project `RestServicesSamples`. For this particular project, we want to walk you through all the pieces included in a web project using REST in Eclipse, so you won't actually use all the built-in wizards for creating REST services that the Eclipse web tools provide. However, you will use some of the features to set up a project that uses JAX-RS and WebSphere Liberty.

Make sure that the Include in EAR check box is unchecked; you only need a WAR file from this project, so you don't need to worry about inclusion in an EAR.

Next, click the **Modify** button. This takes you to the page in Figure 4.2, which enables you to add *facets* to your project. Facets enable you to specify requirements and dependencies on specific technologies—Eclipse automatically handles modifications such as classpaths after you declare that you need those facets. On this page, check the check boxes to include support for JAX-RS and JAXB in your project (we explain why you need JAXB in the later section "JAXB and More Interesting XML-Based Web Services").



Project Facets

Select the facets that should be enabled for this project.

Configuration: <custom>

Project Facet	Version
☐ Axis2 Web Services	
☐ Context and dependency injection (CDI)	1.0
☐ CXF 2.x Web Services	1.0
☐ Default style sheet (CSS file)	1.0
☐ Default synchronization policy for CVS repository	1.0
✓ Dynamic Web Module	3.0
✓ Java	1.6
☐ JavaScript	1.0
☐ JavaServer Faces	2.1
✓ JAX-RS (REST Web Services)	1.1
✓ JAXB	2.1
☐ JPA	2.0
☐ OSGi Bundle	
☐ Web 2.0	
☐ WebDoclet (XDoclet)	1.2.3

Dynamic Web Module 3.0

Adds support for the Java Servlet API, for generation of dynamic Web page content.

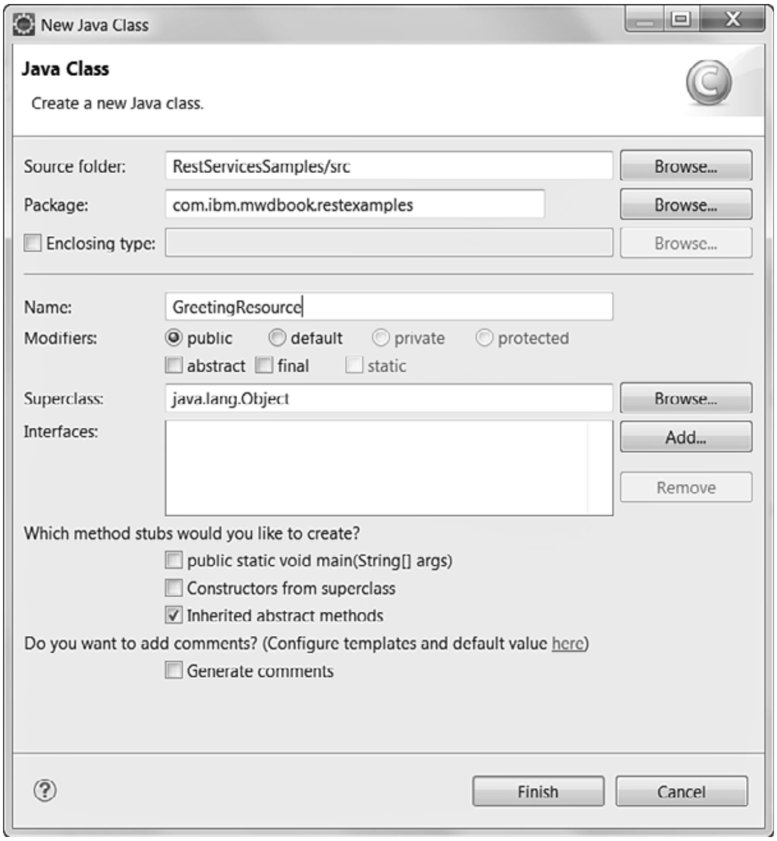
Requires the following facet:

Java 1.6 or newer

Figure 4.2 Modifying Facets

Finally, back on the Dynamic Web Project creation page, click **Finish**. Eclipse creates the project and might inform you through a dialog box that this type of project is best viewed in the JEE perspective; it asks if you want to open that perspective now. If you are not already in that perspective, answer Yes; you can work in the JEE perspective from then on.

Next, you need to create your first resource class. This resource class simply introduces you to many of the common annotations in JAX-RS and familiarizes you with the way you start and test REST services using Eclipse and the WebSphere Liberty Profile. Go to **File > New > Class** from within the web perspective, and open the dialog box in Figure 4.3 that enables you to create your first resource class.


New Java Class

Java Class

Create a new Java class.

Source folder: RestServicesSamples/src
Package: com.ibm.mwdbook.restexamples
[ ] Enclosing type:

Name: GreetingResource
Modifiers: [✓ public] default private protected
[ ] abstract [ ] final [ ] static
Superclass: java.lang.Object
Interfaces:

Which method stubs would you like to create?
[ ] public static void main(String[] args)
[ ] Constructors from superclass
[✓] Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)
[ ] Generate comments

Figure 4.3 Creating `GreetingResource`

Name your new class GreetingResource, and put it in a package named com.ibm.mwdbook.restexamples. Click Finish, and the Java Editor for your newly created class opens. At this point, you can go into the Java editor and change the newly created class stub to match the code in Listing 4.1.

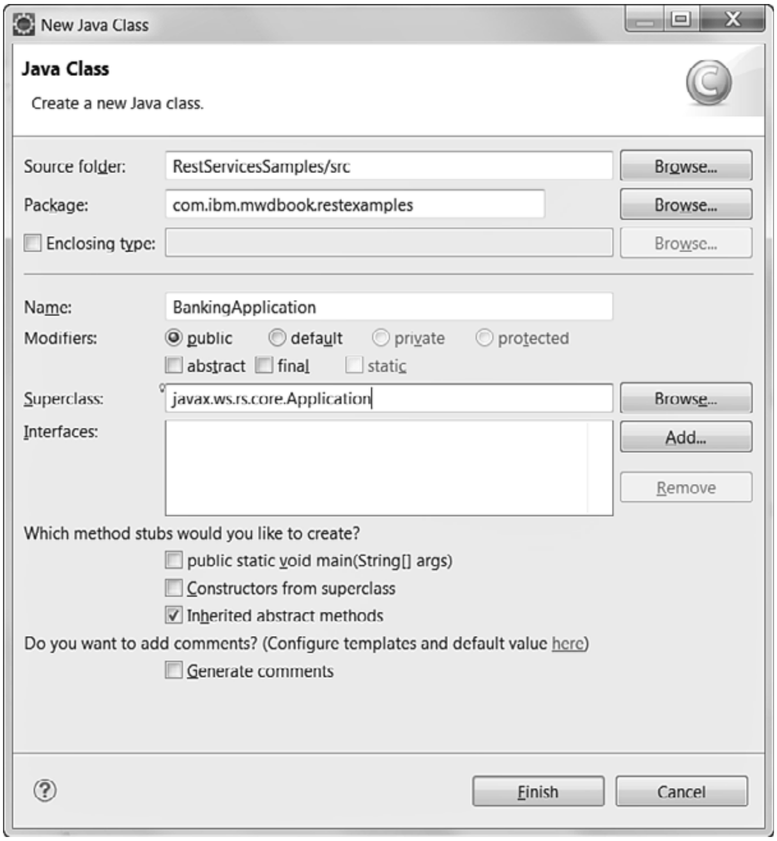
Listing 4.1 GreetingResource

```
package com.ibm.mwdbook.restexamples;

import javax.ws.rs.GET;
import javax.ws.rs.Path;

@Path("/Greeting")
public class GreetingResource {
@GET
      public String getMessage() {
            return "Hello World";
      }
}
```

This little example doesn't do much, but it does point out a few key aspects of resource classes in JAX-RS. First, notice that this class doesn't descend from any specialized subclass, nor does it implement any special interfaces. This is in accordance with the design of the JAX-RS specification, which aimed to allow developers to implement services as annotated POJOs. This was a principle that originated in JEE5 and has continued into later specifications. Next, notice the `@Path` annotation at the class level. As with the other annotations, `@Path` corresponds to a particular class—in this case, `javax.ws.rs.Path`, which you must import. In fact, each of the annotations you import in this example come from the `javax.ws.rs` package. `@Path` determines where in the URI space this particular resource is placed. Adding a path of `/Greeting` states that the last part of the URL (the resource identifier) will end in `/Greeting`. Other parts of the URL can be in front of the path identifier, but at least this identifies the end. In terms of the JAX-RS specification, annotating a class like this makes it a root resource class. This distinction becomes important when we start discussing subresources later.

The final point to notice about this simple example is the `@GET` annotation. Remember that, in the REST model, the HTTP methods are the verbs of the service. If the URI represents the noun that the action is performed against, then the method is the action that is performed. So the meaning of this simple example is that you are `GET`ting a greeting. That makes the response that we are returning, `Hello World!`, very appropriate! Now, of course, `@GET` isn't the only HTTP method annotation you can use; in the later section "Handling Entity Parameters with POST and the Consumes Annotation," we cover a case in which you use `@POST`, and Chapter 6 shows uses for `@DELETE` and `@PUT` as well.

The next piece of the puzzle to put in place is the `Application` subclass. According to the JAX-RS specification, the purpose of the `Application` subclass is to configure the resources and providers (we cover those later) of the JAX-RS application. In fact, you'll be editing and adding to the `Application` subclass as we expand the examples. For now, we start with another **File > New > Class** and bring up the new class dialog box. Your `Application` subclass should be named `BankingApplication` and should be placed in the `com.ibm.mwdbook.rest examples` package. The class needs to inherit from `javax.ws.rs.core.Application`. Figure 4.4 shows the completed fields in the dialog box.



Source folder: RestServicesSamples/src
Package: com.ibm.mwdbook.restexamples
☐ Enclosing type:
Name: BankingApplication
Modifiers: [✓ public] default private protected
☐ abstract ☐ final ☐ static
Superclass: javax.ws.rs.core.Application
Interfaces:
Which method stubs would you like to create?
☐ public static void main(String[] args)
☐ Constructors from superclass
☑ Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)
☐ Generate comments

Figure 4.4 Creating the Application subclass

After you enter these fields, click **Finish** and then replace the template text of the newly created class with the text in Listing 4.2.

Listing 4.2 `BankingApplication` Class

```
package com.ibm.mwdbook.restexamples;
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/banking/*")
public class BankingApplication extends Application {

}
```

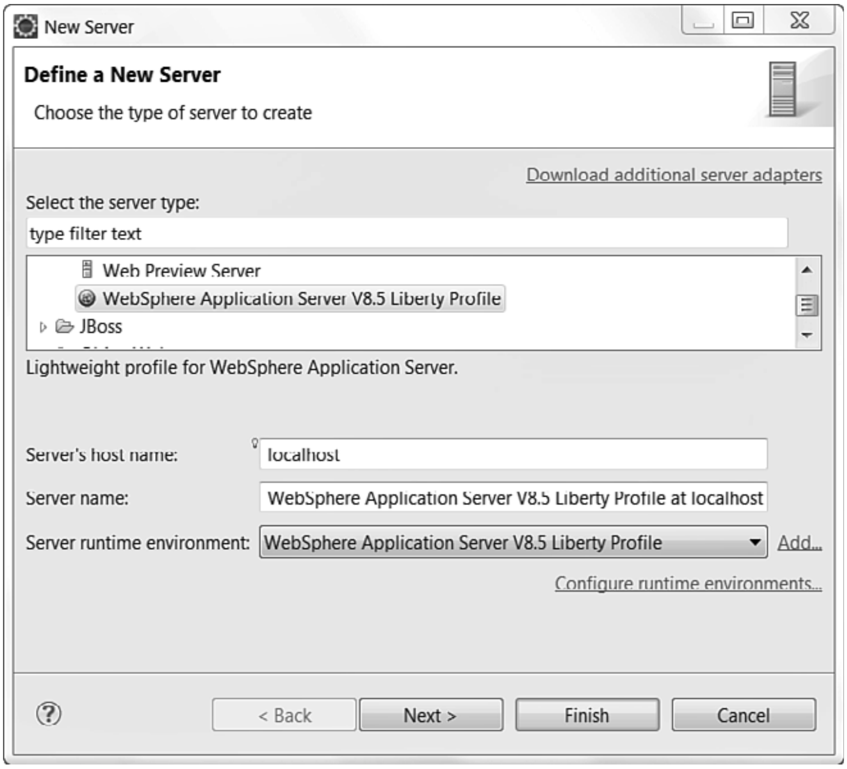
Note that we've added a single annotation in this class, the `@ApplicationPath` annotation. This annotation instructs the JAX-RS runtime what the path for JAX-RS resources will be. The path segment referenced in the annotation is added after the server name and web project context root, so resources are referenced by this pattern:

`http://localhost:9090/RestServicesSamples/banking/`*`your_resource_here`*

This is only one of three mechanisms defined in the Infocenter for configuring JAX-RS in the WebSphere Liberty Profile. This approach enables you to specify multiple JAX-RS applications with different application paths in the same JAR file, but it doesn't allow you to set up security constraints for the applications. For information on how to set up JAX-RS to allow that, refer to the Infocenter.[2]

Creating the WebSphere Liberty Server

You're almost ready to put the final piece in place for this simple example. Now that you've created all the artifacts necessary to implement a service, you need to deploy those artifacts into the WebSphere Liberty Profile. To do so, you must define a server in Eclipse. In this section, we assume that you created one server back in Chapter 2 when you tested a simple Web 1.0 example in Eclipse. If you haven't done so, your panels might differ slightly. Remember the discussion in Chapter 2 on how you might have different servers for different layers in your application. One advantage of defining multiple servers is that it keeps your servers from being cluttered by association with projects you don't need that might slow the startup of your particular server. For now, begin in the JEE Perspective by clicking the **Servers** tab at the bottom of the page, selecting the existing server you created in Chapter 2, and then using the right mouse button menu to select **New > Server**. The dialog box in Figure 4.5 then appears.


New Server

Define a New Server

Choose the type of server to create

Select the server type:

type filter text

Web Preview Server
WebSphere Application Server V8.5 Liberty Profile
JBoss

Lightweight profile for WebSphere Application Server.

Server's host name: localhost

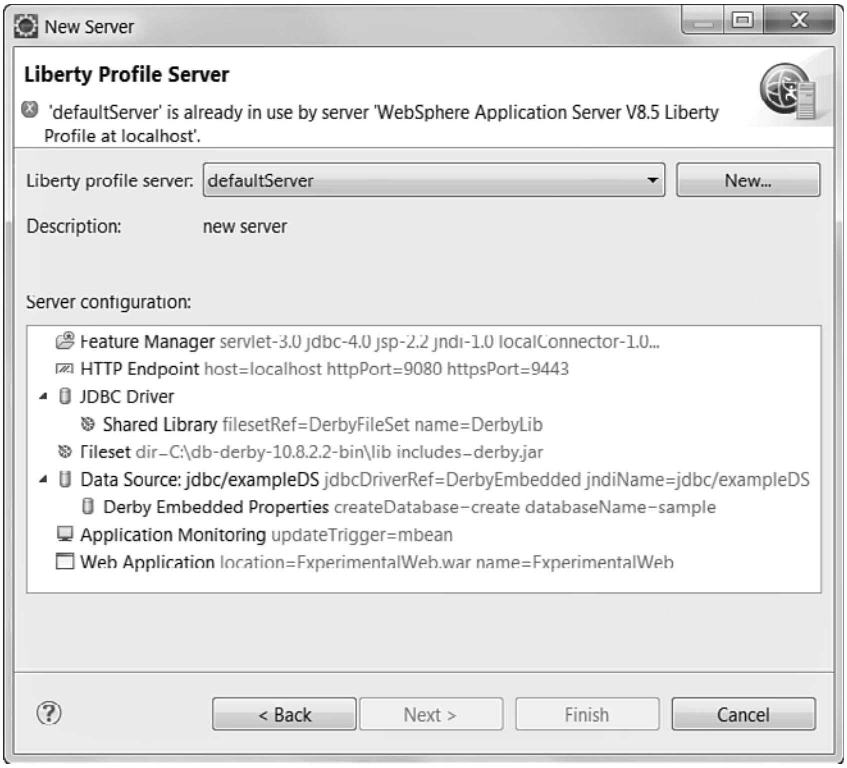
Server name: WebSphere Application Server V8.5 Liberty Profile at localhost

Server runtime environment: WebSphere Application Server V8.5 Liberty Profile

Download additional server adapters
Add...
Configure runtime environments...
< Back
Next >
Finish
Cancel

Figure 4.5 Define a New Server dialog box

In the Define a New Server dialog box, make sure you have selected WebSphere Application Server V8.5 Liberty Profile, and then click **Next**. The page in Figure 4.6 appears.



Liberty Profile Server

'defaultServer' is already in use by server 'WebSphere Application Server V8.5 Liberty Profile at localhost'.

Liberty profile server: defaultServer

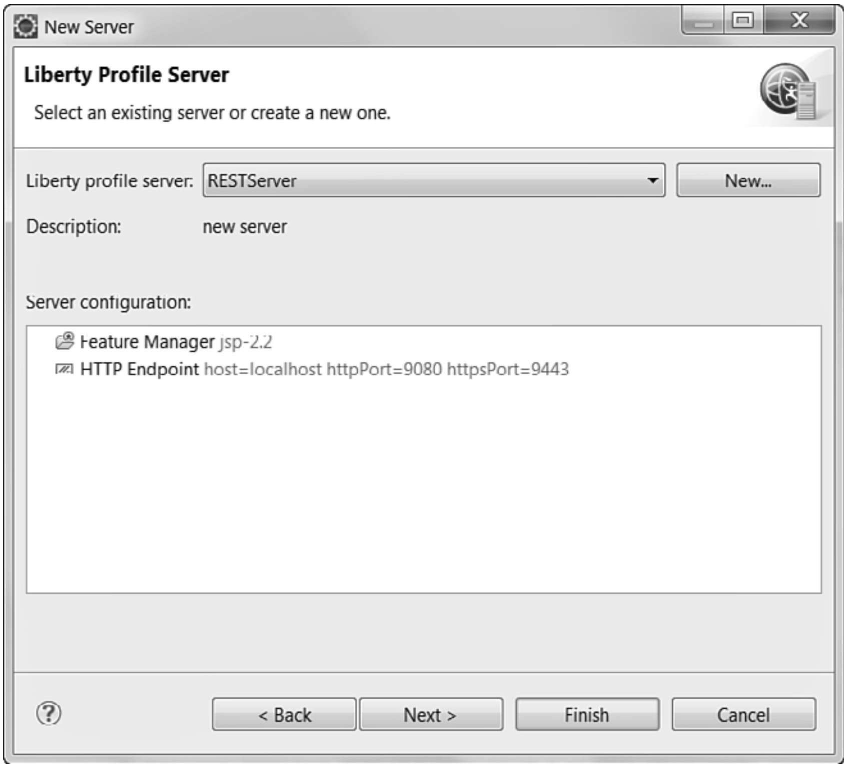
Description: new server

Server configuration:

```
Feature Manager servlet-3.0 jdbc-4.0 jsp-2.2 jndi-1.0 localConnector-1.0...
HTTP Endpoint host=localhost httpPort=9080 httpsPort=9443
JDBC Driver
    Shared Library filesetRef=DerbyFileSet name=DerbyLib
Fileset dir=C:\db-derby-10.8.2.2-bin\lib includes=derby.jar
Data Source: jdbc/exampleDS jdbcDriverRef=DerbyEmbedded jndiName=jdbc/exampleDS
    Derby Embedded Properties createDatabase=create databaseName=sample
Application Monitoring updateTrigger=mbean
Web Application location=ExperimentalWeb.war name=ExperimentalWeb
```

Figure 4.6 Creating a new server from an existing server

This dialog box informs you that you have already created one server named defaultServer and that this name is in use. On this page, click the **New** button. That action brings up the next page of this dialog box (see Figure 4.7).



Liberty profile server: RESTServer

Description: new server

Server configuration:

Feature Manager jsp-2.2
HTTP Endpoint host=localhost httpPort=9080 httpsPort=9443

Figure 4.7 New RESTServer creation

Give the server the name RESTServer. When you click **Finish** on the dialog box in Figure 4.7, you are taken back to the new Servers page, but this time you see a description of the server configuration for your new server, as in Figure 4.8. Note that your server is pretty bare bones at this time—only the basic configuration for your HTTP host and port is defined. That changes in the next step.

The final task in creating your server is associating your project with the server you just created. Click the **Next** button one final time. The dialog box in Figure 4.8 enables you to associate your project with the server by clicking the **Add** button to move the project from the list of available projects on the left side over to the list of configured servers on the right side.

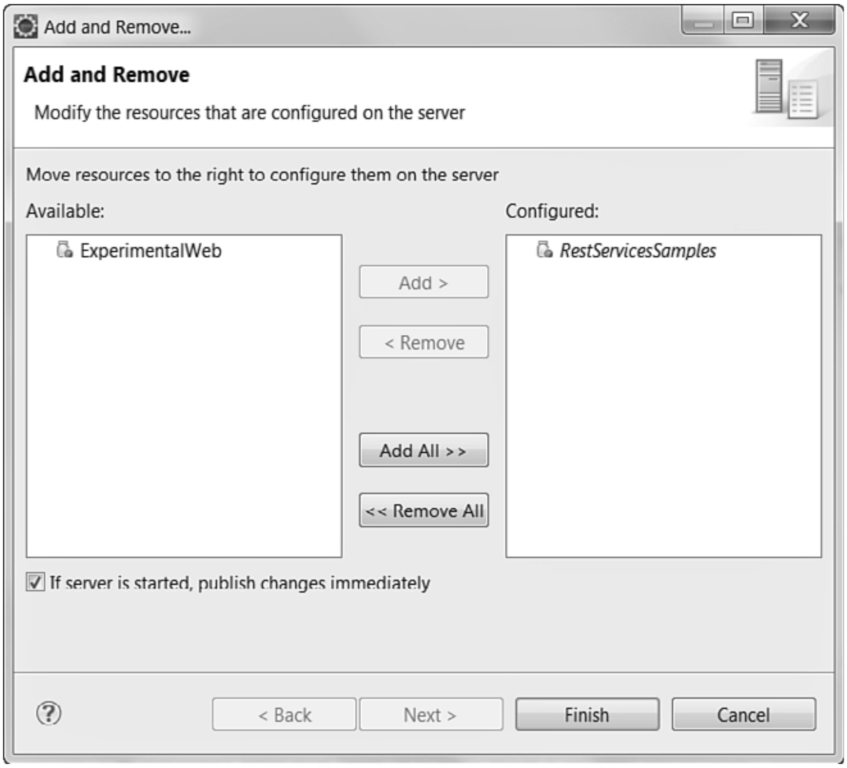
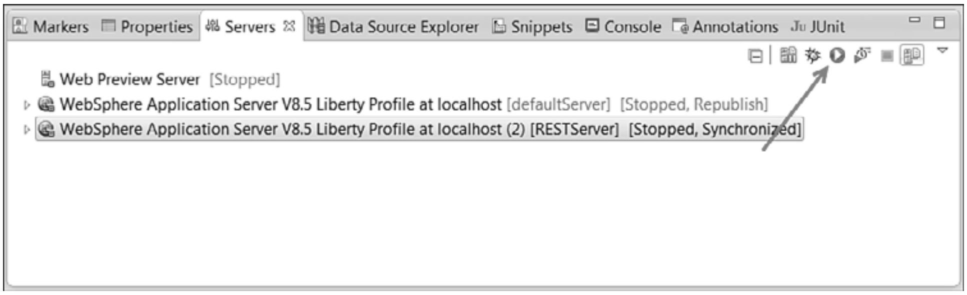

Add and Remove...
Add and Remove
Modify the resources that are configured on the server
Move resources to the right to configure them on the server
Available:
ExperimentalWeb
Configured:
RestServicesSamples
Add >
< Remove
Add All >>
<< Remove All
If server is started, publish changes immediately
< Back
Next >
Finish
Cancel

Figure 4.8 Add Project dialog box

At this point, you can finally click the **Finish** button to finish configuring the server. This adds the required features (in this case, JAX-RS support) to the `server.xml` file. Feel free to examine the `server.xml` file to verify that it has been reconfigured.

Starting the Server and Testing the Application

You're finally ready to test your application. Begin the process by starting the server you just created. On the **Servers** tab, click the green **Start** button in Figure 4.9.



Web Preview Server [Stopped]
WebSphere Application Server V8.5 Liberty Profile at localhost [defaultServer] [Stopped, Republish]
WebSphere Application Server V8.5 Liberty Profile at localhost (2) [RESTServer] [Stopped, Synchronized]

Figure 4.9 Starting the server

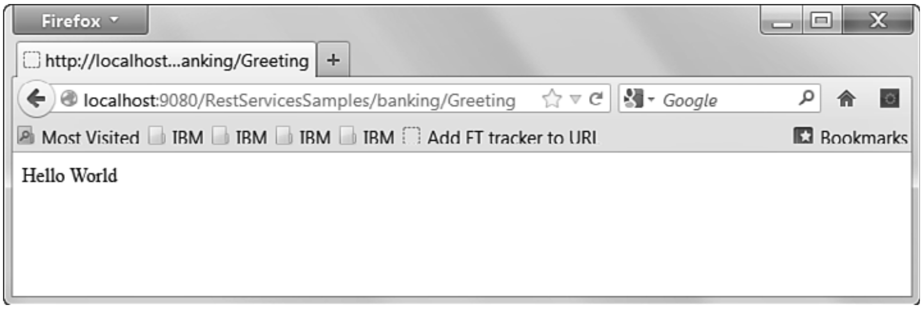
Now switch to the Console tab and make sure you see a message stating something like the following:

```
[AUDIT   ] CWWKZ0001I: Application SimpleBankingProject started in
0.419 seconds.
```

If you don't see this message, or if see an error message instead, take a look through the earlier messages in the console to find out what you did wrong. You can also turn to the end of this chapter and look at the debugging hints for JAX-RS services. Finally, presuming that everything went well, you can open a browser and type the following into the URL line:

```
http://localhost:9080/RestServicesSamples/banking/Greeting
```

If everything worked correctly, you should see your REST service greeting you with `Hello World!` (see Figure 4.10).


Firefox
http://localhost...anking/Greeting
localhost:9080/RestServicesSamples/banking/Greeting
Google
Most Visited
IBM
IBM
IBM
IBM
Add FT tracker to URI
Bookmarks

Hello World

Figure 4.10 Greeting results

More JAX-RS Annotations

Having a REST service greet you is nice, but it's hardly a very useful service. Next you'll implement a simple service that ties directly to our Banking example. Remember from the description of the sample in Chapter 1 that often a development team wants to mock up services so that the UI and Java development teams can test independently. The rest of this chapter walks you through the implementation of such a set of services—in Chapter 7, "Introduction to IBM Worklight," we show you a complete example that is much more like the production code that would be used to implement these services. Table 4.2 shows a set of services you need to implement as part of the example online and mobile banking solution.

Table 4.2 Services for Online Banking

URI	Description
`/banking/accounts`	List of accounts
`/banking/accounts/{id}`	Detail of given account ID
`/banking/accounts/{id}/transactions`	List of transactions for a given account
`/banking/accounts/{id}/transactions/{id}`	Detail of given account ID/transaction ID

We begin by looking at the Accounts resource. You can see two interesting resource references here: one service to return a list of accounts and another service to return the detail for a specific account. From this point on in the chapter, we assume that you know how to create new resource classes, so we just look at the code for each new class. Let's start by creating a new class in `com.ibm.mwdbook.restexamples` named `SimpleAccountResource`, with the code shown in Listing 4.3. We begin with this class in the listing and use it to learn some more features of JAX-RS; then we replace it with a more complete implementation in Listing 4.4. Remember that this class, as with all resource classes, has no special base class.

Listing 4.3 `SimpleAccountResource` Class

```
package com.ibm.mwdbook.restexamples;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;

@Path("/simple/accounts")
public class SimpleAccountResource {
      @GET
```

```
    @Produces("application/xml")
    public String getAccounts() {
          return "<accounts><account>123</account>"+
                      "<account>234</account></accounts>";
    }
    @Path("{id}")
    @GET
    @Produces("application/xml")
    public String getAccount(@PathParam(value="id") int accountId){
          if (accountId==123)
                return "<account><id>123</id>" +
                           "<description>Savings</description>" +
                           "<balance>110.00</balance></account>";
          else
                return "<error>No account having that id</error>";
    }
}
```

In this example, the code implementing the functionality of the service isn't the interesting part; it's the annotations that surround the functionality. The first `@Path` annotation simply sets up the basic URI of this example—remember that you're initially building a throwaway example that you will replace, so you don't use the actual URI in Table 4.2 for this example. Instead, to differentiate this example from others later, you prefix the end of this test URI with `simple` instead of just `accounts`, as the table shows.

As in the previous example, you begin with the method named `getAccounts()`, which returns an XML string representing a collection of two different accounts. The first point to notice is a new tag, `@Produces`, which states what type of content the method returns. In the case of both our new methods, this is `application/xml`. At this point, you might be wondering why we didn't need this for our previous example. The answer is simple—if you don't add the `@Produces` annotation to a resource method, the JAX-RS provider assumes that the content type is `text/html`.

As useful as that is, you can see a much more interesting new feature in the second `@Path` annotation added to the bottom method. Here we're adding the mechanism to handle a second part of the URI that comes after the `/accounts` portion handled by `getAccounts()`. In the example, we want to provide access to a specific account that is identified by an integer account number placed after the `/accounts` portion of the URI. The `@Path({id})` annotation identifies that specific account number. But then the question becomes, how do we manage to map the account number information from the URI to the `getAccount(int)` method? That's the role of the `@PathParam(value="id")` annotation.

`@PathParam` is probably the single most useful source of parameter information for resource methods, but it's not the only source. Table 4.3 shows some other common sources of information that you can use as parameters in your resource classes.

Table 4.3 Sources of Parameter Information in Resource Classes

Source	Description
`@QueryParam`	Individual query string parameter attached to the URI in the form `?name=value`
`@PathParam`	Parameter from URI template
`@CookieParam`	Http cookies value
`@HeaderParam`	Http header value
`@FormParam`	HTML form elements
`@Context`	Limited set of context objects

Testing the New Example

Entering that little bit of code is all you need to do for this new example. At this point, you should be able to test your new examples in your browser. You don't even need to restart the server—when you change the classes, Eclipse and Liberty automatically update the files on the server (using the `dropins` directory on the Liberty server that we mentioned in Chapter 2), so your changes take effect immediately.

Enter the following URLs into your browser to view the results. First, to see the list type, use this:

```
http://localhost:9080/RestServicesSamples/banking/simple/accounts
```

Then to see the individual account, type this:

```
http://localhost:9080/RestServicesSamples/banking/simple/accounts/123
```

Finally, to see the error path, type this:

```
http://localhost:9080/RestServicesSamples/banking/simple/accounts/234
```

JAXB and More Interesting XML-Based Web Services

We've now implemented a simple RESTful web service for the banking example, but it still leaves a lot to be desired. Hand-crafting XML might have been an appealing thought at the dawn of the REST services era, but it's hardly a scalable solution. Instead, we need to discuss ways of generating the XML produced by our services from our POJOs. This can be accomplished in several ways. One is using the `org.w3c.dom.Document` interface to generate a document from its parts. In some cases, this is the best possible approach, especially if you have to handle the generation of several different XML schemas. However, you usually don't need the flexibility of a dynamic approach. In such a case, a simple static approach that ties your POJOs to a single XML schema is best. The JAXB standard gives you the capability to easily generate XML documents from your POJO objects with just a few annotations.

The JAXB Annotations

Essentially, only two annotations are needed to get going with JAXB:

- **`@XmlRootElement:`** This annotation maps an entire Java class or enum type to an XML element. It's called the "RootElement" because it's the root of the tree of XML tags (with the attributes of the class being the leaves of the tree).
- **`@XmlElement:`** This annotation maps a JavaBean property or a nonstatic, nontransient field to an XML element.

One of the common changes made to the XML tags that are output by a JAXB mapping is to change the name of the tag (which, by default, is the same as the name of the field or property). This is done with the `name` parameter to the annotation. Consider the following example where you have a field named `foo` in your code, which you annotate with a standard `@XMLElement` tag:

```
@XmlElement
public int foo;
```

Your XML output then is of the form `<foo>123</foo>`. That might not be helpful for someone trying to read and parse the XML without knowledge of your special variable naming conventions. Instead, using `name` as follows

```
@XmlElement(name="accountNumber")
public int foo;
```

results in more readable output:

```
<accountNumber>123</accountNumber>.
```

Combining these annotations is easy. Consider a simple POJO class that represents an account in our banking example. As in previous examples, you create a new class in your project, named `com.ibm.mwdbook.restexamples.Account`, and then fill in the code from the example (see Listing 4.4).

Listing 4.4 Account Class

```
package com.ibm.mwdbook.restexamples;

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

@XmlRootElement
public class Account{


      int id;
      String accountType;
      String description;
      String currency;
      double balance;

      // For JAXB Serialization to work every class must have a
      // default no-arg constructor
      // if there are any other constructors defined!
      public Account() {
      }

      public Account(int i, String name, String type, double balance) {
            setId(i);
            setDescription(name);
            setAccountType(type);
            setBalance(balance);
            setCurrency("USD");
      }

      @XmlElement
      public int getId() {
            return id;
      }

      public void setId(int id) {
            this.id = id;
      }

      @XmlElement(name="name")
      public String getDescription() {
            return description;
      }

      public void setDescription(String description) {
            this.description = description;
      }

      @XmlElement(name="type")
      public String getAccountType() {
            return accountType;
      }

      public void setAccountType(String accountType) {
            this.accountType = accountType;
      }

      @XmlElement
      public String getCurrency() {
            return currency;
      }

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

      @XmlElement
      public double getBalance() {
            return balance;
      }

      public void setBalance(double balance) {
            this.balance = balance;
      }
}
```

A couple of points are worth calling out in this example. The first is the use of the `no-arg` constructor. Even if your code doesn't use a `no-arg` constructor, one is necessary for any class serialized by JAXB because the JAXB framework itself expects to use it. The second point to notice is that we've annotated the getter methods. This means that we've annotated the *properties* for this example; later in Listing 4.7, we annotate the *fields* and discuss the differences. In one particular case, we've even illustrated a common aspect of properties annotation—note this annotation:

```
@XmlElement(name="type")
```

Here, we want the name of the tag in the XML to differ from the name of the property itself. You can do this by specifying the `name=` property within the annotation. This way, the segment of XML generated would be of this form

```
<type> somevalue </type>
```

instead of the default:

```
<accountType> somevalue </accountType>
```

You'll find yourself substituting names like this fairly often when you have to work with existing XML schemas or JSON formats.

A Trivial DAO (and Its Use)

Now that we have an annotated POJO class representing our accounts, we need to turn our attention to how accounts are created and managed. One of the biggest contributions to the field of Java EE design over the last 15 years is the book *Core J2EE Patterns,* by Alur, et. al. Later developments in JEE have superseded many of the patterns called out in this book, but some are still as appropriate as ever. One in particular that is extremely helpful in many different situations, and one that we will follow in this book, is the Data Access Object (DAO) pattern. (You might remember that you built a simple JDBC-based DAO in Chapter 2.) The benefit of this pattern is that it provides an interface that encapsulates and abstracts away all the details of access to any data source. This enables you to replace one implementation of a DAO with another, without having to change any of the code that uses the DAO. So in this case, we're building a very trivial DAO that's useful for testing and hides the details of retrieving an account from an account list. Listing 4.5 shows the code for this DAO.

Note

Most of the example snippets in the rest of the chapter leave out the package declaration (always the same, `com.ibm.mwdbook.restexamples`) and the `includes` statements. For the most part, you've seen everything that needs to be included—besides, Eclipse can automatically patch these up for you using Source > Organize Imports. This makes the examples much shorter.

Listing 4.5 `AccountDao` Class

```
public class AccountDao {
      HashMap<Integer, Account> accounts = new HashMap<Integer,
Account>();
      public AccountDao() {
            Account anAccount = new Account(123,"savings",110.0);
            accounts.put(123, anAccount);
      }
      public List<Account> getAccounts() {
            List<Account> accountslist = new Vector<Account>();
            accountslist.addAll(accounts.values());
            return accountslist;
      }
      public Account get(int id) {
            return (Account) accounts.get(id);
      }
}
```

That's all you need for now—just a simple constructor that creates a `HashMap` of accounts and adds one to the list, and then a getter for both a list of all accounts and an account stored at a specific ID. However, that's enough to help implement our next, more useful example of a JAX-RS resource (see Listing 4.6).

Listing 4.6 `AccountResource` Class

```
@Path("/accounts")
public class AccountResource {
      AccountDao dao = new AccountDao();
      public AccountResource() {
      }
      @GET
      @Produces(MediaType.APPLICATION_XML)
      public List<Account> getAccounts() {
            return dao.getAccounts();
      }
      @GET
      @Path("{id}")
      @Produces(MediaType.APPLICATION_XML)
      public Account getAccount(@PathParam(value="id")  int id){
            return dao.get(id);
      }
}
```

We now have an example that's complete enough to be of some use. This is exactly the type of simple resource that you would implement when testing an AJAX UI that relies on a number of REST services to provide information that you can manipulate through JavaScript and HTML. You can see that we've used all the different annotations we've examined already, and we've also used the constants in the class `MediaType` instead of hand-coding `application/xml` for the `@Produces` annotation (which is a best practice to eliminate the possibility of mistyping).

To test the example, simply type the following into your browser:

```
http://localhost:9080/RestServicesSamples/banking/accounts/123
```

JSON Serialization

Although RESTful services that produce XML are common, it is perhaps even more common for the service to produce JSON, or the JavaScript Object Notation. JSON is a simple notation based on name/value pairs and ordered lists that are both easy to produce in many languages and extremely easy (in fact, part of the language) for JavaScript to parse and create. It's actually nothing more than a proper subset of the JavaScript object literal notation.[3]

When it comes to Java, especially inside the WebSphere Liberty Profile, you have several ways to produce JSON, just as you have different ways of producing XML. You can, of course, manually hand-code it, although that is not recommended. For more complex situations requiring a great deal of flexibility, a dynamic method of producing JSON might be needed, just as a dynamic approach to producing XML is sometimes helpful. However, the most common approach for producing JSON is the same as that for XML—using static annotations with JAXB.

A Simple Transaction Example with JAX-RS

To show how annotations for JSON work, we going to introduce another service from the list earlier in the chapter. The Transaction service enables you to view a transaction with `GET`, view a list of transactions with `GET`, and also create a new transaction by `POST`ing to the appropriate URL.

Let's start by introducing our `BankingTransaction` class (see Listing 4.7).

Listing 4.7 `BankingTransaction` Class

```
package com.ibm.mwdbook.restexamples;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class BankingTransaction {
      @XmlElement
      protected String id;
      @XmlElement
      protected Date;
      @XmlElement
      protected double amount;
      @XmlElement
      protected String currency;
      @XmlElement
      protected String merchant;
      @XmlElement(name="memo")
      protected String description;
      @XmlElement
```

```
	protected String tranType;

	public BankingTransaction() {
	}

	public BankingTransaction(String id, long date,String currency,
➥String memo, double amount, String tranType, String merchant) {
		setId(id);
		setDescription(memo);
		setAmount(amount);
		setCurrency(currency);
		setTranType(tranType);
		setMerchant(merchant);
		setDate(new Date(date));
	}
	public String getDescription() {
		return description;
	}
	public void setDescription(String aDescription) {
		description = aDescription;
	}
	public double getAmount() {
		return amount;
	}
	public void setAmount(double anAmount) {
		amount = anAmount;
	}
	public String getCurrency() {
		return currency;
	}
	public void setCurrency(String currency) {
		this.currency = currency;
	}
	public String getId() {
		return id;
	}
	public void setId(String id) {
		this.id = id;
	}
	public String getTranType() {
		return tranType;
	}
	public void setTranType(String tranType) {
		this.tranType = tranType;
	}
	public String getMerchant() {
		return merchant;
	}
	public void setMerchant(String merchant) {
		this.merchant = merchant;
	}
	public Date getDate() {
		return date;
	}
	public void setDate(Date date) {
		this.date = date;
	}
}
```

At this point, you might be thinking that this looks exactly like the annotations in the previous example. That's the point. If you use JAXB annotations, you have to annotate the class only once; you don't have to put in separate annotations for JSON and XML. Also, it's not entirely the same. Note that, in this case, we annotated the fields and not the properties—in practice, there is little difference between the two, and you can use either.

Handling Entity Parameters with `POST` and the Consumes Annotation

Now that you've seen how you can create a class with annotations that work for both XML and JSON, you can explore how to add methods to a service to take advantage of that. We're only introducing a couple new concepts in this part of the example—take a look at the following new method from the `AccountResource` class:

```
@Path("{account}/transaction")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@POST
public int putTransaction(@PathParam(value="account") int account,
                          BankingTransaction aTrans){
    return txnDao.putTransaction(account, aTrans);
}
```

The first new annotation is the `@Consumes` annotation. None of the previous services we've written have taken in any message bodies, so this is the first time we've needed to use it. As you can see, it's essentially similar to the `@Produces` annotation. The interesting part is how the browser interacts with the server based on these annotations. In this case, we're being very restrictive—we insist that the format of the message body be in JSON, but we also provide the response back in JSON. This information about what format is acceptable to the server and the client is communicated in specific HTTP headers. Figure 4.11 shows the interaction.

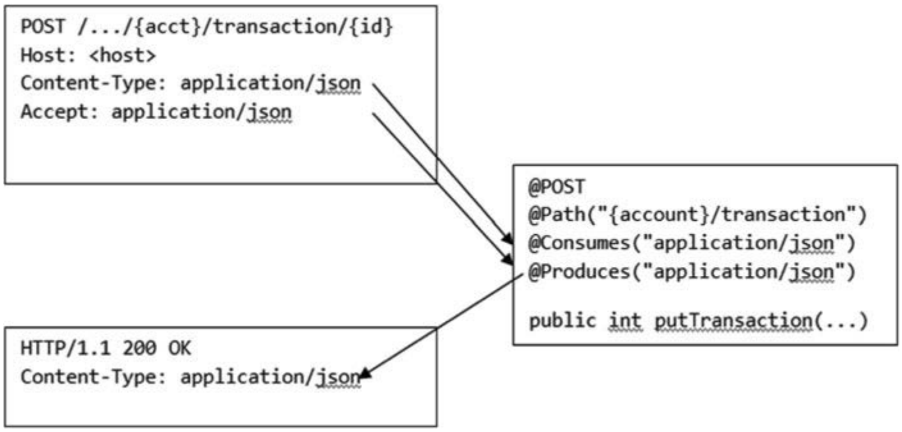

POST /.../{acct}/transaction/{id}
Host: <host>
Content-Type: application/json
Accept: application/json
@POST
@Path("{account}/transaction")
@Consumes("application/json")
@Produces("application/json")
public int putTransaction(...)
HTTP/1.1 200 OK
Content-Type: application/json

Figure 4.11 Header and annotation interaction

For a resource method to process a request, the `Content-Type` header of the request must be compatible with the supported type of the `@Consumes` annotation (if one is provided). Likewise, the `Accept` header of the request must be compatible with the supported type of the `@Produces` annotation. The `Content-type` header of the response will be set to a type listed in the `@Produces` annotation. This case is very simple—we're allowing only a single content type (`application/json`) into our service and a single content type (also `application/json`) out of the service; more complex cases might require *content negotiation*, which we discuss in more detail in the later section "More on Content Negotiation."

Looking back at the code for our `BankingTransaction` class, you see one more interesting fact about the `putTransaction()` method. Not only does it take a `@PathParam`, as have several of our preceding examples, but another method parameter is not attached to a `@PathParam` annotation: an instance of a `BankingTransaction` named `aTrans`. Where does this parameter come from? The JAX-RS specification is very clear on this: A resource method may have only one nonannotated parameter; that special parameter is called the entity parameter and is mapped from the request body. It is possible to handle that mapping yourself, but in our case (and in most cases), that mapping will be handled by a mapping framework such as JAXB.

The Use of Singletons in Application Classes

Before you can test your simple transaction-posting method, you need to understand a couple more concepts. The first is how we're implementing the DAO for this example. All the previous DAOs we implemented were just for prepopulating a collection with examples that we could retrieve with a `GET`. However, if we are now enabling `POST`, we want to be able to check that the information that we `POST` to the resource will be available on the next `GET` to that resource. In a "real" implementation of a DAO, that would be fine—we'd just fetch the values from a relational database on a `GET` and create the new rows on a `POST`. However, in our simplified example, we don't yet have that option (we show that in Chapter 6). Our solution for this case is very simple—we add a static variable that is an instance of our DAO to the DAO class. That way, we implement what in Design Patterns parlance is often called a singleton, a class that has a single instance. You can see this in the source code (see Listing 4.8) of our very simple `BankingTransactionDao`, which holds on to a single static variable that is an instance of the class that we name `instance`.

Listing 4.8 `BankingTransactionDao` Class

```
package com.ibm.mwdbook.restexamples;

import java.util.HashMap;

public class BankingTransactionDao {

      static BankingTransactionDao instance = new
➥BankingTransactionDao();

      public static BankingTransactionDao getInstance() {
            return instance;
      }

      HashMap<String, BankingTransaction> accounts =
              new HashMap<String, BankingTransaction>();
      int lastId=123;

      public BankingTransactionDao() {
            String key=deriveKey(123,123);
            BankingTransaction aTrans = new BankingTransaction("123",
➥1388249396976L,"USD", "paycheck", 110.0, "DEPOSIT", "DIRECT");
            accounts.put(key, aTrans);
      }

      private String deriveKey(int account, int id) {
            StringBuffer buf = new StringBuffer();
            buf.append(account);
            buf.append("-");
            buf.append(id);
            String key = buf.toString();
            return key;
      }

      public BankingTransaction getTransaction(int account, int id){
            String key = deriveKey(account, id);
            return (BankingTransaction)accounts.get(key);
```

```
    }

    public int putTransaction(int account, BankingTransaction aTrans)
    {
        int id=getNextID();
        String key = deriveKey(account,id);
        aTrans.setId(Integer.toString(id));
        accounts.put(key, aTrans);
        return id;
    }

    private int getNextID() {
        return ++lastId;
    }

}
```

Now the variable declaration of `txnDao` within our `AccountResource` class simply needs to obtain the instance of the `Dao` by invoking the `getInstance()` method, as follows:

```
BankingTransactionDao txnDao = BankingTransactionDao.getInstance();
```

However, although this is simple, it's not the best solution for most cases. A better approach is to consider that JAX-RS provides you with the capability to produce singleton instances of resource classes. In JAX-RS, the normal process is that a new instance of the resource class is created for every request. However, this might not always be the best choice. Even though it is a best practice that resources be stateless (as are the REST services themselves), sometimes a singleton instance can be useful—notably, when it needs to contain cached information to improve performance. Our simple service has another reason for this—to provide a stateful test service that mimics a service implemented on a backing store such as a relational database. You achieve this through the use of the `@Singleton` annotation. When your resource class contains this annotation, the resource class itself is considered to be a singleton and will live through the lifetime of the server. We show you many examples of `@Singleton`-annotated resource classes in our more fully fleshed-out example in Chapter 7.

Note

If you're interested in learning about the Singleton pattern, see *Design Patterns, Elements of Reusable Object Oriented Software,* by Gamma, et. al.

To complete this example, simply create a new class for your `BankingTransactionDao` and enter the previous code and then modify the `AccountResource` class to add the variable declaration for the `txnDao` and the new `putTransaction()` method we earlier described. You then need to let Eclipse patch up your import list using **Source > Organize Imports** so that the example compiles cleanly.

Testing `POST` and Other Actions with RESTClient

Now it's time to test adding a banking transaction to our newly defined `POST` resource method in our `AccountResource` class. However, that brings up a problem: In all the previous examples, we've been testing only `GET`ting a response from a resource, which can be tested in any browser. How can we test `POST`ing to a resource? That requires you to provide a message body and also (as you've already seen) a special `Content-Type` HTTP header. Essentially, a basic browser won't do for this case. You can look into several testing options:

- Curl (http://curl.haxx.se/) is a commonly used command-line client tool for transferring data with a URL syntax that can be used over a variety of protocols, including HTTP. Curl is especially useful for scripting if you need to write reusable test scripts.
- rest-client (http://code.google.com/p/rest-client/) is a simple Java GUI application from Google (although it also comes in a command-line version) that can be used for testing REST resources.

The solution we demonstrate in this chapter fits better with our methodology of testing resource methods within the browser: We use RESTClient, a free Mozilla add-on written by Chao Zhou that is available on the Mozilla add-ons site (see https://addons.mozilla.org/en-US/firefox/addon/restclient/). Chapter 9 discusses similar plug-ins for Chrome.

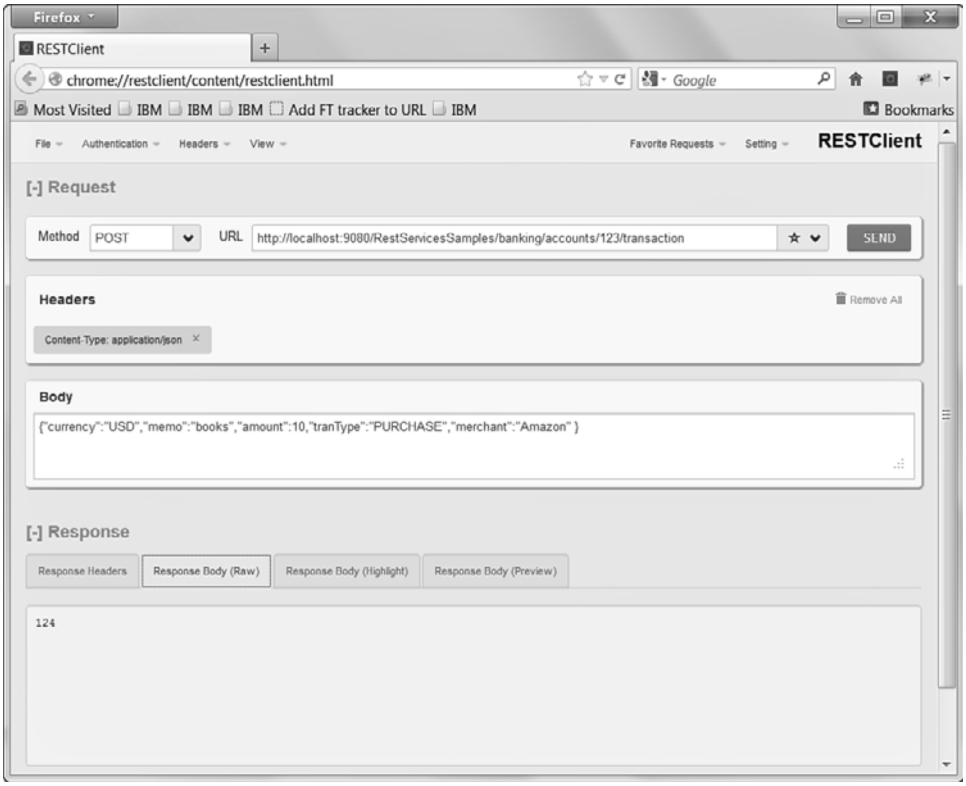
Obtaining RESTClient and installing it into Firefox is easy; just visit the site and follow the instructions. To start a RESTClient session, click the red RESTClient icon in the upper-right corner of your screen that is added during the installation process. You will see a new tab that looks something like the one in Figure 4.12.

When you are ready to test your new service, first select POST from the Method drop-down list. Then type the following URL on the URL line:

```
http://localhost:9080/RestServicesSamples/banking/accounts/123/
transaction
```

You need to add an appropriate `Content-Type` header. Click the Headers menu and choose Custom Header; then in the Request Header dialog box, type **Content-Type** as the Name and **application/json** as the value before clicking **OK** to dismiss the dialog box. Finally, type this into the body text area and click Send:

```
{"currency":"USD","memo":"books","amount":10,"tranType":"PURCHASE",
"merchant":"Amazon" }
```



Method POST URL http://localhost:9080/RestServicesSamples/banking/accounts/123/transaction

Headers

Content-Type: application/json

Body

```
{"currency":"USD","memo":"books","amount":10,"tranType":"PURCHASE","merchant":"Amazon" }
```

```
124
```

Figure 4.12 RESTClient for `POST` testing

If you look at either of the Response Body tabs, they should show the new ID number of your transaction (`124` if you've added only one `BankingTransaction`). Likewise, the Response Header tab should show a status code of `200 OK` and a `Content-Type` of `application/json`, as explained in our earlier diagram.

More on Content Negotiation

One of the most challenging parts of writing a resource is determining what `Content-Type` the resource for each method will accept and what `Content-Type` will be returned. The problem is that different `Content-Types` are better suited for different purposes, as we hinted at earlier. For instance, XML is extremely well suited to Enterprise-level SOA services because a number of languages can parse and generate XML. Likewise, XML has the benefit of a commonly accepted schema language (XML Schema) for defining valid XML documents. This enables you to associate a particular schema document representing the entire range of valid request or response bodies with each service that you write—this capability can be useful for creating an Enterprise registry of your services.

However, a great number of services will be mostly consumed by JavaScript code as part of the Modern Web Application architecture we've described. Thus, the simplicity and efficiency of JSON needs to be strongly considered also. So in practice, many of the actual services you write will have to handle the possibility of consuming and producing multiple `Content-Types`. People have suggested handling this content negotiation problem in a few ways.

You could accept different URI parameters for each content type and then implement different methods in your `Resource` class (each having a different `@Path` annotation) to differentiate between the two. The problem with this approach is that, although it's simple, it's not natural. Appending `.xml` or `.json` to the end of URI is not something most developers would think to do. Also, it has the problem that you now have two different methods that effectively do the same thing—so any changes thus have to be made in two places.

Another possibility is to use `QueryParams`. With this solution, you append a `?format=`*`someformat`* query parameter to the end of each URI or for cases when you want to use a format other than the default (presuming that you remembered to test for the query parameter being null). Although this avoids the two-method problem of the previous solution, it's still not natural. It makes the format request not part of the structure of the request URI itself, but something that hangs off the end. The problem with that kind of extension by query parameter is that when it's begun, it's hard to stop.[4]

The best way to avoid this kind of pain is simply not to follow *any* of these approaches. HTTP already gives you the right mechanism for negotiating the content type that should be returned, and JAX-RS provides easy support for this approach. To illustrate this, take a look at the following example, which is a new method in the `AccountResource` class:

```
@Path("{account}/transaction/{id}")
@Produces(MediaType.APPLICATION_XML+ ","+MediaType.APPLICATION_JSON)
@GET
public BankingTransaction getTransaction(@PathParam(value="account")
➥int account,@PathParam(value="id") int id){
      BankingTransaction aTrans = txnDao.getTransaction(account, id);
      return aTrans;
}
```

Note that, in this method, we've simply expanded on our earlier examples by adding two different `MediaTypes` into the `@Produces` method. Here, if the client states that it wants to receive back JSON, it needs to send along `application/json` in the `Accept` header. If the client wants to receive back XML, it sends `application/xml` instead. Likewise, this method interprets either XML or JSON correctly as its input; it uses the `Content-Type` header (looking for those same two values) to figure out which is which and determine how to interpret what is sent in.

Testing the new method is simple: After it has been added to the class, go back into the RESTClient in your browser and set the `Accept` header to `application/json`. Then perform a `GET` on the following URL:

```
http://localhost:9080/RestServicesSamples/banking/accounts/123/
transaction/123
```

The returned value in the body should be in JSON. However, you change the `Accept` header value to `application/xml`, you'll receive the value in XML.

Introducing the JAX-RS Response

So far in our examples, the only thing we've ever returned from a `Resource` method is what corresponds to the value the client should receive in the best of all possible cases—the case in which the request works. However, in the real world, you often need some more sophisticated error-handling techniques to deal with more complex problems. The JAX-RS spec essentially says that you can return three things from a `Resource` method. If you return `void`, that gives you back an empty message body and an HTTP status code 204 (`No Content`). We've seen the second case several times: You return an entity of some type, and the status code sent back is HTTP status code 200 (`OK`). However, sometimes it's important to be able to set your own status codes for more complicated error handling cases. That is where the JAX-RS spec defines one more thing you can return: an instance of `javax.ws.rs.core.Response`. `Response` contains methods for adding bodies that have other status codes—for instance, `temporaryRedirect()` for redirect (HTTP status code 307) and, the one we are interested in, `status`, which we use to send the correct response for a missing resource, HTTP status code 404 (`Not Found`). We demonstrate this in the following code snippet, which is a rewritten version of the `getAccount()` method from the `AccountResource` class:

```
@GET
@Path("/{id}")
@Produces(MediaType.APPLICATION_XML)
public Response getAccount(@PathParam(value="id") int id){
      Account value = dao.get(id);
      if (value != null)
            return Response.ok(value).build();
      else
            return Response.status(404).build();
}
```

Note a couple important points about what we've done to this method. First, the return type of the method is no longer `Account`, but `Response`. We are using two helpful static methods in `Response`: The method `ok(Object)` simply indicates that the response should be built around the object that is passed in as a parameter, and that the HTTP return code should be status code 200. By contrast, the `status(int)` method indicates that you simply want to return an HTTP status code. Both of these methods actually return an instance of a `ResponseBuilder` object. You create a `Response` from the `ResponseBuilder` by sending it the message `build()`. You might want to investigate the many other useful methods on the `Response` class on your own.

Hints on Debugging: Tips and Techniques

So far, we've assumed that everything has gone well for you in writing and running these examples. However, that's not always the case; sometimes you need to debug problems in the code. Eclipse itself provides a lot of helpful features. You can always examine the console log (which is STDOUT), and you can start the server in debug mode (using the bug icon to start it) to set breakpoints and step through your code. However, when the console doesn't give you enough information to help you determine the cause of your problems, another helpful place to look for more detailed information that is unique to the WebSphere Liberty profile is in your server's logs directory. We briefly passed by this directory in Chapter 2 when we examined the Liberty directory structure. Look under your WAS Liberty Profile installation directory for `/wlp/usr/RESTServer/logs`. In particular, if you encounter a problem that keeps a service from executing (for example, a mismatch between the `@Produces Content-Type` and the annotations that you provide in your entity classes), you can often find helpful information in the `TRACEXXX` logs that are created in this directory. The FFDC (First Failure Data Capture) logs are also good places to look for additional information in debugging problems.

NOTE

As we've mentioned, if you get tired of typing in these examples on your own, you can always download them from the book's website:
www.ibm.com/developerworks/dwbooks/modernwebdev/index.html

Simple Testing with JUnit

One of the best approaches to software development that has emerged in the last 15 years is the notion of Test Driven Development, popularized by Kent Beck as a part of the Extreme Programming method. Many other agile methods have adopted this principle, and we've found it to be extremely useful in our own development as well. A helpful tool that has emerged from this movement is the JUnit testing tool. JUnit enables you to write simple test cases that you can use to validate your own code.

From what you've learned in this chapter, you've probably concluded that testing at the browser is helpful for fast little validation tests, but it quickly becomes tedious. What's more, you've probably noticed that it is quite error prone and hardly repeatable, especially when you have to use a tool such as RESTClient to manually enter JSON or XML request bodies.

Let’s finish the code examples in this chapter with an example of how to write a JUnit test for one of the previous examples (see Listing 4.9).

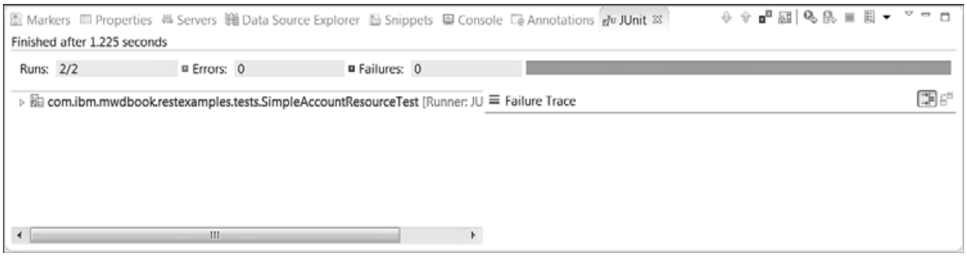
Listing 4.9 JUnit Test for `SimpleAccountResource`

```
package com.ibm.mwdbook.restexamples.tests;
import static org.junit.Assert.*;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLConnection;

import org.junit.Test;

public class SimpleAccountResourceTest {

      @Test
      public void testGetAccounts() {
            String address = "http://localhost:9080/
➥RestServicesSamples/banking/simple/accounts";
            StringBuffer result = new StringBuffer();
            String expectedResult = "<accounts><account>123</account>"+
                        "<account>234</account></accounts>";
            try {

                  fetchResult(address,result);
            } catch (MalformedURLException e) {
                  e.printStackTrace();
                  fail("MalformedURLException caught");
            } catch (IOException e) {
                  e.printStackTrace();
                  fail("IOException caught");
            }
            assertEquals(expectedResult, result.toString());
      }


      private void fetchResult(String address,StringBuffer result)
➥throws MalformedURLException,IOException
            {
            URL;
            url = new URL(address);
            URLConnection conn = url.openConnection();
            BufferedReader in = new BufferedReader(new
            InputStreamReader(
                        conn.getInputStream()));
            String inputLine;
            while ((inputLine = in.readLine()) != null)
                  result.append(inputLine);
            in.close();
      }

}
```

JUnit tests are like most other classes in Java today—they are annotated POJOs. In this case, the annotation `@Test` (from `org.junit.test`) identifies a method as a JUnit test method. A single class can have many test methods, each individually identified. In this excerpt, we show only the test method for the `getAccounts()` method, which corresponds to a `GET` to the URL http://localhost:9080/RestServicesSamples/banking/simple/accounts. If you browse the full class definition as included in the sample code, you'll see that we also include test methods that test retrieving an account that has an account number defined, as well as the error case of retrieving an account that does not have an account number defined. All these methods use a utility method named `fetchResult()` that uses a `URLConnection` to fetch the contents of a particular URL.

After fetching the results, we use JUnit assertions (from `org.junit.Assert`) to compare the value we receive against an expected value. So if you know the XML or JSON you want to compare against ahead of time, you can easily set up a test that can validate that your code still functions, even after multiple changes.

Running the JUnit Test is extremely easy. Just select the `SimpleAccountResourceTest` class in the Java EE explorer pane and then right-click and select **Run As > JUnit Test** from the menu. You should see a result in the bottom-right pane that looks like Figure 4.13.



Finished after 1.225 seconds

Runs: 2/2 Errors: 0 Failures: 0

com.ibm.mwdbook.restexamples.tests.SimpleAccountResourceTest [Runner: JU

Failure Trace

Figure 4.13 JUnit test results

This is an improvement over using a browser for testing, but as you can tell from close inspection of the code, even this approach leaves much to be desired. In particular, the `fetchResult()` utility method is limited to `GET`s and does not provide a way to pass in header values or even a message body. You can address each of these issues, but the method becomes more complicated as a result (see the corresponding method in the class `AccountResource-Test` for an example of exactly how complicated). Thus, you'll want a better mechanism for invoking the REST APIs than hand-coding it with a `URLConnection`. In Chapter 7, you examine using the Apache Wink client for just that purpose.

RESTful SOA

Before closing this chapter, which has been a whirlwind tour of REST, JAX-RS, and JAXB, as well as how to use them in the WebSphere Liberty Profile and Eclipse, we need to talk about the way in which the services we've been talking about here relate to the Enterprise services you might also want to implement. The problem is exactly how often the technologies discussed here, as powerful as they are, should be applied.

An old adage that still rings true is, "If all you have is a hammer, everything looks like a nail." That's true of technologies as well as physical tools. Another truism is that all technologies follow the "hype cycle," popularized by the analyst firm Gartner, Inc.[5] In the hype cycle, all technologies first encounter a period of inflated expectations, followed by the inevitable "trough of disillusionment." In the years since the publication of the predecessor volume to this book,[6] the hype cycle has definitely run its course for Enterprise Web Services with SOAP and the WS-* standards. Many people tried applying the WS-* standards to situations for which they were not appropriate, or when something simpler would definitely have worked better. However, it's easy to see that the REST services that are rapidly replacing WS-* in the hearts and minds of Enterprise developers could soon suffer the same fate. You have to be smart about what you're building when you're crafting a service and what you're building it for. Otherwise, REST can simply become another hammer used inappropriately for trying to insert a wood screw.

It also comes down determining what each different service implementation approach is best suited for. REST is about exposing content through HTTP; it does not replace traditional WS-* web services. Traditional web services are much more about program-to-program integration. The WS-* approach allows for a variety of protocols and integrating logic. For example, you might need distributed transactions and message delivery on top of a reliable protocol such as JMS when assured delivery is essential.

Traditional web services built using WS-* standards are about technology abstraction to allow for a wide variety of infrastructures. REST-based web services are about HTTP exploitation, to take advantage of a single delivery channel. For example, WS-* abstracts security through WS- Security, enabling you to apply security policies away from the code and abstracting the encryption methods and decisions such as token providers. REST-based services, on the other hand, make assumptions, using HTTP security mechanisms such as Open ID or HTTPS for encryption.

The real key lies in channel abstraction rather than channel exploitation. REST does not address transactions, reliability, or standard context propagation, or other protocols such as messaging. However, these are important considerations for the enterprise as a whole. REST is great for the responsibilities it is meant to handle, but it can't do everything. The two approaches can coexist in the same enterprise—and they should. Always make sure you choose the right tool for the job.

Summary

This chapter discussed a lot. You've seen how the JAX-RS standard defines REST services in Java, you've explored how to annotate POJOs with JAXB, and you examined how to implement these services in the WAS Liberty Profile. In the following chapters, you use the things you've learned in this chapter to build more complex REST services using JAX-RS; then you connect these to front ends built using JavaScript.

Endnotes

1. See Eclipse Java IDE – Tutorial: www.eclipse.org/resources/resource.php?id=505.
2. See Configuring JAX-RS applications using JAX-RS 1.1 methods: http://tinyurl.com/latksog.
3. See JSON in JavaScript: www.json.org/js.html.
4. The Google Search REST API (https://developers.google.com/custom-search/v1/using_rest) is a particularly egregious example of this: its documentation has multiple pages of required and optional parameters.
5. See Hype Cycles, www.gartner.com/technology/research/methodologies/hype-cycle.jsp.
6. Brown, Kyle et al. *Enterprise Java Programming with IBM WebSphere* (Second Edition). IBM Press 2004.

This page intentionally left blank

Index