Designing and Developing Distributed Web Services

RESTful Java

with JAX-RS

O'REILLY®

Bill Burke

RESTful Java with JAX-RS

Learn how to design and develop distributed web services in Java, using RESTful architectural principles and the JAX-RS specification in Java EE 6. With this hands-on reference, you'll focus on implementation rather than theory, and discover why the RESTful method is far better than technologies like CORBA and SOAP.

It's easy to get started with services based on the REST architecture. *RESTful Java with JAX-RS* includes a technical guide that explains REST and JAX-RS, how they work, and when to use them. With the RESTEasy workbook that follows, you get step-by-step instructions for installing, configuring, and running several working JAX-RS examples, using the JBoss RESTEasy implementation of JAX-RS.

- Work on the design of a distributed RESTful interface and develop it in Java as a JAX-RS service

- Dispatch HTTP requests in JAX-RS, and learn how to extract information from them

- Deploy your web services within Java Enterprise Edition, using the Application class, Default Component Model, EJB Integration, Spring Integration, and JPA

- Explore several options for securing your web services

- Learn how to use JAX-RS to implement RESTful design patterns

- Write RESTful clients in Java, using libraries and frameworks such as java.net.URL, Apache HTTP Client, and RESTEasy Proxy

" *Essential reading for Java programmers who want to learn and apply the philosophy of the Web.* "

— **Leonard Richardson**
author, RESTful Web Services

Bill Burke is a Fellow at the JBoss division of Red Hat, Inc. A longtime JBoss contributor and architect, his current project is RESTEasy, RESTful Web Services for Java.

INTRODUCTORY   INTERMEDIATE   ADVANCED

Previous programming experience strongly recommended

O'REILLY®
oreilly.com

US $39.99   CAN $49.99
ISBN: 978-0-596-15804-0

53999

9 780596 158040

Safari
Books Online

Free online edition
for 45 days with purchase of this book. Details on last page.

RESTful Java with JAX-RS

RESTful Java with JAX-RS

Bill Burke

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Sebastopol · Taipei · Tokyo

RESTful Java with JAX-RS

by Bill Burke

Copyright © 2010 William J. Burke, Jr. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *RESTful Java with JAX-RS*, the image of an Australian bee-eater, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Table of Contents

Part I. REST and the JAX-RS Standard

Part II. JAX-RS Workbook

Foreword

REST is an architectural style that defines a set of constraints that, when applied to the architecture of a distributed system, induce desirable properties like loose coupling and horizontal scalability. RESTful web services are the result of applying these constraints to services that utilize web standards such as URIs, HTTP, XML, and JSON. Such services become part of the fabric of the Web and can take advantage of years of web engineering to satisfy their clients' needs.

The Java API for RESTful web services (JAX-RS) is a new API that aims to make development of RESTful web services in Java simple and intuitive. The initial impetus for the API came from the observation that existing Java Web APIs were generally either:

- Very low-level, leaving the developer to do a lot of repetitive and error-prone work such as URI parsing and content negotiation, or
- Rather high-level and proscriptive, making it easy to build services that conform to a particular pattern but lacking the necessary flexibility to tackle more general problems.

A Java Specification Request (JSR 311) was filed with the Java Community Process (JCP) in January 2007 and approved unanimously in February. The expert group began work in April 2007 with the charter to design an API that was flexible, easy to use, and that encouraged developers to follow the REST style. The resulting API, finalized in October 2008, has already seen a remarkable level of adoption, and we were fortunate to have multiple implementations of the API underway throughout the development of JAX-RS. The combination of implementation experience and feedback from users of those implementations was invaluable and allowed us to refine the specification, clarify edge-cases, and reduce API friction.

JAX-RS is one of the latest generations of Java APIs that make use of Java annotations to reduce the need for standard base classes, implementing required interfaces, and out-of-band configuration files. Annotations are used to route client requests to matching Java class methods and declaratively map request data to the parameters of those methods. Annotations are also used to provide static metadata to create responses. JAX-RS also provides more traditional classes and interfaces for dynamic access to request data and for customizing responses.

Bill Burke led the development of one of the JAX-RS implementations mentioned earlier (RESTEasy) and was an active and attentive member of the expert group. His contributions to expert group discussions are too numerous to list, but a few of the areas where his input was instrumental include rules for annotation inheritance, use of regular expressions for matching request URIs, annotation-driven support for cookies and form data, and support for streamed output.

This book, *RESTful Java with JAX-RS*, provides an in-depth tutorial on JAX-RS and shows how to get the most from this new API while adhering to the REST architectural style. I hope you enjoy the book and working with JAX-RS.

—Marc Hadley
JAX-RS Specification Lead
Sun Microsystems, Inc.
Burlington, MA

Preface

Author's Note

The bulk of my career has been spent working with and implementing distributed middleware. In the mid-'90s I worked for the parent company of Open Environment Corporation, working on DCE tools. Later on, I worked for Iona, developing their next-generation CORBA ORB. Currently, I work for the JBoss division of Red Hat, which is entrenched in Java middleware, specifically Java EE. So, you could say that I have a pretty rich perspective when it comes to middleware.

I must tell you that I was originally very skeptical of REST as a way of writing SOA applications. It seemed way too simple and shortsighted, so I sort of blew it off for a while. One day though, back in mid-2007, I ran into my old Iona boss and mentor Steve Vinoski while grabbing a sandwich at D'Angelo in Westford, MA near Red Hat's offices. We ended up sitting down, having lunch, and talking for hours. The first shocker for me was that Steve had left Iona to go work for a start-up. The second was when he said, "Bill, I've abandoned CORBA and WS-* for REST." For those of you who don't know Steve, he contributed heavily to the CORBA specification, wrote a book on the subject (which is basically the CORBA bible), and is a giant in the distributed computing field, writing regularly for C++ Report and IEEE. How could the guy I looked up to and was responsible for my foundation in distributed computing abandon CORBA, WS-*, and the distributed framework landscape he was instrumental in creating? I felt a little betrayed and very unnerved (OK, maybe I'm exaggerating a little...).

We ended up arguing for a few hours on which was better, WS-*/CORBA or REST. This conversation spilled into many other lengthy email messages, with me trying to promote WS-* and him defending REST. The funniest thing to me was that as I researched REST more and more I found that my arguments with Steve were just another endless replay of debates that had been raging across the blogosphere for years. They are still raging to this day.

Anyway, it took months for me to change my mind and embrace REST. You would figure that my distributed computing background was an asset, but it was not. DCE, CORBA, WS-*, and Java EE were all baggage. All were an invisible barrier for me to accept REST as a viable (and better) alternative for writing SOA applications. I think

that's what I liked most about REST. It required me to rethink and reformulate the foundation of my distributed computing knowledge. Hopefully your journey isn't as difficult as mine and you will be a little less stubborn and more open-minded than I was.

Who Should Read This Book

This book teaches you how to design and develop distributed web services in Java using RESTful architectural principles on top of the HTTP protocol. It is mostly a comprehensive reference guide on the JAX-RS specification, which is a JCP standardized annotation framework for writing RESTful web services in Java.

While this book does go into many of the fundamentals of REST, it does not cover them all and focuses more on implementation rather than theory. You can satisfy your craving for more RESTful theory be obtaining *RESTful Web Services* by Leonard Richardson and Sam Ruby (O'Reilly). If you are familiar writing Java EE applications, you will be very comfortable reading this book. If you are not, you will be at a disadvantage, but some experience with web application development, HTTP, and XML is a huge plus. Before reading this book, you should also be fairly fluent in the Java language and specifically know how to use and apply Java annotations. If you are unfamiliar with the Java language, I recommend *Learning Java* by Patrick Niemeyer and Jonathan Knudsen (O'Reilly).

How This Book Is Organized

This book is organized into two parts: the technical manuscript, followed by the JAX-RS workbook. The technical manuscript explains what REST and JAX-RS are, how they work, and when to use them. The JAX-RS workbook provides step-by-step instructions for installing, configuring, and running the JAX-RS examples from the manuscript with the JBoss RESTEasy framework, an implementation of JAX-RS.

Part I, REST and the JAX-RS Standard

Part I starts off with a brief introduction to REST and HTTP. It then guides you through the basics of the JAX-RS specification, and then in later chapters shows you how you can apply JAX-RS to build RESTful web services:

Chapter 1, *Introduction to REST*
> This chapter gives you a brief introduction to REST and HTTP.

Chapter 2, *Designing RESTful Services*
> This chapter walks you through the design of a distributed RESTful interface for an e-commerce order entry system.

Chapter 3, *Your First JAX-RS Service*

This chapter walks you through the development of a RESTful web service written in Java as a JAX-RS service.

Chapter 4, *HTTP Method and URI Matching*

This chapter defines how HTTP requests are dispatched in JAX-RS and how you can use the `@Path` annotation and subresources.

Chapter 5, *JAX-RS Injection*

This chapter walks you through various annotations that allow you to extract information from an HTTP request (URI parameters, headers, query parameters, form data, cookies, matrix parameters, encoding, and defining default values).

Chapter 6, *JAX-RS Content Handlers*

This chapter explains how to marshal HTTP message bodies to and from Java objects using built-in handlers or writing your own custom marshallers.

Chapter 7, *Response Codes, Complex Responses, and Exception Handling*

This chapter walks through the JAX-RS Response object and how you use it to return complex responses to your client (ResponseBuilder). It also explains how exception and error handling work in JAX-RS.

Chapter 8, *HTTP Content Negotiation*

This chapter explains how HTTP content negotiation works, its relationship to JAX-RS, and how you can leverage this within RESTful architectures.

Chapter 9, *HATEOAS*

This chapter dives into Hypermedia As The Engine Of Application State and how it relates to JAX-RS (UriInfo and UriBuilder).

Chapter 10, *Scaling JAX-RS Applications*

This chapter explains how you can increase the performance of your services by leveraging HTTP caching protocols. It also shows you how to manage concurrency conflicts in high-volume sites.

Chapter 11, *Deployment and Integration*

This chapter explains how you can deploy and integrate your JAX-RS services within Java Enterprise Edition, servlet containers, EJB, Spring, and JPA.

Chapter 12, *Securing JAX-RS*

This chapter walks you through the most popular mechanisms to perform authentication on the Web. It then shows you how to implement secure applications with JAX-RS.

Chapter 13, *RESTful Java Clients*

This chapter shows you how to write RESTful clients in Java using various libraries and frameworks as an example (java.net.URL, Apache HTTP Client, and RESTEasy).

Chapter 14, *JAX-RS Implementations*

This chapter provides a short review of available JAX-RS implementations.

Part II, JAX-RS Workbook

The JAX-RS workbook shows you how to execute examples from chapters in the book that include at least one significant example. You'll want to read the introduction to the workbook to set up RESTEasy and configure it for the examples. After that, just go to the workbook chapter that matches the chapter you're reading. For example, if you are reading Chapter 3 on writing your first JAX-RS service, use Chapter 16, *Examples for Chapter 3*, of the workbook to develop and run the examples with RESTEasy.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
> Indicates new terms, URLs, filenames, and file extensions

`Constant width`
> Indicates variables, method names, and other code elements, as well as the contents of files

`Constant width bold`
> Highlights new code in an example

`Constant width italic`
> Shows text that should be replaced with user-supplied values

> This icon signifies a tip, suggestion, or general note.

> This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books *does* require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation *does* require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*RESTful Java with JAX-RS* by Bill Burke. Copyright 2010 William J. Burke, Jr., 978-0-596-15804-0."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at *permissions@oreilly.com*.

We'd Like to Hear from You

Every example in this book has been tested, but occasionally you may encounter problems. Mistakes and oversights can occur and we will gratefully receive details of any that you find, as well as any suggestions you would like to make for future editions. You can contact the authors and editors at:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596158040

To comment or ask technical questions about this book, send email to the following, quoting the book's ISBN number (9780596158040):

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://www.oreilly.com

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

Acknowledgments

REST and the JAX-RS Standard

Introduction to REST

For those of us with computers, the World Wide Web is an intricate part of our lives. We use it to read the newspaper in the morning, pay our bills, perform stock trades, and buy goods and services, all through the browser, all over the network. "Googling" has become a part of our daily vocabulary as we use search engines to do research for school, find what time a movie is playing, or to just search for information on old friends. Door-to-door encyclopedia salesmen have gone the way of the dinosaur as Wikipedia has become the summarized source of human knowledge. People even socialize over the network using sites like Facebook and MySpace. Professional social networks are sprouting up in all industries as doctors, lawyers and all sorts of professionals use them to collaborate. As programmers, the Web is an intricate part of our daily jobs. We search for and download open source libraries to help us develop applications and frameworks for our companies. We build web-enabled applications so that anybody on the Internet or intranet can use a browser to interact with our systems.

Really, most of us take the Web for granted. Have you, as a programmer, sat down and tried to understand why the Web has been so successful? How has it grown from a simple network of researchers and academics to an interconnected worldwide community? What properties of the Web make it so viral?

One man, Roy Fielding, did ask these questions in his doctoral thesis, "Architectural Styles and the Design of Network-based Software Architectures."[*] In it, he identifies specific architectural principles that answer the following questions:

- Why is the Web so prevalent and ubiquitous?
- What makes the Web scale?
- How can I apply the architecture of the Web to my own applications?

The set of these architectural principles is called REpresentational State Transfer (REST) and are defined as:

---

[*] *www.ics.uci.edu/~fielding/pubs/dissertation/top.htm*

Addressable resources
> The key abstraction of information and data in REST is a resource, and each resource must be addressable via a URI (Uniform Resource Identifier).

A uniform, constrained interface
> Use a small set of well-defined methods to manipulate your resources.

Representation-oriented
> You interact with services using representations of that service. A resource referenced by one URI can have different formats. Different platforms need different formats. For example, browsers need HTML, JavaScript needs JSON (JavaScript Object Notation), and a Java application may need XML.

Communicate statelessly
> Stateless applications are easier to scale.

Hypermedia As The Engine Of Application State (HATEOAS)
> Let your data formats drive state transitions in your applications.

For a PhD thesis, Roy's paper is actually very readable and, thankfully, not very long. It, along with Leonard Richardson and Sam Ruby's book *RESTful Web Services* (O'Reilly), is an excellent reference for understanding REST. I will give a much briefer introduction to REST and the Internet protocol it uses (HTTP) within this chapter.

REST and the Rebirth of HTTP

REST isn't protocol-specific, but when people talk about REST, they usually mean REST over HTTP. Learning about REST was as much of a rediscovery and reappreciation of the HTTP protocol for me as learning a new style of distributed application development. Browser-based web applications see only a tiny fraction of the features of HTTP. Non-RESTful technologies like SOAP and WS-* use HTTP strictly as a transport protocol and thus use a very small subset of its capabilities. Many would say that SOAP and WS-* use HTTP solely to tunnel through firewalls. HTTP is actually a very rich application protocol that provides a multitude of interesting and useful capabilities for application developers. You will need a good understanding of HTTP in order to write RESTful web services.

HTTP is a synchronous request/response-based application network protocol used for distributed, collaborative, document-based systems. It is the primary protocol used on the Web, in particular, by browsers such as Firefox, MS Internet Explorer, Safari, and Netscape. The protocol is very simple: the client sends a request message made up of the HTTP method being invoked, the location of the resource you are interested in invoking, a variable set of headers, and an optional message body that can basically be anything you want, including HTML, plain text, XML, JSON, even binary data. Here's an example:

```
GET /resteasy/index.html HTTP/1.1
Host: jboss.org
```

```
User-Agent: Mozilla/5.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language:      en-us,en;q=0.5
Accept-Encoding:      gzip,deflate
```

Your browser would send this request if you wanted to look at *http://jboss.org/resteasy/ index.html*. GET is the method we are invoking on the server. `/resteasy/index.html` is the object we are interested in. HTTP/1.1 is the version of the protocol. `Host`, `User-Agent`, `Accept`, `Accept-Language`, and `Accept-Encoding` are all message headers. There is no request body, as we are querying information from the server.

The response message from the server is very similar. It contains the version of HTTP we are using, a response code, a short message that explains the response code, a variable set of optional headers, and an optional message body. Here's the message the server might respond with using the previous GET query:

```
HTTP/1.1 200 OK
X-Powered-By: Servlet 2.4; JBoss-4.2.2.GA
Content-Type: text/html

<head>
<title>JBoss RESTEasy Project</title>
</head>
<body>
<h1>JBoss RESTEasy</h1>
<p>JBoss RESTEasy is an open source implementation of the JAX-RS specification...
```

The response code of this message is 200, and the status message is "OK." This code means that the request was processed successfully and that the client is receiving the information it requested. HTTP has a large set of response codes. They can be informational codes like 200, "OK," or error codes like 500, "Internal Server Error." Visit *www.w3.org/Protocols/rfc2616/rfc2616-sec10.html* for a more complete and verbose listing of these codes.

This response message also has a message body that is a chunk of HTML. We know it is HTML by the `Content-Type` header.

RESTful Architectural Principles

Roy Fielding's PhD thesis describing REST was really an explanation of why the human readable Web had become so pervasive in the past 18 years. As time went on though, programmers started to realize that they could use the concepts of REST to build distributed services and model service-oriented architectures (SOAs).

The idea of SOA is that application developers design their systems as a set of reusable, decoupled, distributed services. Since these services are published on the network, conceptually, it should be easier to compose larger and more complex systems. SOA has been around for a long time. Developers have used technologies like DCE, CORBA,

and Java RMI to build them in the past. Nowadays, though, when you think of SOA, you think of SOAP-based web services.

While REST has many similarities to the more traditional ways of writing SOA applications, it is very different in many important ways. You would think that a background in distributed computing would be an asset to understanding this new way of creating web services, but unfortunately this is not always the case. The reason is that some of the concepts of REST are hard to swallow, especially if you have written successful SOAP or CORBA applications. If your career has a foundation in one of these older technologies, there's a bit of emotional baggage you will have to overcome. For me, it took a few months of reading, researching, and intense arguing with REST evangelists (aka RESTafarians). For you, it may be easier. Others will never pick REST over something like SOAP and WS-*.

Let's examine each of the architectural principles of REST in detail and why they are important when writing a web service.

Addressability

Addressability is the idea that every object and resource in your system is reachable through a unique identifier. This seems like a no-brainer, but if you think about it, standardized object identity isn't available in many environments. If you have tried to implement a portable J2EE application, you probably know what I mean. In J2EE, distributed and even local references to services are not standardized, which makes portability really difficult. This isn't such a big deal for one application, but with the new popularity of SOA, we're heading to a world where disparate applications must integrate and interact. Not having something as simple as standardized service addressability adds a whole complex dimension to integration efforts.

In the REST world, addressability is managed through the use of URIs. When you make a request for information in your browser, you are typing in a URI. Each HTTP request must contain the URI of the object you are requesting information from or posting information to. The format of a URI is standardized as follows:

```
scheme://host:port/path?queryString#fragment
```

The `scheme` is the protocol you are using to communicate with. For RESTful web services, it is usually `http` or `https`. The `host` is a DNS name or IP address. It is followed by an optional `port`, which is numeric. The `host` and `port` represent the location of your resource on the network. Following `host` and `port` is a `path` expression. This `path` expression is a set of text segments delimited by the "/" character. Think of the `path` expression as a directory list of a file on your machine. Following the path expression is an optional query string. The "?" character separates the path from the query string. The query string is a list of parameters represented as name/value pairs. Each pair is delimited with the "&" character. Here's an example query string within a URI:

```
http://example.com/customers?lastName=Burke&zipcode=02115
```

A specific parameter name can be repeated in the query string. In this case, there are multiple values for the same parameter.

The last part of the URI is the `fragment`. It is delimited by a "#" character. The fragment is usually used to point to a certain place in the document you are querying.

Not all characters are allowed within a URI string. Some characters must be encoded using the following rules. The characters a–z, A–Z, 0–9, ., -, *, and _ remain the same. The space character is converted to +. The other characters are first converted into a sequence of bytes using specific encoding scheme. Next, a two-digit hexadecimal number prefixed by % represents each byte.

Using a unique URI to identify each of your services makes each of your resources linkable. Service references can be embedded in documents or even email messages. For instance, consider the situation where somebody calls your company's help desk with a problem related to your SOA application. A link could represent the exact problem the user is having. Customer support can email the link to a developer who can fix the problem. The developer can reproduce the problem by clicking on the link. Furthermore, the data that services publish can also be composed into larger data streams fairly easily:

```
<order id="111">
   <customer>http://customers.myintranet.com/customers/32133</customer>
   <order-entries>
     <order-entry>
        <quantity>5</quantity>
        <product>http://products.myintranet.com/products/111</product>
...
```

In this example, an XML document describes an e-commerce order entry. We can reference data provided by different divisions in a company. From this reference, we can not only obtain information about the linked customer and products that were bought, but we also have the identifier of the service this data comes from. We know exactly where we can further interact and manipulate this data if we so desired.

The Uniform, Constrained Interface

The REST principle of a constrained interface is perhaps the hardest pill for an experienced CORBA or SOAP developer to swallow. The idea behind it is that you stick to the finite set of operations of the application protocol you're distributing your services upon. This means that you don't have an "action" parameter in your URI and use only the methods of HTTP for your web services. HTTP has a small, fixed set of operational methods. Each method has a specific purpose and meaning. Let's review them:

GET

> GET is a read-only operation. It is used to query the server for specific information. It is both an *idempotent* and *safe* operation. Idempotent means that no matter how many times you apply the operation, the result is always the same. The act of reading an HTML document shouldn't change the document. Safe means that invoking a GET does not change the state of the server at all. This means that, other than request load, the operation will not affect the server.

PUT

> PUT requests that the server store the message body sent with the request under the location provided in the HTTP message. It is usually modeled as an insert or update. It is also idempotent. When using PUT, the client knows the identity of the resource it is creating or updating. It is idempotent because sending the same PUT message more than once has no effect on the underlying service. An analogy is an MS Word document that you are editing. No matter how many times you click the Save button, the file that stores your document will logically be the same document.

DELETE

> DELETE is used to remove resources. It is idempotent as well.

POST

> POST is the only nonidempotent and unsafe operation of HTTP. Each POST method is allowed to modify the service in a unique way. You may or may not send information with the request. You may or may not receive information from the response.

HEAD

> HEAD is exactly like GET except that instead of returning a response body, it returns only a response code and any headers associated with the request.

OPTIONS

> OPTIONS is used to request information about the communication options of the resource you are interested in. It allows the client to determine the capabilities of a server and a resource without triggering any resource action or retrieval.

There are other HTTP methods (like TRACE and CONNECT), but they are unimportant when designing and implementing RESTful web services.

You may be scratching your head and thinking, "How is it possible to write a distributed service with only four to six methods?" Well...SQL only has four operations: SELECT, INSERT, UPDATE, and DELETE. JMS and other Message Oriented Middleware (MOM) really only have two logical operations: *send* and *receive*. How powerful are these tools? For both SQL and JMS, the complexity of the interaction is confined purely to the data model. The addressability and operations are well defined and finite and the hard stuff is delegated to the data model (in the case of SQL) or the message body (in the case of JMS).

Why Is the Uniform Interface Important?

Constraining the interface for your web services has many more advantages than disadvantages. Let's look at a few:

Familiarity

If you have a URI that points to a service, you know exactly which methods are available on that resource. You don't need an IDL-like file describing which methods are available. You don't need stubs. All you need is an HTTP client library. If you have a document that is composed of links to data provided by many different services, you already know which method to call to pull in data from those links.

Interoperability

HTTP is a very ubiquitous protocol. Most programming languages have an HTTP client library available to them. So, if your web service is exposed over HTTP, there is a very high probability that people who want to use your service will be able to do so without any additional requirements beyond being able to exchange the data formats the service is expecting. With CORBA or WS-*, you have to install vendor-specific client libraries as well as loads and loads of IDL or WSDL-generated stub code. How many of you have had a problem getting CORBA or WS-* vendors to interoperate? It has traditionally been very problematic. The WS-* set of specifications has also been a moving target over the years. So with WS-* and CORBA, you not only have to worry about vendor interoperability, you also have to make sure that your client and server are using the same specification version of the protocol. With REST over HTTP, you don't have to worry about either of these things and can just focus on understanding the data format of the service. I like to think that you are focusing on what is really important: *application interoperability*, rather than *vendor interoperability*.

Scalability

Because REST constrains you to a well-defined set of methods, you have predictable behavior that can have incredible performance benefits. GET is the strongest example. When surfing the Internet, have you noticed that the second time you browse to a specific page it comes up faster? This is because your browser caches already visited pages and images. HTTP has a fairly rich and configurable protocol for defining caching semantics. Because GET is a read method that is both idempotent and safe, browsers and HTTP proxies can cache responses to servers, and this can save a huge amount of network traffic and hits to your website. Add HTTP caching semantics to your web services and you have an incredibly rich way of defining caching policies for your services. We will discuss HTTP caching in detail within Chapter 10.

It doesn't end with caching, though. Consider both PUT and DELETE. Because they are idempotent, neither the client nor the server has to worry about handling duplicate message delivery. This saves a lot of bookkeeping and complex code.

Representation-Oriented

The third architectural principle of REST is that your services should be representation-oriented. Each service is addressable through a specific URI and representations are exchanged between the client and service. With a GET operation, you are receiving a representation of the current state of that resource. A PUT or POST passes a representation of the resource to the server so that the underlying resource's state can change.

In a RESTful system, the complexity of the client-server interaction is within the representations being passed back and forth. These representations could be XML, JSON, YAML, or really any format you can come up with.

With HTTP, the representation is the message body of your request or response. An HTTP message body may be in any format the server and client want to exchange. HTTP uses the `Content-Type` header to tell the client or server what data format it is receiving. The `Content-Type` header value string is in the Multipurpose Internet Mail Extension (MIME) format. The MIME format is very simple:

```
type/subtype;name=value;name=value...
```

`type` is the main format family and `subtype` is a category. Optionally, the MIME type can have a set of name/value pair properties delimited by the ";" character. Some examples are:

```
text/plain
text/html
application/xml
text/html; charset=iso-8859-1
```

One of the more interesting features of HTTP that leverages MIME types is the capability of the client and server to negotiate the message formats being exchanged between them. While not used very much by your browser, HTTP content negotiation is a very powerful tool when writing web services. With the `Accept` header, a client can list its preferred response formats. Ajax clients can ask for JSON, Java for XML, Ruby for YAML. Another thing this is very useful for is versioning of services. The same service can be available through the same URI with the same methods (GET, POST, etc.), and all that changes is the MIME type. For example, the MIME type could be `application/vnd+xml` for an old service while newer services could exchange `application/vnd+xml;version=1.1` MIME types. You can read more about all these concepts in Chapter 8.

All in all, because REST and HTTP have a layered approach to addressability, method choice, and data format, you have a much more decoupled protocol that allows your service to interact with a wide variety of different clients in a consistent way.

Communicate Statelessly

The fourth RESTful principle I will discuss is the idea of statelessness. When I talk about statelessness, though, I don't mean that your applications can't have state. In

REST, stateless means that there is no client session data stored on the server. The server only records and manages the state of the resources it exposes. If there needs to be session-specific data, it should be held and maintained by the client and transferred to the server with each request as needed. A service layer that does not have to maintain client sessions is a lot easier to scale, as it has to do a lot fewer expensive replications in a clustered environment. It's a lot easier to scale up, because all you have to do is add machines.

A world without server-maintained session data isn't so hard to imagine if you look back 12–15 years ago. Back then, many distributed applications had a fat GUI client written in Visual Basic, Power Builder, or Visual C++ talking RPCs to a middle tier that sat in front of a database. The server was stateless and just processed data. The fat client held all session state. The problem with this architecture was an IT operations one. It was very hard for operations to upgrade, patch, and maintain client GUIs in large environments. Web applications solved this problem because the applications could be delivered from a central server and rendered by the browser. We started maintaining client sessions on the server because of the limitations of the browser. Now, circa 2008, with the growing popularity of Ajax, Flex, and Java FX, the browsers are sophisticated enough to maintain their own session state like their fat-client counterparts in the mid-'90s used to do. We can now go back to that stateless scalable middle tier that we enjoyed in the past. It's funny how things go full circle sometimes.

HATEOAS

The final principle of REST is the idea of using Hypermedia As The Engine Of Application State (HATEOAS). Hypermedia is a document-centric approach with the added support for embedding links to other services and information within that document format. I did indirectly talk about HATEOAS in "Addressability" on page 6 when I discussed the idea of using hyperlinks within the data format received from a service.

One of the uses of hypermedia and hyperlinks is composing complex sets of information from disparate sources. The information could be within a company intranet or dispersed across the Internet. Hyperlinks allow us to reference and aggregate additional data without bloating our responses. The e-commerce order in "Addressability" on page 6 is an example of this:

```
<order id="111">
  <customer>http://customers.myintranet.com/customers/32133</customer>
  <order-entries>
    <order-entry>
      <quantity>5</quantity>
      <product>http://products.myintranet.com/products/111</product>
...
```

In that example, links embedded within the document allowed us to bring in additional information as needed. Aggregation isn't the full concept of HATEOAS, though. The more interesting part of HATEOAS is the "engine."

The engine of application state

If you're on Amazon.com buying a book, you follow a series of links and fill out one or two forms before your credit card is charged. You transition through the ordering process by examining and interacting with the responses returned by each link you follow and each form you submit. The server guides you through the order process by embedding where you should go next within the HTML data it provides your browser.

This is very different from the way traditional distributed applications work. Older applications usually have a list of precanned services they know exist and interact with a central directory server to locate these services on the network. HATEOAS is a bit different because with each request returned from a server it tells you what new interactions you can do next, as well as where to go to transition the state of your applications.

For example, let's say we wanted to get a list of products available on a web store. We do an HTTP GET on *http://example.com/webstore/products* and receive back:

```
<products>
  <product id="123">
      <name>headphones</name>
      <price>$16.99</price>
  </product>
  <product id="124">
      <name>USB Cable</name>
    <price>$5.99</price>
  </product>
...
</products>
```

This could be problematic if we had thousands of products to send back to our client. We might overload it, or the client might wait forever for the response to finish downloading. We could instead list only the first five products and provide a link to get the next set:

```
<products>
    <link rel="next" href="http://example.com/webstore/products?startIndex=5"/>
    <product id="123">
        <name>headphones</name>
        <price>$16.99</price>
    </product>
...
</products>
```

When first querying for a list of products, clients don't have to know they're only getting back a list of five products. The data format can tell them that they didn't get a full set and to get the `next` set follow a specific link. Following the `next` link could get them back a new document with additional links:

```
<products>
    <link rel="previous" href="http://example.com/webstore/products?startIndex=1"/>
    <link rel="next" href="http://example.com/webstore/products?startIndex=5"/>
    <product id="128">
        <name>stuff</name>
        <price>$16.99</price>
    </product>
...
</products>
```

In this case, there is the additional state transition of `previous` so that clients can browse an earlier part of the product list. The `next` and `previous` links seem a bit trivial, but imagine if we had other transition types like `payment`, `inventory`, or `sales`.

This sort of approach gives the server a lot of flexibility, as it can change where and how state transitions happen on the fly. It could provide new and interesting opportunities to surf to. In Chapter 9, we'll dive into HATEOAS again.

Wrapping Up

REST identifies the key architectural principles of why the Web is so prevalent and scalable. The next step in the evolution of the Web is to apply these principles to the semantic Web and the world of web services. REST offers a simple, interoperable, and flexible way of writing web services that can be very different than the RPC mechanisms like CORBA and WS-* that so many of us have had training in. In the next chapter we will apply the concepts of REST by defining a distributed RESTful interface to an existing business object model.

Designing RESTful Services

In Chapter 1, I gave you a brief overview of REST and how it relates to HTTP. Although it is good to obtain a solid foundation in theory, nothing can take the place of seeing theory be put into practice. So, let's define a RESTful interface for a simple order entry system of a hypothetical e-commerce web store. Remote distributed clients will use this web service to purchase goods, modify existing orders in the system, and view information about customers and products.

In this chapter, we will start off by examining the simple underlying object model of our service. After walking through the model, we will add a distributed interface to our system using HTTP and the architectural guidelines of REST. To satisfy the addressability requirements of REST, we will first have to define a set of URIs that represent the entry points into our system. Since RESTful systems are representation-oriented, we will next define the data format that we will use to exchange information between our services and clients. Finally, we will decide which HTTP methods are allowed by each exposed URI and what those methods do. We will make sure to conform to the uniform, constrained interface of HTTP when doing this step.

The Object Model

The object model of our order entry system is very simple. Each order in the system represents a single transaction or purchase and is associated with a particular customer. Orders are made up of one or more line items. Line items represent the type and number of each product purchased.

Based on this description of our system, we can deduce that the objects in our model are `Order`, `Customer`, `LineItem`, and `Product`. Each data object in our model has a unique identifier, which is the integer `id` property. Figure 2-1 shows a UML diagram of our object model.

We will want to browse all orders as well as each individual order in our system. We will also want to submit new orders and update existing ones. Finally, we will want to have the ability to cancel and delete existing orders. The `OrderEntryService` object
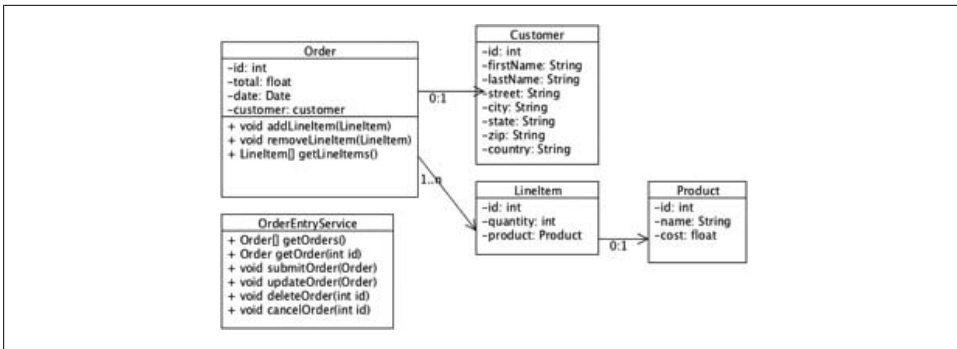
Figure 2-1. Order Entry System Object Model

represents the operations we want to perform on our `Order`, `Customer`, `LineItem`, and `Product` objects.

Model the URIs

The first thing we are going to do to create our distributed interface is define and name each of the distributed endpoints in our system. In a RESTful system, endpoints are usually referred to as *resources* and are identified using a URI. URIs satisfy the addressability requirements of a RESTful service.

In our object model, we will be interacting with `Orders`, `Customers`, and `Products`. These will be our main, top-level resources. We want to be able to obtain lists of each of these top-level items and to interact with individual items. `LineItems` are aggregated within `Order` objects so they will not be a top-level resource. We could expose them as a sub-resource under one particular `Order`, but for now, let's assume they are hidden by the data format. Knowing this, here is a list of URIs that will be exposed in our system:

```
/orders
/orders/{id}
/products
/products/{id}
/customers
/customers/{id}
```

The `/orders` URI represents all the orders in our system. To access an individual order, we will use a pattern: `/orders/{id}`. The {id} string is a pattern that represents an individual order. It can be any integer value that corresponds to the `id` property of our `Order` object. `Products` and `Customers` interactions will also be modeled in the same way.

> You'll notice that the nouns in our object model have been represented as URIs. URIs shouldn't be used as mini-RPC mechanisms and should not identify operations. Instead, you should use a combination of HTTP methods and the data format to model unique operations in your distributed RESTful system.

Defining the Data Format

One of the most important things we have to do when defining a RESTful interface is determine how our resources will be represented over the wire to our clients. XML is perhaps one of the most popular formats on the Web and can be processed by most modern languages, so let's choose that. JSON is also a popular format, as it is more condensed and JavaScript can interpret it directly (great for Ajax applications), but let's stick to XML for now.

Generally, you would define an XML schema for each representation you want to send across the wire. An XML schema defines the grammar of a data format. It defines the rules about how a document can be put together. I do find, though, that when explaining things within an article (or a book), providing examples rather than schema makes things much easier to read and understand.

Read and Update Format

The XML format of our representations will look a tiny bit different from when we read or update resources from the server than when we create resources on the server. Let's look at our read and update format first.

Common link element

Each format for `Order`, `Customer`, and `Product` will have a common XML element called `link`:

```
<link rel="self" href="http://example.com/..."/>
```

The `link`* element tells any client that obtains an XML document describing one of the objects in our e-commerce system where on the network the client can interact with that particular resource. The `rel` attribute tells the client what relationship the link has with the resource the URI points to (contained within the `href` attribute). The `self` value just means it is pointing to itself. While not that interesting on its own, `link` becomes very useful when we aggregate or compose information into one larger XML document.

The details

So, with the common elements described, let's start diving into the details by first looking at our `Customer` representation format:

```
<customer id="771">
   <link rel="self" href="http://example.com/customers/771"/>
   <first-name>Bill</first-name>
   <last-name>Burke</last-name>
```

---

* I actually borrowed the `link` element from the Atom format. Atom is a syndication format that is used to aggregate and publish blogs and news feeds. You can find out more about Atom at *www3.org/2005/Atom*.

```
            <street>555 Beacon St.<street>
            <city>Boston</city>
            <state>MA</state>
            <zip>02115</zip>
        </customer>
```

Pretty straightforward. We just take the object model of `Customer` from Figure 2-1 and expand its attributes as XML elements. `Product` looks much the same in terms of simplicity:

```
        <product id="543">
            <link rel="self" href="http://example.com/products/543"/>
            <name>iPhone</name>
            <cost>$199.99</cost>
        </product>
```

In a real system, we would, of course, have a lot more attributes for `Customer` and `Product`, but let's keep our example simple so that it's easier to illustrate these RESTful concepts:

```
        <order id="133">
            <link rel="self" href="http://example.com/orders/133"/>
            <total>$199.02</total>
            <date>December 22, 2008 06:56</date>
            <customer id="117">
                <link rel="self" href="http://example.com/customers/117"/>
                <first-name>Bill</first-name>
                <last-name>Burke</last-name>
                <street>555 Beacon St.<street>
                <city>Boston</city>
                <state>MA</state>
                <zip>02115</zip>
            </customer>
            <line-items>
                <line-item id="144">
                    <product id="543">
                        <link rel="self" href="http://example.com/products/543"/>
                        <name>iPhone</name>
                        <cost>$199.99</cost>
                    </product>
                    <quantity>1</quantity>
                </line-item>
            </line-items>
        </order>
```

The `Order` data format has the top-level elements of `total` and `date` that specify the total cost of the order and the date the `Order` was made. `Order` is a great example of data composition as it includes `Customer` and `Product` information. This is where the `link` element becomes particularly useful. If the client is interested in interacting with a `Customer` or `Product` that makes up the `Order`, it has the URI needed to interact with one of these resources.

Create Format

When we are creating new `Order`s, `Customer`s, or `Product`s, it doesn't make a lot of sense to include an `id` attribute and `link` element with our XML document. The server will generate IDs when it inserts our new object into a database. We also don't know the URI of a new object, because the server also generates this. So, the XML for creating a new Product would look something like this:

```
<product>
    <name>iPhone</name>
    <cost>$199.99</cost>
</product>
```

Orders and Customers would follow the same pattern and leave out the `id` attribute and `link` element.

Assigning HTTP Methods

The final thing we have to do is decide which HTTP methods will be exposed for each of our resources and what these methods will do. It is crucial that we do not assign functionality to an HTTP method that supersedes the specification-defined boundaries of that method. For example, an HTTP GET on a particular resource should be read-only. It should not change the state of the resource it is invoking on. Intermediate services like a proxy-cache, a CDN (Akamai), or your browser rely on you to follow the semantics of HTTP strictly so that they can perform built-in tasks like caching effectively. If you do not follow the definition of each HTTP method strictly, clients and administration tools cannot make assumptions about your services and your system becomes more complex.

Let's walk through each method of our object model to determine which URIs and HTTP methods are used to represent them.

Browsing All Orders, Customers, or Products

The `Order`, `Customer`, and `Product` objects in our object model are all very similar in how they are accessed and manipulated. One thing our remote clients will want to do is to browse all the `Order`s, `Customer`s, or `Product`s in the system. These URIs represent these objects as a group:

```
/orders
/products
/customers
```

To get a list of `Order`s, `Product`s, or `Customer`s, the remote client will call an HTTP GET on the URI of the object group it is interested in. An example request would look like the following:

```
GET /products HTTP/1.1
```

Our service will respond with a data format that represents all `Orders`, `Products`, or `Customers` within our system. Here's what a response would look like:

```
HTTP/1.1 200 OK
Content-Type: application/xml

<products>
    <product id="111">
        <link rel="self" href="http://example.com/products/111"/>
        <name>iPhone</name>
        <cost>$199.99</cost>
    </product>
    <product id="222">
        <link rel="self" href="http://example.com/products/222"/>
        <name>Macbook</name>
        <cost>$1599.99</cost>
    </product>
...
</products>
```

One problem with this bulk operation is that we may have thousands of `Orders`, `Customers`, or `Products` in our system and we may overload our client and hurt our response times. To mitigate this problem, we will allow the client to specify query parameters on the URI to limit the size of the dataset returned:

```
GET /orders?startIndex=0&size=5 HTTP/1.1
GET /products?startIndex=0&size=5 HTTP/1.1
GET /customers?startIndex=0&size=5 HTTP/1.1
```

Here we have defined two query parameters: `startIndex` and `size`. The `startIndex` parameter represents where in our large list of Orders, Products, or Customers we want to start sending objects from. It is a numeric index into the object group being queried. The `size` parameter specifies how many of those objects in the list we want to return. These parameters will be optional. The client does not have to specify them in its URI when crafting its request to the server.

Obtaining Individual Orders, Customers, or Products

I mentioned in the previous section that we would use a URI pattern to obtain individual `Orders`, `Customers`, or `Products`:

```
/orders/{id}
/products/{id}
/customers/{id}
```

We will use the HTTP GET method to retrieve individual objects in our system. Each GET invocation will return a data format that represents the object being obtained:

```
GET /orders/232 HTTP/1.1
```

For this request, the client is interested in getting a representation of the `Order` represented by the `232` Order `id`. GET requests for `Products` and `Customers` would work the same. The HTTP response message would look something like this:

```
HTTP/1.1 200 OK
Content-Type: application/xml

<order id="232">...</order>
```

The response code is 200, "OK," indicating that the request was successful. The `Content-Type` header specifies the format of our message body as XML, and finally we have the actual representation of the Order.

Creating an Order, Customer, or Product

There are two possible ways in which a client could create an `Order`, `Customer`, or `Product` within our order entry system: by using either the HTTP PUT or POST method. Let's look at both ways.

Creating with PUT

The HTTP definition of PUT states that it can be used to create or update a resource on the server. To create an `Order`, `Customer`, or `Product` with PUT, the client simply sends a representation of the new object it is creating to the exact URI location that represents the object:

```
PUT /orders/233 HTTP/1.1
PUT /customers/112 HTTP/1.1
PUT /products/664 HTTP/1.1
```

PUT is required by the specification to send a response code of 201, "Created" if a new resource was created on the server as a result of the request.

The HTTP specification also states that PUT is idempotent. Our PUT is idempotent, because no matter how many times we tell the server to "create" our Order, the same bits are stored at the `/orders/233` location. Sometimes a PUT request will fail and the client won't know if the request was delivered and processed at the server. Idempotency guarantees that it's OK for the client to retransmit the PUT operation and not worry about any adverse side effects.

The disadvantage of using PUT to create resources is that the client has to provide the unique ID that represents the object it is creating. While it usually possible for the client to generate this unique ID, most application designers prefer that their servers (usually through their databases) create this ID. In our hypothetical order entry system, we want our server to control the generation of resource IDs. So what do we do? We can switch to using POST instead of PUT.

Creating with POST

Creating an Order, Customer, or Product using the POST method is a little more complex than using PUT. To create an `Order`, `Customer`, or `Product` with POST, the client sends a representation of the new object it is creating to the parent URI of its representation, leaving out the numeric target ID. For example:

```
POST /orders HTTP/1.1
Content-Type: application/xml

<order>
    <total>$199.02</total>
    <date>December 22, 2008 06:56</date>
...
</order>
```

The service receives the POST message, processes the XML, and creates a new order in the database using a database-generated unique ID. While this approach works perfectly fine, we've left our client in a quandary. What if the client wants to edit, update, or cancel the order it just posted? What is the ID of the new order? What URI can we use to interact with the new resource? To resolve this issue, we will add a bit of additional information to the HTTP response message. The client would receive a message something like this:

```
HTTP/1.1 201 Created
Content-Type: application/xml
Location: http://example.com/orders/233

<order id="233">
    <link rel="self" href="http://example.com/orders/133"/>
    <total>$199.02</total>
    <date>December 22, 2008 06:56</date>
...
</order>
```

HTTP requires that if POST creates a new resource that it respond with a code of 201, "Created" (just like PUT). The `Location` header in the response message provides a URI to the client so it knows where to further interact with the `Order` that was created, i.e., if the client wanted to update the `Order`. It is optional whether the server sends the representation of the newly created `Order` with the response. Here, we send back an XML representation of the `Order` that was just created with the ID attribute set to the one generated by our database as well as a link element.

> I didn't pull the `Location` header out of thin air. The beauty of this approach is that it is defined within the HTTP specification. That's an important part of REST—to follow the predefined behavior defined within the specification of the protocol you are using. Because of this, most systems are self-documenting, as the distributed interactions are already mostly defined by the HTTP specification.

Updating an Order, Customer, or Product

We will model updating an `Order`, `Customer`, or `Product` using the HTTP PUT method. The client PUTs a new representation of the object it is updating to the exact URI location that represents the object. For example, let's say we wanted to change the price of a product from $199.99 to $149.99. Here's what the request would look like:

```
PUT /orders/232 HTTP/1.1
Content-Type: application/xml

<product id="111">
    <name>iPhone</name>
    <cost>$149.99</cost>
</product>
```

As I stated earlier in this chapter, PUT is great because it is idempotent. No matter how many times we transmit this PUT request, the underlying Product will still have the same final state.

When a resource is updated with PUT, the HTTP specification requires that you send a response code of 200, "OK" and a response message body or a response code of 204, "No Content" without any response body. In our system, we will send a status of 204 and no response message.

> We could use POST to update an individual `Order`, but then the client would have to assume the update was nonidempotent and we would have to take duplicate message processing into account.

Removing an Order, Customer, or Product

We will model deleting an Order, Customer, or Product using the HTTP DELETE method. The client simply invokes the DELETE method on the exact URI that represents the object we want to remove. Removing an object will wipe its existence from the system.

When a resource is removed with DELETE, the HTTP specification requires that you send a response code of 200, "OK" and a response message body or a response code of 204, "No Content" without any response body. In our application, we will send a status of 204 and no response message.

Cancelling an Order

So far, the operations of our object model have fit quite nicely into corresponding HTTP methods. We're using GET for reading, PUT for updating, POST for creating, and DELETE for removing. We do have an operation in our object model that doesn't fit so nicely. In our system, Orders can be cancelled as well as removed. While removing an object wipes it clean from our databases, cancelling only changes the state of the Order and retains it within the system. How should we model such an operation?

Overloading the meaning of DELETE

Cancelling an `Order` is very similar to removing it. Since we are already modeling remove with the HTTP DELETE method, one thing we could do is add an extra query parameter to the request:

```
DELETE /orders/233?cancel=true
```

Here, the cancel query parameter would tell our service that we don't really want to remove the `Order`, but cancel it. In other words, we are overloading the meaning DELETE.

While I'm not going to tell you not to do this, I will tell you that you shouldn't do it. It is not good RESTful design. In this case, you are changing the meaning of the uniform interface. Using a query parameter in this way is actually creating a mini-RPC mechanism. HTTP specifically states that DELETE is used to delete a resource from the server, not cancel it.

States versus operations

When modeling a RESTful interface for the operations of your object model, you should ask yourself a simple question: is the operation a state of the resource? If you answer yes to this question, the operation should be modeled within the data format.

Cancelling an `Order` is a perfect example of this. The key thing with cancelling is that it is a specific state of an `Order`. When a client follows a particular URI that links to a specific `Order`, the client will want to know whether the `Order` was cancelled or not. Information about the cancellation needs to be in the data format of the `Order`. So let's add a cancelled element to our `Order` data format:

```
<order id="133">
    <link rel="self" href="http://example.com/orders/133"/>
    <total>$199.02</total>
    <date>December 22, 2008 06:56</date>
    <cancelled>false</cancelled>
...
</order>
```

Since the state of being cancelled is modeled in the data format, we can now use our already defined mechanism of updating an `Order` to model the cancel operation. For example, we could PUT this message to our service:

```
PUT /orders/233 HTTP/1.1
Content-Type: application/xml

<order id="233">
    <total>$199.02</total>
    <date>December 22, 2008 06:56</date>
    <cancelled>true</cancelled>
...
</order>
```

In this example, we PUT a new representation of our order with the cancelled element set to true. By doing this, we've changed the state of our order from viable to cancelled.

This pattern of modeling an operation as the state of the resource doesn't always fit, though. What if we expanded on our cancel example by saying that we wanted a way to clean up all cancelled orders? In other words, we want to purge all cancelled orders from our database. We can't really model purging the same way we did cancel. While purge does change the state of our application, it is not in and of itself a state of the application.

To solve this problem, we model this operation as a subresource of `/orders` and we trigger a purging by doing a POST on that resource. For example:

```
POST /orders/purge HTTP/1.1
```

An interesting side effect of this is that because purge is now a URI, we can evolve its interface over time. For example, maybe `GET /orders/purge` returns a document that states the last time a purge was executed and which orders were deleted. What if we wanted to add some criteria for purging as well? Form parameters could be passed stating that we only want to purge orders older than a certain date. In doing this, we're giving ourselves a lot of flexibility as well as honoring the uniform interface contract of REST.

Wrapping Up

So, we've taken an existing object diagram and modeled it as a RESTful distributed service. We used URIs to represent the endpoints in our system. These endpoints are called resources. For each resource, we defined which HTTP methods each resource will allow and how those individual HTTP methods behave. Finally, we defined the data format that our clients and services will use to exchange information. The next step is to actually implement these services in Java. This will be the main topic for the rest of this book.

Your First JAX-RS Service

The first two chapters of this book focused on the theory of REST and designing the RESTful interface for a simple e-commerce order entry system. Now it's time to implement a part of our system in the Java language.

Writing RESTful services in Java has been possible for years with the servlet API. If you have written a web application in Java, you are probably already very familiar with servlets. Servlets bring you very close to the HTTP protocol and require a lot of boilerplate code to move information to and from an HTTP request. In 2008, a new specification called JAX-RS was defined to simplify RESTful service implementation.

JAX-RS is a framework that focuses on applying Java annotations to plain Java objects. It has annotations to bind specific URI patterns and HTTP operations to individual methods of your Java class. It has parameter injection annotations so that you can easily pull in information from the HTTP request. It has message body readers and writers that allow you to decouple data format marshalling and unmarshalling from your Java data objects. It has exception mappers that can map an application-thrown exception to an HTTP response code and message. Finally, it has some nice facilities for HTTP content negotiation.

This chapter gives a brief introduction to writing a JAX-RS service. You'll find that getting up and running is fairly simple.

Developing a JAX-RS RESTful Service

Let's start by implementing one of the resources of the order entry system we defined in Chapter 2. Specifically, we'll define a JAX-RS service that allows us to read, create, and update `Customers`. To do this, we will need to implement two Java classes. One class will be used to represent actual Customers. The other will be our JAX-RS service.

Customer: The Data Class

First, we will need a Java class to represent customers in our system. We will name this class `Customer`. `Customer` is a simple Java class that defines eight properties: `id`, `first Name`, `lastName`, `street`, `city`, `state`, `zip`, and `country`. *Properties* are attributes that can be accessed via the class's fields or through public set and get methods. A Java class that follows this pattern is also called a *Java bean*:

```java
package com.restfully.shop.domain;

public class Customer {
    private int id;
    private String firstName;
    private String lastName;
    private String street;
    private String city;
    private String state;
    private String zip;
    private String country;

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

    public String getFirstName() { return firstName; }
    public void setFirstName(String firstName) {
                    this.firstName = firstName; }

    public String getLastName() { return lastName; }
    public void setLastName(String lastName) {
                         this.lastName = lastName; }

    public String getStreet() { return street; }
    public void setStreet(String street) { this.street = street; }

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

    public String getState() { return state; }
    public void setState(String state) { this.state = state; }

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

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

In an Enterprise Java application, the `Customer` class would usually be a Java Persistence (JPA) Entity bean and would be used to interact with a relational database. It could also be annotated with JAXB annotations that allow you to map a Java class directly to XML. To keep our example simple, `Customer` will be just a plain Java object and stored in memory. In Chapter 6, I'll show how you can use JAXB with JAX-RS to make translating between your customer's data format (XML) and your `Customer` objects easier.

Chapter 11 will show you how JAX-RS works in the context of a Java EE application and things like JPA.

CustomerResource: Our JAX-RS Service

Now that we have defined a domain object that will represent our customers at runtime, we need to implement our JAX-RS service so that remote clients can interact with our customer database. A JAX-RS service is a Java class that uses JAX-RS annotations to bind and map specific incoming HTTP requests to Java methods that can service these requests. While JAX-RS can integrate with popular component models like EJB, Web Beans, JBoss Seam, and Spring, it does define its own lightweight model.

In vanilla JAX-RS, services can either be singletons or per-request objects. A *singleton* means that one and only one Java object services HTTP requests. *Per-request* means that a Java object is created to process each incoming request and is thrown away at the end of that request. Per-request also implies statelessness, as no service state is held between requests.

For our example, we will write a `CustomerResource` class to implement our JAX-RS service and assume it will be a singleton. In this example, we need `CustomerResource` to be a singleton because it is going to hold state. It is going to keep a map of `Customer` objects in memory that our remote clients can access. In a real system, `CustomerResource` would probably interact with a database to retrieve and store customers and wouldn't need to hold state between requests. In this database scenario, we could make `CustomerResource` per-request and thus stateless. Let's start by looking at the first few lines of our class to see how to start writing a JAX-RS service:

```
package com.restfully.shop.services;

import ...;

@Path("/customers")
public class CustomerResource {

    private Map<Integer, Customer> customerDB =
                            new ConcurrentHashMap<Integer, Customer>();
    private AtomicInteger idCounter = new AtomicInteger();
```

As you can see, `CustomerResource` is a plain Java class and doesn't implement any particular JAX-RS interface. The `@javax.ws.rs.Path` annotation placed on the `CustomerResource` class designates the class as a JAX-RS service. Java classes that you want to be recognized as JAX-RS services must have this annotation. Also notice that the `@Path` annotation has the value of `/customers`. This value represents the relative root URI of our customer service. If the absolute base URI of our server is *http://shop.restfully.com*, methods exposed by our `CustomerResource` class would be available under *http://shop.restfully.com/customers*.

In our class, we define a simple map in the `customerDB` field that will store created `Customer` objects in memory. We use a `java.util.concurrent.ConcurrentHashMap` for `customerDB` because `CustomerResource` is a singleton and will have concurrent requests accessing the map. Using a `java.util.HashMap` would trigger concurrent access exceptions in a multithreaded environment. Using a `java.util.Hashtable` creates a synchronization bottleneck. `ConcurrentHashMap` is our best bet. The `idCounter` field will be used to generate IDs for newly created `Customer` objects. For concurrency reasons, we use a `java.util.concurrent.atomic.AtomicInteger`, as we want to always have a unique number generated. Of course, these two lines of code have nothing to do with JAX-RS and are solely artifacts required by our simple example.

Creating customers

Let's now take a look at how to create customers in our `CustomerResource` class:

```
@POST
@Consumes("application/xml")
public Response createCustomer(InputStream is) {
    Customer customer = readCustomer(is);
    customer.setId(idCounter.incrementAndGet());
    customerDB.put(customer.getId(), customer);
    System.out.println("Created customer " + customer.getId());
    return Response.created(URI.create("/customers/"
                                   + customer.getId())).build();
}
```

We will implement customer creation using the same model as that used in Chapter 2. An HTTP POST request sends an XML document representing the customer we want to create. The `createCustomer()` method receives the request, parses the document, creates a `Customer` object from the document, and adds it to our `customerDB` map. The `createCustomer()` method returns a response code of 201, "Created" along with a `Location` header pointing to the absolute URI of the customer we just created. So how does the `createCustomer()` method do all this? Let's examine further.

To bind HTTP POST requests to the `createCustomer()` method, we annotate it with the `@javax.ws.rs.POST` annotation. The `@Path` annotation we put on the `CustomerResource` class, combined with this `@POST` annotation, binds all POST requests going to the relative URI `/customers` to the Java method `createCustomer()`.

The `@javax.ws.rs.Consumes` annotation applied to `createCustomer()` specifies which media type the method is expecting in the message body of the HTTP input request. If the client POSTs a media type other than XML, an error code is sent back to the client.

The `createCustomer()` method takes one `java.io.InputStream` parameter. In JAX-RS, any non-JAX-RS-annotated parameter is considered to be a representation of the HTTP input request's message body. In this case, we want access to the method body in its most basic form, an `InputStream`.

Only one Java method parameter can represent the HTTP message body. This means any other parameters must be annotated with one of the JAX-RS annotations discussed in Chapter 5.

The implementation of the method reads and transforms the POSTed XML into a `Customer` object and stores it in the `customerDB` map. The method returns a complex response to the client using the `javax.ws.rs.core.Response` class. The static `Response.created()` method creates a Response object that contains an HTTP status code of 201, "Created." It also adds a `Location` header to the HTTP response with the value of something like *http://shop.restfully.com/customers/333*, depending on the base URI of the server and the generated ID of the `Customer` object (333 in this example).

Retrieving customers

Now let's look at how we retrieve this information. In Chapter 2, we decided to use `GET /customers/{id}` to retrieve customer representations, where `{id}` represents the customer code. We implement this functionality in the `getCustomer()` method of `CustomerResource`:

```
@GET
@Path("{id}")
@Produces("application/xml")
public StreamingOutput getCustomer(@PathParam("id") int id) {
   final Customer customer = customerDB.get(id);
   if (customer == null) {
      throw new WebApplicationException(
                                  Response.Status.NOT_FOUND);
   }
   return new StreamingOutput() {
      public void write(OutputStream outputStream)
                  throws IOException, WebApplicationException {
         outputCustomer(outputStream, customer);
      }
   };
}
```

We annotate the `getCustomer()` method with the `@javax.ws.rs.GET` annotation to bind HTTP GET operations to this Java method.

We use an additional `@Path` annotation on `getCustomer()` to specify which URI will be bound to the method. The value of this annotation is concatenated with the value of the `@Path` annotation we applied to the `CustomerResource` class. This concatenation defines a URI matching pattern of `/customers/{id}`. `{id}` matches a URI path segment. A path segment is any sequence of characters that is not the "/" character.

We also annotate `getCustomer()` with the `@javax.ws.rs.Produces` annotation. This annotation tells JAX-RS which HTTP `Content-Type` the GET response will be. In this case, it is `application/xml`.

The `getCustomer()` takes one `id` parameter that represents the ID of the `Customer` object we are interested in retrieving. We will use this value later in the method to query the `customerDB` map. The `@javax.ws.rs.PathParam` annotation tells the JAX-RS provider that you want to inject a piece of the incoming URI into the `id` parameter. The `"id"` value of `@PathParam` must match to a URI pattern defined in the method's and class's `@Path` annotations. In this case, the `{id}` pattern is `/customers/{id}`. For example, if the incoming URI of the GET request is *http://shop.restfully.com/customers/333*, the `333` string will be extracted from the URI, converted into an integer, and injected into the `id` parameter of the `getCustomer()` method.

In the implementation of the method, we use the `id` parameter to query for a `Customer` object in the `customerDB` map. If this customer does not exist, we throw the `javax.ws.rs.WebApplicationException`. This exception will set the HTTP response code to 404, "Not Found," meaning that the customer resource does not exist. We'll discuss more about exception handling in Chapter 7, so I won't go into more detail about the `WebApplicationException` here.

We will write the response manually to the client through a `java.io.OutputStream`. In JAX-RS, when you want to do streaming manually, you must implement and return an instance of the `javax.ws.rs.core.StreamingOutput` interface from your JAX-RS method. `StreamingOutput` is a callback interface with one callback method `write()`:

```
package javax.ws.rs.core;

public interface StreamingOutput {
    public void write(OutputStream os) throws IOException,
                                         WebApplicationException;
}
```

In the last line of our `getCustomer()` method, we implement and return an inner class implementation of `StreamingOutput`. Within the `write()` method of this inner class, we delegate back to a utility method called `outputCustomer()` that exists in our `Customer Resource` class. When the JAX-RS provider is ready to send an HTTP response body back over the network to the client, it will callback to the `write()` method we implemented to output the XML representation of our `Customer` object.

In general, you will not use the `StreamingOutput` interface to output responses. In Chapter 6, you will see that JAX-RS has a bunch of nice content handlers that can automatically convert Java objects straight into the data format you are sending across the wire. I didn't want to introduce too many new concepts in the first introductory chapter, so the example only does simple streaming.

Updating a customer

The last RESTful operation we have to implement is updating customers. In Chapter 2, we used `PUT /customers/{id}`, while passing along an updated XML representation of the customer. This is implemented in the `updateCustomer()` method of our `CustomerResource` class:

```
@PUT
@Path("{id}")
@Consumes("application/xml")
public void updateCustomer(@PathParam("id") int id,
                           InputStream is) {
    Customer update = readCustomer(is);
    Customer current = customerDB.get(id);
    if (current == null)
      throw new WebApplicationException(Response.Status.NOT_FOUND);

    current.setFirstName(update.getFirstName());
    current.setLastName(update.getLastName());
    current.setStreet(update.getStreet());
    current.setState(update.getState());
    current.setZip(update.getZip());
    current.setCountry(update.getCountry());
}
```

We annotate the `updateCustomer()` method with `@javax.ws.rs.PUT` to bind HTTP PUT requests to this method. Like our `getCustomer()` method, `updateCustomer()` is annotated with an additional `@Path` annotation so that we can match `/customers/{id}` URIs.

The `updateCustomer()` method takes two parameters. The first is an `id` parameter that represents the `Customer` object we are updating. Like `getCustomer()`, we use the `@PathParam` annotation to extract the ID from the incoming request URI. The second parameter is an `InputStream` that will allow us to read in the XML document that was sent with the PUT request. Like `createCustomer()`, a parameter that is not annotated with a JAX-RS annotation is considered a representation of the body of the incoming message.

In the first part of the method implementation, we read in the XML document and create a `Customer` object out of it. The method then tries to find an existing `Customer` object in the `customerDB` map. If it doesn't exist, we throw a `WebApplicationException` that will send a 404, "Not Found" response code back to the client. If the `Customer` object does exist, we update our existing `Customer` object with new updated values.

Utility methods

The final thing we have to implement is the utility methods that were used in `createCustomer()`, `getcustomer()`, and `updateCustomer()` to transform `Customer` objects to and from XML. The `outputCustomer()` method takes a `Customer` object and writes it as XML to the response's `OutputStream`:

```
protected void outputCustomer(OutputStream os, Customer cust)
                                         throws IOException {
    PrintStream writer = new PrintStream(os);
    writer.println("<customer id=\"" + cust.getId() + "\">");
    writer.println("   <first-name>" + cust.getFirstName()
                    + "</first-name>");
    writer.println("   <last-name>" + cust.getLastName()
                    + "</last-name>");
    writer.println("   <street>" + cust.getStreet() + "</street>");
```

```
        writer.println("   <city>" + cust.getCity() + "</city>");
        writer.println("   <state>" + cust.getState() + "</state>");
        writer.println("   <zip>" + cust.getZip() + "</zip>");
        writer.println("   <country>" + cust.getCountry() + "</country>");
        writer.println("</customer>");
    }
```

As you can see, this is a pretty straightforward method. Through string manipulations, it does a brute-force conversion of the `Customer` object to XML text.

The next method is `readCustomer()`. The method is responsible for reading XML text from an `InputStream` and creating a `Customer` object:

```
    protected Customer readCustomer(InputStream is) {
        try {
            DocumentBuilder builder =
                DocumentBuilderFactory.newInstance().newDocumentBuilder();
            Document doc = builder.parse(is);
            Element root = doc.getDocumentElement();
```

Unlike `outputCustomer()`, we don't manually parse the `InputStream`. The JDK has a built-in XML parser, so we do not need to write it ourselves or download a third-party library to do it. The `readCustomer()` method starts off by parsing the `InputStream` and creating a Java object model that represents the XML document. The rest of the `readCustomer()` method moves data from the XML model into a newly created `Customer` object:

```
            Customer cust = new Customer();
            if (root.getAttribute("id") != null
                    && !root.getAttribute("id").trim().equals("")) {
                cust.setId(Integer.valueOf(root.getAttribute("id")));
            }
            NodeList nodes = root.getChildNodes();
            for (int i = 0; i < nodes.getLength(); i++) {
                Element element = (Element) nodes.item(i);
                if (element.getTagName().equals("first-name")) {
                    cust.setFirstName(element.getTextContent());
                }
                else if (element.getTagName().equals("last-name")) {
                    cust.setLastName(element.getTextContent());
                }
                else if (element.getTagName().equals("street")) {
                    cust.setStreet(element.getTextContent());
                }
                else if (element.getTagName().equals("city")) {
                    cust.setCity(element.getTextContent());
                }
                else if (element.getTagName().equals("state")) {
                    cust.setState(element.getTextContent());
                }
                else if (element.getTagName().equals("zip")) {
                    cust.setZip(element.getTextContent());
                }
                else if (element.getTagName().equals("country")) {
                    cust.setCountry(element.getTextContent());
```

```
                }
            }
            return cust;
        }
        catch (Exception e) {
            throw new WebApplicationException(e,
                        Response.Status.BAD_REQUEST);
        }
    }
}
```

I'll admit, this example was a bit contrived. In a real system, we would not manually output XML or write all this boilerplate code to read in an XML document and convert it to a business object, but I don't want to distract you from learning JAX-RS basics by introducing another API. In Chapter 6, I will show how you can use JAXB to map your `Customer` object to XML and have JAX-RS automatically transform your HTTP message body to and from XML.

JAX-RS and Java Interfaces

In our example so far, we've applied JAX-RS annotations directly on the Java class that implements our service. In JAX-RS, you are also allowed to define a Java interface that contains all your JAX-RS annotation metadata instead of applying all your annotations to your implementation class.

Interfaces are a great way to scope out how you want to model your services. With an interface, you can write something that defines what your RESTful API will look like along with what Java methods they will map to before you write a single line of business logic. Also, many developers like to use this approach so that their business logic isn't "polluted" with so many annotations. They think the code is more readable if it has fewer annotations. Finally, sometimes you do have the case where the same business logic must be exposed not only RESTfully, but also through SOAP and JAX-WS. In this case, your business logic would look more like an explosion of annotations than actual code. Interfaces are a great way to isolate all this metadata into one logical and readable construct.

Let's transform our customer resource example into something that is interface-based:

```
package com.restfully.shop.services;

import ...;

@Path("/customers")
public interface CustomerResource {

    @POST
    @Consumes("application/xml")
    public Response createCustomer(InputStream is);

    @GET
    @Path("{id}")
```

```
@Produces("application/xml")
public StreamingOutput getCustomer(@PathParam("id") int id);

@PUT
@Path("{id}")
@Consumes("application/xml")
public void updateCustomer(@PathParam("id") int id, InputStream is);
}
```

Here, our `CustomerResource` is defined as an interface and all the JAX-RS annotations are applied to methods within that interface. We can then define a class that implements this interface:

```
package com.restfully.shop.services;

import ...;

public class CustomerResourceService implements CustomerResource {

    public Response createCustomer(InputStream is) {
        ... the implementation ...
    }

    public StreamingOutput getCustomer(int id)
        ... the implementation ...
    }

    public void updateCustomer(int id, InputStream is) {
        ... the implementation ...
    }
}
```

As you can see, no JAX-RS annotations are needed within the implementing class. All our metadata is confined to the `CustomerResource` interface.

If you need to, you can override the metadata defined in your interfaces by reapplying annotations within your implementation class. For example, maybe we want to enforce a specific character set for POST XML:

```
public class CustomerResourceService implements CustomerResource {

    @POST
    @Consumes("application/xml;charset=utf-8")
    public Response createCustomer(InputStream is) {
        ... the implementation ...
    }
```

In this example, we are overriding the metadata defined in an interface for one specific method. When overriding metadata for a method, you must respecify all the annotation metadata for that method even if you are changing only one small thing.

Overall, I do not recommend that you do this sort of thing. The whole point of using an interface to apply your JAX-RS metadata is to isolate the information and define it in one place. If your annotations are scattered about between your implementation class and interface, your code becomes a lot harder to read and understand.

Inheritance

The JAX-RS specification also allows you to define class and interface hierarchies if you so desire. For example, let's say we wanted to make our outputCustomer() and readCustomer() methods abstract so that different implementations could transform XML how they wanted:

```
package com.restfully.shop.services;

import ...;

public abstract class AbstractCustomerResource {

    @POST
    @Consumes("application/xml")
    public Response createCustomer(InputStream is) {
        ... complete implementation ...
    }

    @GET
    @Path("{id}")
    @Produces("application/xml")
    public StreamingOutput getCustomer(@PathParam("id") int id) {
        ... complete implementation
    }
    @PUT
    @Path("{id}")
    @Consumes("application/xml")
    public void updateCustomer(@PathParam("id") int id,
                               InputStream is) {
        ... complete implementation ...
    }

    abstract protected void outputCustomer(OutputStream os,
                                   Customer cust) throws IOException;

    abstract protected Customer readCustomer(InputStream is);

}
```

You could then extend this abstract class and define the outputCustomer() and readCustomer() methods:

```
package com.restfully.shop.services;

import ...;

@Path("/customers")
public class CustomerResource extends AbstractCustomerResource {

    protected void outputCustomer(OutputStream os, Customer cust)
                                                throws IOException {
        ... the implementation ...
    }
```

```
    protected Customer readCustomer(InputStream is) {
        ... the implementation ...
    }
```

The only caveat with this approach is that the concrete subclass must annotate itself
with the @Path annotation to identify it as a service class to the JAX-RS provider.

Deploying Our Service

The last thing we need to do is to deploy our JAX-RS service. To do this, we need some
way of telling our environment (an application server, for instance) which
JAX-RS services we want registered. We will write one simple class that extends
javax.ws.rs.core.Application:

```
package javax.ws.rs.core;

import java.util.Collections;
import java.util.Set;

public abstract class Application {
    private static final Set<Object> emptySet = Collections.emptySet();

    public abstract Set<Class<?>> getClasses();


    public Set<Object> getSingletons() {
        return emptySet;
    }

}
```

The getClasses() method returns a list of JAX-RS service classes (and providers, but
I'll get to that in Chapter 6). Any JAX-RS service class returned by this method will
follow the per-request model mentioned earlier. When the JAX-RS vendor implemen-
tation determines that an HTTP request needs to be delivered to a method of one of
these classes, an instance of it will be created for the duration of the request and thrown
away. You are delegating the creation of these objects to the JAX-RS runtime.

The getSingletons() method returns a list of JAX-RS service objects (and providers,
too—again, see Chapter 6). You, as the application programmer, are responsible for
creating and initializing these objects.

These two methods tell the JAX-RS vendor which services you want deployed. Here is
an example:

```
package com.restfully.shop.services;

import javax.ws.rs.core.Application;
import java.util.HashSet;
import java.util.Set;

public class ShoppingApplication extends Application {
```

```
    private Set<Object> singletons = new HashSet<Object>();
    private Set<Class<?>> empty = new HashSet<Class<?>>();

    public ShoppingApplication() {
        singletons.add(new CustomerResource());
    }

    @Override
    public Set<Class<?>> getClasses() {
        return empty;
    }

    @Override
    public Set<Object> getSingletons() {
        return singletons;
    }
}
```

For our customer service database example, we do not have any per-request services, so our `getClasses()` method returns an empty set. Our `getSingletons()` method returns an instance contained within a `Set` of our `CustomerResource` class.

Deployment Within a Servlet Container

Most JAX-RS applications are written to run within an application server's servlet container as a Web ARchive (WAR). Think of a servlet container as your application server's web server. A WAR is a JAR file that, in addition to Java class files, also contains other Java libraries along with dynamic (like JSPs) and static content (like HTML files or images) that you want to publish on your website. We need to place our Java classes within this archive so that our application server can deploy them. Here's what the structure of a WAR file looks like:

```
<any static content>
WEB-INF/
        web.xml
        classes/
                com/restfully/shop/domain/
                                        Customer.class
                com/restfully/shop/services/
                                        CustomerResource.class
                                        ShoppingApplication.class

        lib/
```

Our application server's servlet container publishes everything outside the *WEB-INF/* directory of the archive. This is where you would put static HTML files and images that you want to expose to the outside world. The *WEB-INF/* directory has two subdirectories. Within the *classes/* directory, you can put any Java classes you want there. They must be in a Java package structure. This is where we place all of the classes we wrote and compiled in this chapter. The *lib/* directory can contain any third-party JARs we

used with our application. Depending on whether your application server has built-in support for JAX-RS or not, you may have to place the JARs for your JAX-RS vendor implementation within this directory. For our customer example, we are not using any third-party libraries so this *lib/* directory may be empty.

We are almost finished, but we still need to point our servlet container to our `ShoppingApplication` class so that the JAX-RS runtime knows what to deploy. To do this, we need to create a *WEB-INF/web.xml* file within our archive. If our application server is JAX-RS-aware or, in other words, is tightly integrated with JAX-RS, we can declare our `ShoppingApplication` class as a servlet:

```xml
<?xml version="1.0"?>
<web-app>
   <servlet>
      <servlet-name>Rest</servlet-name>
      <servlet-class>
         com.restfully.shop.services.ShoppingApplication
      </servlet-class>
   </servlet>

   <servlet-mapping>
      <servlet-name>Rest</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

We also need to bind the application using the `servlet-mapping` to a base URI.

If our application server is not JAX-RS-aware, you will have to specify the JAX-RS provider's servlet that handles JAX-RS invocations. The `Application` class should be specified as an `init-param` of the servlet:

```xml
<?xml version="1.0"?>
<web-app>
   <display-name>Archetype Created Web Application</display-name>
   <servlet>
      <servlet-name>Rest</servlet-name>
      <servlet-class>
         com.jaxrs.vendor.JaxrsVendorServlet
      </servlet-class>
      <init-param>
         <param-name>javax.ws.rs.Application</param-name>
         <param-value>
            com.restfully.shop.services.ShoppingApplication
         </param-value>
      </init-param>
   </servlet>
   <servlet-mapping>
      <servlet-name>Rest</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>

</web-app>
```

That's it! Once you have jarred up your WAR file with the *web.xml* we created and the structure that we defined, you can deploy it within your application server and start invoking on the service.

Wrapping Up

In this chapter, we discussed how to implement a simple customer database as a JAX-RS service. You can test drive this code by flipping to the back of this book and looking at Chapter 16. It will walk you through installing JBoss RESTEasy, a JAX-RS implementation, and running the examples in this chapter within a servlet container.

HTTP Method and URI Matching

Now that we have a foundation in JAX-RS, it's time to start looking into the details. In Chapter 3, you saw how we used the `@GET`, `@PUT`, `@POST`, and `@DELETE` annotations to bind Java methods to a specific HTTP operation. You also saw how we used the `@Path` annotation to bind a URI pattern to a Java method. While applying these annotations seems pretty straightforward, there are some interesting attributes that we're going to examine within this chapter.

Binding HTTP Methods

JAX-RS defines five annotations that map to specific HTTP operations:

- `@javax.ws.rs.GET`
- `@javax.ws.rs.PUT`
- `@javax.ws.rs.POST`
- `@javax.ws.rs.DELETE`
- `@javax.ws.rs.HEAD`

In Chapter 3, we used these annotations to bind HTTP GET requests to a specific Java method. For example:

```
@Path("/customers")
public class CustomerService {

    @GET
    @Produces("application/xml")
    public String getAllCustomers() {
    }
}
```

Here we have a simple method, `getAllCustomers()`. The `@GET` annotation instructs the JAX-RS runtime that this Java method will process HTTP GET requests to the URI `/customers`. You would use one of the other five annotations described earlier to bind to different HTTP operations. One thing to note, though, is that you may only

apply one HTTP method annotation per Java method. A deployment error occurs if you apply more than one.

Beyond simple binding, there are some interesting things to note about the implementation of these types of annotations. Let's take a look at @GET, for instance:

```
package javax.ws.rs;

import ...;

@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@HttpMethod(HttpMethod.GET)
public @interface GET {
}
```

@GET, by itself, does not mean anything special to the JAX-RS provider. In other words, JAX-RS is not hardcoded to look for this annotation when deciding whether or not to dispatch an HTTP GET request. What makes the @GET annotation meaningful to a JAX-RS provider is the meta-annotation @javax.ws.rs.HttpMethod. Meta-annotations are simply annotations that annotate other annotations. When the JAX-RS provider examines a Java method, it looks for any method annotations that use the meta-annotation @HttpMethod. The value of this meta-annotation is the actual HTTP operation that you want your Java method to bind to.

HTTP Method Extensions

What are the implications of this? This means that you can create new annotations that bind to HTTP methods other than GET, POST, DELETE, HEAD, and PUT. While HTTP is a ubiquitous, stable protocol, it is still constantly evolving. For example, consider the WebDAV standard.[*] The WebDAV protocol makes the Web an interactive readable and writable medium. It allows users to create, change, and move documents on web servers. It does this by adding a bunch of new methods to HTTP like MOVE, COPY, MKCOL, LOCK, and UNLOCK.

Although JAX-RS does not define any WebDAV-specific annotations, we could create them ourselves using the @HttpMethod annotation:

```
package org.rest.webdav;

import ...;

@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@HttpMethod("LOCK")
public @interface LOCK {
}
```

---

[*] For more information on WebDAV, see *www.webdav.org*.

Here, we have defined a new `@org.rest.LOCK` annotation using `@HttpMethod` to specify the HTTP operation it binds to. We can then use it on JAX-RS resource methods:

```
@Path("/customers")
public class CustomerResource {

    @Path("{id}")
    @LOCK
    public void lockIt(@PathParam("id") String id) {
        ...
    }
}
```

Now WebDAV clients can invoke `LOCK` operations on our web server and they will be dispatched to the `lockIt()` method.
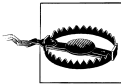
> Do not use `@HttpMethod` to define your own application-specific HTTP methods. `@HttpMethod` exists to hook into new methods defined by standards bodies like the W3C. The purpose of the uniform interface is to define a set of well-known behaviors across companies and organizations on the Web. Defining your own methods breaks this architectural principle.

@Path

There's more to the `@javax.ws.rs.Path` annotation than what we saw in our simple example in Chapter 3. `@Path` can have complex matching expressions so that you can be more specific on what requests get bound to which incoming URIs. `@Path` can also be used on a Java method as sort of an object factory for subresources of your application. We'll examine both in this section.

Binding URIs

The `@javax.ws.rs.Path` annotation in JAX-RS is used to define a URI matching pattern for incoming HTTP requests. It can be placed upon a class or on one or more Java methods. For a Java class to be eligible to receive any HTTP requests, the class must be annotated with at least the `@Path("/")` expression. These types of classes are called JAX-RS *root resources*.

The value of the `@Path` annotation is an expression that denotes a relative URI to the context root of your JAX-RS application. For example, if you are deploying into a WAR archive of a servlet container, that WAR will have a base URI that browsers and remote clients use to access it. `@Path` expressions are relative to this URI.

To receive a request, a Java method must have at least an HTTP method annotation like `@javax.ws.rs.GET` applied to it. This method is not required to have an `@Path` annotation on it, though. For example:

```
@Path("/orders")
public class OrderResource {
   @GET
   public String getAllOrders() {
      ...
   }
}
```

An HTTP request of `GET /orders` would dispatch to the `getAllOrders()` method.

`@Path` can also be applied to your Java method. If you do this, the URI matching pattern is a concatenation of the class's `@Path` expression and that of the method's. For example:

```
@Path("/orders")
public class OrderResource {

   @GET
   @Path("unpaid")
   public String getUnpaidOrders() {
      ...
   }
}
```

So, the URI pattern for `getUnpaidOrders()` would be the relative URI `/orders/unpaid`.

@Path Expressions

The value of the `@Path` annotation is usually a simple string, but you can also define more complex expressions to satisfy your URI matching needs.

Template parameters

In Chapter 3, we wrote a customer access service that allowed us to query for a specific customer using a wildcard URI pattern:

```
@Path("/customers")
public class CustomerResource {

   @GET
   @Path("{id}")
   public String getCustomer(@PathParam("id") int id) {
      ...
   }
}
```

The `{id}` expression represents a template parameter. A template parameter is a named wildcard pattern embedded within the value of an `@Path` annotation. It starts with an open brace, "{", continues with one or more alphanumeric characters, and ends with a closed brace "}". The expression represents one or more characters that are not the slash "/" character. So a `GET /customers/333` request would match `getCustomer()`, but a `GET/customers/333/444` request would not match the path expression.

These template parameters can be embedded anywhere within an `@Path` declaration. For example:

```
@Path("/")
public class CustomerResource {

    @GET
    @Path("customers/{firstname}-{lastname}")
    public String getCustomer(@PathParam("firstname") String first,
                              @PathParam("lastname") String last) {
        ...
    }
}
```

In our example, the URI is constructed with a customer's first name, followed by a hyphen, ending with the last name of the individual. So, the request `GET /customers/333` would no longer match to `getCustomer()`, but a `GET/customers/bill-burke` request would.

Regular expressions

`@Path` expressions are not limited to simple wildcard matching expressions. For example, our `getCustomer()` method takes an integer parameter. We can change our `@Path` value to only match digits:

```
@Path("/customers")
public class CustomerResource {

    @GET
    @Path("{id : \\d+}")
    public String getCustomer(@PathParam("id") int id) {
        ...
    }
}
```

We have taken our `{id}` expression and changed it to `{id : \\d+}`. The `id` string is the identifier of the expression. We'll see in the next chapter how you can reference it with `@PathParam`. The `\\d+` string is a *regular expression*. It is delimited by a ":" character. Regular expressions are part of a string matching language that allows you to express complex string matching patterns. In this case, `\\d+` matches one or more digits. We will not cover regular expressions in this book, but you can read up on how to use them directly from your JDK's javadocs within the `java.util.regex.Pattern` class page.[†]

Regular expressions are not limited in matching one segment of a URI. For example:

```
@Path("/customers")
public class CustomerResource {

    @GET
```

[†] You can find the javadoc for the `Pattern` class at *http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html*.

```
    @Path("{id : .+}")
    public String getCustomer(@PathParam("id") String id) {
        ...
    }

    @GET
    @Path("{id : .+}/address")
    public String getAddress(@PathParam("id") String id) {
        ...
    }

}
```

We've changed getCustomer()'s @Path expression to {id : .+}. The .+ is a regular expression that will match any stream of characters after /customers. So, the GET /customers/bill/burke request would be routed to getCustomer().

The getAddress() method has a more specific expression. It will map any stream of characters after /customers and that ends with /address. So, the GET /customers/bill/burke/address request would be routed to the getAddress() method.

Precedence rules

You may have noticed that, together, the @Path expressions for getCustomer() and getAddress() are ambiguous. A GET /customers/bill/burke/address request could match either getCustomer() or getAddress(), depending on which expression was matched first by the JAX-RS provider. The JAX-RS specification has defined strict sorting and precedence rules for matching URI expressions and is based on a *most specific match wins* algorithm. The JAX-RS provider gathers up the set of deployed URI expressions and sorts them based on the following logic:

1. The primary key of the sort is the number of literal characters in the full URI matching pattern. The sort is in descending order. In our ambiguous example, getCustomer()'s pattern has 11 literal characters: /customers/. The getAddress() method's pattern has 18 literal characters: /customers/ plus address. Therefore, the JAX-RS provider will try and match getAddress()'s pattern before getCustomer()'s.

2. The secondary key of the sort is the number of template expressions embedded within the pattern, i.e., {id} or {id : .+}. This sort is in descending order.

3. The tertiary key of the sort is the number of nondefault template expressions. A default template expression is one that does not define a regular expression, i.e., {id}.

Let's look at a list of sorted URI matching expressions and explain why one would match over another:

```
1 /customers/{id}/{name}/address
2 /customers/{id : .+}/address
3 /customers/{id}/address
4 /customers/{id : .+}
```

Expressions 1–3 come first because they all have more literal characters than expression 4. Although expressions 1–3 all have the same number of literal characters, expression 1 comes first because sorting rule #2 is triggered. It has more template expressions than either pattern 2 or 3. Expressions 2 and 3 have the same number of literal characters and same number of template expressions. Expression 2 is sorted ahead of 3 because it triggers sorting rule #3; it has a template pattern that is a regular expression.

These sorting rules are not perfect. It is still possible to have ambiguities, but the rules cover 90% of use cases. If your application has URI matching ambiguities, your application design is probably too complicated and you need to revisit and refactor your URI scheme.

Encoding

The URI specification only allows certain characters within a URI string. It also reserves certain characters for its own specific use. In other words, you cannot use these characters as part of your URI segments. This is the set of allowable and reserved characters:

- The US-ASCII alphabetic characters a–z and A–Z are allowable.
- The decimal digit characters 0–9 are allowable.
- All these other characters are allowable: `_-!.~'()*`.
- These characters are allowed, but are reserved for URI syntax: `,;:$&+= ?/[]@`.

All other characters must be encoded using the "%" character followed by a two-digit hexadecimal number. This hexadecimal number corresponds to the equivalent hexadecimal character in the ASCII table. So, the string `bill&burke` would be encoded as `bill%26burke`.

When creating `@Path` expressions, you may encode its string, but you do not have to. If a character in your `@Path` pattern is an illegal character, the JAX-RS provider will automatically encode the pattern before trying to match and dispatch incoming HTTP requests. If you do have an encoding within your `@Path` expression, the JAX-RS provider will leave it alone and treat it as an encoding when doing its request dispatching. For example:

```
@Path("/customers"
public class CustomerResource {

    @GET
    @Path("roy&fielding")
    public String getOurBestCustomer() {
        ...
    }
}
```

The `@Path` expression for `getOurBestCustomer()` would match incoming requests like `GET /customers/roy%26fielding`.

Matrix Parameters

One part of the URI specification that we have not touched on yet is *matrix parameters*. Matrix parameters are name-value pairs embedded within the path of a URI string. For example:

```
http://example.cars.com/mercedes/e55;color=black/2006
```

They come after a URI segment and are delimited by the ";" character. The matrix parameter in this example comes after the URI segment `e55`. Its name is `color` and its value is `black`. Matrix parameters are different than query parameters, as they represent attributes of certain segments of the URI and are used for identification purposes. Think of them as adjectives. Query parameters, on the other hand, always come at the end of the URI and always pertain to the full resource you are referencing.

Matrix parameters are ignored when matching incoming requests to JAX-RS resource methods. It is actually illegal to specify matrix parameters within an `@Path` expression. For example:

```
@Path("/mercedes")
public class MercedesService {

    @GET
    @Path("/e55/{year}")
    @Produces("image/jpeg")
    public Jpeg getE55Picture(@PathParam("year") String year) {
        ...
    }
```

If we queried our JAX-RS service with `GET /mercedes/e55;color=black/2006`, the `getE55Picture()` method would match the incoming request and would be invoked. Matrix parameters are not considered part of the matching process, because they are usually variable attributes of the request. We'll see in Chapter 5 how to access matrix parameter information within our JAX-RS resource methods.

Subresource Locators

So far, I've shown you the JAX-RS capability to statically bind URI patterns expressed through the `@Path` annotation to a specific Java method. JAX-RS also allows you to dynamically dispatch requests yourself through subresource locators. Subresource locators are Java methods annotated with `@Path`, but with no HTTP method annotation, like `@GET`, applied to them. This type of method returns an object that is, itself, a JAX-RS annotated service that knows how to dispatch the remainder of the request. This is best described using an example.

Let's continue by expanding our customer database JAX-RS service. This example will be a bit contrived, so please bear with me. Let's say our customer database is partitioned into different databases based on geographic regions. We want to add this information to our URI scheme, but we want to decouple finding a database server from querying

and formatting customer information. We will now add the database partition information to the URI pattern /customers/{database}-db/{customerId}. We can define a CustomerDatabaseResource class and have it delegate to our original CustomerResource class. Here's the example:

```
@Path("/customers")
public class CustomerDatabaseResource {

    @Path("{database}-db")
    public CustomerResource getDatabase(@PathParam("database") String db) {
        // find the instance based on the db parameter
        CustomerResource resource = locateCustomerResource(db);
        return resource;
    }

    protected CustomerResource locateCustomerResource(String db) {
        ...
    }
}
```

The CustomerDatabaseResource class is our root resource. It does not service any HTTP requests directly. It processes the database identifier part of the URI and locates the identified customer database. Once it does this, it allocates a CustomerResource instance passing in a reference to the database. The JAX-RS provider uses this CustomerResource instance to service the remainder of the request:

```
public class CustomerResource {
    private Map<Integer, Customer> customerDB =
                          new ConcurrentHashMap<Integer, Customer>();
    private AtomicInteger idCounter = new AtomicInteger();

    @POST
    @Consumes("application/xml")
    public Response createCustomer(InputStream is) {
        ...
    }

    @GET
    @Path("{id}")
    @Produces("application/xml")
    public StreamingOutput getCustomer(@PathParam("id") int id) {
        ...
    }

    @PUT
    @Path("{id}")
    @Consumes("application/xml")
    public void updateCustomer(@PathParam("id") int id, InputStream is) {
        ...
    }
}
```

So, if a client does a GET /customers/northamerica-db/333, the JAX-RS provider will first match the expression on the method CustomerDatabaseResource.getDatabase().

It will then match and process the remaining part of the request with the method `CustomerResource.getCustomer()`.

Besides the added constructor, another difference in the `CustomerResource` class from previous examples is that it is no longer annotated with `@Path`. It is no longer a root resource in our system; it is a subresource and must not be registered with the JAX-RS runtime within an `Application` class.

Full Dynamic Dispatching

While our previous example does illustrate the concept of subresource locators, it does not show their full dynamic nature. The `CustomerDatabaseResource.getDatabase()` method can return any instance of any class. At runtime, the JAX-RS provider will introspect this instance's class for resource methods that can handle the request.

Let's say that in our example, we have two customer databases with different kinds of identifiers. One database uses a numeric key, as we talked about before. The other uses first and last name as a composite key. We would need to have two different classes to extract the appropriate information from the URI. Let's change our example:

```
@Path("/customers")
public class CustomerDatabaseResource {

    protected CustomerResource europe = new CustomerResource();
    protected FirstLastCustomerResource northamerica =
                            new FirstLastCustomerResource();

    @Path("{database}-db")
    public Object getDatabase(@PathParam("database") String db) {
       if (db.equals("europe")) {
          return europe;
       }
       else if (db.equals("northamerica")) {
          return northamerica;
       }
       else return null;
    }
}
```

Instead of our `getDatabase()` method returning a `CustomerResource`, it will return any `java.lang.Object`. JAX-RS will introspect the instance returned to figure out how to dispatch the request. For this example, if our database is `europe`, we will use our original `CustomerResource` class to service the remainder of the request. If our database is `northamerica`, we will use a new subresource class `FirstLastCustomerResource`:

```
public class FirstLastCustomerResource {
    private Map<String, Customer> customerDB =
                            new ConcurrentHashMap<String, Customer>();
```

```
    @GET
    @Path("{first}-{last}")
    @Produces("application/xml")
    public StreamingOutput getCustomer(@PathParam("first") String firstName,
                                       @PathParam("last") String lastName) {
       ...
    }

    @PUT
    @Path("{first}-{last}")
    @Consumes("application/xml")
    public void updateCustomer(@PathParam("first") String firstName,
                               @PathParam("last") String lastName,
                               InputStream is) {
       ...
    }
}
```

Customer lookup requests routed to `europe` would match the `/customers/{database}-db/{id}` URI pattern defined in `CustomerResource`. Requests routed to `northamerica` would match the `/customers/{database}-db/{first}-{last}` URI pattern defined in `FirstLastCustomerResource`. This type of pattern gives you a lot of freedom to dispatch your own requests.

Wrapping Up

In this chapter, we examined the intricacies of the `@javax.ws.rs.Path` annotation. `@Path` allows you to define complex URI matching patterns that can map to a Java method. These patterns can be defined using regular expressions and also support encoding. We also discussed subresource locators, which allow you to programmatically perform your own dynamic dispatching of HTTP requests. Finally, we looked at how you can hook into new HTTP operations by using the `@HttpMethod` annotation. You can test-drive the code in this chapter in Chapter 17.

JAX-RS Injection

A lot of JAX-RS is pulling information from an HTTP request and injecting it into a Java method. You may be interested in a fragment of the incoming URI. You might be interested in a URI query string value. The client might be sending critical HTTP headers or cookie values that your service needs to process the request. JAX-RS lets you grab this information à la carte, as you need it, through a set of injection annotations and APIs.

The Basics

There are a lot of different things JAX-RS annotations can inject. Here is a list of those provided by the specification:

@javax.ws.rs.PathParam
> This annotation allows you to extract values from URI template parameters.

@javax.ws.rs.MatrixParam
> This annotation allows you to extract values from a URI's matrix parameters.

@javax.ws.rs.QueryParam
> This annotation allows you to extract values from a URI's query parameters.

@javax.ws.rs.FormParam
> This annotation allows you to extract values from posted form data.

@javax.ws.rs.HeaderParam
> This annotation allows you to extract values from HTTP request headers.

@javax.ws.rs.CookieParam
> This annotation allows you to extract values from HTTP cookies set by the client.

@javax.ws.rs.core.Context
> This class is the all-purpose injection annotation. It allows you to inject various helper and informational objects that are provided by the JAX-RS API.

Usually, these annotations are used on the parameters of a JAX-RS resource method. When the JAX-RS provider receives an HTTP request, it finds a Java method that will service this request. If the Java method has parameters that are annotated with any of these injection annotations, it will extract information from the HTTP request and pass it as a parameter when it invokes the method.

For per-request resources, you may alternatively use these injection annotations on fields, setter methods, and even constructor parameters of your JAX-RS resource class. Do not try to use these annotations on fields or setter methods if your component model does not follow per-request instantiation. Singletons process HTTP requests concurrently, so it is not possible to use these annotations on fields or setter methods, as concurrent requests will overrun and conflict with each other.

@PathParam

We looked at `@javax.ws.rs.PathParam` a little bit in Chapters 3 and 4. `@PathParam` allows you to inject the value of named URI path parameters that were defined in `@Path` expressions. Let's revisit the `CustomerResource` example that we defined in Chapter 2 and implemented in Chapter 3:

```
@Path("/customers")
public class CustomerResource {
   ...

   @Path("{id}")
   @GET
   @Produces("application/xml")
   public StreamingOutput getCustomer(@PathParam("id") int id) {
      ...
   }
}
```

In this example, we want to route HTTP GET requests to the relative URI pattern `/customers/{id}`. Our `getCustomer()` method extracts a `Customer` ID from the URI using `@PathParam`. The value of the `@PathParam` annotation, `"id"`, matches the path parameter, `{id}`, that we defined in the `@Path` expression of `getCustomer()`.

While `{id}` represents a string segment in the request's URI, JAX-RS automatically converts the value to an `int` before it invokes the `getCustomer()` method. If the URI path parameter cannot be converted to an integer, the request is considered a client error and the client will receive a 404, "Not Found" response from the server.

More Than One Path Parameter

You can reference more than one URI path parameter in your Java methods. For instance, let's say we are using first and last name to identify a customer in our `CustomerResource`:

```
@Path("/customers")
public class CustomerResource {
   ...

   @Path("{first}-{last}")
   @GET
   @Produces("application/xml")
   public StreamingOutput getCustomer(@PathParam("first") String firstName,
                                      @PathParam("last") String lastName) {
      ...
   }
}
```

Here, we have the URI path parameters {first} and {last}. If our HTTP request is GET /customers/bill-burke, bill will be injected into the firstName parameter and burke will be injected into the lastName parameter.

Scope of Path Parameters

Sometimes a named URI path parameter will be repeated by different @Path expressions that compose the full URI matching pattern of a resource method. The path parameter could be repeated by the class's @Path expression or by a subresource locator. In these cases, the @PathParam annotation will always reference the final path parameter. For example:

```
@Path("/customers/{id}")
public class CustomerResource {

   @Path("/address/{id}")
   @Produces("text/plain")
   @GET
   public String getAddress(@PathParam("id") String addressId) {...}
}
```

If our HTTP request was GET /customers/123/address/456, the addressId parameter in the getAddress() method would have the 456 value injected.

PathSegment and Matrix Parameters

@PathParam can not only inject the value of a path parameter, it can also inject instances of javax.ws.rs.core.PathSegment. The PathSegment class is an abstraction of a specific URI path segment:

```
package javax.ws.rs.core;

public interface PathSegment  {

   String getPath();
   MultivaluedMap<String, String> getMatrixParameters();

}
```

The `getPath()` method is the string value of the actual URI segment minus any matrix parameters. The more interesting method here is `getMatrixParameters()`. This returns a map of the entire matrix parameters applied to a particular URI segment. In combination with `@PathParam`, you can get access to the matrix parameters applied to your request's URI. For example:

```
@Path("/cars/{make}")
public class CarResource {

    @GET
    @Path("/{model}/{year}")
    @Produces("image/jpeg")
    public Jpeg getPicture(@PathParam("make") String make,
                           @PathParam("model") PathSegment car,
                           @PathParam("year") String year) {
        String carColor = car.getMatrixParameters().getFirst("color");
        ...
    }
}
```

In this example, we have a `CarResource` that allows us to get pictures of cars in our database. The `getPicture()` method returns a JPEG image of cars that match the make, model, and year that we specify. The color of the vehicle is expressed as a matrix parameter of the model. For example:

```
GET /cars/mercedes/e55;color=black/2006
```

Here, our make is `mercedes`, the model is `e55` with a color attribute of `black`, and the year is `2006`. While the make, model, and year information can be injected into our `getPicture()` method directly, we need to do some processing to obtain information about the color of the vehicle.

Instead of injecting the model information as a Java string, we inject the path parameter as a `PathSegment` into the `car` parameter. We then use this `PathSegment` instance to obtain the `color` matrix parameter's value.

Matching with multiple PathSegments

Sometimes a particular path parameter matches to more than one URI segment. In these cases, you can inject a list of `PathSegments`. For example, let's say a model in our `CarResource` could be represented by more than one URI segment. Here's how the `getPicture()` method might change:

```
@Path("/cars/{make}")
public class CarResource {

    @GET
    @Path("/{model : .+}/year/{year}")
    @Produces("image/jpeg")
    public Jpeg getPicture(@PathParam("make") String make,
                           @PathParam("model") List<PathSegment> car,
                           @PathParam("year") String year) {
```

```
        }
    }
```

In this example, if our request was GET /cars/mercedes/e55/amg/year/2006, the car parameter would have a list of two PathSegments injected into it, one representing the e55 segment and the other representing the amg segment. We could then query and pull in matrix parameters as needed from these segments.

Programmatic URI Information

All this à la carte injection of path parameter data with the @PathParam annotation is perfect most of the time. Sometimes, though, you need a more general raw API to query and browse information about the incoming request's URI. The interface javax.ws.rs.core.UriInfo provides such an API:

```
public interface UriInfo {
    public String getPath();
    public String getPath(boolean decode);
    public List<PathSegment> getPathSegments();
    public List<PathSegment> getPathSegments(boolean decode);
    public MultivaluedMap<String, String> getPathParameters();
    public MultivaluedMap<String, String> getPathParameters(boolean decode);
    ...
}
```

The getPath() methods allow you obtain the relative path JAX-RS used to match the incoming request. You can receive the path string decoded or encoded. The getPath Segments() methods break up the entire relative path into a series of PathSegment objects. Like getPath(), you can receive this information encoded or decoded. Finally, getPathParameters() returns a map of all the path parameters defined for all matching @Path expressions.

You can obtain an instance of the UriInfo interface by using the @javax.ws.rs.core.Con text injection annotation. Here's an example:

```
@Path("/cars/{make}")
public class CarResource {

    @GET
    @Path("/{model}/{year}")
    @Produces("image/jpeg")
    public Jpeg getPicture(@Context UriInfo info) {
        String make = info.getPathParameters().getFirst("make");
        PathSegment model = info.getPathSegments().get(1);
        String color = model.getMatrixParameters().getFirst("color");
    ...
    }
}
```

In this example, we inject an instance of UriInfo into the getPicture() method's info parameter. We then use this instance to extract information out of the URI.

@MatrixParam

Instead of injecting and processing `PathSegment` objects to obtain matrix parameter values, the JAX-RS specification allows you to inject matrix parameter values directly through the `@javax.ws.rs.MatrixParam` annotation. Let's change our `CarResource` example from the previous section to reflect using this annotation:

```
@Path("/{make}")
public class CarResource {

    @GET
    @Path("/{model}/{year}")
    @Produces("image/jpeg")
    public Jpeg getPicture(@PathParam("make") String make,
                           @PathParam("model") String model,
                           @MatrixParam("color") String color) {
        ...
    }
```

Using the `@MatrixParam` annotation shrinks our code and provides a bit more readability. The only downside of `@MatrixParam` is that sometimes you might have a repeating matrix parameter that is applied to many different path segments in the URI. For example, what if `color` shows up multiple times in our car service example?

```
GET /mercedes/e55;color=black/2006/interior;color=tan
```

Here, the `color` attribute shows up twice: once with the model and once with the interior. Using `@MatrixParam("color")` in this case would be ambiguous and we would have to go back to processing `PathSegments` to obtain this matrix parameter.

@QueryParam

The `@javax.ws.rs.QueryParam` annotation allows you to inject individual URI query parameters into your Java parameters. For example, let's say we wanted to query a customer database and retrieve a subset of all customers in the database. Our URI might look like this:

```
GET /customers?start=0&size=10
```

The `start` query parameter represents the customer index we want to start with and the `size` query parameter represents how many customers we want returned. The JAX-RS service that implemented this might look like this:

```
@Path("/customers")
public class CustomerResource {

    @GET
    @Produces("application/xml")
    public String getCustomers(@QueryParam("start") int start,
                               @QueryParam("size") int size) {
        ...
```

```
        }
    }
```

Here, we use the `@QueryParam` annotation to inject the URI query parameters `"start"` and `"size"` into the Java parameters `start` and `size`. As with other annotation injection, JAX-RS automatically converts the query parameter's string into an integer.

Programmatic Query Parameter Information

You may have the need to iterate through all query parameters defined on the request URI. The `javax.ws.rs.core.UriInfo` interface has a `getQueryParameters()` method that gives you a map containing all query parameters:

```
public interface UriInfo {
...
    public MultivaluedMap<String, String> getQueryParameters();
    public MultivaluedMap<String, String> getQueryParameters(
                                                boolean decode);
...
}
```

You can inject instances of `UriInfo` using the `@javax.ws.rs.core.Context` annotation. Here's an example of injecting this class and using it to obtain the value of a few query parameters:

```
@Path("/customers")
public class CustomerResource {

    @GET
    @Produces("application/xml")
    public String getCustomers(@Context UriInfo info) {
        String start = info.getQueryParameters().getFirst("start");
        String size = info.getQueryParameters().getFirst("size");
        ...
    }
}
```

@FormParam

The `@javax.ws.rs.FormParam` annotation is used to access `application/x-www-form-urlencoded` request bodies. In other words, it's used to access individual entries posted by an HTML form document. For example, let's say we set up a form on our website to register new customers:

```
<FORM action="http://example.com/customers" method="post">
    <P>
    First name: <INPUT type="text" name="firstname"><BR>
    Last name: <INPUT type="text" name="lastname"><BR>
    <INPUT type="submit" value="Send">
    </P>
</FORM>
```

We could post this form directly to a JAX-RS backend service described as follows:

```
@Path("/customers")
public class CustomerResource {

    @POST
    public void createCustomer(@FormParam("firstname") String first,
                               @FormParam("lastname") String last) {
        ...
    }
}
```

Here, we are injecting `firstname` and `lastname` from the HTML form into the Java parameters `first` and `last`. Form data is URL-encoded when it goes across the wire. When using `@FormParam`, JAX-RS will automatically decode the form entry's value before injecting it.

@HeaderParam

The `@javax.ws.rs.HeaderParam` annotation is used to inject HTTP request header values. For example, what if your application was interested in the web page that referred to or linked to your web service? You could access the HTTP `Referer` header using the `@HeaderParam` annotation:

```
@Path("/myservice")
public class MyService {

    @GET
    @Produces("text/html")
    public String get(@HeaderParam("Referer") String referer) {
        ...
    }
}
```

The `@HeaderParam` annotation is pulling the `Referer` header directly from the HTTP request and injecting it into the referer method parameter.

Raw Headers

Sometimes you need programmatic access to view all headers within the incoming request. For instance, you may want to log them. The JAX-RS specification provides the `javax.ws.rs.core.HttpHeaders` interface for such scenarios.

```
public interface HttpHeaders {
    public List<String> getRequestHeader(String name);
    public MultivaluedMap<String, String> getRequestHeaders();
...
}
```

The `getRequestHeader()` method allows you to get access to one particular header, and `getRequestHeaders()` gives you a map that represents all headers.

As with `UriInfo`, you can use the `@Context` annotation to obtain an instance of `HttpHeaders`. Here's an example:

```
@Path("/myservice")
public class MyService {

    @GET
    @Produces("text/html")
    public String get(@Context HttpHeaders headers) {
        String referer = headers.getRequestHeader("Referer").get(0);
        for (String header : headers.getRequestHeaders().keySet())
        {
            System.out.println("This header was set: " + header);
        }
        ...
    }
}
```

@CookieParam

Servers can store state information in cookies on the client, and can retrieve that information when the client makes its next request. Many web applications use cookies to set up a session between the client and the server. They also use cookies to remember identity and user preferences between requests. These cookie values are transmitted back and forth between the client and server via cookie headers.

The `@javax.ws.rs.CookieParam` annotation allows you to inject cookies sent by a client request into your JAX-RS resource methods. For example, let's say our applications push a `customerId` cookie to our clients so that we can track users as they invoke and interact with our web services. Code to pull in this information might look like this:

```
@Path("/myservice")
public class MyService {

    @GET
    @Produces("text/html")
    public String get(@CookieParam("customerId") int custId) {
        ...
    }
}
```

The use of `@CookieParam` here makes the JAX-RS provider search all cookie headers for the `customerId` cookie value. It then converts it into an `int` and injects it into the `custId` parameter.

If you need more information about the cookie other than its base value, you can instead inject a `javax.ws.rs.core.Cookie` object:

```
@Path("/myservice")
public class MyService {
```

```
    @GET
    @Produces("text/html")
    public String get(@CookieParam("customerId") Cookie custId) {
      ...
    }
}
```

The `Cookie` class has additional contextual information about the cookie beyond its name and value:

```
package javax.ws.rs.core;

public class Cookie
{
    public String getName() {...}
    public String getValue() {...}
    public int getVersion() {...}
    public String getDomain() {...}
    public String getPath() {...}

  ...
}
```

The `getName()` and `getValue()` methods correspond to the string name and value of the cookie you are injecting. The `getVersion()` method defines the format of the cookie header, specifically, which version of the cookie specification[*] the header follows. The `getDomain()` method specifies the DNS name that the cookie matched. The `getPath()` method corresponds to the URI path that was used to match the cookie to the incoming request. All these attributes are defined in detail by the IETF cookie specification.

You can also obtain a map of all cookies sent by the client by injecting a reference to `javax.ws.rs.core.HttpHeaders`:

```
public interface HttpHeaders {
...
    public Map<String, Cookie> getCookies();
}
```

As you saw in the previous section, you use the `@Context` annotation to get access to `HttpHeaders`. Here's an example of logging all cookies sent by the client:

```
@Path("/myservice")
public class MyService {

    @GET
    @Produces("text/html")
    public String get(@Context HttpHeaders headers) {
       for (String name : headers.getCookies().keySet())
       {
```

---

[*] For more information, see *www.ietf.org/rfc/rfc2109.txt*.

```
        Cookie cookie = headers.getCookies().get(name);
        System.out.println("Cookie: " +
                        name + "=" + cookie.getValue());
      }
      ...
    }
  }
```

Common Functionality

Each of these injection annotations has a common set of functionality and attributes. Some can automatically be converted from their string representation within an HTTP request into a specific Java type. You can also define default values for an injection parameter when an item does not exist in the request. Finally, you can work with encoded strings directly, rather than having JAX-RS automatically decode values for you. Let's look into a few of these.

Automatic Java Type Conversion

All the injection annotations described in this chapter reference various parts of an HTTP request. These parts are represented as a string of characters within the HTTP request. You are not limited to manipulating strings within your Java code, though. JAX-RS can convert this string data into any Java type that you want, provided that it matches one of the following criteria:

1. It is a primitive type. The `int`, `short`, `float`, `double`, `byte`, `char`, and `boolean` types all fit into this category.
2. It is a Java class that has a constructor with a single `String` parameter.
3. It is a Java class that has a static method named `valueOf()` that takes a single `String` argument and returns an instance of the class.
4. It is a `java.util.List<T>`, `java.util.Set<T>`, or `java.util.SortedSet<T>`, where `T` is a type that satisfies criteria 2 or 3 or is a `String`. Examples are `List<Double>`, `Set<String>`, or `SortedSet<Integer>`.

Primitive type conversion

We've already seen a few examples of automatic string conversion into a primitive type. Let's review a simple example again:

```
@GET
@Path("{id}")
public String get(@PathParam("id") int id) {...}
```

Here, we're extracting an integer ID from a string-encoded segment of our incoming request URI.

Java object conversion

Besides primitives, this string request data can be converted into a Java object before it is injected into your JAX-RS method parameter. This object's class must have a constructor or a static method named `valueOf()` that takes a single String parameter.

For instance, let's go back to the `@HeaderParam` example we used earlier in this chapter. In that example, we used `@HeaderParam` to inject a string that represented the `Referer` header. Since `Referer` is a URL, it would be much more interesting to inject it as an instance of `java.net.URL`:

```
import java.net.URL;

@Path("/myservice")
public class MyService {

    @GET
    @Produces("text/html")
    public String get(@HeaderParam("Referer") URL referer) {
        ...
    }
}
```

The JAX-RS provider can convert the `Referer` string header into a `java.net.URL` because this class has a constructor that takes only one String parameter.

This automatic conversion also works well when only a `valueOf()` method exists within the Java type we want to convert. For instance, let's revisit the `@MatrixParam` example we used in this chapter. In that example, we used the `@MatrixParam` annotation to inject the color of our vehicle into a parameter of a JAX-RS method. Instead of representing color as a string, let's define and use a Java Enum class:

```
public enum Color {
    BLACK,
    BLUE,
    RED,
    WHITE,
    SILVER
}
```

You cannot allocate Java Enums at runtime, but they do have a built-in `valueOf()` method that the JAX-RS provider can use:

```
public class CarResource {

    @GET
    @Path("/{model}/{year}")
    @Produces("image/jpeg")
    public Jpeg getPicture(@PathParam("make") String make,
                           @PathParam("model") String model,
                           @MatrixParam("color") Color color) {
        ...
    }
```

JAX-RS has made our lives a bit easier, as we can now work with more concrete Java objects rather than doing string conversions ourselves.

Collections

All the parameter types described in this chapter may have multiple values for the same named parameter. For instance, let's revisit the `@QueryParam` example from earlier in this chapter. In that example, we wanted to pull down a set of customers from a customer database. Let's expand the functionality of this query so that we can order the data sent back by any number of customer attributes:

```
GET /customers?orderBy=last&orderBy=first
```

In this request, the `orderBy` query parameter is repeated twice with different values. We can let our JAX-RS provider represent these two parameters as a `java.util.List` and inject this list with one `@QueryParam` annotation:

```
import java.util.List;

@Path("/customers")
public class CustomerResource {

    @GET
    @Produces("application/xml")
    public String getCustomers(
                  @QueryParam("start") int start,
                  @QueryParam("size") int size,
                  @QueryParam("orderBy") List<String> orderBy) {
       ...
    }
}
```

You must define the generic type the List will contain; otherwise, JAX-RS won't know which objects to fill it with.

Conversion failures

If the JAX-RS provider fails to convert a string into the Java type specified, it is considered a client error. If this failure happens when processing an injection for an `@MatrixParam`, `@QueryParam`, or `@PathParam`, an error status of 404, "Not Found" is sent back to the client. If the failure happens with `@HeaderParam` or `@CookieParam`, an error response code of 400, "Bad Request" is sent.

@DefaultValue

In many types of JAX-RS services, you may have parameters that are optional. When a client does not provide this optional information within the request, JAX-RS will, by default, inject a `null` value for Object types and a zero value for primitive types.

Many times, though, a null or zero value may not work as a default value for your injection. To solve this problem, you can define your own default value for optional parameters by using the `@javax.ws.rs.DefaultValue` annotation.

For instance, let's look back again at the `@QueryParam` example given earlier in this chapter. In that example, we wanted to pull down a set of customers from a customer database. We used the `start` and `size` query parameters to specify the beginning index and the number of customers desired. While we do want to control the amount of customers sent back as a response, we do not want to require the client to send these query parameters when making a request. We can use `@DefaultValue` to set a base index and dataset size:

```
import java.util.List;

@Path("/customers")
public class CustomerResource {

    @GET
    @Produces("application/xml")
    public String getCustomers(@DefaultValue("0") @QueryParam("start") int start,
                               @DefaultValue("10") @QueryParam("size") int size) {
        ...
    }
}
```

Here, we've used `@DefaultValue` to specify a default start index of `0` and a default dataset size of `10`. JAX-RS will use the string conversion rules to convert the string value of the `@DefaultValue` annotation into the desired Java type.

@Encoded

URI template, matrix, query, and form parameters must all be encoded by the HTTP specification. By default, JAX-RS decodes these values before converting them into the desired Java types. Sometimes, though, you may want to work with the raw encoded values. Using the `@javax.ws.rs.Encoded` annotation gives you the desired effect:

```
@GET
@Produces("application/xml")
public String get(@Encoded @QueryParam("something") String str) {...}
```

Here, we've used the `@Encoded` annotation to specify that we want the encoded value of the `something` query parameter be injected into the `str` Java parameter. If you want to work solely with encoded values within your Java method or even your entire class, you can annotate the method or class with `@Encoded` and only encoded values will be used.

Wrapping Up

In this chapter, we examined how to use JAX-RS injection annotations to insert bits and pieces of an HTTP request à la carte into your JAX-RS resource method parameters. While represented as strings within an HTTP request, JAX-RS can automatically convert this data into the Java type you desire, provided that the type follows certain constraints. These features allow you to write compact, easily understandable code and avoid a lot of the boilerplate code you might need if you were using other frameworks like the servlet specification. You can test-drive the code in this chapter by flipping to the back of this book and looking at Chapter 18.

JAX-RS Content Handlers

Last chapter, we focused on injecting information from the header of an HTTP request. In this chapter, we will focus on the message body of an HTTP request and response. In the examples in the previous chapters, we used low-level streaming to read in requests and write out responses. To make things easier, JAX-RS also allows you to marshal message bodies to and from specific Java types. It has a number of built-in providers, but you can also write and plug in your own providers. Let's look at them all.

Built-in Content Marshalling

JAX-RS has a bunch of built-in handlers that can marshal to and from a few different specific Java types. While most are low-level conversions, they can still be useful to your JAX-RS classes.

javax.ws.rs.core.StreamingOutput

We were first introduced to `StreamingOutput` back in Chapter 3. `StreamingOutput` is a simple callback interface that you implement when you want to do raw streaming of response bodies:

```
public interface StreamingOutput  {
     void write(OutputStream output) throws IOException,
                                      WebApplicationException;
}
```

You allocate implemented instances of this interface and return them from your JAX-RS resource methods. When the JAX-RS runtime is ready to write the response body of the message, the `write()` method is invoked on the `StreamingOutput` instance. Let's look at an example:

```
@Path("/myservice")
public class MyService {

   @GET
   @Produces("text/plain")
   StreamingOutput get() {
      return new StreamingOutput() {
         public void write(OutputStream output) throws IOException,
                                             WebApplicationException {
            output.write("hello world".getBytes());
         }
      };
   }
}
```

Here, we're getting access to the raw `java.io.OutputStream` through the `write()` method and outputting a simple string to the stream. I like to use an anonymous inner class implementation of the `StreamingOutput` interface rather than creating a separate public class. Since the `StreamingOutput` interface is so tiny, I think it's beneficial to keep the output logic embedded within the original JAX-RS resource method so that the code is easier to follow. Usually, you're not going to reuse this logic in other methods, so it doesn't make much sense to create a specific class.

You may be asking yourself, "Why not just inject an `OutputStream` directly? Why have a callback object to do streaming output?" That's a good question! The reason for having a callback object is that it gives the JAX-RS implementation freedom to handle output however it wants. For performance reasons, it may sometimes be beneficial for the JAX-RS implementation to use a different thread other than the calling thread to output responses. More importantly, many JAX-RS implementations have an interceptor model that abstracts things out like automatic GZIP encoding or response caching. Streaming directly can usually bypass these architectural constructs. Finally, the Servlet 3.0 specification has introduced the idea of asynchronous responses. The callback model fits in very nicely with the idea of asynchronous HTTP within the Servlet 3.0 specification.

java.io.InputStream, java.io.Reader

For reading request message bodies, you can use a raw `InputStream` or `Reader` for inputting any media type. For example:

```
@Path("/")
public class MyService {

   @PUT
   @Path("/stuff")
   public void putStuff(InputStream is) {
      byte[] bytes = readFromStream(is);
      String input = new String(bytes);
      System.out.println(input);
   }
```

```
    private byte[] readFromStream(InputStream stream)
        throws IOException
{
    ByteArrayOutputStream baos = new ByteArrayOutputStream();

    byte[] buffer = new byte[1000];
    int wasRead = 0;
    do {
        wasRead = stream.read(buffer);
        if (wasRead > 0) {
            baos.write(buffer, 0, wasRead);
        }
    } while (wasRead > -1);
    return baos.toByteArray();
}
```

Here, we're reading the full raw bytes of the `java.io.InputStream` available and using them to create a String that we output to the screen:

```
@PUT
@Path("/morestuff")
public void putMore(Reader reader) {
    LineNumberReader lineReader = new LineNumberReader(reader);
    do {
        String line = lineReader.readLine();
        if (line != null) System.out.println(line);
    } while (line != null);
}
```

For this example, we're creating a `java.io.LineNumberReader` that wraps our injected `Reader` object and prints out every line in the request body.

You are not limited to using `InputStream` and `Reader` for reading input request message bodies. You can also return these as response objects. For example:

```
@Path("/file")
public class FileService {

    private static final String basePath = "...";
    @GET
    @Path("{filepath: .*}")
    @Produces("text/plain")
    public InputStream getFile(@PathParam("filepath") String path) {
        FileInputStream is = new FileInputStream(basePath + path);
        return is;
    }
```

Here, we're using an injected `@PathParam` to create a reference to a real file that exists on our disk. We create a `java.io.FileInputStream` based on this path and return it as our response body. The JAX-RS implementation will read from this input stream into a buffer and write it back out incrementally to the response output stream. We must specify the `@Produces` annotation so that the JAX-RS implementation knows how to set the `Content-Type` header.

java.io.File

Instances of `java.io.File` can also be used for input and output of any media type. Here's an example for returning a reference to a file on disk:

```
@Path("/file")
public class FileService {

    private static final String basePath = "...";
    @GET
    @Path("{filepath: .*}")
    @Produces("text/plain")
    public File getFile(@PathParam("filepath") String path) {
       return new File(basePath + path);
    }
}
```

In this example, we're using an injected `@PathParam` to create a reference to a real file that exists on our disk. We create a `java.io.File` based on this path and return it as our response body. The JAX-RS implementation will open up an `InputStream` based on this file reference and stream into a buffer that is written back incrementally to the response's output stream. We must specify the `@Produces` annotation so that the JAX-RS implementation knows how to set the `Content-Type` header.

You can also inject `java.io.File` instances that represent the incoming request response body. For example:

```
@POST
@Path("/morestuff")
public void post(File file) {
    Reader reader = new Reader(new FileInputStream(file));
    LineNumberReader lineReader = new LineNumberReader(reader);
    do {
       String line = lineReader.readLine();
       if (line != null) System.out.println(line);
    } while (line != null);
}
```

The way this works is that the JAX-RS implementation creates a temporary file for input on disk. It reads from the network buffer and saves the bytes read into this temporary file. In our example, we create a `java.io.FileInputStream` from the `java.io.File` object that was injected by the JAX-RS runtime. We then use this input stream to create a `LineNumberReader` and output the posted data to the console.

byte[]

A raw array of bytes can be used for the input and output of any media type. Here's an example:

```
@Path("/")
public class MyService {
```

```
    @GET
    @Produces("text/plain")
    public byte[] get() {
       return "hello world".getBytes();
    }

    @POST
    @Consumes("text/plain")
    public void post(byte[] bytes) {
        System.out.println(new String(bytes));
    }
 }
```

For JAX-RS resource methods that return an array of bytes, you must specify the
@Produces annotation so that JAX-RS knows what media to use to set the Content-
Type header.

String, char[]

Most of the data formats on the Internet are text-based. JAX-RS can convert any text-
based format to and from either a String or array of characters. For example:

```
@Path("/")
public class MyService {

    @GET
    @Produces("application/xml")
    public String get() {
       return "<customer><name>Bill Burke</name></customer>";
    }

    @POST
    @Consumes("text/plain")
    public void post(String str) {
        System.out.println(str);
    }
 }
```

For JAX-RS resource methods that return a String or array of characters, you must
specify the @Produces annotation so that JAX-RS knows what media to use to set the
Content-Type header.

The JAX-RS specification does require that implementations be sensitive to the char-
acter set specified by the Content-Type when creating injected String. For example,
here's a client HTTP POST request that is sending some text data to our service:

```
POST /data
Content-Type: application/xml;charset=UTF-8

<customer>...</customer>
```

The `Content-Type` of the request is `application/xml`, but it is also stating the character encoding is `UTF-8`. JAX-RS implementations will make sure that the created Java `String` is encoded as `UTF-8` as well.

MultivaluedMap<String, String> and Form Input

HTML forms are a common way to post data to web servers. Form data is encoded as the `application/x-www-form-urlencoded` media type. In Chapter 5, we saw how you can use the `@FormParam` annotation to inject individual form parameters from the request. You can also inject a `MultivaluedMap<String, String>` that represents all the form data sent with the request. For example:

```
@Path("/")
public class MyService {

    @POST
    @Consumes("application/x-www-form-urlencoded")
    @Produces("application/x-www-form-urlencoded")
    public MultivaluedMap<String,String> post(
                              MultivaluedMap<String, String> form) {

        return form;
    }
}
```

Here, our `post()` method accepts POST requests and receives a `MultivaluedMap<String, String>` containing all our form data. You may also return a `MultivaluedMap` of form data as your response. We do this in our example.

The JAX-RS specification does not say whether the injected `MultivaluedMap` should contain encoded strings or not. Most JAX-RS implementations will automatically decode the map's string keys and values. If you want it encoded, you can use the `@javax.ws.rs.Encoded` annotation to notify the JAX-RS implementation that you want the data in its raw form.

javax.xml.transform.Source

The `javax.xml.transform.Source` interface represents XML input or output. It is usually used to perform XSLT transformations on input documents. Here's an example:

```
@Path("/transform")
public class TransformationService {

    @POST
    @Consumes("application/xml")
    @Produces("application/xml")
    public String post(Source source) {

        javax.xml.transform.TransformerFactory tFactory =
                javax.xml.transform.TransformerFactory.newInstance();
```

```
javax.xml.transform.Transformer transformer =
    tFactory.newTransformer(
      new javax.xml.transform.stream.StreamSource("foo.xsl"));

StringWriter writer = new StringWriter();
transformer.transform(source,
      new javax.xml.transform.stream.StreamResult(writer));

return writer.toString();
}
```

In this example, we're having JAX-RS inject a `javax.xml.transform.Source` instance that represents our request body and we're transforming it using an XSLT transformation.

Except for JAXB, `javax.xml.transform.Source` is the only XML-based construct that the specification requires implementers to support. I find it a little strange that you can't automatically inject and marshal `org.w3c.dom.Document` objects. This was probably just forgotten in the writing of the specification.

JAXB

JAXB is an older Java specification and is not defined by JAX-RS. JAXB is an annotation framework that maps Java classes to XML and XML schema. It is extremely useful because instead of interacting with an abstract representation of an XML document, you can work with real Java objects that are closer to the domain you are modeling. JAX-RS has built-in support for JAXB, but before we review these handlers, let's get a brief overview of the JAXB framework.

Intro to JAXB

A whole book could be devoted to explaining the intricacies of JAXB, but I'm only going to focus here on the very basics of the framework. If you want to map an existing Java class to XML using JAXB, there are a few simple annotations you can use. Let's look at a simple example:

```
@XmlRootElement(name="customer")
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {

    @XmlAttribute
    protected int id;

    @XmlElement
    protected String fullname;

    public Customer() {}

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

```
        public String getFullName() { return this.fullname; }
        public void setFullName(String name} { this.fullname = name; }
    }
```

The `@javax.xml.bind.annotation.XmlRootElement` annotation is put on Java classes to denote that they are XML elements. The `name()` attribute of `@XmlRootElement` specifies the string to use for the name of the XML element. In our example, the annotation `@XmlRootElement` specifies that our `Customer` objects should be marshalled into an XML element named `<customer>`.

The `@javax.xml.bind.annotation.XmlAttribute` annotation was placed on the `id` field of our `Customer` class. This annotation tells JAXB to map the field to an `id` attribute on the main `<Customer>` element of the XML document. The `@XmlAttribute` annotation also has a `name()` attribute that allows you to specify the exact name of the XML attribute within the XML document. By default, it is the same name as the annotated field.

In our example, the `@javax.xml.bind.annotation.XmlElement` annotation was placed on the `fullname` field of our `Customer` class. This annotation tells JAXB to map the field to a `<fullname>` element within the main `<Customer>` element of the XML document. `@XmlElement` does have a `name()` attribute, so you can specify the exact string of the XML element. By default, it is the same name as the annotated field.

If we were to output an instance of our `Customer` class that had an `id` of 42 and a `name` of "Bill Burke," the outputted XML would look like this:

```
<customer id="42">
    <fullname>Bill Burke</fullname>
</customer>
```

You can also use the `@XmlElement` annotation to embed other JAXB-annotated classes. For example, let's say we wanted to add an `Address` class to our `Customer` class:

```
@XmlRootElement(name="address")
@XmlAccessorType(XmlAccessType.FIELD)
public class Address  {

    @XmlElement
    protected String street;

    @XmlElement
    protected String city;

    @XmlElement
    protected String state;

    @XmlElement
    protected String zip;

    // getters and setters

    ...
}
```

We would simply add a field to `Customer` that was of type `Address` as follows:

```
@XmlRootElement(name="customer")
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {

    @XmlAttribute
    protected int id;

    @XmlElement
    protected String name;

    @XmlElement
    protected Address address;

    public Customer() {}

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

If we were to output an instance of our new `Customer` class that had an `id` of 42, a `name` of "Bill Burke," a `street` of "200 Marlboro Street," `city` of "Boston," `state` of "MA," and `zip` of "02115," the outputted XML would look like this:

```
<customer id="42">
    <name>Bill Burke</name>
    <address>
        <street>200 Marlboro Street</street>
        <city>Boston</city>
        <state>MA</state>
        <zip>02115</zip>
    </address>
</customer>
```

There are a number of other annotations and settings that allow you to do some more complex Java-to-XML mappings. JAXB implementations are also required to have command-line tools that can automatically generate JAXB-annotated Java classes from XML schema documents. If you need to integrate with an existing XML schema, these autogeneration tools are the way to go.

To marshal Java classes to and from XML, you need to interact with the `javax.xml.bind.JAXBContext` class. `JAXBContext` instances introspect your classes to understand the structure of your annotated classes. They are used as factories for the `javax.xml.bind.Marshaller` and `javax.xml.bind.Unmarshaller` interfaces. `Marshaller` instances are used to take Java objects and output them as XML. `Unmarshaller` instances are used to take XML input and create Java objects out of it. Here's an example of using JAXB to write an instance of the `Customer` class we defined earlier into XML and then to take that XML and recreate the `Customer` object:

```
    Customer customer = new Customer();
    customer.setId(42);
    customer.setName("Bill Burke");

    JAXBContext ctx = JAXBContext.newInstance(Customer.class);
    StringWriter writer = new StringWriter();

    ctx.createMarshaller().marshal(customer, writer);

    String custString = writer.toString();

    customer = ctx.createUnmarshaller()
                  .unmarshal(new StringReader(custString));
```

We first create an initialized instance of a `Customer` class. We then initialize a
`JAXBContext` to understand how to deal with `Customer` classes. We use a `Marshaller`
instance created by the method `JAXBContext.createMarshaller()` to write the
`Customer` object into a Java string. Next we use the `Unmarshaller` created by the `JAXB`
`Context.createUnmarshaller()` method to recreate the `Customer` object with the XML
string we just created.

Now that we have a general idea how JAXB works, let's look at how JAX-RS integrates
with it.

JAXB JAX-RS Handlers

The JAX-RS specification requires implementations to automatically support the mar-
shalling and unmarshalling of classes that are annotated with `@XmlRootElement` or
`@XmlType` as well as objects wrapped inside `javax.xml.bind.JAXBElement` instances.
Here's an example that interacts using the `Customer` class defined earlier:

```
    @Path("/customers")
    public class CustomerResource {

        @GET
        @Path("{id}")
        @Produces("application/xml")
        public Customer getCustomer(@PathParam("id") int id) {

            Customer cust = findCustomer(id);
            return cust;
        }

        @POST
        @Consumes("application/xml")
        public void createCustomer(Customer cust) {
            ...
        }
    }
```

As you can see, once you've applied JAXB annotations to your Java classes, it is very
easy to exchange XML documents between your client and web services. The built-in

JAXB handlers will handle any JAXB-annotated class for media types of `application/ xml`, `text/xml`, or `application/*+xml`. By default, they will also manage the creation and initialization of `JAXBContext` instances. Because the creation of `JAXBContext` instances can be expensive, JAX-RS implementations usually cache them after they are first initialized.

Managing your own JAXBContexts with ContextResolvers

If you are already familiar with JAXB, you'll know that many times you need to configure your `JAXBContext` instances a certain way to get the output you desire. The JAX-RS built-in JAXB provider allows you to plug in your own `JAXBContext` instances. The way it works is that you have to implement a factory-like interface called `javax.ws.rs.ext.ContextResolver` to override the default `JAXBContext` creation:

```
public interface ContextResolver<T> {

    T getContext(Class<?> type);
}
```

`ContextResolvers` are pluggable factories that create objects of a specific type, for a certain Java type, and for a specific media type. To plug in your own `JAXBContext`, you will have to implement this interface. Here's an example of creating a specific `JAXBContext` for our `Customer` class:

```
@Provider
@Produces("application/xml")
public class CustomerResolver
                     implements ContextResolver<JAXBContext> {
    private JAXBContext ctx;

    public CustomerResolver() {
      this.ctx = ...; // initialize it the way you want
    }


    public JAXBContext getContext(Class<?> type) {
        if (type.equals(Customer.class)) {
          return ctx;
        } else {
          return null;
        }
    }
}
```

Your resolver class must implement `ContextResolver` with the parameterized type of `JAXBContext`. The class must also be annotated with the `@javax.ws.rs.ext.Provider` annotation to identify it as a JAX-RS component. In our example, the `Customer Resolver` constructor initializes a `JAXBContext` specific to our `Customer` class.

You register your `ContextResolver` using the `javax.ws.rs.core.Application` API discussed in Chapters 3 and 11. The built-in JAXB handler will see if there are any registered `ContextResolvers` that can create `JAXBContext` instances. It will iterate

through them, calling the `getContext()` method passing in the Java type it wants a
`JAXBContext` created for. If the `getContext()` method returns `null`, it will go on to the
next `ContextResolver` in the list. If the `getContext()` method returns an instance, it will
use that `JAXBContext` to handle the request. If there are no `ContextResolvers` found, it
will create and manage its own `JAXBContext`. In our example, the `CustomerResolver.get`
`Context()` method checks to see if the type is a `Customer` class. If it is, it returns `null`;
otherwise, it returns the `JAXBContext` we initialized in the constructor.

The `@Produces` annotation on your `CustomerResolver` implementation is optional. It
allows you to specialize a `ContextResolver` for a specific media type. You'll see in the
next section that you can use JAXB to output to formats other than XML. This is a way
to create `JAXBContext` instances for each individual media type in your system.

JAXB and JSON

JAXB is flexible enough to support formats other than XML. The Jettison[*] open source
project has written a JAXB adapter that can input and output the JSON format. JSON
is a text-based format that can be directly interpreted by JavaScript. It is the preferred
exchange format for Ajax applications. Although not required by the JAX-RS specifi-
cation, many JAX-RS implementations use Jettison to support marshalling JAXB an-
notated classes to and from JSON.

JSON is a much simpler format than XML. Objects are enclosed in curly brackets "{}"
and contain key/value pairs. Values can be quoted strings, Booleans (true or false),
numeric values, or arrays of these simple types. Here's an example:

```
{
  "id" : 42,
  "name" : "Bill Burke",
  "married" : true
  "kids" : [ "Molly", "Abby" ]
}
```

Key and value pairs are separated by colon character and delimited by commas. Arrays
are enclosed in brackets "[]." Here, our object has four properties, `id`, `name`, `married`,
and `kids`, with varying values.

XML to JSON using BadgerFish

As you can see, JSON is a much simpler format than XML. While XML has elements,
attributes, and namespaces, JSON only has name/value pairs. There has been some
work in the JSON community to produce a mapping between XML and JSON so that
one XML schema can output documents of both formats. The de facto standard,
BadgerFish, is a widely used XML-to-JSON mapping and is available in most JAX-RS
implementations that have JAXB/JSON support. Let's go over this mapping:

---

[*] For more information, see *http://jettison.codehaus.org*.

1. XML element names become JSON object properties and the text values of these elements are contained within a nested object that has a property named "$." So, if you had the XML `<customer>Bill Burke</customer>`, it would map to `{ "customer" : { "$" : "Bill Burke" }}`.

2. XML elements become properties of their base element. Suppose you had the following XML:

   ```
   <customer>
       <first>Bill</first>
       <last>Burke</last>
   </customer>
   ```

   The JSON mapping would look like:

   ```
   { "customer" :
       { "first" : { "$" : "Bill"},
         "last" : { "$" : "Burke" }
       }
   }
   ```

3. Multiple nested elements of the same name would become an array value. So, this XML:

   ```
   <customer>
       <phone>978-666-5555</phone>
       <phone>978-555-2233</phone>
   </customer
   ```

   would look like this in JSON:

   ```
   { "customer" :
       { "phone" : [ { "$", "978-666-5555"}, { "$", "978-555-2233"} ] }
   }
   ```

4. XML attributes become JSON properties prefixed with the @ character. So, if you had this XML:

   ```
   <customer id="42">
       <name>Bill Burke</name>
   </customer>
   ```

   the JSON mapping would look like:

   ```
   { "customer" :
       { "@id" : 42,
         "name" : "Bill Burke"
       }
   }
   ```

5. Active namespaces are contained in an `@xmlns` JSON property of the element. The "$" represents the default namespace. All nested elements and attributes would use the namespace prefix as part of their names. So, if we had this XML:

```
<customer xmlns="urn:cust" xmlns:address="urn:address">
   <name>Bill Burke</name>
   <address:zip>02115</address:zip>
</customer>
```

the JSON mapping would be:

```
{ "customer" :
    { "@xmlns" : { "$" : "urn:cust",
                   "address" : "urn:address" } ,
      "name" : { "$" : "Bill Burke",
                 "@xmlns" : { "$" : "urn:cust",
                              "address" : "urn:address" } },
      "address:zip" : { "$" : "02115",
                        "@xmlns" : { "$" : "urn:cust",
                        "address" : "urn:address" }}
    }
}
```

BadgerFish is kind of unnatural when writing JavaScript, but if you want to unify your formats under XML schema, it's the way to go.

JSON and JSON Schema

The thing with using XML schema and the BadgerFish mapping to define your JSON data structures is that it is very weird for JavaScript programmers to consume. If you do not need to support XML clients or if you want to provide a cleaner and simpler JSON representation, there are some options available for you.

It doesn't make much sense to use XML schema to define JSON data structures. The main reason is that JSON is a richer data format that supports things like maps, lists, numeric, Boolean, and string data. It is a bit quirky modeling these sorts of simple data structures with XML schema. To solve this problem the JSON community has come up with JSON schema. Here's an example of what it looks like when you define a JSON data structure representing our customer example:

```
{
  "description":"A customer",
  "type":"object",

  "properties":
    {"first": {"type": "string"},
     "last" : {"type" : "string"}
    }
}
```

The `description` property defines the description for the schema. The `type` property defines what is being described. Next, you define a list of properties that make up your object. I don't want to go into a lot of detail about JSON schema, so you should visit *http://www.json-schema.org* to get more information on this subject.

If you do a Google search on Java and JSON, you'll find a plethora of frameworks that help you marshal and unmarshal between Java and JSON. One particularly good one is the Jackson[†] framework. It has a prewritten JAX-RS content handler that can automatically convert Java beans to and from JSON. It also has the capability to generate JSON schema documents from a Java object model.

The way it works by default is that it introspects your Java class, looking for properties, and maps them into JSON. For example, if we had this Java class:

```java
public class Customer {
    private int id;
    private String firstName;
    private String lastName;

    public int getId() {
        return id;
    }

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

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

and sample data like this:

```
{
  "id" : 42,
  "firstName" : "Bill",
  "lastName" : "Burke"
}
```

reading in the data to create a `Customer` object would be as easy as this:

```java
ObjectMapper mapper = new ObjectMapper();
Customer cust = mapper.readValue(inputStream, Customer.class);
```

and writing the data would be as easy as this:

---

† *http://jackson.codehaus.org/*

```
ObjectMapper mapper = new ObjectMapper();
mapper.writeValue(outputStream, customer);
```

The Jackson framework's JAX-RS integration actually does all this work for you, so all you have to do in your JAX-RS classes is specify the output and input format as `application/json` when writing your JAX-RS methods.

Custom Marshalling

So far in this chapter, we've focused on built-in JAX-RS handlers that can marshal and unmarshal message content. Unfortunately, there are hundreds of data formats available on the Internet and the built-in JAX-RS handlers are either too low-level to be useful or may not match the format you need. Luckily, JAX-RS allows you to write your own handlers and plug them into the JAX-RS runtime.

To illustrate how to write your own handlers, we're going to pretend that there is no built-in JAX-RS JAXB support and instead write one ourselves using JAX-RS APIs.

MessageBodyWriter

The first thing we're going to implement is JAXB-marshalling support. To automatically convert Java objects into XML, we have to create a class that implements the `javax.ws.rs.ext.MessageBodyWriter` interface:

```
public interface MessageBodyWriter<T> {

    boolean isWriteable(Class<?> type, Type genericType,
                        Annotation annotations[],
                                    MediaType mediaType);


    long getSize(T t, Class<?> type, Type genericType,
                 Annotation annotations[], MediaType mediaType);


    void writeTo(T t, Class<?> type, Type genericType,
                 Annotation annotations[],
                 MediaType mediaType,
                 MultivaluedMap<String, Object> httpHeaders,
                 OutputStream entityStream)
                         throws IOException, WebApplicationException;
}
```

The `MessageBodyWriter` interface only has three methods. The `isWriteable()` method is called by the JAX-RS runtime to determine if the writer supports marshalling the given type. The `getSize()` method is called by the JAX-RS runtime to determine the `Content-Length` of the output. Finally, the `writeTo()` method does all the heavy lifting and writes the content out to the HTTP response buffer. Let's implement this interface to support JAXB:

```
@Provider
@Produces("application/xml")
public class JAXBMarshaller implements MessageBodyWriter {

    public boolean isWriteable(Class<?> type, Type genericType,
                    Annotation annotations[], MediaType mediaType) {
       return type.isAnnotationPresent(XmlRootElement.class);
    }
```

We start off the implementation of this class by annotating it with the @javax.ws.rs.ext.Provider annotation. This tells JAX-RS that this is a deployable JAX-RS component. We must also annotate it with @Produces to tell JAX-RS which media types this MessageBodyWriter supports. Here, we're saying that our JAXBMarshaller class supports application/xml.

The isWriteable() method is a callback method that tells the JAX-RS runtime whether or not the class can handle writing out this type. JAX-RS follows this algorithm to find an appropriate MessageBodyWriter to write out a Java object into the HTTP response:

1. First, a list of MessageBodyWriters is calculated by looking at each writer's @Produces annotation to see if it supports the media type that the JAX-RS resource method wants to output.

2. This list is sorted with the best match for the desired media type coming first. In other words, if our JAX-RS resource method wants to output application/xml and we have three MessageBodyWriters (one produces application/*, one supports anything */*, and the last supports application/xml), the one producing application/xml will come first.

3. Once this list is calculated, the JAX-RS implementation iterates through the list in order, calling the MessageBodyWriter.isWriteable() method. If the invocation returns true, that MessageBodyWriter is used to output the data.

The isWriteable() method takes four parameters. The first one is a java.lang.Class that is the type of the object that is being marshalled. The type is determined by calling the getClass() method of the object. In our example, we use this parameter to find out if our object's class is annotated with the @XmlRootElement annotation.

The second parameter is a java.lang.reflect.Type. This is generic type information about the object being marshalled. It is determined by introspecting the return type of the JAX-RS resource method. We don't use this parameter in our JAXBMarshaller.isWriteable() implementation. This parameter would be useful, for example, if we wanted to know the type parameter of a java.util.List generic type.

The third parameter is an array of java.lang.annotation.Annotation objects. These annotations are applied to the JAX-RS resource method we are marshalling the response for. Some MessageBodyWriters may be triggered by JAX-RS resource method annotations rather that class annotations. In our JAXBMarshaller class, we do not use this parameter in our isWriteable() implementation.

The fourth parameter is the media type that our JAX-RS resource method wants to produce.

Let's examine the rest of our `JAXBMarshaller` implementation:

```
public long getSize(Object obj, Class<?> type, Type genericType,
                    Annotation[] annotations, MediaType mediaType)
{
   return -1;
}
```

The `getSize()` method is responsible for determining the `Content-Length` of the response. If you cannot easily determine the length, just return −1. The underlying HTTP layer (i.e., a servlet container) will handle populating the `Content-Length` in this scenario or use the chunked transfer encoding.

The first parameter of `getSize()` is the actual object we are outputting. The rest of the parameters serve the same purpose as the parameters for the `isWriteable()` method.

Finally, let's look at how we actually write the JAXB object as XML:

```
public void writeTo(Object target,
                    Class<?> type,
                    Type genericType,
                    Annotation[] annotations,
                    MediaType mediaType,
                    MultivaluedMap<String, Object> httpHeaders,
                    OutputStream outputStream) throws IOException
{
   try {
      JAXBContext ctx = JAXBContext.newInstance(type);
      ctx.createMarshaller().marshal(target, outputStream);
   } catch (JAXBException ex) {
      throw new RuntimeException(ex);
   }
}
```

The `target`, `type`, `genericType`, `annotations`, and `mediaType` parameters of the `writeTo()` method are the same information passed into the `getSize()` and `isWriteable()` methods. The `httpHeaders` parameter is a `javax.ws.rs.core.Multiva luedMap` that represents the HTTP response headers. You may modify this map and add, remove, or change the value of a specific HTTP header as long as you do this before outputting the response body. The `outputStream` parameter is a `java.io.Output Stream` and is used to stream out the data.

Our implementation simply creates a `JAXBContext` using the `type` parameter. It then creates a `javax.xml.bind.Marshaller` and converts the Java object to XML.

Adding pretty printing

By default, JAXB outputs XML without any whitespace or special formatting. The XML output is all one line of text with no new lines or indentation. We may have human clients looking at this data, so we want to give our JAX-RS resource methods the option

to pretty-print the output XML. We will provide this functionality using an @Pretty annotation. For example:

```
@Path("/customers")
public class CustomerService {

    @GET
    @Path("{id}")
    @Produces("application/xml")
    @Pretty
    public Customer getCustomer(@PathParam("id") int id) {...}
}
```

Since the `writeTo()` method of our `MessageBodyWriter` has access to the `getCustomer()` method's annotations, we can implement this easily. Let's modify our `JAXBMarshaller` class:

```
public void writeTo(Object target,
                    Class<?> type,
                    Type genericType,
                    Annotation[] annotations,
                    MediaType mediaType,
                    MultivaluedMap<String, Object> httpHeaders,
                    OutputStream outputStream) throws IOException
{
    try {
        JAXBContext ctx = JAXBContext.newInstance(type);
        Marshaller m = ctx.createMarshaller();

        boolean pretty = false;
        for (Annotation ann : annotations) {
            if (ann.annotationType().equals(Pretty.class)) {
                pretty = true;
                break;
            }
        }
        if (pretty) {
            marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
        }

        m.marshal(target, outputStream);
    } catch (JAXBException ex) {
        throw new RuntimeException(ex);
    }
}
```

Here, we iterate over the `annotations` parameter to see if any of them are the `@Pretty` annotation. If `@Pretty` has been set, we set the `JAXB_FORMATTED_OUTPUT` property on the `Marshaller` so that it will format the XML with line breaks and indentation strings.

Pluggable JAXBContexts using ContextResolvers

Earlier in this chapter, we saw how you could plug in your own JAXBContext using the ContextResolver interface. Let's look at how we can add this functionality to our JAXBMarshaller class.

First, we need a way to locate a ContextResolver that can provide a custom JAXBContext. This is done through the **javax.ws.rs.ext.Providers** interface:

```
public interface Providers {

    <T> ContextResolver<T> getContextResolver(Class<T> contextType,
                                              MediaType mediaType);

    <T> MessageBodyReader<T>
    getMessageBodyReader(Class<T> type, Type genericType,
                         Annotation annotations[], MediaType mediaType);


    <T> MessageBodyWriter<T>
    getMessageBodyWriter(Class<T> type, Type genericType,
                         Annotation annotations[], MediaType mediaType);


    <T extends Throwable> ExceptionMapper<T>
    getExceptionMapper(Class<T> type);

}
```

We use the Providers.getContextResolver() method to find a ContextResolver. We inject a reference to a Providers object using the @Context annotation. Let's modify our JAXBMarshaller class to add this new functionality:

```
@Context
protected Providers providers;

public void writeTo(Object target,
                    Class<?> type,
                    Type genericType,
                    Annotation[] annotations,
                    MediaType mediaType,
                    MultivaluedMap<String, Object> httpHeaders,
                    OutputStream outputStream) throws IOException
{
    try {
        JAXBContext ctx = null;
        ContextResolver<JAXBContext> resolver =
            providers.getContextResolver(JAXBContext.class, mediaType);
        if (resolver != null) {
          ctx = resolver.getContext(type);
        }
        if (ctx == null) {
            // create one ourselves
            ctx = JAXBContext.newInstance(type);
        }
```

```
      ctx.createMarshaller().marshal(target, outputStream);
    } catch (JAXBException ex) {
      throw new RuntimeException(ex);
    }
  }
}
```

In our `writeTo()` method, we now use the `Providers` interface to find a
`ContextResolver` that can give us a customer `JAXBContext`. If one exists, we call
`resolver.getContext()`, passing in the type of the object we want a `JAXBContext` for.

The `ContextResolver` returned by `Providers.getContextResolver()` is actually a proxy
that sits in front of a list of `ContextResolvers` that can provide `JAXBContext` instances.
When `getContextResolver()` is invoked, the proxy iterates on this list, recalling
the `getContextResolver()` on that individual resolver in the list. If it returns a
`JAXBContext` instance, it returns that to the original caller; otherwise, it tries the next
resolver in this list.

MessageBodyReader

Now that we have written a `MessageBodyWriter` to convert a Java object into XML and
output it as the HTTP response body, let's write an unmarshaller that knows how to
convert HTTP XML request bodies back into a Java object. To do this, we need to use
the `javax.ws.rs.ext.MessageBodyReader` interface:

```
public interface MessageBodyReader<T> {

    boolean isReadable(Class<?> type, Type genericType,
                       Annotation annotations[], MediaType mediaType);

    T readFrom(Class<T> type, Type genericType,
               Annotation annotations[], MediaType mediaType,
               MultivaluedMap<String, String> httpHeaders,
               InputStream entityStream)
                        throws IOException, WebApplicationException;

}
```

The `MessageBodyReader` interface has only two methods. The `isReadable()` method is
called by the JAX-RS runtime when it is trying to find a `MessageBodyReader` to unmarshal
the message body of an HTTP request. The `readFrom()` method is responsible for cre-
ating a Java object from the HTTP request body.

Implementing a `MessageBodyReader` is very similar to writing a `MessageBodyWriter`. Let's
look at how we would implement one:

```
@Provider
@Consumes("application/xml")
public class JAXBUnmarshaller implements MessageBodyReader {

    public boolean isReadable(Class<?> type, Type genericType,
                       Annotation annotations[], MediaType mediaType) {
```

```
        return type.isAnnotationPresent(XmlRootElement.class);
    }
```

Our `JAXBUnmarshaller` class is annotated with `@Provider` and `@Consumes`. The latter annotation tells the JAX-RS runtime which media types it can handle. The matching rules for finding a `MessageBodyReader` are the same as the rules for matching `MessageBody Writer`. The difference is that `@Consumes` annotation is used instead of the `@Produces` annotation to correlate media types.

Let's now look at how we read and convert our HTTP message into a Java object:

```
Object readFrom(Class<Object>, Type genericType,
                Annotation annotations[], MediaType mediaType,
                MultivaluedMap<String, String> httpHeaders,
                InputStream entityStream)
                    throws IOException, WebApplicationException {

    try {
        JAXBContext ctx = JAXBContext.newInstance(type);
        Return ctx.createUnmarshaller().unmarshal(outputStream);
    } catch (JAXBException ex) {
        throw new RuntimeException(ex);
    }
}
```

The `readFrom()` method gives us access to the HTTP headers of the incoming request as well as a `java.io.InputStream` that represents the request message body. Here, we just create a `JAXBContext` based on the Java type we want to create and use a `javax.xml.bind.Unmarshaller` to extract it from the stream.

Life Cycle and Environment

By default, only one instance of each `MessageBodyReader`, `MessageBodyWriter`, or `ContextResolver` is created per application. If JAX-RS is allocating instances of these components (see Chapter 11), the classes of these components must provide a public constructor for which the JAX-RS runtime can provide all the parameter values. A public constructor may only include parameters annotated with the `@Context` annotation. For example:

```
@Provider
@Consumes("application/json")
public class MyJsonReader implements MessageBodyReader {

    public MyJsonReader(@Context Providers providers) {
        this.providers = providers;
    }
}
```

Whether or not the JAX-RS runtime is allocating the component instance, JAX-RS will perform injection into properly annotated fields and setter methods. Again, you can only inject JAX-RS objects that are found using the `@Context` annotation.

Wrapping Up

In this chapter, you learned that JAX-RS can automatically convert Java objects to a specific data type format and write it out as an HTTP response. It can also automatically read in HTTP request bodies and create specific Java objects that represent the request. JAX-RS has a number of built-in handlers, but you can also write your own customer marshallers and unmarshallers. Chapter 19 walks you through some sample code that you can use to test-drive many of the concepts and APIs introduced in this chapter.

Response Codes, Complex Responses, and Exception Handling

So far, the examples given in this book have been very clean and tidy. The JAX-RS resource methods we have written have looked like regular vanilla Java methods annotated with JAX-RS annotations. We haven't talked a lot about the default behavior of JAX-RS resource methods, particularly around HTTP response codes in success and failure scenarios. Also, in the real world, you can't always have things so neat and clean. Many times you need to send specific response headers to deal with complex error conditions. This chapter first discusses the default response codes that vanilla JAX-RS resource methods give. It then walks you through writing complex responses using JAX-RS APIs. Finally, it goes over how exceptions can be handled within JAX-RS.

Default Response Codes

The default response codes that JAX-RS uses are pretty straightforward. There is pretty much a one-to-one relationship to the behavior described in the HTTP 1.1 Method Definition* specification. Let's examine what the response codes would be for both success and error conditions for the following JAX-RS resource class:

```
@Path("/customers")
public clas CustomerResource {

    @Path("{id}")
    @GET
    @Produces("application/xml")
    public Customer getCustomer(@PathParam("id") int id) {...}

    @POST
    @Produces("application/xml")
    @Consumes("application/xml")
    public Customer create(Customer newCust) {...}
```

---

* For more information, see *www.w3.org/Protocols/rfc2616/rfc2616-sec9.html*.

```
    @PUT
    @Path("{id}")
    @Consumes("application/xml")
    public void update(@PathParam("id") int id, Customer cust) {...}

    @Path("{id}")
    @DELETE
    public void delete(@PathParam("id") int id) {...}
}
```

Successful Responses

Successful HTTP response code numbers range from 200 to 399. For the `create()`and `getCustomer()` methods of our `CustomerResource` class, they will return a response code of 200, "OK" if the `Customer` object they are returning is not null. If the return value is null, a successful response code of 204, "No Content" is returned. The 204 response is not an error condition. It just tells the client that everything went OK, but that there is no message body to look for in the response. If the JAX-RS resource method's return type is void, a response code of 204, "No Content" is returned. This is the case with our `update()` and `delete()` methods.

The HTTP specification is pretty consistent for the PUT, POST, GET, and DELETE methods. If a successful HTTP response contains a message body, 200, "OK" is the response code. If the response doesn't contain a message body, 204, "No Content" must be returned.

Error Responses

In our `CustomerResource` example, error responses are mostly driven by application code throwing an exception. We will discuss this exception handling later in this chapter. There are some default error conditions that we can talk about right now, though.

Standard HTTP error response code numbers range from 400 to 599. In our example, if a client mistypes the request URI, for example, to `customers`, it will result in the server not finding a JAX-RS resource method that can service the request. In this case, a 404, "Not Found" response code will be sent back to the client.

For our `getCustomer()` and `create()` methods, if the client requests a text/html response, the JAX-RS implementation will automatically return a 406, "Not Acceptable" response code with no response body. This means that JAX-RS has a relative URI path that matches the request, but doesn't have a JAX-RS resource method that can produce the client's desired response media type (Chapter 8 talks in detail about how clients can request certain formats from the server).

If the client invokes an HTTP method on a valid URI to which no JAX-RS resource method is bound, the JAX-RS runtime will send an error code of 405, "Method Not Allowed." So, in our example, if our client does a PUT, GET, or DELETE on

the /customers URI, it will get a 405 response because POST is the only supported method for that URI. The JAX-RS implementation will also return an `Allow` response header back to the client that contains a list of HTTP methods the URI supports. So, if our client did a `GET /customers` in our example, the server would send this response back:

```
HTTP/1.1 405, Method Not Allowed
Allow: POST
```

The exception to this rule is the HTTP `HEAD` and `OPTIONS` methods. If a JAX-RS resource method isn't available that can service `HEAD` requests for that particular URI, but there does exist a method that can handle GET, JAX-RS will invoke the JAX-RS resource method that handles GET and return the response from that minus the request body. If there is no existing method that can handle `OPTIONS`, the JAX-RS implementation is required to send back some meaningful, automatically generated response along with the `Allow` header set.

Complex Responses

Sometimes the web service you are writing can't be implemented using the default request/response behavior inherent in JAX-RS. For the cases in which you need to explicitly control the response sent back to the client, your JAX-RS resource methods can return instances of `javax.ws.rs.core.Response`:

```
public abstract class Response {

    public abstract Object getEntity();
    public abstract int getStatus();
    public abstract MultivaluedMap<String, Object> getMetadata();
    ...
}
```

The `Response` class is an abstract class that contains three simple methods. The `getEntity()` method returns the Java object you want converted into an HTTP message body. The `getStatus()` method returns the HTTP response code. The `getMetadata()` method is a `MultivaluedMap` of response headers.

`Response` objects cannot be created directly; instead, they are created from `javax.ws.rs.core.Response.ResponseBuilder` instances returned by one of the static helper methods of `Response`:

```
public abstract class Response {
...
    public static ResponseBuilder status(Status status) {...}
    public static ResponseBuilder status(int status) {...}
    public static ResponseBuilder ok() {...}
    public static ResponseBuilder ok(Object entity) {...}
    public static ResponseBuilder ok(Object entity, MediaType type) {...}
    public static ResponseBuilder ok(Object entity, String type) {...}
    public static ResponseBuilder ok(Object entity, Variant var) {...}
```

```
        public static ResponseBuilder serverError() {...}
        public static ResponseBuilder created(URI location) {...}
        public static ResponseBuilder noContent() {...}
        public static ResponseBuilder notModified() {...}
        public static ResponseBuilder notModified(EntityTag tag) {...}
        public static ResponseBuilder notModified(String tag) {...}
        public static ResponseBuilder seeOther(URI location) {...}
        public static ResponseBuilder temporaryRedirect(URI location) {...}
        public static ResponseBuilder notAcceptable(List<Variant> variants) {...}
        public static ResponseBuilder fromResponse(Response response) {...}
   ...
   }
```

If you want an explanation of each and every static helper method, the JAX-RS Javadocs are a great place to look. They generally center on the most common use cases for creating custom responses. For example:

```
    public static ResponseBuilder ok(Object entity, MediaType type) {...}
```

The `ok()` method here takes the Java object you want converted into an HTTP response and the `Content-Type` of that response. It returns a preinitialized `ResponseBuilder` with a status code of 200, "OK." The other helper methods work in a similar way, setting appropriate response codes and sometimes setting up response headers automatically.

The `ResponseBuilder` class is a factory that is used to create one individual `Response` instance. You store up state you want to use to create your response and when you're finished, you have the builder instantiate the `Response`:

```
    public static abstract class ResponseBuilder {

        public abstract Response build();
        public abstract ResponseBuilder clone();

        public abstract ResponseBuilder status(int status);
        public ResponseBuilder status(Status status) {...}

        public abstract ResponseBuilder entity(Object entity);
        public abstract ResponseBuilder type(MediaType type);
        public abstract ResponseBuilder type(String type);

        public abstract ResponseBuilder variant(Variant variant);
        public abstract ResponseBuilder variants(List<Variant> variants);

        public abstract ResponseBuilder language(String language);
        public abstract ResponseBuilder language(Locale language);

        public abstract ResponseBuilder location(URI location);
        public abstract ResponseBuilder contentLocation(URI location);

        public abstract ResponseBuilder tag(EntityTag tag);
        public abstract ResponseBuilder tag(String tag);

        public abstract ResponseBuilder lastModified(Date lastModified);
        public abstract ResponseBuilder cacheControl(CacheControl cacheControl);
```

```
    public abstract ResponseBuilder expires(Date expires);
    public abstract ResponseBuilder header(String name, Object value);

    public abstract ResponseBuilder cookie(NewCookie... cookies);
}
```

As you can see, `ResponseBuilder` has a lot of helper methods for initializing various response headers. I don't want to bore you with all the details, so check out the JAX-RS Javadocs for an explanation of each one. I'll be giving examples, using many of them throughout the rest of this book.

Now that we have a rough idea about creating custom responses, let's look at an example of a JAX-RS resource method setting some specific response headers:

```
@Path("/textbook")
public class TextBookService {

    @GET
    @Path("/restfuljava")
    @Produces("text/plain")
    public Response getBook() {

        String book = ...;
        ResponseBuilder builder = Response.ok(book);
        builder.language("fr")
                .header("Some-Header", "some value");

        return builder.build();
    }
}
```

Here, our `getBook()` method is returning a plain-text string that represents a book our client is interested in. We initialize the response body using the `Response.ok()` method. The status code of the `ResponseBuilder` is automatically initialized with 200. Using the `ResponseBuilder.language()` method, we then set the `Content-Language` header to be French. We then use the `ResponseBuilder.header()` method to set a custom response header. Finally, we create and return the `Response` object using the `Response Builder.build()` method.

One interesting thing to note about this code is that we never set the `Content-Type` of the response. Because we have already specified an `@Produces` annotation, the JAX-RS runtime will set the media type of the response for us.

Returning Cookies

JAX-RS also provides a simple class to represent new cookie values. This class is `javax.ws.rs.core.NewCookie`:

```
public class NewCookie extends Cookie {

    public static final int DEFAULT_MAX_AGE = -1;
```

```
    public NewCookie(String name, String value) {}

    public NewCookie(String name, String value, String path,
                       String domain, String comment,
                        int maxAge, boolean secure) {}

    public NewCookie(String name, String value, String path,
                       String domain, int version, String comment,
                        int maxAge, boolean secure) {}

    public NewCookie(Cookie cookie) {}

    public NewCookie(Cookie cookie, String comment,
                       int maxAge, boolean secure) {}

    public static NewCookie valueOf(String value)
                       throws IllegalArgumentException {}

    public String getComment() {}
    public int getMaxAge() {}
    public boolean isSecure() {}
    public Cookie toCookie() {}
}
```

The `NewCookie` class extends the `Cookie` class discussed in Chapter 5. To set response cookies, create instances of `NewCookie` and pass them to the method `ResponseBuilder.cookies()`. For example:

```
@Path("/myservice")
public class MyService {

    @GET
    public Response get() {

        NewCookie cookie = new NewCookie("key", "value);
        ResponseBuilder builder = Response.ok("hello", "text/plain");
        return builder.cookies(cookie).build();
    }
}
```

Here, we're just setting a cookie named `key` to the value `value`.

The Status Enum

Generally, developers like to have constant variables represent raw strings or numeric values within. For instance, instead of using a numeric constant to set a `Response` status code, you may want a static final variable to represent a specific code. The JAX-RS specification provides a Java enum called `javax.ws.rs.core.Response.Status` for this very purpose:

```
public enum Status {
    OK(200, "OK"),
    CREATED(201, "Created"),
    ACCEPTED(202, "Accepted"),
    NO_CONTENT(204, "No Content"),
```

```
        MOVED_PERMANENTLY(301, "Moved Permanently"),
        SEE_OTHER(303, "See Other"),
        NOT_MODIFIED(304, "Not Modified"),
        TEMPORARY_REDIRECT(307, "Temporary Redirect"),
        BAD_REQUEST(400, "Bad Request"),
        UNAUTHORIZED(401, "Unauthorized"),
        FORBIDDEN(403, "Forbidden"),
        NOT_FOUND(404, "Not Found"),
        NOT_ACCEPTABLE(406, "Not Acceptable"),
        CONFLICT(409, "Conflict"),
        GONE(410, "Gone"),
        PRECONDITION_FAILED(412, "Precondition Failed"),
        UNSUPPORTED_MEDIA_TYPE(415, "Unsupported Media Type"),
        INTERNAL_SERVER_ERROR(500, "Internal Server Error"),
        SERVICE_UNAVAILABLE(503, "Service Unavailable");

        public enum Family {
                INFORMATIONAL, SUCCESSFUL, REDIRECTION,
                CLIENT_ERROR, SERVER_ERROR, OTHER
        }

        public Family getFamily()

        public int getStatusCode()

        public static Status fromStatusCode(final int statusCode)
    }
```

Each `Status` enum value is associated with a specific family of HTTP response codes. These families are identified by the `Status.Family` Java enum. Codes in the 100 range are considered *informational*. Codes in the 200 range are considered *successful*. Codes in the 300 range are success codes, but fall under the *redirection* category. Error codes are in the 400 to 500 ranges. The 400s are *client errors* and 500s are *server errors*.

Both the `Response.status()` and `ResponseBuilder.status()` methods can accept a `Status` enum value. For example:

```
    @DELETE
    Response delete() {
        ...

        return Response.status(Status.GONE).build();
    }
```

Here, we're telling the client that the thing we want to delete is already gone (410).

javax.ws.rs.core.GenericEntity

When we're dealing with returning `Response` objects, we do have a problem with `MessageBodyWriter`s that are generic-type-sensitive. For example, what if our built-in JAXB `MessageBodyWriter` can handle lists of JAXB objects? The `isWriteable()` method of our JAXB handler needs to extract parameterized type information of the generic type of the response entity. Unfortunately, there is no easy way in Java to obtain generic

type information at runtime. To solve this problem, JAX-RS provides a helper class called `javax.ws.rs.core.GenericEntity`. This is best explained using an example:

```
@GET
@Produces("application/xml")
public Response getCustomerList() {
    List<Customer> list = new ArrayList<Customer>();
    list.add(new Customer(...));

    GenericEntity entity = new GenericEntity<List<Customer>>(list){};
    return Response.ok(entity);
}
```

The `GenericEntity` class is a Java generic template. What you do here is create an anonymous class that extends `GenericEntity`, initializing the `GenericEntity`'s template with the generic type you're using. If this looks a bit magical, it is. The creators of Java generics made things a bit difficult, so we're stuck with this solution.

Exception Handling

Errors can be reported to a client either by creating and returning the appropriate `Response` object or by throwing an exception. Application code is allowed to throw any checked (classes extending `java.lang.Exception`) or unchecked (classes extending `java.lang.RuntimeException`) exceptions they want. Thrown exceptions are handled by the JAX-RS runtime if you have registered an exception mapper. Exception mappers can convert an exception to an HTTP response. If the thrown exception is not handled by a mapper, it is propagated and handled by the container (i.e., servlet) JAX-RS is running within. JAX-RS also provides the `javax.ws.rs.WebApplicationException`. This can be thrown by application code and automatically processed by JAX-RS without having to write an explicit mapper. Let's look at how to use the `WebApplicationException` first. We'll then examine how to write your own specific exception mappers.

javax.ws.rs.WebApplicationException

JAX-RS has a built-in unchecked exception that applications can throw. This exception is preinitialized with either a `Response` or a particular status code:

```
public class WebApplicationException extends RuntimeException {

    public WebApplicationException() {...}
    public WebApplicationException(Response response) {...}
    public WebApplicationException(int status) {...}
    public WebApplicationException(Response.Status status) {...}
    public WebApplicationException(Throwable cause) {...}
    public WebApplicationException(Throwable cause,
                                          Response response) {...}
    public WebApplicationException(Throwable cause, int status) {...}
    public WebApplicationException(Throwable cause,
                                      Response.Status status) {...}
```

```
        public Response getResponse() {...]
    }
```

When JAX-RS sees that a `WebApplicationException` has been thrown by application code, it catches the exception and calls its `getResponse()` method to obtain a `Response` to send back to the client. If the application has initialized the `WebApplicationException` with a status code or `Response` object, that code or `Response` will be used to create the actual HTTP response. Otherwise, the `WebApplicationException` will return a status code of 500, "Internal Server Error" to the client.

For example, let's say we have a web service that allows clients to query for customers represented in XML:

```
@Path("/customers")
public class CustomerResource {

    @GET
    @Path("{id}")
    @Produces("application/xml")
    public Customer getCustomer(@PathParam("id") int id) {

        Customer cust = findCustomer(id);
        if (cust == null) {
          throw new WebApplicationException(Response.Status.NOT_FOUND);
        }
        return cust;
    }
}
```

In this example, if we do not find a `Customer` instance with the given ID, we throw a `WebApplicationException` that causes a 404, "Not Found" status code to be sent back to the client.

Exception Mapping

Many applications have to deal with a multitude of exceptions thrown from application code and third-party frameworks. Relying on the underlying servlet container to handle the exception doesn't give us much flexibility. Catching and then wrapping all these exceptions within `WebApplicationException` would become quite tedious. Alternatively, you can implement and register instances of `javax.ws.rs.ext.ExceptionMapper`. These objects know how to map a thrown application exception to a `Response` object:

```
public interface ExceptionMapper<E extends Throwable> {
{
    Response toResponse(E exception);
}
```

For example, one exception that is commonly thrown in Java Persistence (JPA)–based database applications is the `javax.persistence.EntityNotFoundException`. It is thrown when JPA cannot find a particular object in the database. Instead of writing code to

handle this exception explicitly, you could write an `ExceptionMapper` to handle this exception for you. Let's do that:

```java
@Provider
public class EntityNotFoundMapper
        implements ExceptionMapper<EntityNotFoundException> {

    public Response toResponse(EntityNotFoundException e) {
        return Response.status(Response.Status.NOT_FOUND).build();
    }
}
```

Our `ExceptionMapper` implementation must be annotated with the `@Provider` annotation. This tells the JAX-RS runtime that it is a component. The class implementing the `ExceptionMapper` interface must provide the parameterized type of the `ExceptionMapper`. JAX-RS uses this generic type information to match up thrown exceptions to `ExceptionMapper`s. Finally, the `toResponse()` method receives the thrown exception and creates a `Response` object that will be used to build the HTTP response.

JAX-RS supports exception inheritance as well. When an exception is thrown, JAX-RS will first try and find an `ExceptionMapper` for that exception's type. If it cannot find one, it will look for a mapper that can handle the exception's superclass. It will continue this process until there are no more superclasses to match against.

Finally, `ExceptionMapper`s are registered with the JAX-RS runtime using the deployment APIs discussed in Chapter 11.

Wrapping Up

In this chapter, you learned that JAX-RS has default response codes for both success and error conditions. For more complex responses, your JAX-RS resource methods can return `javax.ws.rs.core.Response` objects. JAX-RS has a few exception utilities. You can throw instances of `javax.ws.rs.WebApplicationException` or let the underlying servlet container handle the exception. Or, you can write an `ExceptionMapper` that can map a particular exception to an HTTP response. Chapter 20 walks you through some sample code that you can use to test-drive many of the concepts and APIs introduced in this chapter.

HTTP Content Negotiation

Within any meaningfully sized organization or on the Internet, SOA (service-oriented architecture) applications need to be flexible enough to handle and integrate with a variety of different clients and platforms. RESTful services have an advantage in this area because most programming languages can communicate with the HTTP protocol. This is not enough, though. Different clients need different formats in order to run efficiently. Java clients might like their data within an XML format. Ajax clients work a lot better with JSON. Ruby clients prefer YAML. Clients may also want internationalized data so that they can provide translated information to their English, Chinese, Japanese, Spanish, or French users. Finally, as our RESTful applications evolve, older clients need a clean way to interact with newer versions of our web services.

HTTP does have facilities to help with these types of integration problems. One of its most powerful features is a client's capability to specify to a server how it would like its responses formatted. The client can negotiate the content type of the message body, how it is encoded, and even which human language it wants the data translated into. This protocol is called HTTP Content Negotiation, or *conneg* for short. In this chapter, I'll explain how conneg works, how JAX-RS supports it, and most importantly how you can leverage this feature of HTTP within your RESTful web services.

Conneg Explained

The first part of HTTP Content Negotiation is that clients can request a specific media type they would like returned when querying a server for information. Clients set an `Accept` request header that is a comma-delimited list of preferred formats. For example:

```
GET http://example.com/stuff
Accept: application/xml, application/json
```

In this example request, the client is asking the server for `/stuff` formatted in either XML or JSON. If the server is unable to provide the desired format, it will respond with a status code of 406, "Not Acceptable." Otherwise, the server chooses one of the media types and sends a response in that format back to the client.

Wildcards and media type properties can also be used within the `Accept` header listing. For example:

```
GET http://example.com/stuff
Accept: text/*, text/html;level=1
```

The `text/*` media type means any text format.

Preference Ordering

The protocol also has both implicit and explicit rules for choosing a media type to respond with. The implicit rule is that more specific media types take precedence over less specific ones. Take this example:

```
GET http://example.com/stuff
Accept: text/*, text/html;level=1, */*, application/xml
```

The server assumes that the client always wants a concrete media type over a wildcard one, so the server would interpret the client preference as follows:

1. `text/html;level=1`
2. `application/xml`
3. `text/*`
4. `*/*`

The `text/html;level=1` type would come first because it is the most specific. The `application/xml` type would come next because it does not have any MIME type properties like `text/html;level=1` does. After this would come the wildcard types, with `text/*` coming first because it is obviously more concrete than the match-all qualifier `*/*`.

Clients can also be more specific on their preferences by using the `q` MIME type property. This property is a numeric value between 0.0 and 1.0, with 1.0 being the most preferred. For example:

```
GET http://example.com/stuff
Accept: text/*;q=0.9, */*;q=0.1, audio/mpeg, application/xml;q=0.5
```

If no `q` qualifier is given, then a value of `1.0` must be assumed. So, in our example request, the preference order is as follows:

1. `audio/mpeg`
2. `text/*`
3. `application/xml`
4. `*/*`

The `audio/mpeg` type is chosen first because it has an implicit qualifier of `1.0`. Text types come next as `text/*` has a qualifier of `0.9`. Even though `application/xml` is more specific, it has a lower preference value than `text/*` so it follows in the third spot. If none

of those types matches the formats the server can offer, anything can be passed back to the client.

Language Negotiation

HTTP Content Negotiation also has a simple protocol for negotiating the desired human language of the data sent back to the client. Clients use the `Accept-Language` header to specify which human language they would like to receive. For example:

```
GET http://example.com/stuff
Accept-Language: en-us, es, fr
```

Here, the client is asking for a response in English, Spanish, or French. The `Accept-Language` header uses coded format. Two digits represent a language identified by the ISO-639[*] standard. The code can be further specialized by following the two-character language code with an ISO-3166[†] two-character country code. In the previous example, `en-us` represents U.S. English.

The `Accept-Language` header also support preference qualifiers:

```
GET http://example.com/stuff
Accept-Language: fr;q=1.0, es;q=1.0, en=0.1
```

Here, the client prefers French or Spanish, but would accept English as the default translation.

Clients and servers use the `Content-Language` header to specify the human language for message body translation.

Encoding Negotiation

Clients can also negotiate the encoding of a message body. To save on network bandwidth, encodings are generally used to compress messages before they are sent. The most common algorithm for encoding is GZIP compression. Clients use the `Accept-Encoding` header to specify which encodings they support. For example:

```
GET http://example.com/stuff
Accept-Encoding: gzip, deflate
```

Here, the client is saying that it wants its response either compressed using GZIP or it wants it uncompressed (`deflate`).

The `Accept-Encoding` header also supports preference qualifiers:

```
GET http://example.com/stuff
Accept-Encoding: gzip;q=1.0, compress;0.5; deflate;q=0.1
```

---

[*] For more information, see *www.w3.org/WAI/ER/IG/ert/iso639.htm*.

[†] For more information, see *www.iso.org/iso/english_country_names_and_code_elements*.

Here, `gzip` is desired first, then `compress`, followed by `deflate`. In practice, clients use the `Accept-Encoding` header to tell the server which encoding formats they support and they really don't care which one the server uses.

When a client or server encodes a message body, it must set the `Content-Encoding` header. This tells the receiver which encoding was used.

JAX-RS and Conneg

The JAX-RS specification has a few facilities that help you manage conneg. It does method dispatching based on `Accept` header values. It allows you to view this content information directly. It also has complex negotiation APIs that allow you to deal with multiple decision points. Let's look into each of these.

Method Dispatching

In previous chapters, we saw how the `@Produces` annotation denotes which media type a JAX-RS method should respond with. JAX-RS also uses this information to dispatch requests to the appropriate Java method. It matches the preferred media types listed in the `Accept` header of the incoming request to the metadata specified in `@Produces` annotations. Let's look at a simple example:

```
@Path("/customers")
public class CustomerResource {

    @GET
    @Path("{id}")
    @Produces("application/xml")
    public Customer getCustomerXml(@PathParam("id") int id) {...}

    @GET
    @Path("{id}")
    @Produces("text/plain")
    public String getCustomerText(@PathParam("id") int id) {...}

    @GET
    @Path("{id}")
    @Produces("application/json")
    public Customer getCustomerJson(@PathParam("id") int id) {...}
}
```

Here, we have three methods that all service the same URI but produce different data formats. JAX-RS can pick one of these methods based on what is in the `Accept` header. For example, let's say a client made this request:

```
GET http://example.com/customers/1
Accept: application/json;q=1.0, application/xml;q=0.5
```

The JAX-RS provider would dispatch this request to the `getCustomerJson()` method.

Leveraging Conneg with JAXB

In Chapter 6, I showed you how to use JAXB annotations to map Java objects to and from XML and JSON. If you leverage JAX-RS integration with conneg, you can implement one Java method that can service both formats. This can save you from writing a whole lot of boilerplate code:

```
@Path("/service")
public class MyService {

    @GET
    @Produces({"application/xml", "application/json"})
    public Customer getCustomer(@PathParam("id") int id) {...}
}
```

In this example, our `getCustomer()` method produces either XML or JSON, as denoted by the `@Produces` annotation applied to it. The returned object is an instance of a Java class, `Customer`, which is annotated with JAXB annotations. Since most JAX-RS implementations support using JAXB to convert to XML or JSON, the information contained within our `Accept` header can pick which `MessageBodyWriter` to use to marshal the returned Java object.

Complex Negotiation

Sometimes simple matching of the `Accept` header with a JAX-RS method's `@Produces` annotation is not enough. Different JAX-RS methods that service the same URI may be able to deal with different sets of media types, languages, and encodings. Unfortunately, JAX-RS does not have the notion of either a `@ProduceLanguages` or `@ProduceEncodings` annotation. Instead, you must code this yourself by looking at header values directly or by using the JAX-RS API for managing complex conneg. Let's look at both.

Viewing Accept headers

In Chapter 5, you were introduced to `javax.ws.rs.core.HttpHeaders`, the JAX-RS utility interface. This interface contains some preprocessed conneg information about the incoming HTTP request:

```
public interface HttpHeaders {

    public List<MediaType> getAcceptableMediaTypes();

    public List<Locale> getAcceptableLanguages();
...
}
```

The `getAcceptableMediaTypes()` method contains a list of media types defined in the HTTP request's `Accept` header. It is preparsed and represented as a `javax.ws.rs.core.MediaType`. The returned list is also sorted based on the "q" values (explicit or implicit) of the preferred media types with the most desired listed first.

The `getAcceptableLanguages()` processes the HTTP request's `Accept-Language` header. It is preparsed and represented as a list of `java.util.Locale` objects. As with `getAcceptableMediaTypes()`, the returned list is sorted based on the "q" values of the preferred languages, with the most desired listed first.

You inject a reference to `HttpHeaders` using the `@javax.ws.rs.core.Context` annotation. Here's how your code might look:

```
@Path("/myservice")
public class MyService {

    @GET
    public Response get(@Context HttpHeaders headers) {

        MediaType type = headers.getAcceptableMediaTypes().get(0);
        Locale language = headers.getAcceptableLanguages().get(0);

        Object responseObject = ...;

        Response.ResponseBuilder builder = Response.ok(responseObject, type);
        builder.language(language);
        return builder.build();
    }
}
```

Here, we create `Response` using the `ResponseBuilder` interface using the desired media type and language pulled directly from the `HttpHeaders` injected object.

Variant processing

JAX-RS also has an API to deal with situations in which you have multiple sets of media types, languages, and encodings you have to match against. You can use the interface `javax.ws.rs.core.Request` and the class `javax.ws.rs.core.Variant` to perform these complex mappings. Let's look at the `Variant` class first:

```
package javax.ws.rs.core.Variant

public class Variant {

    public Variant(MediaType mediaType, Locale language, String encoding) {...}

    public Locale getLanguage() {...}

    public MediaType getMediaType() {...}

    public String getEncoding() {...}
}
```

The `Variant` class is a simple structure that contains one media type, one language, and one encoding. It represents a single set that your JAX-RS resource method supports. You build a list of these objects to interact with the Request interface:

```
package javax.ws.rs.core.Request

public interface Request {

    Variant selectVariant(List<Variant> variants) throws IllegalArgumentException;
...
}
```

The `selectVariant()` method takes in a list of `Variant` objects that your JAX-RS method supports. It examines the `Accept`, `Accept-Language`, and `Accept-Encoding` headers of the incoming HTTP request and compares them to the `Variant` list you provide to it. It picks the variant that best matches the request. More explicit instances are chosen before less explicit ones. The method will return null if none of the listed variants matches the incoming accept headers. Here's an example of using this API:

```
@Path("/myservice")
public class MyService {

    @GET
    Response getSomething(@Context Request request) {

        List<Variant> variants = new ArrayList();
        variants.add(new Variant(
                        new MediaType("application/xml"),
                        "en", "deflate"));

        variants.add(new Variant(
                        new MediaType("application/xml"),
                        "es", "deflate"));
        variants.add(new Variant(
                        new MediaType("application/json"),
                        "en", "deflate"));

        variants.add(new Variant(
                        new MediaType("application/json"),
                        "es", "deflate"));
        variants.add(new Variant(
                        new MediaType("application/xml"),
                        "en", "gzip"));

        variants.add(new Variant(
                        new MediaType("application/xml"),
                        "es", "gzip"));
        variants.add(new Variant(
                        new MediaType("application/json"),
                        "en", "gzip"));

        variants.add(new Variant(
                        new MediaType("application/json"),
                        "es", "gzip"));
```

```
        // Pick the variant
        Variant v = request.selectVariant(variants);
        Object entity = ...; // get the object you want to return

        ResponseBuilder builder = Response.ok(entity);
        builder.type(v.getMediaType())
               .language(v.getLanguage())
               .header("Content-Encoding", v.getEncoding());

        return builder.build();
    }
```

That's a lot of code to say that the getSomething() JAX-RS method supports XML, JSON, English, Spanish, deflated, and GZIP encodings. You're almost better off not using the selectVariant() API and doing the selection manually. Luckily, JAX-RS offers the javax.ws.rs.core.Variant.VariantBuilder class to make writing these complex selections easier:

```
    public static abstract class VariantListBuilder {

        public static VariantListBuilder newInstance() {...}

        public abstract VariantListBuilder mediaTypes(MediaType... mediaTypes);

        public abstract VariantListBuilder languages(Locale... languages);

        public abstract VariantListBuilder encodings(String... encodings);

        public abstract List<Variant> build();

        public abstract VariantListBuilder add();
    }
```

The VariantBuilder class allows you to add a series of media type, language, and encodings to it. It will then automatically create a list of variants that contains every possible combination of these objects. Let's rewrite our previous example using a VariantBuilder:

```
    @Path("/myservice")
    public class MyService {

        @GET
        Response getSomething(@Context Request request) {

            Variant.VariantBuilder vb = Variant.VariantBuilder.newInstance();
            vb.mediaTypes(new MediaType("application/xml"),
                          new MediaType("application/json"))
              .languages(new Locale("en"), new Locale("es"))
              .encodings("deflate", "gzip");

            List<Variant> variants = vb.build();

            // Pick the variant
            Variant v = request.selectVariant(variants);
```

```
        Object entity = ...; // get the object you want to return

        ResponseBuilder builder = Response.ok(entity);
        builder.type(v.getMediaType())
                .language(v.getLanguage())
                .header("Content-Encoding", v.getEncoding());

        return builder.build();
    }
```

You interact with `VariantBuilder` instances by calling the `mediaTypes()`, `languages()`, and `encodings()` methods. When you are done adding items, you invoke the `build()` method and it generates a `Variant` list containing all the possible combinations of items you built it with.

You might have the case where you want to build two or more different combinations of variants. The `VariantBuilder.add()` method allows you to delimit and differentiate between the combinatorial sets you are trying to build. When invoked, it generates a `Variant` list internally based on the current set of items added to it. It also clears its builder state so that new things added to the builder do not combine with the original set of data. Let's look at another example:

```
Variant.VariantBuilder vb = Variant.VariantBuilder.newInstance();
vb.mediaTypes(new MediaType("application/xml"),
              new MediaType("application/json"))
  .languages(new Locale("en"), new Locale("es"))
  .encodings("deflate", "gzip")
  .add()
  .mediaTypes(new MediaType("text/plain"))
  .languages(new Locale("en"), new Locale("es"), new Locale("fr"))
  .encodings("compress");
```

In this example, we want to add an additional set of variants that our JAX-RS method supports. Our JAX-RS resource method will now also support `text/plain` with English, Spanish, or French, but only the `compress` encoding. The `add()` method delineates between our original set and our new one.

You're not going to find a lot of use for the `Request.selectVariant()` API in the real world. First of all, content encodings are not something you're going to be able to easily work with in JAX-RS. If you wanted to deal with content encodings portably, you'd have to do all the streaming yourself. Most JAX-RS implementations have automatic support for encodings like GZIP anyway and you don't have to write any code for this.

Second, most JAX-RS services pick the response media type automatically based on the `@Produces` annotation and `Accept` header. I have never seen a case in which a given language is not supported for a particular media type. In most cases, you're solely interested in the language desired by the client. You can obtain this information easily through the `HttpHeaders.getAcceptableLanguages()` method.

Negotiation by URI Patterns

Conneg is a power feature of HTTP. The problem is that some clients, specifically browsers, do not support it. For example, the Firefox browser hardcodes the `Accept` header it sends to the web server it connects to as follows:

```
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

If you wanted to view a JSON representation of a specific URI through your browser, you would not be able to if JSON is not one of the preferred formats that your browser is hardcoded to accept.

A common pattern to support such clients is to embed conneg information within the URI instead of passing it along within an `Accept` header. Two examples are:

```
/customers/en-US/xml/3323
/customers/3323.xml.en-US
```

The content information is embedded within separate paths of the URI or as filename suffixes. In these examples, the client is asking for XML translated into English. You could model this within your JAX-RS resource methods by creating simple path parameter patterns within your `@Path` expressions. For example:

```
@Path("/customers/{id}.{type}.{language}")
@GET
public Customer getCustomer(@PathParam("id") int id,
                       @PathParam("type") String type,
                       @PathParam("language") String language) {...}
```

Before the JAX-RS specification went final, a facility revolving around the file name suffix pattern was actually defined as part of the specification. Unfortunately, the expert group could not agree on the full semantics of the feature, so it was removed. Many JAX-RS implementations still support this feature, so I think it is important to go over how it works.

The way the specification worked and the way many JAX-RS implementations now work is that you define a mapping between file suffixes, media types, and languages. An `xml` suffix maps to `application/xml`. An `en` suffix maps to `en-US`. When a request comes in, the JAX-RS implementation extracts the suffix and uses that information as the conneg data instead of any incoming `Accept` or `Accept-Language` header. Consider this JAX-RS resource class:

```
@Path("/customers")
public class CustomerResource {

    @GET
    @Produces("application/xml")
    public Customer getXml() {...}

    @GET
    @Produces("application/json")
    public Customer getJson() {...}
}
```

For this `CustomerService` JAX-RS resource class, if a request of `GET /customers.json` came in, the JAX-RS implementation would extract the `.json` suffix and remove it from the request path. It would then look in its media type mappings for a media type that matched `json`. In this case, let's say `json` mapped to `application/json`. It would use this information instead of the `Accept` header and dispatch this request to the `getJson()` method.

Leveraging Content Negotiation

Most of the examples so far in this chapter have used conneg simply to differentiate between well-known media types like XML and JSON. While this is very useful to help service different types of clients, it's not the main purpose of conneg. Your web services will evolve over time. New features will be added. Expanded datasets will be offered. Data formats will change and evolve. How do you manage these changes? How can you manage older clients that can only work with older versions of your services? Modeling your application design around conneg can address a lot of these issues. Let's discuss some of the design decisions you must make to leverage conneg when designing and building your applications.

Creating New Media Types

An important principle of REST is that the complexities of your resources are encapsulated within the data formats you are exchanging. While location information (URIs) and protocol methods remain fixed, data formats can evolve. This is a very important thing to remember and consider when you are planning how your web services are going to handle versioning.

Since complexity is confined to your data formats, clients can use media types to ask for different format versions. A common way to address this is to design your applications to define their own new media types. The convention is to combine a `vnd` prefix, the name of your new format, and a concrete media type suffix delimited by the "+" character. For example, let's say the company Red Hat had a specific XML format for its customer database. The media type name might look like this:

```
application/vnd.rht.customers+xml
```

The `vnd` prefix stands for vendor. The `rht` string in this example represents Red Hat and, of course, the `customers` string represents our customer database format. We end it with `+xml` to let users know that the format is XML-based. We could do the same with JSON as well:

```
application/vnd.rht.customers+json
```

Now that we have a base media type name for the Red Hat format, we can append versioning information to it so that older clients can still ask for older versions of the format:

```
application/vnd.rht.customers+xml;version=1.0
```

Here, we've kept the subtype name intact and used media type properties to specify version information. Specifying a version property within a custom media type is a common pattern to denote versioning information. As this customer data format evolves over time, we can bump the version number to support newer clients without breaking older ones.

Flexible Schemas

Using media types to version your web services and applications is a great way to mitigate and manage change as your web services and applications evolve over time. While embedding version information within the media type is extremely useful, it shouldn't be the primary way you manage change. When defining the initial and newer versions of your data formats, you should pay special attention to backward compatibility.

Take XML schema, for instance. Your initial schema should allow for extended or custom elements and attributes within each and every schema type in your data format definition. Here's the initial definition of a customer data XML schema:

```xml
<schema targetNamespace="http://www.example.org/customer"
        xmlns="http://www.w3.org/2001/XMLSchema">
   <element name="customer" type="customerType"/>
   <complexType name="customerType">
      <attribute name="id" use="required" type="string"/>
      <anyAttribute/>
      <element name="first" type="string" minOccurs="1"/>
      <element name="last" type="string" minOccurs="1"/>
      <any/>
   </complexType>
</schema>
```

In this example, the schema allows for adding any arbitrary attribute to the `id` attribute. It also allows documents to contain any XML element in addition to the `first` and `last` elements. If new versions of the customer XML data format retain the initial data structure, older clients that still use the older version of the schema can still validate and process newer versions of the format as they receive them.

As the schema evolves, new attributes and elements can be added, but they should be made optional. For example:

```xml
<schema targetNamespace="http://www.example.org/customer"
        xmlns="http://www.w3.org/2001/XMLSchema">
   <element name="customer" type="customerType"/>
   <complexType name="customerType">
      <attribute name="id" use="required" type="string"/>
      <anyAttribute/>
      <element name="first" type="string" minOccurs="1"/>
      <element name="last" type="string" minOccurs="1"/>
      <element name="street" type="string" minOccurs="0"/>
      <element name="city" type="string" minOccurs="0"/>
      <element name="state" type="string" minOccurs="0"/>
```

```
        <element name="zip" type="string" minOccurs="0"/>
        <any/>
    </complexType>
</schema>
```

Here, we have added the street, city, state, and zip elements to our schema, but have made them optional. This allows older clients to still PUT and POST older, yet valid, versions of the data format.

If you combine flexible, backward-compatible schemas with media type versions, you truly have an evolvable system of data formats. Clients that are version-aware can use the media type version scheme to request specific versions of your data formats. Clients that are not version-aware can still request and send the version of the format they understand.

Wrapping Up

In this chapter, you learned how HTTP Content Negotiation works and how you can write JAX-RS-based web services that take advantage of this feature. You saw how clients can provide a list of preferences for data format, language, and encoding. You also saw that JAX-RS has implicit and explicit ways for dealing with conneg. Finally, we discussed general architectural guidelines for modeling your data formats and defining your own media types. You can test-drive the code in this chapter by flipping to the back of this book and looking at Chapter 21.

HATEOAS

The Internet is commonly referred to as "the Web" because information is connected together through a series of hyperlinks embedded within HTML documents. These links create threads between interrelated websites on the Internet. Because of this, humans can "surf" the Web for interesting tidbits of related information by clicking through these links with their browsers. Search engines can crawl these links and create huge indexes of searchable data. Without them, the Internet would never have scaled. There would have been no way to easily index information and registering websites would have been a painful manual process.

Besides links, another key feature of the Internet is HTML forms. Sometimes a website wants you to fill out information to buy something or register for some service. The server is telling you, the client, what information it needs to complete an action described on the web page you are viewing. The browser renders the web page into a format that you can easily understand. You read the web page and fill out and submit the form. An HTML form is an interesting data format because it is a self-describing interaction between the client and server.

The architectural principle that describes linking and form submission is called HATEOAS. HATEOAS stands for Hypermedia As The Engine Of Application State. It is a little bit of a weird name for a key architecture principle, but we're stuck with it (my editor actually thought I was making the acronym up). The idea of HATEOAS is that your data format provides extra information on how to change the state of your application. On the Web, HTML links allow you to change the state of your browser. When reading a web page, a link tells you which possible documents (states) you can view next. When you click a link, your browser's state changes as it visits and renders a new web page. HTML forms, on the other hand, provide a way for you to change the state of a specific resource on your server. When you buy something on the Internet through an HTML form, you are creating two new resources on the server: a credit card transaction and an order entry.

HATEOAS and Web Services

How does HATEOAS relate to web services? When applying HATEOAS to web services, the idea is to embed links within your XML or JSON documents. While this can be as easy as inserting a URL as the value of an element or attribute, most XML-based RESTful applications use syntax from the Atom[*] Syndication Format as a means to implement HATEOAS. From the Atom RFC:

> Atom is an XML-based document format that describes lists of related information known as "feeds." Feeds are composed of a number of items, known as "entries," each with an extensible set of attached metadata.

Think of Atom as the next evolution of RSS. It is generally used to publish blog feeds on the Internet, but a few data structures within the format are particularly useful for web services, particularly Atom links.

Atom Links

The Atom link XML type is a very simple, yet standardized way of embedding links within your XML documents. Let's look at an example:

```
<customers>
    <link rel="next"
          href="http://example.com/customers?start=2&size=2"
          type="application/xml"/>
    <customer id="123">
        <name>Bill Burke</name>
    </customer>
    <customer id="332">
        <name>Roy Fielding</name>
    </customer>
</customers>
```

The Atom link is just a simple XML element with a few specific attributes.

The `rel` *attribute*
> The `rel` attribute is used for link relationships. It is the logical, simple name used to reference the link. This attribute gives meaning to the URL you are linking to, much in the same way that text enclosed in an HTML `<a>` element gives meaning to the URL you can click in your browser.

The `href` *attribute*
> This is the URL you can traverse to get new information or change the state of your application.

The `type` *attribute*
> This is the exchanged media type of the resource the URL points to.

---

[*] For more information, see *www.w3.org/2005/Atom*.

The `hreflang` *attribute*
> Although not shown in the example, this attribute represents the language the data format is translated into. Some examples are French, English, German, and Spanish.

When a client receives a document with embedded Atom links, it looks up the relationship it is interested in and invokes the URI embedded within the `href` link attribute.

Advantages of Using HATEOAS with Web Services

It is pretty obvious why links and forms have done so much to make the Web so prevalent. With one browser, we have a window to a wide world of information and services. Search engines crawl the Internet and index websites, so all that data is at our fingertips. This is all possible because the Web is self-describing. We get a document and we know how to retrieve additional information by following links. We know how to purchase something from Amazon because the HTML form tells us how.

Machine-based clients are a little different, though. Other than browsers, there aren't a lot of generic machine-based clients that know how to interpret self-describing documents. They can't make decisions on the fly like humans can. They require programmers to tell them how to interpret data received from a service and how to transition to other states in the interaction between client and server. So, does that make HATEOAS useless to machine-based clients? Not at all. Let's look at some of the advantages.

Location transparency

One feature that HATEOAS provides is location transparency. In a RESTful system that leverages HATEOAS, very few URIs are published to the outside world. Services and information are represented within links embedded in the data formats returned by accessing these top-level URIs. Clients need to know the logical link names to look for, but don't have to know the actual network locations of the linked services.

For those of you who have written EJBs, this isn't much different than using JNDI. Like a naming service, links provide a level of indirection so that underlying services can change their locations on the network without breaking client logic and code. HATEOAS has an additional advantage in that the top-level web service has control over which links are transferred.

Decoupling interaction details

Consider a request that gives us a list of customers in a customer database: `GET /customers`. If our database has thousands and thousands of entries, we do not want to return them all with one basic query. What we could do is define a RESTful view into our database using URI query parameters:

```
/customers?start={startIndex}&size={numberReturned}
```

The `start` query parameter identifies the starting index for our customer list. The `size` parameter specifies how many customers we want returned from the query.

This is all well and good, but what we've just done is increased the amount of predefined knowledge the client must have to interact with the service beyond a simple URI of `/customers`. Let's say in the future, the server wanted to change how view sets are queried. For instance, maybe the customer database changes rather quickly and a start index isn't enough information anymore to calculate the view. If the service changes the interface, we've broken older clients.

Instead of publishing this RESTful interface to viewing our database, what if, instead, we embedded this information within the returned document?

```
<customers>
   <link rel="next"
         href="http://example.com/customers?start=2&size=2"
         type="application/xml"/>
   <customer id="123">
      <name>Bill Burke</name>
   </customer>
   <customer id="332">
      <name>Roy Fielding</name>
   </customer>
</customers>
```

By embedding an Atom link within a document, we've given a logical name to a state transition. The state transition here is the next set of customers within the database. We are still requiring the client to have predefined knowledge about how to interact with the service, but the knowledge is much simpler. Instead of having to remember which URI query parameters to set, all that's needed is to follow a specific named link. The client doesn't have to do any bookkeeping of the interaction. It doesn't have to remember which section of the database it is currently viewing.

Also, this returned XML is self-contained. What if we were to hand off this document to a third party? We would have to tell the third party that it is only a partial view of the database and specify the start index. Since we now have a link, this information is all a part of the document.

By embedding an Atom link, we've decoupled a specific interaction between the client and server. We've made our web service a little more transparent and change-resistant because we've simplified the predefined knowledge the client must have to interact with the service. Finally, the server has the power to guide the client through interactions by providing links.

Reduced state transition errors

Links are not used only as a mechanism to aggregate and navigate information. They can also be used to change the state of a resource. Consider an order in an e-commerce website obtained by traversing the URI `/orders/333`:

```
<order id="333">
  <customer id="123">...</customer>
  <amount>$99.99</amount>
  <order-entries>
    ...
  </order-entries>
</order>
```

Let's say a customer called up and wanted to cancel her order. We could simply do an HTTP DELETE on `/orders/333`. This isn't always the best approach, as we usually want to retain the order for data warehousing purposes. So instead, we might PUT a new representation of the order with a `cancelled` element sent to true:

```
PUT /orders/333 HTTP/1.1
Content-Type: application/xml

<order id="333">
  <customer id="123">...</customer>
  <amount>$99.99</amount>
  <cancelled>true</cancelled>
  <order-entries>
    ...
  </order-entries>
</order>
```

But what happens if the order can't be cancelled? We may be at a certain state in our order process where such an action is not allowed. For example, if the order has already been shipped, it cannot be cancelled. In this case, there really isn't a good HTTP status code to send back that represents the problem. A better approach would be to embed a `cancel` link:

```
<order id="333">
  <customer id="123">...</customer>
  <amount>$99.99</amount>
  <cancelled>false</cancelled>
  <link rel="cancel"
        href="http://example.com/orders/333/cancelled"/>
  <order-entries>
    ...
  </order-entries>
</order>
```

The client would do a `GET /orders/333` and get the XML document representing the order. If the document contains the `cancel` link, the client is allowed to change the order status to "cancelled" by doing an empty POST or PUT to the URI referenced in the link. If the document doesn't contain the link, the client knows that this operation is not possible. This allows the web service to control how the client is able to interact with it in real time.

W3C standardized relationships

An interesting thing that is happening in the REST community is an effort to define, register, and standardize[†] a common set of link relationship names and their associated behaviors. Some examples are given in Table 9-1.

Table 9-1. W3C standard relationship names

Relationship	Description
previous	A URI that refers to the immediately preceding document in a series of documents.
next	A URI that refers to the immediately following document in a series of documents.
edit	A URI that can be retrieved, updated, and deleted.
payment	A URI where payment is accepted. It is meant as a general way to facilitate acts of payment.

This is not an exhaustive list, but hopefully you get the general idea where this registry is headed. Registered relationships can go a long way to help make data formats even more self-describing and intuitive to work with.

Link Headers Versus Atom Links

While Atom links have become very popular for publishing links in RESTful systems, there is an alternative. Instead of embedding a link directly in your document, you can instead use `Link`[‡] response headers. This is best explained with an example.

Consider the order cancel example described in the previous section. An Atom link is used to specify whether or not the cancelling of an order is allowed and which URL to use to do a POST that will cancel the order. Instead of using an Atom link embedded within the Order XML document, let's use a `Link` header. So, if a user does a `GET /orders/333`, he will get back the following HTTP response:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Link: <http://example.com/orders/333/cancelled>; rel=cancel

<order id="333">
  ...
</order>
```

The `Link` header has all the same characteristics as an Atom link. The URI is enclosed within <> followed by one or more attributes delimited by semicolons. The `rel` attribute is required and means the same thing as the corresponding Atom attribute of the same name. This part isn't shown in the example, but you may also specify a media type via the `type` attribute.

---

† For more information, see *www.iana.org/assignments/link-relations/link-relations.xhtml*.

‡ For more information, see *http://tools.ietf.org/html/draft-nottingham-http-link-header-03*.

Personally, I really like `Link` headers as an alternative to embedding Atom links. Many times, I find that my client isn't interested in the resource representation and is only interested in the link relations. You shouldn't have to parse a whole XML or JSON document just to find the URL you're interested in invoking on. Another nice thing is that instead of doing a GET invocation, you can do a HEAD invocation and avoid getting the XML document entirely. In general, I like to use Atom links for data aggregation and `Link` headers for everything else.

HATEOAS and JAX-RS

JAX-RS doesn't have many facilities to help with HATEOAS. HATEOAS is defined by the application, so there's not much a framework can add. What it does have, though, are helper classes that you can use to build the URIs that you link to in your data formats.

Building URIs with UriBuilder

One such helper class is `javax.ws.rs.core.UriBuilder`. The `UriBuilder` class allows you to construct a URI piece by piece and is also sensitive to template parameters:

```
public abstract class UriBuilder {
    public static UriBuilder fromUri(URI uri)
                                   throws IllegalArgumentException
    public static UriBuilder fromUri(String uri)
                                   throws IllegalArgumentException
    public static UriBuilder fromPath(String path)
                                   throws IllegalArgumentException
    public static UriBuilder fromResource(Class<?> resource)
                                   throws IllegalArgumentException
```

`UriBuilder` instances can only be instantiated from the static helper methods listed. They can be initialized by a URI, path, or the `@Path` annotation of a JAX-RS resource class:

```
public abstract UriBuilder clone();
public abstract UriBuilder uri(URI uri)
                               throws IllegalArgumentException;
public abstract UriBuilder scheme(String scheme)
                               throws IllegalArgumentException;
public abstract UriBuilder schemeSpecificPart(String ssp)
                               throws IllegalArgumentException;

public abstract UriBuilder userInfo(String ui);
public abstract UriBuilder host(String host)
                               throws IllegalArgumentException;
public abstract UriBuilder port(int port)
                               throws IllegalArgumentException;
public abstract UriBuilder replacePath(String path);
public abstract UriBuilder path(String path)
                               throws IllegalArgumentException;
```

```
        public abstract UriBuilder path(Class resource)
                                    throws IllegalArgumentException;
        public abstract UriBuilder path(Class resource, String method)
                                    throws IllegalArgumentException;
        public abstract UriBuilder path(Method method)
                                    throws IllegalArgumentException;
        public abstract UriBuilder segment(String... segments)
                                    throws IllegalArgumentException;
        public abstract UriBuilder replaceMatrix(String matrix)
                                    throws IllegalArgumentException;
        public abstract UriBuilder matrixParam(String name, Object... vals)
                                    throws IllegalArgumentException;
        public abstract UriBuilder replaceMatrixParam(String name,
                    Object... values) throws IllegalArgumentException;
        public abstract UriBuilder replaceQuery(String query)
                                    throws IllegalArgumentException;
        public abstract UriBuilder queryParam(String name, Object... values)
                                    throws IllegalArgumentException;
        public abstract UriBuilder replaceQueryParam(String name,
                    Object... values) throws IllegalArgumentException;
        public abstract UriBuilder fragment(String fragment);
```

These methods are used to piece together various parts of the URI. You can set the values of a specific part of a URI directly or by using the `@Path` annotation values declared on JAX-RS resource methods. Both string values and `@Path` expressions are allowed to contain template parameters:

```
        public abstract URI buildFromMap(Map<String, ? extends Object> values)
                    throws IllegalArgumentException, UriBuilderException;
        public abstract URI buildFromEncodedMap(
                    Map<String, ? extends Object> values)
                throws IllegalArgumentException, UriBuilderException;
        public abstract URI build(Object... values)
                throws IllegalArgumentException, UriBuilderException;
        public abstract URI buildFromEncoded(Object... values)
                throws IllegalArgumentException, UriBuilderException;
    }
```

The `build()` methods create the actual URI. Before building the URI, though, any template parameters you have defined must be filled in. The `build()` methods take either a map of name/value pairs that can match up to named template parameters or you can provide a list of values that will replace template parameters as they appear in the templated URI. These values can either be encoded or decoded values, your choice. Let's look at a few examples:

```
    UriBuilder builder = UriBuilder.fromPath("/customers/{id}");
    builder.scheme("http")
            .host("{hostname}")
            .queryParam("param={param}");
```

In this code block, we have defined a URI pattern that looks like this:

```
    http://{hostname}/customers/{id}?param={param}
```

Since we have template parameters, we need to initialize them with values passed to one of the build arguments to create the final URI. If you want to reuse this builder, you should `clone()` it before calling a `build()` method, as the template parameters will be replaced in the internal structure of the object:

```
UriBuilder clone = builder.clone();
URI uri = clone.build("example.com", "333", "value");
```

This code would create a URI that looks like this:

```
http://example.com/customers/333?param=value
```

We can also define a map that contains the template values:

```
Map<String, Object> map = new HashMap<String, Object>();
map.put("hostname", "example.com");
map.put("id", 333);
map.put("param", "value");

UriBuilder clone = builder.clone();
URI uri = clone.buildFromMap(map);
```

Another interesting example is to create a URI from the `@Path` expressions defined in a JAX-RS annotated class. Here's an example of a JAX-RS resource class:

```
@Path("/customers")
public class CustomerService {

    @Path("{id}")
    public Customer getCustomer(@PathParam("id") int id) {...}
}
```

We can then reference this class and the `getCustomer()` method within our `UriBuilder` initialization to define a new template:

```
UriBuilder builder = UriBuilder.fromClass(CustomerService.class);
builder.host("{hostname}")
builder.path(CustomerService.class, "getCustomer");
```

This builder code defines a URI template with a variable hostname and the patterns defined in the `@Path` expressions of the `CustomerService` class and the `getCustomer()` method. The pattern would look like this in the end:

```
http://{hostname}/customers/{id}
```

You can then build a URI from this template using one of the `build()` methods discussed earlier.

Relative URIs with UriInfo

When writing services that distribute links, there's certain information that you cannot know at the time you write your code. Specifically, you will probably not know the hostnames of the links. Also, if you are linking to other JAX-RS services, you may not know the base paths of the URIs, as you may be deployed within a servlet container.

While there are ways to write your applications to get this base URI information from configuration data, JAX-RS provides a cleaner, simpler way through the use of the `javax.ws.rs.core.UriInfo` interface. You were introduced to a few features of this interface in Chapter 5. Besides basic path information, you can also obtain `UriBuilder` instances preinitialized with the base URI used to define all JAX-RS services or the URI used to invoke the current HTTP request:

```java
public interface UriInfo {
    public URI getRequestUri();
    public UriBuilder getRequestUriBuilder();
    public URI getAbsolutePath();
    public UriBuilder getAbsolutePathBuilder();
    public URI getBaseUri();
    public UriBuilder getBaseUriBuilder();
```

For example, let's say you have a JAX-RS service that exposes the customers in a customer database. Instead of having a base URI that returns all customers in a document, you want to embed `previous` and `next` links so that you can navigate through subsections of the database (I described an example of this earlier in this chapter). You will want to create these link relations using the URI to invoke the request:

```java
@Path("/customers")
public class CustomerService {

    @GET
    @Produces("application/xml")
    public String getCustomers(@Context UriInfo uriInfo) {

        UriBuilder nextLinkBuilder = uriInfo.getAbsolutePathBuilder();
        nextLinkBuilder.queryParam("start", 5);
        nextLinkBuilder.queryParam("size", 10);
        URI next = nextLinkBuilder.build();

        ... set up the rest of the document ...
    }
```

To get access to a `UriInfo` instance that represents the request, we use the `@javax.ws.rs.core.Context` annotation to inject it as a parameter to the JAX-RS resource method `getCustomers()`. Within `getCustomers()`, we call `uriInfo.getAbsolutePathBuilder()` to obtain a preinitialized `UriBuilder`. Depending on how this service was deployed, the URI created might look like this:

```
http://example.com/jaxrs/customers?start=5&size=10
```

There are other interesting tidbits available for building your URIs. In Chapter 4, I talked about the concept of subresource locators and subresources. Code running within a subresource can obtain partial URIs for each JAX-RS class and method that matches the incoming requests. It can get this information for these methods on `UriInfo`:

```
    public interface UriInfo {
...
       public List<String> getMatchedURIs();
       public List<String> getMatchedURIs(boolean decode);
    }
```

So, for example, lets reprint the subresource locator example in Chapter 4:

```
    @Path("/customers")
    public class CustomerDatabaseResource {

       @Path("{database}-db")
       public CustomerResource getDatabase(@PathParam("database") String db) {
          Map map = ...; // find the database based on the db parameter
          return new CustomerResource(map);
       }
    }
```

CustomerDatabaseResource is the subresource locator. Let's also reprint the subresource example from Chapter 4 with a minor change using these getMatchedURIs() methods:

```
    public class CustomerResource {
       private Map customerDB;

       public CustomerResource(Map db) {
          this.customerDB = db;
       }

       @GET
       @Path("{id}")
       @Produces("application/xml")
       public StreamingOutput getCustomer(@PathParam("id") int id,
                                          @Context UriInfo uriInfo) {

          for(String uri : uriInfo.getMatchedURIs()) {
            System.out.println(uri);
          }
          ...
       }
    }
```

If the request is GET http://example.com/customers/usa-db/333, the output of the for-loop in the getCustomer() method would print out the following:

```
    http://example.com/customers
    http://example.com/customers/usa-db
    http://example.com/customers/usa-db/333
```

The matched URIs correspond to the @Path expressions on the following:

- CustomerDatabaseResource
- CustomerDatabaseResource.getDatabase()
- CustomerResource.getCustomer()

Honestly, I had a very hard time coming up with a use case for the `getMatchedURIs()` methods, so I can't really tell you why you might want to use them.

The final method in `UriInfo` of this category is the `getMatchedResources()` method:

```
public interface UriInfo {
...
   public List<Object> getMatchedResources();
}
```

This method returns a list of JAX-RS resource objects that have serviced the request. Let's modify our `CustomerResource.getCustomer()` method again to illustrate how this method works:

```
public class CustomerResource {
  private Map customerDB;

  public CustomerResource(Map db) {
     this.customerDB = db;
  }

  @GET
  @Path("{id}")
  @Produces("application/xml")
  public StreamingOutput getCustomer(@PathParam("id") int id,
                                     @Context UriInfo uriInfo) {

     for(Object match : uriInfo.getMatchedResources()) {
       System.out.println(match.getClass().getName());
     }
     ...
  }
}
```

The for-loop in `getCustomer()` prints out the class names of the JAX-RS resource objects that were used to process the request. If the request is GET `http://example.com/customers/usa-db/333`, the output of the for-loop would be:

```
com.acme.CustomerDatabaseResource
com.acme.CustomerResource
```

Again, I'm hard pressed to find a use case for this method, but it's in the specification and you should be aware of it.

Wrapping Up

In this chapter, we discussed how links and forms have allowed the Web to scale. You learned the advantages of applying HATEOAS to RESTful web service design. Finally, you saw some JAX-RS utilities that can help make enabling HATEOAS in your JAX-RS services easier. Chapter 22 contains some code you can use to test-drive many of the concepts in this chapter.

Scaling JAX-RS Applications

When studying the Web, one can't help but notice how massively scalable it is. There are hundreds of thousands of websites and billions of requests per day traveling across it. Terabytes of data are downloaded from the Internet every hour. Websites like Amazon and Bank of America process millions of transactions per day. In this chapter, I'll discuss some features of the Web, specifically within HTTP, that make it more scalable and how you can take advantage of these features within JAX-RS applications.

Caching

Caching is one of the more important features of the Web. When you visit a website for the first time, your browser stores images and static text in memory and on disk. If you revisit the site within minutes, hours, days, or even months, your browser doesn't have to reload the data over the network and can instead pick it up locally. This greatly speeds up the rendering of revisited web pages and makes the browsing experience much more fluid. Browser caching not only helps page viewing, it also cuts down on server load. If the browser is obtaining images or text locally, it is not eating up scarce server bandwidth or CPU cycles.

Besides browser caching, there are also proxy caches. Proxy caches are pseudo web servers that work as middlemen between browsers and websites. Their sole purpose is to ease the load on master servers by caching static content and serving it to clients directly, bypassing the main servers. Content delivery networks (CDNs) like Akamai have made multimillion-dollar businesses out of this concept. These CDNs provide you with a worldwide network of proxy caches that you can use to publish your website and scale to hundreds of thousand of users.

If your web services are RESTful, there's no reason you can't leverage the caching semantics of the Web within your applications. If you have followed the HTTP constrained interface religiously, any service URI that can be reached with an HTTP GET is a candidate for caching, as they are, by definition, read-only and idempotent.

So when do you cache? Any service that provides static unchanging data is an obvious candidate. Also, if you have more dynamic data that is being accessed concurrently, you may also want to consider caching, even if your data is valid for only a few seconds or minutes. For example, consider the free stock quote services available on many websites. If you read the fine print, you'll see that these stock quotes are between 5 and 15 minutes old. Caching is viable in this scenario because there is a high chance that a given quote is accessed more than once within the small window of validity. So, even if you have dynamic web services, there's still a good chance that web caching is viable for these services.

HTTP Caching

Before we can leverage web caching, proxy caches, and CDNs for our web services, we need to understand how caching on the Web works. The HTTP protocol defines a rich set of built-in caching semantics. Through the exchange of various request and response headers, the HTTP protocol gives you fine-grained control over the caching behavior of both browser and proxy caches. The protocol also has validation semantics to make managing caches much more efficient. Let's dive into the specifics.

Expires Header

How does a browser know when to cache? In HTTP 1.0, a simple response header called `Expires` tells the browser that it can cache and for how long. The value of this header is a date in the future when the data is no longer valid. When this date is reached, the client should no longer use the cached data and reretrieve the data from the server. For example, if a client did a `GET /customers/123`, an example response using the `Expires` header would look like this:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Expires: Tue, 15 May 2010 16:00 GMT

<customer id="123">...</customers>
```

This cacheable XML data is valid until Tuesday, May 15, 2010.

We can implement this within JAX-RS by using a `javax.ws.rs.core.Response` object. For example:

```
@Path("/customers")
public class CustomerResource {

    @Path("{id}")
    @GET
    @Produces("application/xml")
    public Response getCustomer(@PathParam("id") int id) {
        Customer cust = findCustomer(id);
        ResponseBuilder builder = Response.ok(cust, "application/xml");
        Date date = Calendar.getInstance().set(2010, 5, 15, 16, 0);
```

```
        builder.expires(date);
        return builder.build();
    }
```

In this example, we initialize a `java.util.Date` object and pass it to the `Response Builder.expires()` method. This method sets the `Expires` header to the string date format the header expects.

Cache-Control

HTTP caching semantics were completely redone for the HTTP 1.1 specification. It includes a much richer feature set that has more explicit controls over browser and CDN/proxy caches. The idea of cache revalidation was also introduced. To provide all this new functionality, the `Expires` header was deprecated in favor of the `Cache-Control` header. Instead of a date, `Cache-Control` has a variable set of comma-delimited directives that define who can cache, how, and for how long. Let's take a look at them:

`private`

> The `private` directive states that no shared intermediary (proxy or CDN) is allowed to cache the response. This is a great way to make sure that the client, and only the client, caches the data.

`public`

> The `public` directive is the opposite of `private`. It indicates that the response may be cached by any entity within the request/response chain.

`no-cache`

> Usually, this directive simply means that the response should not be cached. If it is cached anyway, the data should not be used to satisfy a request unless it is revalidated with the server (more on revalidation later).

`no-store`

> A browser will store cacheable responses on disk so that they can be used after a browser restart or computer reboot. You can direct the browser or proxy cache to not store cached data on disk by using the `no-store` directive.

`no-transform`

> Some intermediary caches have the option to automatically transform their cached data to save memory or disk space or to simply reduce network traffic. An example is compressing images. For some applications, you might want to disallow this using the `no-transform` directive.

`max-age`

> This directive is how long (in seconds) the cache is valid. If both an `Expires` header and a `max-age` directive are set in the same response, the `max-age` always takes precedence.

s-maxage

The s-maxage directive is the same as the max-age directive, but it specifies the maximum time a shared, intermediary cache (like a proxy) is allowed to hold the data. This directive allows you to have different expiration times than the client.

Let's take a look at a simple example of a response to see Cache-Control in action:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Cache-Control: private, no-store, max-age=300

<customers>...</customers>
```

In this example, the response is saying that only the client may cache the response. This response is valid for 300 seconds and must not be stored on disk.

The JAX-RS specification provides javax.ws.rs.core.CacheControl, a simple class to represent the Cache-Control header:

```
public class CacheControl {
    public CacheControl() {...}

    public static CacheControl valueOf(String value)
                throws IllegalArgumentException {...}
    public boolean isMustRevalidate() {...}
    public void setMustRevalidate(boolean mustRevalidate) {...}
    public boolean isProxyRevalidate() {...}
    public void setProxyRevalidate(boolean proxyRevalidate) {...}
    public int getMaxAge() {...}
    public void setMaxAge(int maxAge) {...}
    public int getSMaxAge() {...}
    public void setSMaxAge(int sMaxAge) {...}
    public List<String> getNoCacheFields() {...}
    public void setNoCache(boolean noCache) {...}
    public boolean isNoCache() {...}
    public boolean isPrivate() {...}
    public List<String> getPrivateFields() {...}
    public void setPrivate(boolean _private) {...}
    public boolean isNoTransform() {...}
    public void setNoTransform(boolean noTransform) {...}
    public boolean isNoStore() {...}
    public void setNoStore(boolean noStore) {...}
    public Map<String, String> getCacheExtension() {...}
}
```

The ResponseBuilder class has a method called cacheControl() that can accept a CacheControl object:

```
@Path("/customers")
public class CustomerResource {

    @Path("{id}")
    @GET
    @Produces("application/xml")
    public Response getCustomer(@PathParam("id") int id) {
```

```
        Customer cust = findCustomer(id);

        CacheControl cc = new CacheControl();
        cc.setMaxAge(300);
        cc.setPrivate(true);
        cc.setNoStore(true);
        ResponseBuilder builder = Response.ok(cust, "application/xml");
        builder.cacheControl(cc);
        return builder.build();
    }
```

In this example, we initialize a `CacheControl` object and pass it to the `Response Builder.cacheControl()` method to set the `Cache-Control` header of the response. Unfortunately, JAX-RS doesn't yet have any nice annotations to do this for you automatically.

Revalidation and Conditional GETs

One interesting aspect of the caching protocol is that when the cache is stale, the cacher can ask the server if the data it is holding is still valid. This is called *revalidation*. To be able to perform revalidation, the client needs some extra information from the server about the resource it is caching. The server will send back a `Last-Modified` and/or an `ETag` header with its initial response to the client.

Last-Modified

The `Last-Modified` header represents a timestamp of the data sent by the server. Here's an example response:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Cache-Control: max-age=1000
Last-Modified: Tue, 15 May 2009 09:56 EST

<customer id="123">...</customer>
```

This initial response from the server is stating that the XML returned is valid for 1,000 seconds and has a timestamp of Tuesday, May 15, 2009, 9:56 AM EST. If the client supports revalidation, it will store this timestamp along with the cached data. After 1,000 seconds, the client may opt to revalidate its cache of the item. To do this it does a *conditional* GET request by passing a request header called `If-Modified-Since` with the value of the cached `Last-Modified` header. For example:

```
GET /customers/123 HTTP/1.1
If-Modified-Since: Tue, 15 May 2009 09:56 EST
```

When a service receives this GET request, it checks to see if its resource has been modified since the date provided within the `If-Modified-Since` header. If it has been changed since the timestamp provided, the server will send back a 200, "OK" response with the new representation of the resource. If it hasn't been changed, the server will

respond with 304, "Not Modified" and return no representation. In both cases, the server should send an updated `Cache-Control` and `Last-Modified` header if appropriate.

ETag

The `ETag` header is a pseudounique identifier that represents the version of the data sent back. Its value is any arbitrary quoted string and is usually an MD5 hash. Here's an example response:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Cache-Control: max-age=1000
ETag: "3141271342554322343200"

<customer id="123">...</customer>
```

Like the `Last-Modified` header, when the client caches this response, it should also cache the `ETag` value. When the cache expires after 1,000 seconds, the client performs a revalidation request with the `If-None-Match` header that contains the value of the cached `ETag`. For example:

```
GET /customers/123 HTTP/1.1
If-None-Match: "3141271342554322343200"
```

When a service receives this GET request, it tries to match the current `ETag` hash of the resource with the one provided within the `If-Modified-Since header`. If the tags don't match, the server will send back a 200, "OK" response with the new representation of the resource. If it hasn't been changed, the server will respond with 304, "Not Modified" and return no representation. In both cases, the server should send an updated `Cache-Control` and `ETag` header if appropriate.

One final thing about `ETag`s is they come in two flavors: strong and weak. A strong `ETag` should change whenever any bit of the resource's representation changes. A weak `ETag` changes only on semantically significant events. Weak `ETag`s are identified with a `W/` prefix. For example:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Cache-Control: max-age=1000
ETag: W/"3141271342554322343200"

<customer id="123">...</customer>
```

Weak `ETag`s give applications a bit more flexibility to reduce network traffic, as a cache can be revalidated when there have been only minor changes to the resource.

JAX-RS has a simple class called `javax.ws.rs.core.EntityTag` that represents the `ETag` header:

```
public class EntityTag {

    public EntityTag(String value) {...}
    public EntityTag(String value, boolean weak) {...}
```

```
    public static EntityTag valueOf(String value)
                 throws IllegalArgumentException {...}
    public boolean isWeak() {...}
    public String getValue() {...}
}
```

It is constructed with a string value and optionally with a flag telling the object if it is a weak `ETag` or not. The `getValue()` and `isWeak()` methods return these values on demand.

JAX-RS and conditional GETs

To help with conditional GETs, JAX-RS provides an injectable helper class called `javax.ws.rs.core.Request`:

```
public interface Request {
   ...

   ResponseBuilder evaluatePreconditions(EntityTag eTag);
   ResponseBuilder evaluatePreconditions(Date lastModified);
   ResponseBuilder evaluatePreconditions(Date lastModified, EntityTag eTag);
}
```

The overloaded `evaluatePreconditions()` method takes a `javax.ws.rs.core.Entity Tag`, a `java.util.Date` that represents the last modified timestamp, or both. These values should be current and up-to-date, as they will be compared with the values of the `If-Not-Modified` or `If-None-Match` headers sent with the request. If these headers don't exist or if the request header values don't pass revalidation, this method returns null and you should send back a 200, "OK" response with the new representation of the resource. If the method does not return null, it returns a preinitialized instance of a `ResponseBuilder` with a response code of 304 preset. For example:

```
@Path("/customers")
public class CustomerResource {

   @Path("{id}")
   @GET
   @Produces("application/xml")
   public Response getCustomer(@PathParam("id") int id,
                               @Context Request request) {
      Customer cust = findCustomer(id);
      EntityTag tag = new EntityTag(
                              Integer.toString(cust.hashCode()));

      CacheControl cc = new CacheControl();
      cc.setMaxAge(1000);

      ResponseBuilder builder = request.evaluatePreconditions(tag);
      if (builder != null) {
         builder.cacheControl(cc);
         return builder.build();
      }
```

```
    // Preconditions not met!

    builder = Response.ok(cust, "application/xml");
    builder.cacheControl(cc);
    builder.tag(tag);
    return builder.build();
}
```

In this example, we have a `getCustomer()` method that handles GET requests for the `/customers/{id}` URI pattern. An instance of `javax.ws.rs.core.Request` is injected into the method using the `@Context` annotation. We then find a `Customer` instance and create a current `ETag` value for it from the hash code of the object (this isn't the best way to create the `EntityTag`, but for simplicity's sake, let's keep it that way). We then call `Request.evaluatePreconditions()`, passing in the up-to-date tag. If the tags match, we reset the client's cache expiration by sending a new `Cache-Control` header and return. If the tags don't match, we build a `Response` with the new, current version of the `ETag` and `Customer`.

Concurrency

Now that we have a good idea of how to boost the performance of our JAX-RS services using HTTP caching, we now need to look at how to scale applications that update resources on our server. The way RESTful updates work is that the client fetches a representation of a resource through a GET request. It then modifies the representation locally and PUTs or POSTs the modified representation back to the server. This is all fine and dandy if there is only one client at a time modifying the resource, but what if the resource is being modified concurrently? Because the client is working with a snapshot, this data could become stale if another client modifies the resource while the snapshot is being processed.

The HTTP specification has a solution to this problem through the use of conditional PUTs or POSTs. This technique is very similar to how cache revalidation and conditional GETs work. The client first starts out by fetching the resource. For example, let's say our client wants to update a customer in a RESTful customer directory. It would first start off by doing a `GET /customers/123` to pull down the current representation of the specific customer it wants to update. The response might look something like this:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Cache-Control: max-age=1000
ETag: "3141271342554322343200"
Last-Modified: Tue, 15 May 2009 09:56 EST

<customer id="123">...</customer>
```

In order to do a conditional update, we need either an `ETag` or `Last-Modified` header. This information tells the server which snapshot version we have modified when we perform our update. It is sent along within the `If-Match` or `If-Unmodified-Since` header

when we do our PUT or POST request. The `If-Match` header is initialized with the
`ETag` value of the snapshot. The `If-Unmodified-Since` header is initialized with the value
of `Last-Modified` header. So, our update request might look like this:

```
PUT /customers/123 HTTP/1.1
If-Match: "3141271342554322343200"
If-Unmodified-Since: Tue, 15 May 2009 09:56 EST
Content-Type: application/xml

<customer id="123">...</customer>
```

You are not required to send both the `If-Match` and `If-Unmodified-Since` headers. One
or the other is sufficient to perform a conditional PUT or POST. When the server
receives this request, it checks to see if the current `ETag` of the resource matches the
value of the `If-Match` header and also to see if the timestamp on the resource matches
the `If-Unmodified-Since` header. If these conditions are not met, the server will return
an error response code of 412, "Precondition Failed." This tells the client that the rep-
resentation it is updating was modified concurrently and that it should retry. If the
conditions are met, the service performs the update and sends a success response code
back to the client.

JAX-RS and Conditional Updates

To do conditional updates with JAX-RS, you use the `Request.evaluatePrecondi
tions()` method again. Let's look at how we can implement it within Java code:

```
@Path("/customers")
public class CustomerResource {

    @Path("{id}")
    @PUT
    @Consumes("application/xml")
    public Response updateCustomer(@PathParam("id") int id,
                                   @Context Request request,
                                    Customer update ) {
       Customer cust = findCustomer(id);
       EntityTag tag = new EntityTag(
                                  Integer.toString(cust.hashCode()));
       Date timestamp = ...; // get the timestampe

       ResponseBuilder builder =
                     request.evaluatePreconditions(timestamp, tag);

       if (builder != null) {
          // Preconditions not met!
          return builder.build();
       }
```

```
    ... perform the update ...

    builder = Response.noContent();
    return builder.build();
}
```

The `updateCustomer()` method obtains a customer ID and an instance of `javax.ws.rs.core.Request` from injected parameters. It then locates an instance of a `Customer` object in some application-specific way (for example, from a database). From this current instance of `Customer`, it creates an `EntityTag` from the hash code of the object. It also finds the current timestamp of the `Customer` instance in some application-specific way. The `Request.evaluatePreconditions()` method is then called with time-stamp and tag variables. If these values do not match the values within the `If-Match` and `If-Unmodified-Since` headers sent with the request, `evaluatePreconditions()` sends back an instance of a `ResponseBuilder` initialized with the error code 412, "Precondition Failed." A `Response` object is built and sent back to the client. If the preconditions are met, the service performs the update and sends back a success code of 204, "No Content."

With this code in place, we can now worry less about concurrent updates of our resources. One interesting thought is that we did not have to come up with this scheme ourselves. It is already defined within the HTTP specification. This is one of the beauties of REST, in that it fully leverages the HTTP protocol.

Wrapping Up

In this chapter, you learned that HTTP has built-in facilities to help scale the performance of our distributed systems. HTTP caching is a rich protocol that gives us a lot of control over browser, proxy, and client caches. It helps tremendously in reducing network traffic and speeding up response times for applications. Besides caching, distributed systems also have the problem of multiple clients trying to update the same resource. The HTTP protocol again comes to the rescue with well-defined semantics for handling concurrent updates. For both caching and concurrent updates, JAX-RS provides some helper classes to make it easier to enable these features in your Java applications. Chapter 23 contains some code you can use to test-drive many of the concepts in this chapter.

Deployment and Integration

Throughout this book, I have focused on teaching you the basics of JAX-RS and REST with simple examples that have very few moving parts. In the real world, though, your JAX-RS services are going to interact with databases and a variety of server-side component models. They will need to be secure and sometimes transactional. Also, except for Chapter 3, I pretty much ignored how to assemble JAX-RS-based services in complex environments like Java EE. In this chapter, we'll look into deployment details of JAX-RS and how it integrates with Java EE and other component models.

Deployment

JAX-RS applications are deployed within a servlet container, like Apache Tomcat, Jetty, JBossWeb, or the servlet container of your favorite application server, like JBoss, Weblogic, Websphere, or Glassfish. Think of a servlet container as a web server. It understands the HTTP protocol and provides a low-level component model (the servlet API) for receiving HTTP requests.

Servlet-based applications are organized in deployment units called Web ARchives (WAR). A WAR is a JAR-based packaging format that contains the Java classes and libraries used by the deployment as well as static content like images and HTML files that the web server will publish. Here's what the structure of a WAR file looks like:

```
<any static content>
WEB-INF/
        web.xml
        classes/
        lib/
```

Any files outside and above the *WEB-INF/* directory of the archive are published and available directly through HTTP. This is where you would put static HTML files and images that you want to expose to the outside world. The *WEB-INF/* directory has two subdirectories. Within the *classes/* directory, you can put any Java classes you want there. They must be in a Java package structure. The *lib/* directory can contain any application or third-party libraries that will be used by the deployment. The

WEB-INF/ directory also contains a *web.xml* deployment descriptor file. This file defines the configuration of the WAR and how the servlet container should initialize it.

You will need to define a *web.xml* file for your JAX-RS. How JAX-RS is deployed within a servlet container varies between JAX-RS-aware and JAX-RS-unaware servlet containers. Additional deployment options are available in Java EE 6 as well. Let's dive into these details.

The Application Class

Before looking at what we have to do to configure a *web.xml* file, we need to learn about the `javax.ws.rs.core.Application` class. Although Java EE 6 has additional discovery options, the `Application` class is the only portable way of telling JAX-RS which web services (`@Path` annotated classes) as well as which `MessageBodyReader`s, `MessageBody Writer`s, and `ContextResolver`s (`@Provider` annotated classes) you want deployed. I first introduced you to the `Application` class back in Chapter 3:

```
package javax.ws.rs.core;

import java.util.Collections;
import java.util.Set;

public abstract class Application {
   private static final Set<Object> emptySet =
                                       Collections.emptySet();

   public abstract Set<Class<?>> getClasses();


   public Set<Object> getSingletons() {
      return emptySet;
   }

}
```

The `Application` class is very simple. All it does is list classes and objects that JAX-RS is supposed to deploy. The `getClasses()` method returns a list of JAX-RS web service and `@Provider`-annotated classes. JAX-RS web service classes follow the *per-request* model mentioned Chapter 3. `@Provider` classes are instantiated by the JAX-RS container and registered once per application.

The `getSingletons()` method returns a list of preallocated JAX-RS web services and `@Provider`-annotated classes. You, as the application programmer, are responsible for creating these objects. The JAX-RS runtime will iterate through the list of objects and register them internally. When these objects are registered, JAX-RS will also inject values for `@Context` annotated fields and setter methods.

Let's look at a simple example of an `Application` class:

```
import javax.ws.rs.core.Application;
```

```
public class ShoppingApplication extends Application {

    public ShoppingApplication() {}

    public Set<Class<?>> getClasses() {
        HashSet<Class<?>> set = new HashSet<Class<?>>();
        set.add(CustomerResource.class);
        set.add(OrderResource.class);
        set.add(ProduceResource.class);
        return set;
    }

    public Set<Object> getSingletons() {

        JsonWriter json = new JsonWriter();
        CreditCardResource service = new CreditCardResource();

        HashSet<Object> set = new HashSet();
        set.add(json);
        set.add(service);
        return set;
    }
}
```

Here, we have a class `ShoppingApplication` that extends the `Application` class. The `getClasses()` method allocates a `HashSet` and populates it with `@Path` annotated classes and returns the set. The `getSingletons()` method allocates a `MessageBodyWriter` class named `JsonWriter` and an `@Path` annotated class `CreditCardResource`. It then creates a `HashSet` and adds these instances to it. This set is returned by the method.

Deployment Within a JAX-RS-Unaware Container

If you are running within a JAX-RS-unaware servlet container, the JAX-RS implementation will give you a specific class that implements the `javax.servlet.Servlet` interface. You must define and configure this vendor-specific servlet within the WAR file of your application. This is done within the WAR's *WEB-INF/web.xml* deployment descriptor. You must also write an implementation of the `Application` class and specify it as an `<init-param>` of this vendor-specific servlet. For example:

```
<?xml version="1.0"?>
<web-app>
    <servlet>
        <servlet-name>JAXRS</servlet-name>
        <servlet-class>
            com.sun.jersey.spi.container.servlet.ServletContainer
        </servlet-class>
        <init-param>
            <param-name>
                javax.ws.rs.Application
            </param-name>
            <param-value>
                com.restfully.shop.services.ShoppingApplication
```

```
            </param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>Rest</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Here, we've registered and initialized the Jersey JAX-RS implementation with the Shop
pingApplication class we created earlier in this chapter (Jersey is the reference imple-
mentation for the JAX-RS specification). The <servlet-mapping> element specifies the
base URI path for the JAX-RS runtime. The /* <url-pattern> specifies that all incoming
requests should be routed through our JAX-RS implementation.

Deployment Within a JAX-RS-Aware Container

If you are deploying into a JAX-RS-aware servlet container, all you have to do is register
your Application class as a servlet within your WAR's *WEB-INF/web.xml* deployment
descriptor. For example:

```
<?xml version="1.0"?>
<web-app>
    <servlet>
        <servlet-name>Rest</servlet-name>
        <servlet-class>
            com.restfully.shop.services.ShoppingApplication
        </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>Rest</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

</web-app>
```

Here, we have registered the Application class we created earlier in this chapter directly
as a servlet class. A JAX-RS-aware servlet container will detect that
ShoppingApplication is a JAX-RS Application class and will do the necessary initiali-
zation internally behind the scenes. This saves you a few lines of XML.

Deployment Within Java EE 6

Java EE stands for Java Enterprise Edition. It is the umbrella specification of JAX-RS
and defines a complete enterprise platform that includes services like a servlet con-
tainer, EJB, a transaction manager (JTA), messaging (JMS), connection pooling (JCA),
database persistence (JPA), a web framework (JSF), and a multitude of other services.
Application servers that are certified under Java EE 6 are required to support JAX-RS.

In Java EE 6, the servlet container is JAX-RS-aware and supports declaring an `Application` class as a servlet within *web.xml*, as described earlier. The `Application` class is optional, though. When it is not present, the WAR file will be scanned for classes that are annotated with `@Path` or `@Provider`. So, your *web.xml* file would be very simple in such an environment:

```
<?xml version="1.0"?>
<web-app>

</web-app>
```

When scanning for classes, the application server will look within *WEB-INF/classes* and any JAR file within the *WEB-INF/lib* directory. When it finds a JAX-RS annotated class, it will add it to the list of things that need to be deployed and registered with the JAX-RS runtime.

Configuration

All the examples in this book so far have been simple and pretty self-contained. Your RESTful web services will probably need to sit in front of a database and interact with other local and remote services. Your services will also need configuration settings that are described outside of code. I don't want to get into too much detail, but the servlet and Java EE specifications provide annotations and XML configuration that allow you to get access to various Java EE services and configuration information. Let's look at how JAX-RS can take advantage of these features.

Older Java EE Containers

Any JAX-RS implementation, whether it sits within a JAX-RS-aware or -unaware servlet container, must support the `@Context` injection of the `javax.servlet.ServletContext` and `javax.servlet.ServletConfig` interfaces. Through these interfaces, you can get access to configuration information expressed in the WAR's *web.xml* deployment descriptor. Let's take this *web.xml* file, for example:

```
<?xml version="1.0"?>
<web-app>
   <context-param>
      <param-name>max-customers-size</param-name>
      <param-value>10</param-value>
   </context-param>

   <servlet>
      <servlet-name>JAXRS</servlet-name>
      <servlet-class>
         com.sun.jersey.spi.container.servlet.ServletContainer
      </servlet-class>
      <init-param>
         <param-name>
            javax.ws.rs.Application
```

```
            </param-name>
            <param-value>
                com.restfully.shop.services.ShoppingApplication
            </param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>Rest</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

In this *web.xml* file, we want to define a default maximum dataset size for a
JAX-RS-based customer database that returns a collection of customers through XML.
We do this by defining a `<context-param>` named `max-customers-size` and set the value
to 10. We can get access to this value within our JAX-RS service by injecting a reference
to `ServletContext` with the `@Context` annotation. For example:

```
@Path("/customers")
public class CustomerResource {

    protected int defaultPageSize = 5;

    @Context
    public void setServletContext(ServletContext context) {
        String size = context.getInitParameter("max-customers-size");
        if (size != null) {
            defaultPageSize = Integer.parseInt(size);
        }
    }

    @GET
    @Produces("application/xml")
    public String getCustomerList() {
        ... use defaultPageSize to create
                and return list of XML customers...
    }
}
```

Here, we use the `@Context` annotation on the `setServletContext()` method of our
`CustomerResource` class. When an instance of `CustomerResource` gets instantiated, the
`setServletContext()` method is called with access to a `javax.servlet.ServletContext`.
From this, we can obtain the value of `max-customers-size` that we defined in our
web.xml and save it in the member variable `defaultPageSize` for later use.

Another way you might want to do this is to use your `javax.ws.rs.core.Application`
class as a factory for your JAX-RS services. You could define or pull in configuration
information through this class and use it to construct your JAX-RS service. Let's first
rewrite our `CustomerResource` class to illustrate this technique:

```
@Path("/customers")
public class CustomerResource {

    protected int defaultPageSize = 5;

    public void setDefaultPageSize(int size) {
        defaultPageSize = size;
    }

    @GET
    @Produces("application/xml")
    public String getCustomerList() {
        ... use defaultPageSize to create and return list of XML customers...
    }
}
```

We first remove all references to the `ServletContext` injection we did in our previous incarnation of the `CustomerResource` class. We replace it with a setter method, `setDefaultPageSize()`, which initializes the `defaultPageSize` member variable. This is a better design for our `CustomerResource` class because we've abstracted away how it obtains configuration information. This gives the class more flexibility as it evolves over time.

It would be nice if we could use `@Context` injection of `ServletContext` within our `Application` class implementation. Unfortunately, the specification does not define or require that JAX-RS implementations support `@Context` injection into this class (to be honest, this was a simple mistake made by the JAX-RS specification committee and will be fixed in later versions of the specification). Luckily, Java EE has defined a general-purpose way of exposing configuration variables that we can use within our `Application` class. It is done using an `<env-entry>` within our *web.xml* file:

```xml
<?xml version="1.0"?>
<web-app>
    <servlet>
        <servlet-name>JAXRS</servlet-name>
        <servlet-class>
            com.sun.jersey.spi.container.servlet.ServletContainer
        </servlet-class>
        <init-param>
            <param-name>
                javax.ws.rs.Application
            </param-name>
            <param-value>
                com.restfully.shop.services.ShoppingApplication
            </param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>Rest</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
```

```
<env-entry>
    <env-entry-name>max-customers-size</env-entry-name>
    <env-entry-type>java.lang.Integer</env-entry-type>
    <env-entry-value>10</env-entry-value>
</env-entry>
</web-app>
```

Instead of using a `<context-param>`, we instead use an `<env-entry>`. The `<env-entry-name>` element defines the name of our configuration variable and the value is declared within the `<env-entry-value>` element. From our `Application` class, we can then look up the value through a standard directory service called JNDI:

```
import javax.ws.rs.core.Application;
import javax.naming.InitialContext;

public class ShoppingApplication extends Application {

    public ShoppingApplication() {}

    public Set<Class<?>> getClasses() {
        return Collections.emptySet();
    }

    public Set<Object> getSingletons() {
        int pageSize = 0;

        try {
            InitialContext ctx = new InitialContext();
            Integer size =
                (Integer)ctx.lookup("java:comp/env/max-customers-size");
            pageSize = size.getValue();
        } catch (Exception ex) {
            ... handle example ...
        }
        CustomerResource custService = new CustomerResource();
        custService.setDefaultPageSize(pageSize);

        HashSet<Object> set = new HashSet();
        set.add(custService);
        return set;
    }
}
```

To get access to the `<env-entry>` we defined in our *web.xml* file, we allocate an instance of `javax.naming.InitialContext()` and perform a `lookup()` on it prefixing our variable name, `max-customers-size`, with `java:comp/env/` to get `java:comp/env/max-customers-size`. This returns a `java.lang.Integer` that we convert to an `int` and use to initialize the `CustomerService` object. Any more explanation of JNDI and how all this process works is beyond the scope of this book.

Within Java EE 6 Containers

Java EE 6 offers a few more mechanisms for configuration that makes things a tiny bit easier. For one, JAX-RS resource classes are required to support Java EE injection annotations like `@Resource`, `@PersistenceContext`, `@PersistenceUnit`, and `@EJB`. This makes our previous configuration example with `<env-entry>` a little bit easier. Instead of doing all those JNDI lookups within our `Application` class, we can inject it directly into our JAX-RS service class:

```
@Path("/customers")
public class CustomerService {

    protected int defaultPageSize = 5;

    @Resource("max-customers-size")
    public void setDefaultPageSize(int size) {
        defaultPageSize = size;
    }

    @GET
    @Produces("application/xml")
    public String getCustomerList() {
        ... use defaultPageSize to create and return
                list of XML customers...
    }
}
```

Here, we've used the `@javax.annotation.Resource` annotation to inject that `<env-entry>` value we described in *web.xml* directly.

Java EE 6 also requires that the JAX-RS default component model support JSR-299,[*] Context and Dependency Injection for Java EE. This is a powerful new specification that fills in all the dependency injection holes that have lingered for years in the Java EE specification.

EJB Integration

EJBs are Java EE components that help you write business logic more easily. They support integration with security, transactions, and persistence. Further explanation of EJB is beyond the scope of this book. I suggest reading the book that I co-wrote with Richard Monson-Haefel, *Enterprise JavaBeans 3.0* (O'Reilly), if you want more information. Java EE 6 requires that EJB containers support integration with JAX-RS. You are allowed to use JAX-RS annotations on local interfaces or no-interface beans of stateless session beans. No other integration with other bean types is supported.

Since it is fairly new, you may not be able to use an application server that supports Java EE 6. This doesn't mean you can't use EJBs and JAX-RS together. There is a

---

[*] For more information, see *http://jcp.org/en/jsr/detail?id=299*.

portable workaround for this centered on registering EJB references through the `Application` class. You first need to declare EJB references within your *web.xml* file:

```
<?xml version="1.0"?>
<web-app>
    <servlet>
        <servlet-name>JAXRS</servlet-name>
        <servlet-class>
            com.sun.jersey.spi.container.servlet.ServletContainer
        </servlet-class>
        <init-param>
            <param-name>
                javax.ws.rs.Application
            </param-name>
            <param-value>
                com.restfully.shop.services.ShoppingApplication
            </param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>Rest</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

    <ejb-local-ref>
        <ejb-ref-name>ejb/CustomerResource</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <local>com.restfully.shop.services.CustomerResource</local>
        <ejb-link>com.restfull.shop.services.CustomerResourceBean</ejb-link>
    </ejb-local-ref>
</web-app>
```

This *web.xml* file uses an `<ejb-local-ref>` to create a reference to a `CustomerResource` Bean. We will use this reference later to register this EJB with JAX-RS. Let's look at the implementation of this EJB:

```
@Path("/customers")
@Local
public interface CustomerResource {

    @GET
    @Produces("application/xml")
    public String getCustomers();

    @GET
    @Produces("application/xml")
    @Path("{id}")
    public String getCustomer(@PathParam("id") int id);
}
```

To expose an EJB as a JAX-RS service, we use JAX-RS annotations on the local interface of the bean. Our EJB bean class would simply implement this interface:

```
@Stateless
public clas CustomerResourceBean implements CustomerResource {
```

```
        public String getCustomers() {...}
        public String getCustomer(int id) {...}
    }
```

Using JAX-RS annotations on the bean's local interface will not automatically register the EJB as a JAX-RS service. We still have some work to do in the `Application` class of our WAR to get this to work. Let's look at our `Application` class:

```
import javax.ws.rs.core.Application;

public class ShoppingApplication extends Application {

    public ShoppingApplication() {}

    public Set<Class<?>> getClasses() {
        return Collections.emptySet();
    }

    public Set<Object> getSingletons() {
        HashSet<Object> set = new HashSet();
        try {
            InitialContext ctx = new InitialContext();
            Object custService = ctx.lookup(
                        "java:comp/env/ejb/CustomerResource");
            set.add(custService);
        } catch (Exception ex) {
            ... handle example ...
        }
        return set;
    }
}
```

In our `Application` class's `getSingletons()` method, we look up our EJB reference within JNDI. We then create a `HashSet` and add the EJB reference to it. EJB references are proxies that must implement the local interface of the EJB. Since this proxy is an object that implements the `CustomerResource` interface, JAX-RS can introspect and find the JAX-RS annotations it needs to register the `CustomerResource` bean with the JAX-RS runtime.

There are a few limitations to this approach. The first is that you can only use JAX-RS annotations on stateless session beans and only on the bean's interface. Second, you cannot use setter method injection. This is because stateless session beans are allocated per-request and you are not guaranteed to get the same bean instance between invocations (you may even get a new instance). Why does this matter? JAX-RS would need to make a full EJB invocation on the setter method. This would result in the creation of a new bean instance that may or may not be pooled by a subsequent invocation on the same EJB container.

Spring Integration

Spring is an open source framework similar to EJB. Like EJB, it provides a great abstraction for transactions, persistence, and security. Further explanation of Spring is beyond the scope of this book. If you want more information on it, check out *Spring: A Developer's Notebook* by Bruce A. Tate and Justin Gehtland (O'Reilly). Most JAX-RS implementations have their own proprietary support for Spring and allow you to write Spring beans that are JAX-RS web services. If portability is not an issue for you, I suggest that you use the integration with Spring provided by your JAX-RS implementation.

There is a simple, portable way to integrate with Spring that we can talk about in this chapter. What you can do is write an `Application` class that loads your Spring XML files and then registers your Spring beans with JAX-RS through the `getSingletons()` method. First, let's define a Spring bean that represents a customer database. It will pretty much look like `CustomerResource` bean described in the EJB section of this chapter:

```
@Path("/customers")
public interface CustomerResource {

    @GET
    @Produces("application/xml")
    public String getCustomers();

    @GET
    @Produces("application/xml")
    @Path("{id}")
    public String getCustomer(@PathParam("id") int id);
}
```

In this example, we first create an interface for our `CustomerResource` that is annotated with JAX-RS annotations:

```
public clas CustomerResourceBean implements CustomerResource {

    public String getCustomers() {...}
    public String getCustomer(int id) {...}
}
```

Our Spring bean class, `CustomerResourceBean`, simply implements the `CustomerResource` interface. Although you can opt to not define an interface and use JAX-RS annotations directly on the bean class, I highly suggest that you use an interface. Interfaces work better in Spring when you use features like Spring transactions and such.

Now that we have a bean class, we should declare it within a Spring XML file called *spring-beans.xml* (or whatever you want to name the file):

```
<beans xmlns="http://www.springframework.org/schema/beans"
    <bean id="custService"
        class="com.shopping.restful.services.CustomerResourceBean"/>
</beans>
```

Place this *spring-beans.xml* file within your WAR's *WEB-INF/classes* directory or within a jar within the *WEB-INF/lib* directory. For this example, we'll put it in the *WEB-INF/classes* directory. We will find this file through a class loader resource lookup later on when we write our `Application` class.

Next we write our *web.xml* file:

```
<?xml version="1.0"?>
<web-app>
    <servlet>
        <servlet-name>JAXRS</servlet-name>
        <servlet-class>
            com.sun.jersey.spi.container.servlet.ServletContainer
        </servlet-class>
        <init-param>
            <param-name>
                javax.ws.rs.Application
            </param-name>
            <param-value>
                com.restfully.shop.services.ShoppingApplication
            </param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>Rest</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

    <env-entry>
        <env-entry-name>spring-beans-file</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>spring-beans.xml</env-entry-value>
    </env-entry>
</web-app>
```

In our *web.xml* file, we define an `<env-entry>` that contains the classpath location of our Spring XML file. We use an `<env-entry>` so that we can change this value in the future if needed. We then need to wire everything together in our `Application` class:

```
import javax.ws.rs.core.Application;
import org.springframework.context.ApplicatioContext;
import org.springframework.context
            .support.ClassPathXmlApplicationContext;

public class ShoppingApplication extends Application {

    public ShoppingApplication() {}

    public Set<Class<?>> getClasses() {
        return Collections.emptySet();
    }

    protected ApplicationContext springContext;
```

```
public Set<Object> getSingletons() {
   try {
      InitialContext ctx = new InitialContext();
      String xmlFile = (String)ctx.lookup(
                        "java:comp/env/spring-beans-file");
      springContext = new ClassPathXmlApplicationContext(xmlFile);

   } catch (Exception ex) {
      ... handle example ...
   }
   CustomerResource custService =
         (CustomerResource)springContext.getBean("custService");
   HashSet<Object> set = new HashSet();
   set.add(custService);
   return set;
}
}
```

In this `Application` class, we look up the classpath location of the Spring XML file that we defined in the `<env-entry>` of our *web.xml* deployment descriptor. We then load this XML file through Spring's `ClassPathXmlApplicationContext`. This will also create the beans defined in this file. From the Spring `ApplicationContext`, we look up the bean instance for our `CustomerResource` using the `ApplicationContext.getBean()` method. We then create a `HashSet` and add the `CustomerResource` bean to it and return it to be registered with the JAX-RS runtime.

Wrapping Up

In this chapter, you learned how deployment works within Java EE 6 containers and in environments that are not JAX-RS-aware. We also looked at some portable ways to configure your JAX-RS applications. Finally, you saw how you can portably integrate with EJB and Spring. Chapter 24 will allow you to test-drive some of the concepts presented in this chapter. It will walk you through the deployment of a full application that integrates with EJB, Spring, and Java Persistence (JPA).

Securing JAX-RS

Many RESTful web services will want secure access to data and functionality they provide. This is especially true for services that will be performing updates. They will want to prevent sniffers on the network from reading their messages. They may also want to fine-tune which users are allowed to interact with a specific service and disallow certain actions for specific users. The Web and the umbrella specification for JAX-RS, Java EE, provide a core set of security services and protocols that you can leverage from within your RESTful web services. These include:

Authentication

> Authentication is about validating the identity of a client that is trying to access your services. It usually involves checking to see if the client has provided an existing user with valid credentials, such as a password. The Web has a few standardized protocols you can use for authentication. Java EE, specifically your servlet container, has facilities to understand and configure these Internet security authentication protocols.

Authorization

> Once a client is authenticated, it will want to interact with your RESTful web service. Authorization is about deciding whether or not a certain user is allowed to access and invoke on a specific URI. For example, you may want to allow write access (PUT/POST/DELETE operations) for one set of users and disallow it for others. Authorization is not part of any Internet protocol and is really the domain of your servlet container and Java EE.

Encryption

> When a client is interacting with a RESTful web service, it is possible for hostile individuals to intercept network packets and read requests and responses if your HTTP connection is not secure. Sensitive data should be protected with cryptographic services like SSL. The Web defines the HTTPS protocol to leverage SSL and encryption.

JAX-RS has a small programmatic API for interacting with servlet and Java EE security, but enabling security in a JAX-RS environment is usually an exercise in configuration

and applying annotation metadata. This chapter focuses on various web protocols for authentication and how to configure your JAX-RS applications to use authentication, authorization, and encryption.

Authentication

When you want to enforce authentication for your RESTful web services, the first thing you have to do is decide which authentication protocol you want to use. Internet protocols for authentication vary in their complexity and their perceived reliability. In Java land, most servlet containers support the protocols of Basic authentication, Digest authentication, and authentication using X.509 certificates. Let's look into how each of these protocols works.

Basic Authentication

Basic authentication is the simplest protocol available for performing authentication over HTTP. It involves sending a Base64-encoded username and password within a request header to the server. The server checks to see if the username exists within its system and verifies the sent password. To understand the details of this protocol, let's look at an example.

Say an unauthorized client tries to access one of our secure RESTful web services:

```
GET /customers/333 HTTP/1.1
```

Since the request does not contain any authentication information, the server would reply with an HTTP response of:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="CustomerDB Realm"
```

The 401 response tells the client that it is not authorized to access the URI it tried to invoke on. The `WWW-Authenticate` header specifies which authentication protocol the client should use. In this case, `Basic` means basic authentication should be used. The `realm` attribute identifies a collection of secured resources on a website. The client can use the realm information to match against a username and password that is required for this specific URI.

To perform authentication, the client must send a request with the `Authorization` header set to a Base64-encoded string of our username and a colon character, followed by the password. If our username is `bburke` and our password `geheim`, the Base64-encoded string of `bburke:geheim` will be `YmJ1cmtlOmdlaGVpbQ==`. Put all this together and our authenticated GET request would look like this:

```
GET /customers/333 HTTP/1.1
Authorization: Basic YmJ1cmtlOmdlaGVpbQ==
```

The client needs to send this `Authorization` header with each and every request it makes to the server.

The problem with this approach is that if this request is intercepted by a hostile entity on the network, the hacker can easily obtain the username and password and use it to invoke its own requests. Using an encrypted HTTP connection, HTTPS, solves this problem. With an encrypted connection, a rogue programmer on the network will be unable to decode the transmission and get at the `Authorization` header. Still, security-paranoid network administrators are very squeamish about sending passwords over the network, even if they are encrypted within SSL packets.

Digest Authentication

Digest authentication was invented so that clients would not have to send clear text passwords over HTTP. It involves exchanging a set of secure MD5 hashes of the username, password, operation, URI, and optionally the hash of the message body itself. The protocol starts off with the client invoking an insecure request on the server:

```
GET /customers/333 HTTP/1.1
```

Since the request does not contain any authentication information, the server replies with an HTTP response of:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Digest realm="CustomerDB Realm",
                         qop="auth,auth-int",
                         nonce="12dcde223152321ab99cd",
                         opaque="aa9321534253bcd00121"
```

Like before, a 401 error code is returned along with a `WWW-Authenticate` header. The `nonce` and `opaque` attributes are special server-generated keys that will be used to build the subsequent authenticated request.

Like basic authentication, the client uses the `Authorization` header, but with digest-specific attributes. Here's a request example:

```
GET /customers/333 HTTP/1.1
Authorization: Digest username="bburke",
                       realm="CustomerDB Realm",
                       nonce="12dcde223152321ab99cd",
                       uri="/customer/333",
                       qop="auth",
                       nc=00000001,
                       cnonce="43fea",
                       response="11132fffdeab993421",
                       opaque="aa9321534253bcd00121"
```

The `nonce` and `opaque` attributes are a copy of the values sent with the earlier `WWW-Authenticate` header. The `uri` attribute is the base URI you are invoking on. The `nc` attribute is a request counter that should be incremented by the client with each request. This prevents hostile clients from replaying a request. The `cnonce` attribute is a unique key generated by the client and can be anything the client wants. The `response` attribute is where all the meat is. It is a hash value generated with the following pseudocode:

```
H1 = md5("username:realm:password")
H2 = md5("httpmethod:uri")
response = md5("H1:nonce:nc:cnonce:qop:H2")
```

If our username is `bburke` and our password `geheim`, the algorithm will resolve to this pseudocode:

```
H1 = md5("bburke:CustomerDB Realm:geheim")
H2 = md5("GET:/customers/333")
response = md5("H1:12dcde223152321ab99cd:00000001:43fea:auth:H2")
```

When the server receives this request, it builds its own version of the response hash using its stored, secret values of the username and password. If the hashes match, the user and its credentials are valid.

One advantage of this approach is that the password is never used directly by the protocol. For example, the server doesn't even need to store clear text passwords. It can instead initialize its authorization store with prehashed values. Also, since request hashes are built with a `nonce` value, the server can expire these nonce values over time. This, combined with a request counter, can greatly reduce replay attacks.

The disadvantage to this approach is that unless you use HTTPS, you are still vulnerable to man-in-the-middle attacks, where the middleman can tell a client to use Basic authentication to obtain a password.

Client Certificate Authentication

When you buy things or trade stocks on the Internet, you use the HTTPS protocol to obtain a secure connection with the server. HTTPS isn't only an encryption mechanism—it can also be used for authentication. When you first interact with a secure website, your browser receives a digitally signed certificate from the server that identifies it. Your browser verifies this certificate with a central authority like VeriSign. This is how you guarantee the identity of the server you are interacting with and make sure you're not dealing with some man-in-the-middle security breach.

HTTPS can also perform two-way authentication. In addition to the client receiving a signed digital certificate representing the server, the server can receive a certificate that represents and identifies the client. When a client initially connects to a server, it exchanges its certificate and the server matches it against its internal store. Once this link is established, there is no further need for user authentication, since the certificate has already positively identified the user.

Client certificate authentication is perhaps the most secure way to perform authentication on the Web. The only disadvantage of this approach is the managing of the certificates themselves. The server must create a unique certificate for each client that wants to connect to the service. From the browser/human perspective, this can be a pain, as the user has to do some extra configuration to interact with the server.

Authorization

While authentication is about establishing and verifying user identity, authorization is about permissions. Is my user allowed to perform the operation it is invoking? None of the standards-based Internet authorization protocols discussed so far deals with authorization. The server and application know the permissions for each user and do not need to share this information over a communication protocol. This is why authorization is the domain of the server and application.

JAX-RS relies on the servlet and Java EE specifications to define how authorization works. Authorization is performed in Java EE by associating one or more roles with a given user and then assigning permissions based on that role. While an example of a user might be "Bill" or "Monica," roles are used to identify a group of users, for instance, "adminstrator," "manager," or "employee." You do not assign access control on a per-user basis, but rather on a per-role basis.

Authentication and Authorization in JAX-RS

To enable authentication, you need to modify the *WEB-INF/web.xml* deployment descriptor of the WAR file your JAX-RS application is deployed in. Authorization is enabled through XML or by applying annotations to your JAX-RS resource classes. To see how all this is put together, let's do a simple example. We have a customer database that allows us to create new customers by posting an XML document to the JAX-RS resource located at the URI `/customers`. We want to secure our customer service so that only administrators are allowed to create new customers. Let's look at a full XML-based implementation of this example:

```
<?xml version="1.0"?>
<web-app>
    <servlet>
        <servlet-name>JAXRS</servlet-name>
        <servlet-class>
     org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
        </servlet-class>
        <init-param>
            <param-name>
                javax.ws.rs.Application
            </param-name>
            <param-value>
                com.restfully.shop.services.ShoppingApplication
            </param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>Rest</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
```

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>customer creation</web-resource-name>
        <url-pattern>/customers</url-pattern>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
</security-constraint>

<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>jaxrs</realm-name>
</login-config>

<security-role>
    <role-name>admin</role-name>
</security-role>
```

```
</web-app>
```

The `<login-config>` element defines how we want our HTTP requests to be authenticated for our entire deployment. The `<auth-method>` subelement can be `BASIC`, `DIGEST`, or `CLIENT_CERT`. These values correspond to Basic, Digest, and Client Certificate authentication, respectively.

The `<login-config>` element doesn't turn on authentication. By default, any client can access any URL provided by your web application with no constraints. To enforce authentication, you must specify a URL pattern you want to secure. In our example, we use the `<url-pattern>` element to specify that we want to secure the `/customers` URL. The `<http-method>` element says that we only want to secure POST requests to this URL. If we leave out the `<http-method>` element, all HTTP methods are secured. In our example, we only want to secure POST requests, so we must define the `<http-method>` element.

Next, we have to specify which roles are allowed to POST to `/customers`. In the *web.xml* file example, we define an `<auth-constraint>` element within a `<security-constraint>`. This element has one or more `<role-name>` elements that define which roles are allowed to access the defined constraint. In our example, we give the `admin` role access to only the URL. A `<role-name>` of * means that every defined role in our deployment descriptor is allowed to access the constraint. In this case, even though all roles are allowed to access the URL pattern, authentication still happens. This is really the same as allowing access to anybody that is a valid user.

Finally, there's an additional bit of syntactic sugar we need to specify in *web.xml*. For every `<role-name>` we use in our `<auth-constraints>` declarations, we must define a corresponding `<security-role>` in the deployment descriptor.

There is a minor limitation when declaring `<security-constraints>` for JAX-RS resources. The `<url-pattern>` element does not have as rich an expression syntax as JAX-RS `@Path` annotation values. In fact, it is extremely limited. It supports only simple wildcard matches via the * character. No regular expressions. For example:

```
/*
/foo/*
*.txt
```

The wildcard pattern can only be used at the end of a URL pattern or to match file extensions. When used at the end of a URL pattern, the wildcard matches every character in the incoming URL. For example, `/foo/*` would match any URL that starts with `/foo`. To match file extensions, you use the format `*.<suffix>`. For example, the `*.txt` pattern matches any URL that ends with `.txt`. No other uses of the wildcard character are permitted in URL patterns. For example, here are some illegal expressions:

```
/foo/*/bar
/foo/*.txt
```

Enforcing Encryption

By default, the servlet specification will not require access over HTTPS to any user constraints you declare in your *web.xml* file. If you want to enforce HTTPS access for these constraints, you can specify a `<user-data-constraint>` within your `<security-constraint>` definitions. Let's modify our previous example to enforce HTTPS:

```
<web-app>
...

    <security-constraint>
       <web-resource-collection>
          <web-resource-name>customer creation</web-resource-name>
          <url-pattern>/customers</url-pattern>
          <http-method>POST</http-method>
       </web-resource-collection>
       <auth-constraint>
          <role-name>admin</role-name>
       </auth-constraint>
       <user-data-constraint>
          <transport-guarantee>CONFIDENTIAL</transport-guarantee>
       </user-data-constraint>
    </security-constraint>
...
</web-app>
```

All you have to do is declare a `<transport-guarantee>` element within a `<user-data-constraint>` that has a value of `CONFIDENTIAL`. If a user tries to access the URL pattern with HTTP, she will be redirected to an HTTPS-based URL.

Authorization Annotations

Java EE defines a common set of annotations that can define authorization metadata. The JAX-RS specification suggests, but does not require, vendor implementations to support these annotations in a non-Java EE 6 environment. These annotations live in the `javax.annotation.security` package and are `@RolesAllowed`, `@DenyAll`, `@PermitAll`, and `@RunAs`.

The `@RolesAllowed` annotation defines the roles permitted to execute a specific operation. When placed on a JAX-RS annotated class, it defines the default access control list for all HTTP operations defined in the JAX-RS class. If placed on a JAX-RS method, the constraint applies only to the method that is annotated.

The `@PermitAll` annotation specifies that any authenticated user is permitted to invoke your operation. As with `@RolesAllowed`, you can use this annotation on the class to define the default for the entire class or you can use it on a per-method basis. Let's look at an example:

```
@Path("/customers")
@RolesAllowed({"ADMIN", "CUSTOMER"})
public class CustomerResource {

    @GET
    @Path("{id}")
    @Produces("application/xml")
    public Customer getCustomer(@PathParam("id") int id) {...}

    @RolesAllowed("ADMIN")
    @POST
    @Consumes("application/xml")
    public void createCustomer(Customer cust) {...}

    @PermitAll
    @GET
    @Produces("application/xml")
    public Customer[] getCustomers() {}
}
```

Our `CustomerResource` class is annotated with `@RolesAllowed` to specify that, by default, only `ADMIN` and `CUSTOMER` users can execute HTTP operations and paths defined in that class. The `getCustomer()` method is not annotated with any security annotations, so it inherits this default behavior. The `createCustomer()` method is annotated with `@RolesAllowed` to override the default behavior. For this method, we only want to allow `ADMIN` access. The `getCustomers()` method is annotated with `@PermitAll`. This overrides the default behavior so that any authenticated user can access that URI and operation.

In practice, I don't like to specify security metadata using annotations. Security generally does not affect the behavior of the business logic being executed and falls more under the domain of configuration. Administrators may want to add or remove role constraints periodically. You don't want to have to recompile your whole application

when they want to make a simple change. So, if I can avoid it, I usually use *web.xml* to define my authorization metadata.

There are some advantages to using annotations, though. For one, it is a workaround for doing fine-grained constraints that are just not possible in *web.xml* because of the limited expression capabilities of `<url-pattern>`. Also, because you can apply constraints per-method using these annotations, you can have fine-tune authorization per media type. For example:

```
@Path("/customers")
public class CustomerService {

    @GET
    @Produces("application/xml")
    @RolesAllowed("XML-USERS")
    public Customer getXmlCustomers() {}


    @GET
    @Produces("application/json")
    @RolesAllowed("JSON-USERS")
    public Customer getJsonCustomers() {}
}
```

Here we only allow `XML-USERS` to obtain `application/xml` content and `JSON-USERS` to obtain `application/json` content. This might be useful for limiting users in getting data formats that are expensive to create.

Programmatic Security

The security features defined in this chapter have so far focused on declarative security metadata, or metadata that is statically defined before an application even runs. JAX-RS also has a small programmatic API for gathering security information about a secured request. Specifically, the `javax.ws.rs.core.SecurityContext` interface has a method for determining the identity of the user making the secured HTTP invocation. It also has a method that allows you to check whether or not the current user belongs to a certain role:

```
public interface SecurityContext {

    public Principal getUserPrincipal();
    public boolean isUserInRole(String role);
    public boolean isSecure();
    public String getAuthenticationScheme();
}
```

The `getUserPrincipal()` method returns a standard Java Standard Edition (SE) `javax.security.Principal` security interface. A `Principal` object represents the individual user that is currently invoking the HTTP request. The `isUserInRole()` method allows you to determine whether the current calling user belongs to a certain role. The `isSecure()` method returns true if the current request is a secure connection. The `getAuthenticationScheme()` tells you which authentication mechanism was used to secure the request. `BASIC`, `DIGEST`, `CLIENT_CERT`, and `FORM` are valid values returned by this method. You get access to a `SecurityContext` instance by injecting it into a field, setter method, or resource method parameter using the `@Context` annotation.

Let's examine this security interface with an example. Let's say we want to have a security log of all access to a customer database by users who are not administrators. Here is how it might look:

```
@Path("/customers")
public class CustomerService {

    @GET
    @Produces("application/xml")
    public Customer[] getCustomers(@Context SecurityContext sec) {

        if (sec.isSecure() && !sec.isUserInRole("ADMIN")) {
          logger.log(sec.getUserPrincipal() +
                        " accessed customer database.");
        }
        ...
    }
}
```

In this example, we inject the `SecurityContext` as a parameter to our `getCustomer()` JAX-RS resource method. We use the method `SecurityContext.isSecure()` to determine whether or not this is an authenticated request. We then use the method `SecurityContext.isUserInRole()` to find out if the caller is an `ADMIN` or not. Finally, we print out to our audit log.

Wrapping Up

In this chapter, we discussed a few of the authentication protocols used on the Internet, specifically Basic, Digest, and Client Certificate authentication. You learned how to configure your JAX-RS applications to be secure using the metadata provided by the servlet and Java EE specifications.

RESTful Java Clients

So far, we've talked a lot about writing RESTful web services in Java, but we haven't talked a lot about writing RESTful clients. The beauty of REST is that if your programming language has support for HTTP, that's all you need. Sure, there are frameworks out there that can help you write client code more productively, but you are not required, or may not even need, to use them. Unfortunately, JAX-RS is only a server-side framework for writing actual web services. It does not provide a client API, but there is strong interest in defining one for the JAX-RS 2.0 specification. In the meantime, there are a few frameworks available that you can use to write RESTful Java clients.

java.net.URL

Like most programming languages, Java has a built-in HTTP client library. It's nothing fancy, but it's good enough to perform most of the basic functions you need. The API is built around two classes, `java.net.URL` and `java.net.HttpURLConnection`. The `URL` class is just a Java representation of a URL. Here are some of the pertinent constructors and methods:

```
public class URL {

   public URL(java.lang.String s)
           throws java.net.MalformedURLException {}

   public java.net.URLConnection
           openConnection() throws java.io.IOException {}
...
   }
```

From a `URL`, you can create an `HttpURLConnection` that allows you to invoke specific requests. Here's an example of doing a simple GET request:

```
URL url = new URL("http://example.com/customers/1");
connection = (HttpURLConnection) getUrl.openConnection();
connection.setRequestMethod("GET");
connection.setRequestProperty("Accept", "application/xml");

if (connection.getResponseCode() != 200) {
  throw new RuntimeExceptioin("Operation failed: "
                             + connection.getResponseCode());
}

System.out.println("Content-Type: " + connection.getContentType());

BufferedReader reader = new BufferedReader(new
              InputStreamReader(connection.getInputStream()));

String line = reader.readLine();
while (line != null) {
   System.out.println(line);
   line = reader.readLine();
}
connection.disconnect();
```

In this example, we instantiate a URL instance and then open a connection using the URL.openConnection() method. This method returns a generic URLConnection type, so we need to typecast it to an HttpURLConnection. Once we have a connection, we set the HTTP method we are invoking by calling HttpURLConnection.setMethod(). We want XML from the server, so we call the setRequestProperty() method to set the Accept header. We get the response code and Content-Type by calling getResponseCode() and getContentType(), respectively. The getInputStream() method allows us to read the content sent from the server using the Java streaming API. We finish up by calling disconnect().

Sending content to the server via a PUT or POST is a little different. Here's an example of that:

```
URL url = new URL("http://example.com/customers");
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
connection.setDoOutput(true);
connection.setInstanceFollowRedirects(false);
connection.setRequestMethod("POST");
connection.setRequestProperty("Content-Type", "application/xml");
OutputStream os = connection.getOutputStream();
os.write("<customer id='333'/>".getBytes());
os.flush();
if (connection.getResponseCode() != HttpURLConnection.HTTP_CREATED) {
   throw new RuntimeException("Failed to create customer");
}
System.out.println("Location: " + connection.getHeaderField("Location"));
connection.disconnect();
```

In this example, we create a customer by doing POST. We're expecting a response of 201, "Created," as well as a Location header in the response that points to the URL of our newly created customer. We need to call HttpURLConnection.setDoOutput(true).

This allows us to write a body for the request. By default, `HttpURLConnection` will automatically follow redirects. We want to look at our `Location` header, so we call `setInstanceFollowRedirects(false)` to disable this feature. We then call `setRequestMethod()` to tell the connection we're making a POST request. The `setRequestProperty()` method is called to set the `Content-Type` of our request. We then get a `java.io.OutputStream` to write out the data and the `Location` response header by calling `getHeaderField()`. Finally, we call `disconnect()` to clean up our connection.

Caching

By default, `HttpURLConnection` will cache results based on the caching response headers discussed in Chapter 10. You must invoke `HttpURLConnection.setUseCaches(false)` to turn off this feature.

Authentication

The `HttpURLConnection` supports Basic, Digest, and Client Certificate authentication. Basic and Digest authentication use the `java.net.Authenticator` API. Here's an example:

```
Authenticator.setDefault(new Authenticator() {
    protected PasswordAuthentication getPasswordAuthentication() {
        return new PasswordAuthentication ("username, "password".toCharArray());
    }
});
```

The `setDefault()` method is a static method of `Authenticator`. You pass in an `Authenticator` instance that overrides the class's `getPasswordAuthentication()` method. You return a `java.net.PasswordAuthentication` object that encapsulates the username and password to access your server. When you do `HttpURLConnection` invocations, authentication will automatically be set up for you using either Basic or Digest, depending on what the server requires.

The weirdest part of the API is that it is driven by the static method `setDefault()`. The problem with this is that your `Authenticator` is set VM-wide. So, doing authenticated requests in multiple threads to different servers is a bit problematic with the basic example just shown. This can be addressed using `java.lang.ThreadLocal` variables to store username and passwords:

```
public class MultiThreadedAuthenticator extends Authenticator {

    private static ThreadLocal<String> username = new ThreadLocal<String>();
    private static ThreadLocal<String> password = new ThreadLocal<String>();

    public static void setThreadUsername(String user) {
       username.set(user);
    }
```

```
        public static void setThreadPassword(String pwd) {
           password.set(pwd);
        }

         protected PasswordAuthentication getPasswordAuthentication() {
            return new PasswordAuthentication (username.get(),
                                        password.get().toCharArray());
        }
    }
```

The `ThreadLocal` class is a standard class that comes with the JDK. When you call `set()` on it, the value will be stored and associated with the calling thread. Each thread can have its own value. `ThreadLocal.get()` returns the thread's current stored value. So, using this class would look like this:

```
Authenticator.setDefault(new ThreadSafeAuthenticator());

ThreadSafeAuthenticator.setThreadUsername("bill");
ThreadSafeAuthenticator.setThreadPassword("geheim");
```

Client Certificate authentication

Client Certificate authentication is a little different. First, you must generate a client certificate using the `keytool` command-line utility that comes with the JDK:

```
$ <JAVA_HOME>/bin/keytool -genkey -alias client-alias -keyalg RSA -keypass changeit
-storepass changeit -keystore keystore.jks
```

Next, you must export the certificate into a file so it can be imported into a truststore:

```
$ <JAVA_HOME>/bin/keytool -export -alias client-alias
-storepass changeit -file client.cer -keystore keystore.jks
```

Finally, you create a truststore and import the created client certificate:

```
$ <JAVA_HOME>\bin\keytool -import -v -trustcacerts
-alias client-alias -file client.cer
-keystore cacerts.jks
-keypass changeit -storepass changeit
```

Now that you have a truststore, use it to create a **javax.net.ssl.SSLSocketFactory** within your client code:

```
import javax.net.ssl.SSLContext;
import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLSocketFactory;
import java.security.SecureRandom;
import java.security.KeyStore;
import java.io.FileInputStream;
import java.io.InputStream;
import java.io.File;

public class MyClient {
```

```
    public static SSLSocketFactory
            getFactory( File pKeyFile, String pKeyPassword )
                                          throws Exception {
    KeyManagerFactory keyManagerFactory =
                        KeyManagerFactory.getInstance("SunX509");
    KeyStore keyStore = KeyStore.getInstance("PKCS12");

    InputStream keyInput = new FileInputStream(pKeyFile);
    keyStore.load(keyInput, pKeyPassword.toCharArray());
    keyInput.close();

    keyManagerFactory.init(keyStore, pKeyPassword.toCharArray());

    SSLContext context = SSLContext.getInstance("TLS");
    context.init(keyManagerFactory.getKeyManagers(), null
                 , new SecureRandom());

    return context.getSocketFactory();
    }
```

This code loads the truststore into memory and creates an `SSLSocketFactory`. The factory can then be registered with a `java.net.ssl.HttpsURLConnection`:

```
    public static void main(String args[]) throws Exception {
      URL url = new URL("https://someurl");
      HttpsURLConnection con = (HttpsURLConnection) url.openConnection();
      con.setSSLSocketFactory(getFactory(new File("cacerts.jks"),
                              "changeit"));
    }
  }
```

You may then make invocations to the URL, and the client certificate will be used for authentication.

Advantages and Disadvantages

The biggest advantage of using the `java.net` package as a RESTful client is that it is built-in to the JDK. You don't need to download and install a different client framework.

There are a few disadvantages to the `java.net` API. First, it is not JAX-RS-aware. You will have to do your own stream processing and will not be able to take advantage of any of the `MessageBodyReaders` and `MessageBodyWriters` that come with your JAX-RS implementation.

Second, the framework does not do preemptive authentication for Basic or Digest authentication. This means that `HttpURLConnection` will first try to invoke a request without any authentication headers set. If the server requires authentication, the initial request will fail with a 401, "Unauthorized" response code. The `HttpURLConnection` implementation then looks at the `WWW-Authenticate` header to see whether Basic or Digest authentication should be used and then retry the request. This can have an

impact on the performance of your system, because each authenticated request will actually be two requests between the client and server.

Third, the framework can't do something as simple as forming parameters. All you have to work with are `java.io.OutputStream` and `java.io.InputStream` to perform your input and output.

Finally, the framework only allows you to invoke the HTTP methods GET, POST, DELETE, PUT, TRACE, OPTIONS, and HEAD. If you try to invoke any HTTP method other than those, an exception is thrown and your invocation will abort. In general, this is not that important unless you want to invoke newer HTTP methods like those defined in the WebDav specification.

Apache HttpClient

The Apache foundation has written a nice, extendible, HTTP client library called HttpClient.* It is currently on version 4.0 as of the writing of this book. Although it is not JAX-RS-aware, it does have facilities for preemptive authentication and APIs for dealing with a few different media types like forms and multipart. Some of its other features are a full interceptor model, automatic cookie handling between requests, and pluggable authentication to name a few. Let's look at a simple example:

```java
import org.apache.http.*;
import org.apache.http.client.*;

public class MyClient {

  public static void main(String[] args) throws Exception {

      DefaultHttpClient client = new DefaultHttpClient();
      HttpGet get = new HttpGet("http://example.com/customers/1");
      get.addHeader("accept", "application/xml");

      HttpResponse response = client.execute(get);
      if (response.getStatusLine().getStatusCode() != 200) {
         throw new RuntimeExceptioin("Operation failed: " +
                  response.getStatusLine().getStatusCode());
      }

      System.out.println("Content-Type: " +
          response.getEntity().getContentType().getValue());

      BufferedReader reader = new BufferedReader(new
             InputStreamReader(response.getEntity()
                                        .getInputStream()));

      String line = reader.readLine();
      while (line != null) {
```

---

* For more information, see *http://hc.apache.org*.

```
            System.out.println(line);
            line = reader.readLine();
         }
         client.getConnectionManager().shutdown();
      }
   }
```

In Apache HttpClient 4.x, the `org.apache.http.client.DefaultHttpClient` class is responsible for managing HTTP connections. It handles the default authentication settings, pools and manages persistent HTTP connections (keepalive), and any other default configuration settings. It is also responsible for executing requests. The `org.apache.http.client.methods.HttpGet` class is used to build an actual HTTP GET request. You initialize it with a URL and set any request headers you want using the `HttpGet.addHeader()` method. There are similar classes in this package for doing POST, PUT, and DELETE invocations. Once you have built your request, you execute it by calling `DefaultHttpClient.execute()`, passing in the request you built. This returns an `org.apache.http.HttpResponse` object. To get the response code from this object, execute `HttpResponse.getStatusLine().getStatusCode()`. The `HttpResponse.getEntity()` method returns an `org.apache.http.HttpEntity` object, which represents the message body of the response. From it you can get the `Content-Type` by executing `HttpEntity.getContentType()` as well as a `java.io.InputStream` so you can read the response. When you are done invoking requests, you clean up your connections by calling `HttpClient.getConnectionManager().shutdown()`.

To push data to the server via a POST or PUT operation, you need to encapsulate your data within an instance of the `org.apache.http.HttpEntity` interface. The framework has some simple prebuilt ones for sending strings, forms, byte arrays, and input streams. Let's look at sending some XML.

In this example, we want to create a customer in a RESTful customer database. The API works by POSTing an XML representation of the new customer to a specific URI. A successful response is 201, "Created." Also, a `Location` response header is returned that points to the newly created customer:

```
import org.apache.http.*;
import org.apache.http.client.*;

public class MyClient {

   public static void main(String[] args) throws Exception {

      DefaultHttpClient client = new DefaultHttpClient();
      HttpPost post = new HttpPost("http://example.com/customers");
      StringEntity entity = new StringEntity("<customer id='333'/>");
      entity.setContentType("application/xml");
      post.setEntity(entity);
      HttpClientParams.setRedirection(post.getParams(), false);
      HttpResponse response = client.execute(post);
      if (response.getStatusLine().getStatusCode() != 201) {
         throw new RuntimeExceptioin("Operation failed: " +
                 response.getStatusLine().getStatusCode());
```

```
        }

        String location = response.getLastHeader("Location")
                                   .getValue();

        System.out.println("Object created at: " + location);
        System.out.println("Content-Type: " +
            response.getEntity().getContentType().getValue());

        BufferedReader reader = new BufferedReader(new
            InputStreamReader(response.getEntity().getContent()));

        String line = reader.readLine();
        while (line != null) {
            System.out.println(line);
            line = reader.readLine();
        }
        client.getConnectionManager().shutdown();
    }
}
```

We create an `org.apache.http.entity.StringEntity` to encapsulate the XML we want to send across the wire. We set its `Content-Type` by calling `StringEntity.setContent Type()`. We add the entity to the request by calling `HttpPost.setEntity()`. Since we are expecting a redirection header with our response and we do not want to be automatically redirected, we must configure the request to not do automatic redirects. We do this by calling `HttpClientParams.setRedirection()`. We execute the request the same way we did with our GET example. We get the `Location` header by calling `HttpResponse.getLastHeader()`.

Authentication

The Apache HttpClient 4.x supports Basic, Digest, and Client Certificate authentication. Basic and Digest authentication are done through the `DefaultHttpClient.getCre dentialsProvider().setCredentials()` method. Here's an example:

```
DefaultHttpClient client = new DefaultHttpClient();
client.getCredentialsProvider().setCredentials(
    new AuthScope("example.com", 443),
    new UsernamePasswordCredentials("bill", "geheim");
);
```

The `org.apache.http.auth.AuthScope` class defines the server and port that you want to associate with a username and password. The `org.apache.http.auth.UsernamePass wordCredentials` class encapsulates the username and password into an object. You can call `setCredentials()` for every domain you need to communicate with securely.

Apache HttpClient, by default, does not do preemptive authentication for the Basic and Digest protocols, but does support it. Since the code to do this is a bit verbose, we won't cover it in this book.

Client Certificate authentication

Apache HttpClient also supports Client Certificate authentication. As with `HttpsURL` `Connection`, you have to load in a `KeyStore` that contains your client certificates. "java.net.URL" on page 165 describes how to do this. You initialize an `org.apache.http.conn.ssl.SSLSocketFactory` with a loaded `KeyStore` and associate it with the `DefaultHttpClient`. Here is an example of doing this:

```java
import java.io.File;
import java.io.FileInputStream;
import java.security.KeyStore;

import org.apache.http.*;
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.*;
import org.apache.http.conn.scheme.*;
import org.apache.http.conn.ssl.*;
import org.apache.http.impl.client.DefaultHttpClient;

public class MyClient {

    public final static void main(String[] args) throws Exception {
        DefaultHttpClient client = new DefaultHttpClient();

        KeyStore trustStore  = KeyStore.getInstance(
                                    KeyStore.getDefaultType());
        FileInputStream instream = new FileInputStream(
                                    new File("my.keystore"));
        try {
            trustStore.load(instream, "changeit".toCharArray());
        } finally {
            instream.close();
        }

        SSLSocketFactory socketFactory =
                                    new SSLSocketFactory(trustStore);
        Scheme scheme = new Scheme("https", socketFactory, 443);
        client.getConnectionManager()
                .getSchemeRegistry().register(scheme);

        HttpGet httpget = new HttpGet("https://localhost/");

        ... proceed with the invocation ...
    }
}
```

Advantages and Disadvantages

Apache HttpClient is a more complete solution and is better designed than `java.net.HttpURLConnection`. Although you have to download it separately from the JDK, I highly recommend you take a look at it. It has none of the disadvantages of `HttpURLConnection`, except that it is not JAX-RS-aware.

RESTEasy Client Framework

RESTEasy[†] is the JBoss (Red Hat) JAX-RS implementation. As with `java.net.HttpURL` `Connection` and Apache HTTP Client, it provides a programmatic API to submit HTTP requests with the additional feature of it being JAX-RS-aware. To use the API, you create `org.jboss.resteasy.client.ClientRequest` objects. You build up the request using the following constructor and methods:

```
public class ClientRequest {
    public ClientRequest(String uriTemplate) {}

    public ClientRequest followRedirects(boolean follow) {}
    public ClientRequest accept(String accept) {}
    public ClientRequest formParameter(String parameterName,
                                       Object value) {}
    public ClientRequest queryParameter(String parameterName,
                                        Object value) {}
    public ClientRequest matrixParameter(String parameterName,
                                         Object value) {}
    public ClientRequest header(String headerName, Object value) {}
    public ClientRequest cookie(String cookieName, Object value) {}
    public ClientRequest cookie(Cookie cookie) {}
    public ClientRequest pathParameter(String parameterName,
                                       Object value)
    public ClientRequest body(String contentType, Object data) {}
...
}
```

The `ClientRequest` constructor can take any expression you put in an `@Path` annotation, but it must also have scheme, host, and port information. Here's an example of building up a GET request:

```
ClientRequest request =
                new ClientRequest("http://example.com/customers/{id}");
request.accept("application/xml")
       .pathParameter("id", 333);
```

We allocate the `ClientRequest` instance, passing in a URL template. We set the `Accept` header to state we want XML returned and the `id` path parameter to `333`. To invoke the request, call one of `ClientRequest`'s HTTP invocation methods listed here:

```
public class ClientRequest {
...
    // HTTP METHODS

    public <T> T getTarget(Class<T> returnType) throws Exception {}
    public <T> ClientResponse<T> get(Class<T> returnType)
                                                throws Exception {}
    public ClientResponse put() throws Exception {}
    public <T> ClientResponse<T> put(Class<T> returnType)
                                                throws Exception {}
```

---

† For more information, see *http://jboss.org/resteasy*.

```
    public ClientResponse post() throws Exception {}
    public <T> ClientResponse<T> post(Class<T> returnType)
                                        throws Exception {}
    public <T> T postTarget(Class<T> returnType) throws Exception {}
    public ClientResponse delete() throws Exception {}
    public <T> ClientResponse<T> delete(Class<T> returnType)
                                        throws Exception {}
...
}
```

Finishing up our GET request example, we invoke either the `get()` or `getTarget()` methods. The `getTarget()` method invokes the request and automatically converts it to the type you want returned. For example:

```
ClientRequest request =
            new ClientRequest("http://example.com/customers/{id}");
request.accept("application/xml")
        .pathParameter("id", 333);

Customer customer = request.getTarget(Customer.class);
```

In this example, our `Customer` class is a JAXB annotated class. When `ClientRequest.get Target()` is invoked, the request object builds the URL to invoke on using the passed in path parameter, sets the `Accept` header with the desired media type, and then calls the server. If a successful response comes back, the client framework matches the `Customer` class with an appropriate `MessageBodyReader` and unmarshalls the returned XML into a `Customer` instance. If an unsuccessful response comes back from the server, the framework throws an `org.jboss.resteasy.client.ClientResponseFailure` instead of returning a `Customer` object:

```
public class ClientResponseFailure extends RuntimeException {
...
 public ClientResponse<byte[]> getResponse() {}
}
```

If you need to get the response code or want to look at a response header from your HTTP invocation, you can opt to invoke the `ClientRequest.get()` method instead. This method returns an `org.jboss.resteasy.client.ClientResponse` object instead of the unmarshalled requested type. Here's what `ClientResponse` looks like:

```
public abstract class ClientResponse<T> extends Response {
    public abstract MultivaluedMap<String, String> getHeaders();
    public abstract Response.Status getResponseStatus();
    public abstract T getEntity();
...
    public abstract void releaseConnection();
}
```

The `ClientResponse` class extends the `javax.ws.rs.core.Response` class. You can obtain the status code by calling `ClientResponse.getStatus()` or `getResponseStatus()` if you want an enum. The `getEntity()` method unmarshalls the HTTP message body into the type you want. Finally, you must also call `releaseConnection()` when you are finished. Here's an example of using this class with the `ClientRequest.get()` method:

```
ClientRequest request =
                new ClientRequest("http://example.com/customers/{id}");
request.accept("application/xml")
      .pathParameter("id", 333);

ClientResponse<Customer> response = request.get(Customer.class);
try {
   if (response.getStatus() != 200)
      throw new RuntimeException("Failed!");

   Customer customer = response.getEntity();
} finally {
   response.releaseConnection();
}
```

Authentication

The RESTEasy Client Framework runs on top of Apache HttpClient. If you need to execute authenticated requests, you do so by configuring the Apache HttpClient backbone RESTEasy uses. You first set up a `DefaultHttpClient` with the appropriate configuration parameters set. Then you initialize an `org.jboss.resteasy.client.core.executors.ApacheHttpClient4Executor` passing in the `DefaultHttpClient` instance you created. This executor is used when instantiating `ClientRequest` objects. For example:

```
DefaultHttpClient client = new DefaultHttpClient();
client.getCredentialsProvider().setCredentials(
    new AuthScope("example.com", 443),
    new UsernamePasswordCredentials("bill", "geheim");
);
ApacheHttpClient4Executor executor =
                       new ApacheHttpClient4Executor(client);

ClientRequest request =
              new ClientRequest("https://example.com/customers/{id}",
                                    executor);
request.accept("application/xml")
      .pathParameter("id", 333);

ClientResponse<Customer> response = request.get(Customer.class);
try {
   if (response.getStatus() != 200)
      throw new RuntimeException("Failed!");

   Customer customer = response.getEntity();
} finally {
   response.releaseConnection();
}
```

RESTEasy also supports using Apache HttpClient 3.1 and `java.net.URLConnection` as a backbone if you can't use Apache HttpClient 4.x.

Advantages and Disadvantages

Using something like RESTEasy's Client Framework makes writing RESTful clients a lot easier, as you can take advantage of the variety of content handlers available in the Java community for JAX-RS. The biggest disadvantage of RESTEasy's Client Framework is that, although open source, it is not a standard. I mention it in this book not only because it is the project I am currently leading, but also because JAX-RS implementations like Jersey‡ and Apache CXF§ have very similar frameworks. There is a lot of interest from the JAX-RS specification lead and expert group members to get a standardized client framework baked into JAX-RS 2.0. I hope that by the time you finish reading this chapter, the effort is already well under way!

RESTEasy Client Proxies

The RESTEasy Client Proxy Framework is a different way of writing RESTful Java clients. The idea of the framework is to reuse the JAX-RS annotations on the client side. When you write JAX-RS services you are using the specification's annotations to turn an HTTP invocation into a Java method call. The RESTEasy Client Proxy Framework flips this around to instead use the annotations to turn a method call into an HTTP request.

You start off by writing a Java interface with methods annotated with JAX-RS annotations. For example, let's define a RESTful client interface to the customer service application we talked about over and over again throughout this book:

```
@Path("/customers")
public interface CustomerResource {

    @GET
    @Produces("application/xml")
    @Path("{id}")
    public Customer getCustomer(@PathParam("id") int id);

    @POST
    @Consumes("application/xml")
    public Response createCustomer(Customer customer);

    @PUT
    @Consumes("application/xml")
    @Path("{id}")
    public void updateCustomer(@PathParam("id") int id, Customer cust);
}
```

‡ For more information, see *http://jersey.dev.java.net*.

§ For more information, see *http://cxf.apache.org/*.

This interface looks exactly like the interface a JAX-RS service might implement. Through RESTEasy, we can turn this interface into a Java object that can invoke HTTP requests. To do this, we use the `org.jboss.resteasy.client.ProxyFactory` class:

```
CustomerResource client = ProxyFactory.create(CustomerResource.class,
                                     "http://example.com");
```

The `ProxyFactory.create()` method takes the interface you want to convert to a client proxy as well as a base URL to direct the invocations to. It returns an instance that you can invoke on. Here's the proxy in use:

```
// create a customer
Customer newCust = new Customer();
newCust.setName("bill");
Response response = createCustomer(newCust);

// Get a customer
Customer cust = client.getCustomer(333);

// Update a customer
cust.setName("burke");
client.updateCustomer(333, cust);
```

When you invoke one of the methods of the returned `CustomerResource` proxy, it converts the Java method call into an HTTP request to the server using the metadata defined in the annotations applied to the `CustomerResource` interface. For example, the `getCustomer()` invocation in the example code knows that it must do a GET request on the *http://example.com/customers/333* URI, because it has introspected the values of the `@Path`, `@GET`, and `@PathParam` annotations on the method. It knows that it should be getting back XML from the `@Produces` annotation. It also knows that it should unmarshal it using a JAXB `MessageBodyReader`, because the `getCustomer()` method returns a JAXB annotated class.

Advantages and Disadvantages

A nice side effect of writing Java clients with this proxy framework is that you can use the Java interface for Java clients and JAX-RS services. With one Java interface, you also have a nice, clear way of documenting how to interact with your RESTful Java service. As you can see from the example code, it also cuts down on a lot of boilerplate code. The disadvantage, of course, is that this framework, although open source, is proprietary. Like the client API discussed earlier, the JAX-RS expert group has shown a lot of interest in making this a part of JAX-RS 2.0.

Wrapping Up

In this chapter, you learned four ways to write RESTful clients in Java using the JDK's `java.net.HttpURLConnection` class, Apache HttpClient, RESTEasy's Client Framework, and RESTEasy's Proxy Framework. All four have their merits.

JAX-RS Implementations

The JAX-RS specification has been final for over a year now and there are a bunch of implementations out there that you can download and try. This chapter provides an overview of the most popular implementations by highlighting various custom features of each provider. It isn't meant as documentation for each of the frameworks described, but rather to show you some of the innovations that are going on in the JAX-RS community.

Jersey

Jersey is the open source reference implementation of JAX-RS. It is dual-licensed under both the Common Development and Distribution License (CDDL) version 1.0 and the GNU General Public License (GPL) version 2.

Jersey is built, assembled, and installed using Maven. The main project site, *https://jersey.dev.java.net*, contains a download button that links to instructions on how to get started with Jersey, its prerequisites, and links to samples. Jersey is also shipped with the application server GlassFish.[*]

Embeddable Jersey

While Jersey is usually deployed within a servlet container, it can also be embedded within simple Java programs. The following Java class represents a very simple example that uses the Grizzly embedded HTTP server to deploy a root resource class called `MyResource`. It uses the Jersey Client API to communicate with the resource:

```
import com.sun.grizzly.http.SelectorThread;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.container.grizzly.GrizzlyServerFactory;
import java.io.IOException;
```

[*] For more information, see *https://glassfish.dev.java.net*.

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

public class Main {

    @Path("resource")
    public static class MyResource {
        @GET
@Produces("text/plain")
        public String get() {
            return "Hello from Grizzly";
        }
    }

    public static void main(String[] args) throws IOException {
        SelectorThread st =
                GrizzlyServerFactory.create("http://localhost:9999");

        try {
            Client c = Client.create();
            WebResource r = c.resource("http://localhost:9999/resource");

            String s = r.get(String.class);
            System.out.println(s);
        } finally {
            st.stopEndpoint();
        }
    }
}
```

As you can see, it is very easy to get a JAX-RS service up and running with a few lines of code. You can also use this embeddable container easily with unit tests.

Client API

The Jersey client API is a high-level Java-based API for interoperating with any RESTful web service. Developers familiar with JAX-RS should find the Jersey client API complementary to their services, especially if the client API is utilized by those services themselves, or to test those services (the API is used extensively to test Jersey itself).

The goals of the Jersey client API are threefold:

- Encapsulate a key constraint of the REST architectural style, namely the Uniform Interface Constraint and associated data elements, as client-side Java artifacts.
- Make it as easy to interoperate with RESTful web services, as JAX-RS makes it easy to build RESTful web services.
- Leverage artifacts of the JAX-RS API for the client side. Note that JAX-RS is currently a server-side-only API.

Overall the Jersey Client API lets developers concisely and efficiently implement a reusable client-side solution that leverages existing and well-established HTTP client implementations (specifically `HttpURLConnection` and the Apache HTTP client).

Since the Jersey client API is very similar to the RESTEasy client API discussed in Chapter 13, I'm not going to talk a lot about it here. You can find more details on it in a whitepaper available at *www.sun.com/offers/details/Java_Jersey_Client.xml*.

WADL

The Web Application Description Language[†] (WADL) is an XML document format that defines a vocabulary for describing web applications. WADL aims to be the WSDL of RESTful web services. It allows service providers to document the resources offered, the HTTP methods that you may use on those resources, and the representation formats supported. Clients of the service can use WADL both for documentation, configuration, and code generation.

At runtime, Jersey will automatically generate a WADL document for a JAX-RS application. The application WADL is made available at */root/*`application.wadl`, where *root* is the base URI of the web application deployed. This generated WADL includes all of the root resources and as much metadata as can be extracted from the compiled resource classes—typically, the metadata supplied by the JAX-RS annotations. For example, let's say we had the following JAX-RS resource deployed within our application:

```
@Path("/helloworld")
public class MyResource {

    @GET
    @Produces("text/plain")
    public String getClichedMessage() {
        return "Hello world";
    }
}
```

If our JAX-RS application base URI is "/", a `GET /application.wadl` will return the following document:

```
<application xmlns="http://research.sun.com/wadl/2006/10">
    <doc xmlns:jersey="http://jersey.dev.java.net/"
        jersey:generatedBy="Jersey: 08/27/2008 08:24 PM"/>
    <resources base="http://localhost:9998/">
        <resource path="/helloworld">
            <method name="GET" id="getClichedMessage">
                <response>
                    <representation mediaType="text/plain"/>
                </response>
            </method>
        </resource>
```

---

† For more information, see *http://wadl.dev.java.net*.

```
            </resources>
        </application>
```

In addition to the application WADL document, Jersey will also generate a WADL document for an individual resource. Clients may obtain the WADL for an individual resource using an OPTIONS request with an `Accept` header value of `application/vnd.sun.wadl+xml` header. This allows clients to obtain WADL documents for dynamic resources made available via JAX-RS subresource locators.

WADL support is included in the Jersey core and is demonstrated by the *generate-wadl* and *extended-wadl-webapp* samples.

Data Formats

JAX-RS makes it easy to add support for any data type using `MessageBodyReader` and `MessageBodyWriter` providers. Jersey includes JAX-RS extension modules for working with common formats including Atom, JSON, and MIME Multipart data.

Atom

Two separate Jersey modules provide Atom support:

- `jersey-atom` is based on Rome[‡] and is used in the *simple-atom-server* sample.
- `jersey-atom-abdera` is based on Apache Abdera[§] and is used in the *atompub-contacts-model*, *atompub-contacts-server*, and *atompub-contacts-client* samples.

JSON

JSON support is provided by the `jersey-json` module, which integrates heavily with the Jettison framework discussed in Chapter 6. JSON can be used in two ways:

- You can use Jettison classes JSONObject and JSONArray as method parameters and return types.
- You can marshal and unmarshal JAXB classes to and from JSON in addition to XML. They are used in the *json-from-jaxb* and *jsonp* samples.

The latter is very convenient, since a method that uses JAXB classes as its input or output can support both XML and JSON without any additional work.

MIME multipart

A high-level API for working with multipart data is provided by the *jersey-multipart* module. You can use the `MultiPart` and `FormDataMultiPart` classes as resource method parameters or return types. The methods of the class make it easy to work with the

---

‡ For more information, see *http://rome.dev.java.net*.

§ For more information, see *http://abdera.apache.org*.

contents. In addition, you can use the annotation `@FormDataParam` to extract information from a `multipart/form-data` request entity. For example:

```
@Path("/service")
public class MyService {

    @POST
    @Consumes("multpart/form-data")
    public void post(@FormDataParam("file1") File image,
                     @FormDataParam("description") String description) {
     ...
    }
}
```

As you can see, this annotation is very similar to the `@FormParam` annotation that is defined in JAX-RS, except that it works with the `multipart/form-data` format instead of the `application/x-www-form-urlencoded`.

Model, View, and Controller

One of the more interesting features of Jersey is its capability to provide support for the (Model, View, and Controller) MVC pattern. The Controller is a JAX-RS resource class, the Model is any Java object, and the View is a reference to a template capable of processing the Model to produce a representation (such as an HTML document).

A JAX-RS resource method can return an instance of the `com.sun.jersey.api.view.View` `able` class that has a reference to the template you want to invoke and the Model you want to initialize it with. The `Viewable` will then be processed by a template processor that supports the referenced template. Jersey supplies template processors for Java Server Pages (JSP) and Lift templates. (Lift is a Web framework written in Scala.) For example, consider this JAX-RS Controller and a View that is a JSP:

```
@Path("/service")
public class MyResource {
    @GET
    public Viewable get() {
      return new Viewable("/index", "String Model Data")
    }
}
```

If there is a `/index.jsp` in your web application, Jersey will forward the HTTP request to this JSP and initialize a request attribute called `it` with the "String Model Data" model data you initialized the `Viewable` class with. The JSP can reference it through regular mechanisms:

```
<h1>${it}</h1>
```

Developers may integrate their own template engines by implementing the interface `com.sun.jersey.spi.template.TemplateProcessor`. For example, some developers have implemented support for Freemarker and the StringTemplate.

Component Integration

Jersey provides a framework for developers to plug in dependency injection frameworks. Support for EJB, Spring, and Guice 2.0 are provided such that Spring and Guice 2.0 can manage instances of resource or provider classes. I talked a bit about Spring integration in Chapter 11, but let's take a look at how Jersey does it.

Jersey works pretty seamlessly with Spring. All you need to do to integrate JAX-RS with a Spring application is to add a little configuration to your *web.xml* file. Here is an example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:applicationContext.xml</param-value>
    </context-param>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <listener>
        <listener-class>
         org.springframework.web.context.request.RequestContextListener
        </listener-class>
    </listener>
    <servlet>
        <servlet-name>Jersey Spring Web Application</servlet-name>
        <servlet-class>
            com.sun.jersey.spi.spring.container.servlet.SpringServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
         <servlet-name>Jersey Spring Web Application</servlet-name>
         <url-pattern>/webresources/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Using Spring with Jersey first involves setting up some basic Spring configuration. The `contextConfigLocation` context parameter and the `ContextLoaderListener` and `RequestContextListener` are standard configuration elements of any typical Spring-based web application. They load and initialize the beans defined in the XML file referenced in the `contextConfigLocation` variable. Jersey has a special servlet class called `com.sun.jersey.spi.spring.container.servlet.SpringServlet` that manages the bindings between JAX-RS and Spring. Other than annotating your Spring beans with JAX-RS annotations, this is all you have to do to get up and running.

Apache CXF

Apache CXF[||] is a popular open source web services framework. It was originally just a SOAP stack, but they recently added support for JAX-RS. The CXF philosophy is that different web service styles can coexist. You'll see a lot of this when you use the framework to write your RESTful web services.

Apache CXF has a few nice extensions to JAX-RS that I'd like to point out.

Aggregating Request Parameters into Beans

RESTful services can often depend on a large number of request parameters. Listing multiple parameters in resource method signatures may end up not very readable and quite brittle, for example:

```
@Path("stores/{storeid}")
public class BookStores {
    @GET @Path("books/{publication}")
    public Books getBooks(
                    @PathParam("storeid") int storeid,
                    @PathParam("publication") Date publicationDate,
                    @MatrixParam("sortKey1") String sortKey1,
                    @MatrixParam("sortKey2") String sortKey2,
                    @MatrixParam("publication") Date publicationDate,
                    @QueryParam("author") String name,
                    @QueryParam("count") int responseListSize) {}
}
```

If you need access to a lot of injectable data, your JAX-RS resource methods can get quite large and cumbersome. Apache CXF allows you to aggregate this information into specific injectable bean classes instead. For example:

```
@Path("stores/{storeid}")
public class BookStores {
    @GET @Path("books/{publication}")
    public Books getBooks(@PathParam("") PathBean pathParams,
                          @MatrixParam("") MatrixBean matrixParams,
                          @QueryParam("") QueryBean queryParams) {}
}
```

`PathBean` is a plain Java bean class that has `storeid` and `publication` properties:

```
Public class PathBean {
    ...
    public int getStoreid() {...}
    public void setStoreid(int id) {...}

    public Date getPublication() {...}
    public void setPublication(Date date) {...}
}
```

[||] For more information, see *http://cxf.apache.org*.

Apache CXF will automatically instantiate the PathBean and inject the appropriate path parameters directly into the properties of the PathBean instance. This pattern would work similarly for the other injected parameters.

You can get even funkier with this method of injection. CXF also supports creating and injecting nested Java objects. For example, consider this object model:

```
class Name
{
    String first;
    String last;
}
class Address
{
    String city;
    String state;
}
class Person
{
    Name legalName;
    Address homeAddr;
    String race;
    String sex;
    Date birthDate;
}
```

Let's say you also had the following GET request posted to your JAX-RS system:

```
GET /getPerson?sex=M&legalName.first=John&legalName.last=Doe&
              homeAddr.city=Reno&homeAddr.state=NV
```

You could have CXF instantiate a Person object and initialize it with all its contained objects from data within the URI's query string:

```
@GET
public String doQuery(@QueryParam("") Person person) {...}
```

Converting Request Parameters into Custom Types

The JAX-RS 1.0 specification explains how to convert request parameters into Java types using a static valueOf() method or a constructor that takes a String parameter. In some cases, it is not possible to convert a given parameter value into a given type instance using the default conversion algorithm. For example, users sometimes wish to pass encoded XML fragments as query parameters. CXF JAX-RS has introduced a ParameterHandler extension to deal with such cases. For example:

```
public XmlToBean implements ParameterHandler<CustomBean> {
    public CustomBean fromString(String decodedQueryParameter) {
        // convert to XML
    }
}
```

This custom parameter handler has to be registered as a provider with the Apache CXF endpoint and will be checked by the runtime after the default conversion algorithm has been tried.

Static Resolution of Subresources

As you saw in Chapter 4, JAX-RS subresource locators and classes are resolved dynamically at runtime to process incoming requests. Apache CXF offers an option to resolve all the subresource bindings statically. Resolving subresources statically leads to faster processing times if your JAX-RS application makes use of subresource classes.

Client API

Apache CXF has both a programmatic API and a proxy-based mechanism similar to what was discussed in Chapter 13. Since it is so similar, I won't discuss it in detail here.

Supporting Services Without JAX-RS Annotations

JAX-RS is an annotation framework. All metadata about request processing is described by annotating your Java classes. Some developers find describing metadata through annotations too inflexible, as they have to recompile and build their applications to make one simple change. To placate these users, Apache CXF offers a simple extension to JAX-RS to define this metadata within an XML deployment descriptor. For example:

```
<model xmlns="http://cxf.apache.org/jaxrs">
   <resource name="org.bookstores.BookInterface" path="book/{id}">
      <operation name="getBook" verb="GET">
         <param name="id" type="PATH"/>
      </operation>
   </resource>
</model>
```

You can use this model instance in both the client and server frameworks of Apache CXF.

Intercepting Requests and Responses

It is often necessary to do some pre- and postprocessing of HTTP invocations on the client and server in a nonintrusive way. Apache CXF has many options for doing request interception. It has common interception facilities that you can use with both JAX-RS and Apache CXF's SOAP stack. In addition, Apache CXF allows you to register request and response filters, custom invokers, and custom STAX readers and writers for pre- or postprocessing XML data.

Promoting XSLT and XPath As First-Class Citizens

JAX-RS resources may consume and produce data in various formats. XML is by far the most popular format. XSLT and XPath are common powerful utilities that developers use to consume XML documents.

Apache CXF provides a JAXB-based XSLT provider that you can use to produce and consume XML instances and adapt them as expected. This XSLT provider can produce not only XML, but also JSON and XHTML.

XPath is supported at the provider level, where XPath expressions can be applied to produced or consumed XML. It can also be utilized explicitly by the HTTP-centric CXF web clients.

Support for Suspended Invocations

A suspended invocation is one that can be temporarily released by a transport thread and processed by a different thread. Apache CXF has a transport-neutral Continuations API, which has been inspired by the Jetty Continuations project.[#] It gives the application code explicit control of the life cycle of a given invocation so that it can perform asynchronous activity. You can use this API across both the JAX-RS and SOAP stacks of the Apache CXF framework.

Support for Multipart Formats

Apache CXF supports most multipart formats and has a similar API to both Jersey and RESTEasy. Since it is so similar, I won't go into a lot of detail here.

Integration with Distributed OSGi RI

Apache CXF has both an implementation of the Distributed OSGi (DOSGi) specification and JAX-RS integration.[*] In a nutshell, DOSGi is an OSGi alliance attempt to standardize how Java services can be autoexposed and consumed as web services in OSGi-based containers. Apache CXF provides support for exposing Java objects as RESTful services within a DOSGi environment.

Support for WADL

Currently, CXF supports the autogeneration of WADL instances. The whole application can be described if a `staticSubresourceResolution` property has been set.

[#] For more information, see *http://docs.codehaus.org/display/JETTY/Continuations*.

[*] For more information, see *http://cxf.apache.org/distributed-osgi.html*.

Component Integration

Apache CXF provides integration with the Spring framework. It is very similar to both the Jersey and RESTEasy implementations, so I won't go into detail here.

JBoss RESTEasy

JBoss RESTEasy is Red Hat's implementation of JAX-RS and is the project I lead and run at Red Hat. It is licensed under the GNU Lesser General Public License (LGPL) and can be used in any environment that has a servlet container. Many of its features overlap with other JAX-RS implementations, so I'll highlight only distinguishing features here.

Embedded Container

An important part of software development is unit testing. Personally, I prefer to work with as many of the real subsystems as possible when writing unit tests instead of mocking them up. Mocks can be useful, but many times your code behavior is different when you run it within your deployment environment. When I created RESTEasy, I wanted it to be embeddable within unit tests so that developers could make over-the-wire invocations to their JAX-RS resources without having to install, build, and run an entire servlet container. Given that, speed is such an important factor when running unit tests because of how often you run them during your development cycle. With this in mind, I wanted to pick an embeddable servlet container that had the best boot-time performance. After some research I picked TJWS,[†] a tiny, relatively unknown, embeddable servlet container.

Using the embeddable container is really easy. Here's an example:

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import org.jboss.resteasy.plugins.server.tjws.TJWSEmbeddedJaxrsServer;
import org.jboss.resteasy.spi.ResteasyDeployment;
import org.jboss.resteasy.client.ClientRequest;

public class Main {

   @Path("resource")
   public static class MyResource {
   @GET
   @Produces("text/plain")
      public String get() {
         return "Hello from RESTEasy";
      }
   }
```

---

† For more information, see *http://tjws.sourceforge.net*.

```
    public static void main(String[] args) throws Exception {
        TJWSEmbeddedJaxrsServer server = new TJWSEmbeddedJaxrsServer();
        server.setPort(9095);
        server.getDeployment()
                .getRegistry().addPerRequestResource(MyResource.class);

        try {
            ClientRequest request =
                    new ClientRequet("http://localhost:9095/resource");
            String msg = request.getTarget(String.class);
            Client c = Client.create();
            System.out.println(s);
        } finally {
            server.stop();
        }
    }
}
```

Asynchronous HTTP

Asynchronous HTTP is a relatively new technique that allows you to process a single
HTTP request using nonblocking I/O and, if desired, within separate threads. Some
refer to it as COMET. The primary use case for Asynchronous HTTP is when you have
a lot of clients making blocking requests. The typical scenario is when a client polls a
server, waiting for an event to be fired off by the server. This happens a lot in Ajax
clients, where you have the server pushing data to the client. The most common ex-
ample is a chat application.

These scenarios have the client blocking a long time on the server's socket, waiting for
a new message. This can create havoc on the performance of your server in high-load
situations. In synchronous I/O, you have a thread-per-socket model. In most applica-
tions, this isn't such a big deal, but when you start getting thousands of clients blocking
and waiting, the number of threads your OS has spawned starts to become an issue.
Although lightweight, threads are still a bit expensive, as they eat up memory and
system resources.

A number of servlet containers have solved this problem by allowing application code
to process HTTP responses within a separate thread. Recently, this has also been
standardized in the Servlet 3.0 specification. RESTEasy provides a very simple JAX-RS-
friendly abstraction that works on all these various containers. Here's a simple example
of using it:

```
import org.jboss.resteasy.annotations.Suspend;
import org.jboss.resteasy.spi.AsynchronousResponse;

@Path("/")
public class SimpleResource
{
```

```
            @GET
            @Path("basic")
            @Produces("text/plain")
            public void getBasic(final
                              @Suspend(10000) AsynchronousResponse response)
                                                   throws Exception
            {
               Thread t = new Thread()
               {
                  @Override
                  public void run()
                  {
                     try
                     {
                        Thread.sleep(3000);
                        Response jaxrs = Response.ok("basic")
                                            .type(MediaType.TEXT_PLAIN)
                                            .build();
                        response.setResponse(jaxrs);
                     }
                     catch (Exception e)
                     {
                        e.printStackTrace();
                     }
                  }
               };
               t.start();
            }
         }
```

The `@org.jboss.resteasy.annotations.Suspend` annotation tells RESTEasy to suspend
the request processing of the current thread so that a different thread can process
the HTTP response. `@Suspend` injects a simple callback interface `org.jboss.rest`
`easy.spi.AsynchronousResponse`. The thread responsible for processing the response
creates a JAX-RS `Response` object and calls the `setResponse()` method of the
`AsynchronousResponse` class to finish up the request.

Interceptor Framework

RESTEasy has the capability to intercept JAX-RS invocations and route them through
listener-like objects called interceptors. There are four different interception points on
the server side: wrapping around `MessageBodyWriter` invocations, wrapping around
`MessageBodyReader` invocations, preprocessors that intercept the incoming request be-
fore anything is unmarshalled, and postprocessors that are invoked right after the
JAX-RS method is finished. On the client side, there are three different interception
points. You can intercept `MessageBodyReader` and `Writer` processing as well as the re-
mote invocation to the server.

Client "Browser" Cache

RESTEasy has the capability to set up a client-side cache that can store HTTP responses much in the same way your browser can. It is really easy to set up. Here's an example:

```
import org.jboss.resteasy.client.ProxyFactory;
import org.jboss.resteasy.client.cache.CacheFactory;
import org.jboss.resteasy.client.cache.LightweightBrowserCache;

public static void main(String[] args) throws Exception
{
    RegisterBuiltin.register(ResteasyProviderFactory.getInstance());

    // This line enables caching
    LightweightBrowserCache cache = new LightweightBrowserCache();

    ClientRequest request =
                new ClientRequest("http://example.com/orders/333");
    CacheFactory.makeCacheable(request, cache);
}
```

The feature obeys the semantics of the `Cache-Control` header discussed in Chapter 10, including conditional GETs and cache revalidation.

Server-Side Caching

RESTEasy also has an in-memory server-side cache that sits on top of your application. If turned on, it will cache marshalled representations of your JAX-RS resource method invocations and serve them up to requesting clients. This can greatly improve the performance of your system if you have business logic that is expensive to execute. Beyond caching, the service handles the generation and return of `Cache-Control` and `ETag` headers. It also manages conditional GET calls from the client by matching cached `ETag` headers with `If-None-Match` headers.

GZIP Compression

As you learned in Chapter 10, request and response body compression can help maximize the throughput of your HTTP invocations. RESTEasy has built-in support for GZIP compression for both the client and server-side frameworks. Both the client and the server will automatically compress and decompress requests and responses that have the `Content-Encoding` header set to `gzip`. On the client side, the framework automatically sets the `Accept-Encoding` header to `deflate, gzip`.

Data Formats

Like the other JAX-RS implementations discussed in this chapter, RESTEasy has support for the most popularly exchanged data formats on the Internet, including XML (through JAXB), JSON (Jettison and Jackson), Atom, XOP, Fastinfoset, and Multipart.

I won't get into a lot of details here because the support is very similar to that of the other frameworks, with a few exceptions.

Atom

Like Jersey, RESTEasy has support for the Apache Abdera project. While I like the Abdera and the Sun ROME project, I do think they are a bit of overkill for JAX-RS applications. They also don't have support for JAXB. If you're using JAX-RS and Atom together, you're probably not writing blog feeds, but instead writing web services that are exchanging XML content. You'll want to be able to map this XML content to Java objects, and JAXB is crucial for this. Because of these two things, I decided to write a simple JAXB annotated object model for Atom that supports custom JAXB content. It's very lightweight and simple.

Multipart

Like Jersey and Apache CXF, RESTEasy has an explicit API for managing multipart data. From a usability perspective, I decided to take this a bit further. Working with a specific API can be a bit tedious sometimes. If you have uniform sets of data encapsulated in a multipart request, there's no reason a framework like RESTEasy can't make your job a little bit easier.

For the `multipart/mixed` format, RESTEasy allows you to use a `java.util.List` of your own custom objects and not worry about interacting with an explicit multipart API. Here's an example:

```
@Path("/customers")
public class CustomerResource {

    @POST
    @Produces("multipart/mixed")
    @PartType("application/xml")
    @Consumes("multipart/mixed")
    public List<Customer> post(List<Customer> input) {...}
}
```

For the `CustomerResource.post()` method, RESTEasy unmarshalls the incoming request into a list of `Customer` objects. For each part of the incoming message body, it will find a `MessageBodyReader` based on the `Content-Type` of the part and the generic type of the `List`. In this case, it is `application/xml` and the `Customer` class.

Converting a `List` of Java objects to a `multipart/mixed` message requires a bit more information, as RESTEasy won't know which data format to use for each part. The `@PartType` annotation defines the format of the part.

RESTEasy has similar support for `multipart/form-data` and `java.util.Map`. Here's an example:

```
@Path("/customers")
public class CustomerResource {

    @POST
    @Produces("multipart/mixed")
    @PartType("application/xml")
    @Consumes("multipart/mixed")
    public Map<String, Customer>
        updateCustomersByName(Map<String, Customer> input) {...}
}
```

For a map, RESTEasy assumes that the key is the name of the part.

Component Integration

RESTEasy supports tight integration with JBoss Seam, EJB, Spring, and Guice. Since its integration is very similar to that of Jersey, I won't go into a lot of detail on these features.

Wrapping Up

In this chapter, you got a brief overview of three JAX-RS implementations. Even though they all implement the same specification, you can see there is still a lot of room for innovation. If you like some of these vendor-specific features, I suggest you bug the JAX-RS expert group to get them into version 2.0 of the specification.

JAX-RS Workbook

Workbook Introduction

Reading a book on a new technology gives you a nice foundation to build on, but you cannot truly understand and appreciate a new technology until you see it in action. The following workbook chapters were designed to be a companion to the main chapters of this book. Their goal is to provide step-by-step instructions for installing, configuring, and running various JAX-RS examples found throughout this book with the JBoss RESTEasy framework.

This chapter focuses on downloading and installing RESTEasy and the workbook examples. Following this, each workbook chapter corresponds to a specific chapter in the book. For example, if you are reading Chapter 3 on writing your first JAX-RS service, use Chapter 16 of the workbook to develop and run the examples shown in that chapter with RESTEasy.

This workbook is based on the production release of JBoss RESTEasy JAX-RS 1.2. I picked RESTEasy as the JAX-RS framework for the workbook for no other reason than I am the project lead for it and I know it backward and forward. That said, I took great care to ensure you can easily port the examples to other JAX-RS implementations.

Installing RESTEasy and the Examples

The workbook examples are embedded within the RESTEasy distribution so that as future versions of RESTEasy are released, the workbook examples will be updated along with that release. (I discovered that having a separate download for the workbook examples causes various problems—users can get confused about which package to download, and the examples can get out of sync with specific software versions.)

You can download the distribution by following the download links at:

http://jboss.org/resteasy

Download the latest RESTEasy JAX-RS distribution, e.g., *resteasy-jaxrs-1.2.GA.zip*. Figure 15-1 shows the directory structure of the distribution.
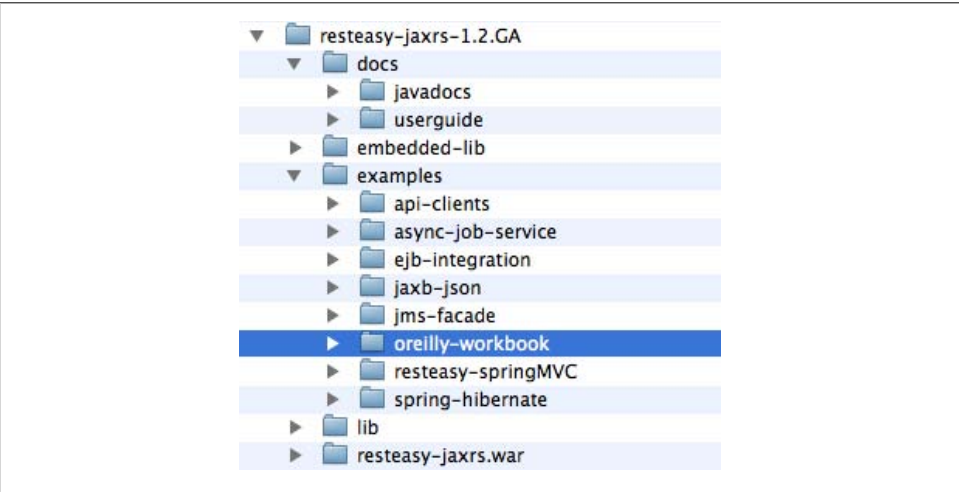
Figure 15-1. RESTEasy directory structure

Table 15-1 describes the purpose of the various directories.

Table 15-1. RESTEasy directories

Directory	Description
docs/javadocs	Generated Javadocs for both the JAX-RS APIs and RESTEasy
docs/userguide	Reference guide for RESTEasy in both HTML and PDF format
examples	Top-level directory containing all RESTEasy examples
examples/oreilly-workbook	Contains all the workbook example code for each workbook chapter
lib	All the RESTEasy jars and the third-party libraries they depend on
embedded-lib	Optional jar files used when you are running RESTEasy in embedded mode
resteasy-jaxrs.war	Sample RESTEasy servlet deployment

For Apache Maven users, RESTEasy also has a Maven repository at:

http://repository.jboss.org/maven2

The `groupId` for all RESTEasy artifacts is `org.jboss.resteasy`. You can view all available artifacts at this URL:

http://repository.jboss.org/maven2/org/jboss/resteasy

Example Requirements and Structure

The RESTEasy distribution does not have all the software you need to run the examples. You will also need the following components:

- JDK 5.0 or later. You will, of course, need Java installed on your computer.
- Maven 2.0.10. Maven is the build system used to compile and run the examples. Later versions of Maven may work, but it is recommended that you use 2.0.10. You can download Maven from *http://maven.apache.org*.

Code Directory Structure

The example code is organized as a set of directories, one for each exercise (see Figure 15-2). You'll find the server source code for each example in the *src/main/java* directory. The servlet configuration for each example lives in the *src/main/webapp/WEB-INF* directory. The client code that runs the example is in *src/test/java*.
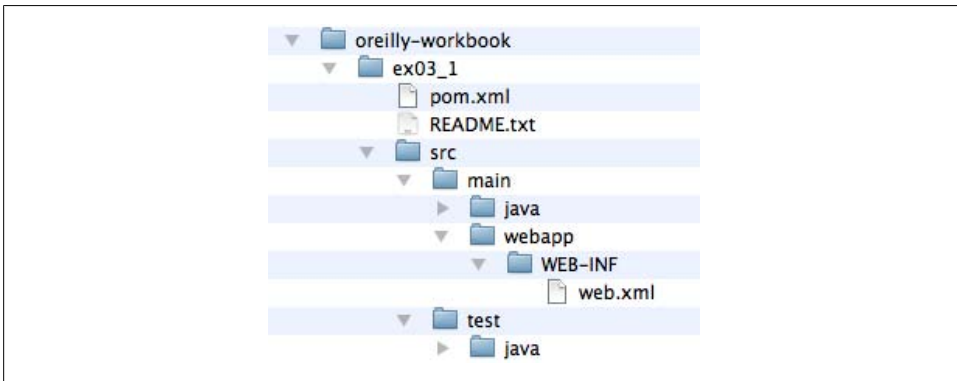
Figure 15-2. Code directory structure

To build and run the exercises, you'll use the Maven build tool. A *pom.xml* is provided at the top-level directory of each example. It contains the Maven configuration needed to compile, build, and run the specific example.

Environment Setup

For Maven to work correctly, you will have to make sure the Maven scripts are in your path. Depending on your platform, you'll have to execute commands like these:

- Windows:

```
C:\> set PATH=\maven\bin;%PATH%
```

- Unix:

```
$ export PATH=/home/username/maven/bin:$PATH
```

In each chapter, you'll find detailed instructions on how to build, deploy, and run the exercise using Maven.

Examples for Chapter 3

Chapter 3 walked you through a very basic example of creating a JAX-RS service. This service was a simple in-memory customer database. It was modeled as a singleton JAX-RS resource class and exchanged simple XML documents.

This chapter takes the code from Chapter 3 and shows you how to run it using the downloadable workbook example code. I'll walk you through how the code is structured on disk as well as how the examples use the Maven build system to compile, build, and run it.

Build and Run the Example Program

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex03_1* directory of the workbook example code.
2. Make sure your PATH is set up to include both the JDK and Maven, as described in Chapter 15.
3. Perform the build by typing `maven install`. Maven uses *pom.xml* to figure out what to compile, build, and run the example code.

Before we examine the build file for this example, you might want to take a quick look at the Maven utility at its Apache website at *http://maven.apache.org*.

Maven is a build-by-convention tool. It expects that your source code be laid out in a certain directory structure. From this standard directory structure, it knows how to automatically find, compile, and package your main class files. It also knows where your test code is and will compile and run it.

Every exercise in this book will follow the directory structure shown in Figure 16-1. Table 16-1 describes the purpose of the various directories.
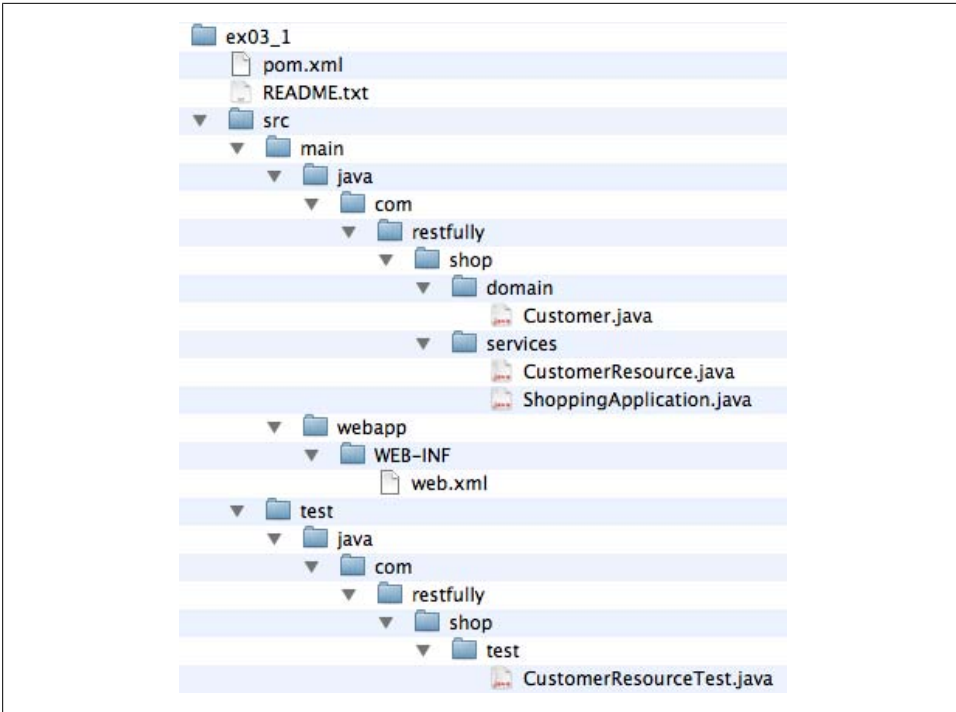
Figure 16-1. Example directory structure

Table 16-1. Directory structure description

Directory	Description
src	Top-level directory that contains all source and configuration files.
src/main	Contains all Java source code and configuration files that are used to create your package. In this case, we're creating a WAR file.
src/main/java	Contains server-side Java source code.
src/main/webapp	Contains servlet configuration files, specifically *web.xml*.
src/test/java	Contains Java source code that will be used to run tests on the packaged archive. This code will not be included within our WAR file.

Deconstructing pom.xml

The *pom.xml* file provided for each workbook exercise gives the Maven utility information about how to compile and deploy your Java programs. In our case, Maven will use the information within the *pom.xml* file to compile the code within *src/main/java*, create a WAR file using the *web.xml* file within *src/main/webapp*, deploy the WAR file automatically using the Jetty-embedded servlet container, and finally, run any test code that is within the *src/test/java* directory.

Here's a breakdown of what is contained with *pom.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.oreilly.rest.workbook</groupId>
    <artifactId>ex03_1</artifactId>
```

`artifactId` is the name of the project. It will also be used for the name of the WAR file that is created by the build. This artifact belongs to a family of packages defined by the element `groupId`:

```
<version>1.0</version>
```

The `version` element identifies the version of the project we are creating. This version text will also be appended to the `artifactId` when Maven creates the WAR file. After building, you will see that the created file is named *target/ex03_1-1.0.war*:

```
<packaging>war</packaging>
```

The `packaging` element tells Maven that this project is building a WAR file. Other values for packaging could be `jar`, if we're creating a jar, or an `ear` for a Java EE enterprise archive:

```
<name/>
<description/>

<repositories>
    <repository>
        <id>java.net</id>
        <url>http://download.java.net/maven/1</url>
        <layout>legacy</layout>
    </repository>
    <repository>
        <id>maven repo</id>
        <name>maven repo</name>
        <url>http://repo1.maven.org/maven2/</url>
    </repository>
    <!-- For resteasy -->
    <repository>
        <id>jboss</id>
        <name>jboss repo</name>
        <url>http://repository.jboss.org/maven2</url>
    </repository>
</repositories>
```

The next interesting piece of configuration is the `repositories` element. It defines a list of locations where Maven should look for the binary dependencies this project may have. What's cool about Maven is that you do not need to package every library your project depends on. Maven will try to find the third-party dependencies of your project from the URLs listed in the `repository` elements defined:

```
<dependencies>
    <dependency>
        <groupId>org.jboss.resteasy</groupId>
        <artifactId>resteasy-jaxrs</artifactId>
        <version>1.2</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.1</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

The `dependencies` element lists all library dependencies our `ex03_1` project needs to compile and run. We are dependent on the RESTEasy project, as this is the JAX-RS implementation we are using. We are also dependent on the JUnit library for running the test code in our project. Prior to building, Maven will search for these libraries within the repositories listed under the `repositories` element. It will then download these libraries to your machine along with each of the transitive dependencies that these libraries have. What do I mean by transitive dependencies? Well, for example, RESTEasy depends on a multitude of third-party libraries like the servlet and JAXB APIs. The repository in which RESTEasy resides contains metadata about RESTEasy's dependencies. Maven will discover these extra dependencies when it tries to download the RESTEasy jar:

```
<build>
    <plugins>
```

The `build` element contains a set of `plugins` that will be used to compile and run the code within the project:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <source>1.5</source>
        <target>1.5</target>
    </configuration>
</plugin>
```

The first `plugin` listed is the compiler plug-in, which is used to configure the Java compiler. Here, we're just saying we want our source code compiled into the Java 5 bytecode format:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <configuration>
        <skip>true</skip>
    </configuration>
    <executions>
        <execution>
            <id>surefire-it</id>
```

```
                    <phase>integration-test</phase>
                    <goals>
                        <goal>test</goal>
                    </goals>
                    <configuration>
                        <skip>false</skip>
                    </configuration>
                </execution>
            </executions>
        </plugin>
```

The next plug-in we need to configure is `surefire-it`. This plug-in controls how our test execution works. By default, Maven will compile the source code under *src/main/java* and *src/test/java* and then try to run the tests under *src/test/java*. If the tests succeed, it packages the main source code into the package you want to create. In our case, though, we want to create the WAR file and deploy it to the Jetty-embedded servlet container before we run our test code. The *surefire-it* configuration listed tells Maven not to run the test code until the WAR file has been built and deployed to Jetty:

```
        <plugin>
            <groupId>org.mortbay.jetty</groupId>
            <artifactId>maven-jetty-plugin</artifactId>
            <version>6.1.15</version>
            <configuration>
                <contextPath>/</contextPath>
                <scanIntervalSeconds>2</scanIntervalSeconds>
                <stopKey>foo</stopKey>
                <stopPort>9999</stopPort>
                <connectors>
                    <connector implementation=
            "org.mortbay.jetty.nio.SelectChannelConnector">
                        <port>9095</port>
                        <maxIdleTime>60000</maxIdleTime>
                    </connector>
                </connectors>
            </configuration>
    ...
        </plugin>
```

The final plug-in is the Jetty plug-in, which is responsible for running the Jetty-embedded servlet container. After the WAR file is built, the Jetty container will boot up an HTTP server under port 9095. The WAR file is then deployed into Jetty.

I don't really need to explain the specifics of the entire Jetty plug-in configuration. The interesting bits that you might want to tweak are the `port` (9095) and the `stopPort` (9999). You may have to change these if there is a service on your computer already using these network ports:

```
        </plugins>
    </build>
</project>
```

Running the Build

To run the build, simply type `mvn install` at the command prompt from the *ex03_1* directory. The output will look something like this:

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------
[INFO] Building
[INFO]    task-segment: [install]
[INFO] ------------------------------------------------------------
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
Downloading: http://download.java.net/maven/1
                    /org.jboss.resteasy/poms/resteasy-jaxrs-1.1.GA.pom
...
```

You'll see Maven downloading a bunch of files from the repositories. This may take a while the first time you run the build script, as Maven needs to pull down a huge number of dependencies:

```
[INFO] [compiler:compile]
[INFO] Compiling 3 source files to /oreilly-workbook/ex03_1
                                            /target/classes
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] Compiling 1 source file to /oreilly-workbook/ex03_1
                                            /target/test-classes
```

Next, you'll see Maven compiling your main and test source code:

```
[INFO] [surefire:test]
[INFO] Tests are skipped.
[INFO] [war:war]
[INFO] Packaging webapp
[INFO] Assembling webapp[ex03_1] in
[/Users/billburke/jboss/resteasy-jaxrs/examples/oreilly-workbook/ex03_1/
target/ex03_1-1.0]
[INFO] Processing war project
[INFO] Webapp assembled in[154 msecs]
[INFO] Building war: /oreilly-workbook/ex03_1/target/ex03_1-1.0.war
```

Then you'll see that the WAR file is built:

```
...
[INFO] Started Jetty Server
[INFO] [surefire:test {execution: surefire-it}]
[INFO] Surefire report directory:
/Users/billburke/jboss/resteasy-jaxrs/examples/oreilly-workbook/ex03_1/target/
surefire-reports


-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.restfully.shop.test.CustomerResourceTest
```

```
*** Create a new Customer ***
Created customer 1
Location: http://localhost:9095/rest-services/customers/1
*** GET Created Customer **
Content-Type: application/xml
<customer id="1">
   <first-name>Bill</first-name>
   <last-name>Burke</last-name>
   <street>256 Clarendon Street</street>
   <city>Boston</city>
   <state>MA</state>
   <zip>02115</zip>
   <country>USA</country>
</customer>
**** After Update ***
Content-Type: application/xml
<customer id="1">
   <first-name>William</first-name>
   <last-name>Burke</last-name>
   <street>256 Clarendon Street</street>
   <city>Boston</city>
   <state>MA</state>
   <zip>02115</zip>
   <country>USA</country>
</customer>
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0,
              Time elapsed: 0.169 sec
```

Finally, Maven will start Jetty, deploy the WAR file created, and run the test code under *src/test/java*:

```
Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0


...
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------
[INFO] Total time: 35 seconds
[INFO] Finished at: Tue Jul 07 16:50:41 EDT 2009
[INFO] Final Memory: 12M/22M
[INFO] ------------------------------------------------------------
2009-07-07 16:50:42.022::INFO:  Shutdown hook executing
2009-07-07 16:50:42.024::INFO:  Shutdown hook complete
bill-burkes-computer:ex03_1 billburke$
```

The output of the build should end with BUILD SUCCESSFUL.

Examining the Source Code

The server-side source code is exactly as posted in Chapter 3. What we haven't already gone over is the client code for this example. The client code is structured as a JUnit class. JUnit is an open source Java library for defining unit tests. Maven automatically

knows how to find JUnit-enabled test code and run it with the build. It scans the classes within the *src/test/java* directory, looking for classes that have methods annotated with @org.junit.Test. This example has only one: com.restfully.shop.test.Customer ResourceTest. Let's go over the code for it:

src/test/java/com/restfully.shop.test.CustomerResourceTest.java

```java
package com.restfully.shop.test;

import org.junit.Assert;
import org.junit.Test;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.URL;


public class CustomerResourceTest {
    @Test
    public void testCustomerResource() throws Exception  {
```

Our test class has only one method: testCustomerResource(). It is annotated with @Test. This tells Maven that this is a JUnit test:

```java
        System.out.println("*** Create a new Customer ***");
        // Create a new customer
        String newCustomer = "<customer>"
                + "<first-name>Bill</first-name>"
                + "<last-name>Burke</last-name>"
                + "<street>256 Clarendon Street</street>"
                + "<city>Boston</city>"
                + "<state>MA</state>"
                + "<zip>02115</zip>"
                + "<country>USA</country>"
                + "</customer>";

        URL postUrl = new URL("http://localhost:9095/customers");
        HttpURLConnection connection =
                        (HttpURLConnection) postUrl.openConnection();
        connection.setDoOutput(true);
        connection.setInstanceFollowRedirects(false);
        connection.setRequestMethod("POST");
        connection.setRequestProperty("Content-Type",
                                      "application/xml");
        OutputStream os = connection.getOutputStream();
        os.write(newCustomer.getBytes());
        os.flush();
```

Our first bit of code creates a customer by invoking on the JAX-RS service defined in Chapter 3. We use the java.net.HttpURLConnection class to open a connection to our service and to POST an XML representation of the new customer that we want to create. Chapter 13 talks in detail about how to use HttpURLConnection:

```
    Assert.assertEquals(HttpURLConnection.HTTP_CREATED,
    connection.getResponseCode());
    System.out.println("Location: "
                        + connection.getHeaderField("Location"));
    connection.disconnect();
```

After posting the data, we check that the response code is 201, "Created." The
`org.junit.Assert.assertEquals()` method makes sure that the value is 201. If the check
fails, the test fails:

```
    // Get the new customer
    System.out.println("*** GET Created Customer **");
    URL getUrl = new URL("http://localhost:9095/customers/1");
    connection = (HttpURLConnection) getUrl.openConnection();
    connection.setRequestMethod("GET");
    System.out.println("Content-Type: "
                        + connection.getContentType());

    BufferedReader reader = new BufferedReader(new
            InputStreamReader(connection.getInputStream()));

    String line = reader.readLine();
    while (line != null)
    {
        System.out.println(line);
        line = reader.readLine();
    }
    Assert.assertEquals(HttpURLConnection.HTTP_OK,
                        connection.getResponseCode());
    connection.disconnect();
```

Next, we query the server to obtain an XML representation of the customer we just
created. Again, we use the `HttpURLConnection` class to do the invocation:

```
    // Update the new customer.  Change Bill's name to William
    String updateCustomer = "<customer>"
            + "<first-name>William</first-name>"
            + "<last-name>Burke</last-name>"
            + "<street>256 Clarendon Street</street>"
            + "<city>Boston</city>"
            + "<state>MA</state>"
            + "<zip>02115</zip>"
            + "<country>USA</country>"
            + "</customer>";
    connection = (HttpURLConnection) getUrl.openConnection();
    connection.setDoOutput(true);
    connection.setRequestMethod("PUT");
    connection.setRequestProperty("Content-Type", "application/xml");
    os = connection.getOutputStream();
    os.write(updateCustomer.getBytes());
    os.flush();
    Assert.assertEquals(HttpURLConnection.HTTP_NO_CONTENT,
                        connection.gectResponseCode());
    connection.disconnect();
```

```
        // Show the update
        System.out.println("**** After Update ***");
        connection = (HttpURLConnection) getUrl.openConnection();
        connection.setRequestMethod("GET");

        System.out.println("Content-Type: "
                                + connection.getContentType());
        reader = new BufferedReader(new
                InputStreamReader(connection.getInputStream()));

        line = reader.readLine();
        while (line != null)
        {
            System.out.println(line);
            line = reader.readLine();
        }
        Assert.assertEquals(HttpURLConnection.HTTP_OK,
                                connection.getResponseCode());
        connection.disconnect();
    }
}
```

Finally, we test updating a customer by changing the first name of the customer from Bill, to William. We do this by creating a new XML document and doing a PUT to the URL of our customer resource.

That's it! The rest of the examples in this book have the same Maven structure as *ex03_1* and are tested using JUnit.

Examples for Chapter 4

Chapter 4 discussed three things. First, it mentions how the `@javax.ws.rs.HttpMe thod` annotation works and how to define and bind Java methods to new HTTP methods. Next, the chapter talks about the intricacies of the `@Path` annotation. It explains how you can use complex regular expressions to define your application's published URIs. Finally, the chapter goes over the concept of subresource locators.

This chapter walks you through three different example programs that you can build and run to illustrate the concepts in Chapter 4. The first example uses `@HttpMethod` to define a new HTTP method called PATCH. The second example expands on the customer service database example from Chapter 16 by adding some funky regular expression mappings with `@Path`. The third example implements the subresource locator example shown in "Full Dynamic Dispatching" on page 52 in Chapter 4.

Example ex04_1: HTTP Method Extension

This example shows you how your JAX-RS services can consume HTTP methods other than the common standard ones defined in HTTP 1.1. Specifically, the example implements the PATCH method. The PATCH method was originally mentioned in an earlier draft version of the HTTP 1.1 specification:[*]

> The PATCH method is similar to PUT except that the entity contains a list of differences between the original version of the resource identified by the Request-URI and the desired content of the resource after the PATCH action has been applied.

The idea of PATCH is that instead of transmitting the entire representation of a resource to update it, you only have to provide a partial representation in your update request. PUT requires that you transmit the entire representation, so the original plan was to include PATCH for scenarios where sending everything is not optimal.

---

[*] For more information, see *www.ietf.org/rfc/rfc2068.txt*.

Build and Run the Example Program

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex04_1* directory of the workbook example code.
2. Make sure your PATH is set up to include both the JDK and Maven, as described in Chapter 15.
3. Perform the build and run the example by typing `maven install`.

The Server Code

Using PATCH within JAX-RS is very simple. The source code under the *ex04_1* directory contains a simple annotation that implements PATCH:

src/main/java/org/ieft/annotations/PATCH.java

```java
package org.ieft.annotations;

import javax.ws.rs.HttpMethod;
import java.lang.annotation.*;

@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@HttpMethod("PATCH")
public @interface PATCH
{
}
```

As described in Chapter 4, all you need to do to use a custom HTTP method is annotate an annotation class with `@javax.ws.rs.HttpMethod`. This `@HttpMethod` declaration must contain the value of the new HTTP method you are defining.

To illustrate the use of our new `@PATCH` annotation, I expanded a little bit on the example code discussed in Chapter 16. A simple JAX-RS method is added to the `CustomerResource` class that can handle PATCH requests:

src/main/java/com/restfully.shop.services.CustomerResource.java

```java
package com.restfully.shop.services;

@Path("/customers")
public class CustomerResource {
...

    @PATCH
    @Path("{id}")
    @Consumes("application/xml")
    public void patchCustomer(@PathParam("id") int id, InputStream is)
    {
        updateCustomer(id, is);
    }
```

```
    ...
    }
```

The `@PATCH` annotation is used on the `patchCustomer()` method. The implementation of this method simply delegates to the original `updateCustomer()` method.

The Client Code

The client code for *ex04_1* is going to be a bit different than the client code in the previous chapter. As discussed in Chapter 13, one of the disadvantages of using `java.net.HttpURLConnection` is that you cannot use custom HTTP methods like PATCH. To get around this problem, the client is implemented using Apache HttpClient 4.x. Chapter 13 covered this client library in detail, so I will only cover how a PATCH request is made:

src/test/java/com/restfully/shop/test/PatchTest.java

```
package com.restfully.shop.test;

import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.client.params.HttpClientParams;
import org.apache.http.entity.StringEntity;
import org.apache.http.impl.client.DefaultHttpClient;
import org.junit.Assert;
import org.junit.Test;

import java.io.BufferedReader;
import java.io.InputStreamReader;


/**
 * @author <a href="mailto:bill@burkecentral.com">Bill Burke</a>
 * @version $Revision: 1 $
 */
public class PatchTest
{
    private static class HttpPatch extends HttpPost
    {
        public HttpPatch(String s)
        {
            super(s);
        }

        public String getMethod()
        {
            return "PATCH";
        }
    }
```

Other than to extend `org.apache.http.client.methods.HttpPost` and override the `getMethod()` method, there is no simple way to invoke our PATCH method in Apache

HttpClient 4.x. As you can see, the new class, `HttpPatch`, does just that. The `getMethod()` method simply returns the `PATCH` string. This tells the Apache `HttpClient` runtime which HTTP method the request object is invoking.

The rest of the class is pretty straightforward. I'll highlight only the interesting parts:

```
    @Test
    public void testCustomerResource() throws Exception {
...

        HttpPatch patch =
                new HttpPatch("http://localhost:9095/customers/1");

        // Update the new customer.  Change Bill's name to William
        String patchCustomer = "<customer>"
                + "<first-name>William</first-name>"
                + "</customer>";
        entity = new StringEntity(patchCustomer);
        entity.setContentType("application/xml");
        patch.setEntity(entity);
        response = client.execute(patch);
...
    }
}
```

As you can see, we allocate an instance of `HttpPatch` initialized with the URL of the customer we want to modify. We execute it as we would a POST operation. That's it!

Example ex04_2: @Path with Expressions

For this section, I wanted to show you `@Path` being used with regular expressions. The example is a direct copy of the code in *ex03_1* with a few minor modifications.

Build and Run the Example Program

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex04_2* directory of the workbook example code.
2. Make sure your PATH is set up to include both the JDK and Maven, as described in Chapter 15.
3. Perform the build and run the example by typing `maven install`.

The Server Code

The `CustomerResource` class copied from the *ex03_1* example is pretty much the same in *ex04_2*, except that a few of the `@Path` expressions have been modified. I also added an extra method that allows you to reference customers by their first and last names within the URL path:

```
@Path("/customers")
public class CustomerResource {
...

    @GET
    @Path("{id : \\d+}")
    @Produces("application/xml")
    public StreamingOutput getCustomer(@PathParam("id") int id)
    {
        ...
    }

    @PUT
    @Path("{id : \\d+}")
    @Consumes("application/xml")
    public void updateCustomer(@PathParam("id") int id, InputStream is)
    {
        ...
    }
```

The `@Path` expression for `getCustomer()` and `updateCustomer()` was changed a little bit
to use a Java regular expression for the URI matching. The expression dictates that the
`id` segment of the URI can only be a string of digits. So, `/customers/333` is a legal URI,
but `/customers/a32ab` would result in a 404, "Not Found" being returned to the client:

```
@GET
@Path("{first : [a-zA-Z]+}-{last:[a-zA-Z]+}")
@Produces("application/xml")
public StreamingOutput getCustomerFirstLast(
                            @PathParam("first") String first,
                            @PathParam("last") String last)
{
    ...
}
```

To show a more complex regular expression, I added the `getCustomerFirstLast()`
method to the resource class. This method provides a URI pointing to a specific cus-
tomer, using the customer's first and last names instead of a numeric ID. This `@Path`
expression matches a string of the first name and last name separated by a hyphen
character. A legal URI is `/customers/Bill-Burke`. The name can only have letters within
it, so `/customers/Bill7-Burke` would result in a 404, "Not Found" being returned to
the client.

The Client Code

The client code is in *src/test/java/com/restfully/shop/test/ClientResourceTest.java*. It is
really not much different than the code in example *ex03_1*, other than the fact that it
additionally invokes the URI represented by the `getCustomerFirstLast()` method. If
you've examined the code from Chapter 16, you can probably understand what is going
on in this client example, so I won't elaborate further.

Example ex04_3: Subresource Locators

The *ex04_3* example implements the subresource locator example shown in "Full Dynamic Dispatching" on page 52 in Chapter 4. The workbook code for this section really doesn't add much more, so I won't go into much detail in this section.

Build and Run the Example Program

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex04_3* directory of the workbook example code.
2. Make sure your PATH is set up to include both the JDK and Maven, as described in Chapter 15.
3. Perform the build and run the example by typing `maven install`.

The Server Code

The only real interesting thing to point out with the server-side code is how it is deployed. Because we are dealing with subresource locators, the `com.restfully.shop.serv ices.ShoppingApplication` class is a bit different than described in Chapter 3:

src/main/java/com/restfully/shop/services/ShoppingApplication.java

```java
public class ShoppingApplication extends Application {
    private Set<Object> singletons = new HashSet<Object>();
    private Set<Class<?>> empty = new HashSet<Class<?>>();

    public ShoppingApplication() {
        singletons.add(new CustomerDatabaseResource());
    }

    @Override
    public Set<Class<?>> getClasses() {
        return empty;
    }

    @Override
    public Set<Object> getSingletons() {
        return singletons;
    }
}
```

What's different from earlier implementations of `ShoppingApplication` is that you only need to register the top-level resource classes with the JAX-RS runtime through the `getClasses()` or `getSingletons()` method. Subresource classes like `CustomerResource` and `FirstLastCustomerResource` do not need to be registered. In fact, JAX-RS will give you a deployment-time error if you try to register these subresource classes.

The Client Code

The client code lives in *src/test/java/com/restfully/shop/test/CustomerResourceTest.java*:

```
public class CustomerResourceTest
{
    @Test
    public void testCustomerResource() throws Exception {
        ...
    }

    @Test
    public void testFirstLastCustomerResource() throws Exception {
        ...
    }
}
```

The code contains two methods: `testCustomerResource()` and `testFirstLastCustomer Resource()`.

The `testCustomerResource()` method first performs a POST to `/customers/europe-db` to create a customer using the `CustomerResource` subresource. It then retrieves the created customer using GET `/customers/europe-db/1`.

The `testFirstLastCustomerResource()` method performs a POST to `/customers/north america-db` to create a customer using the `FirstLastCustomerResource` subresource. It then uses GET `/customers/northamerica-db/Bill-Burke` to retrieve the created customer.

Examples for Chapter 5

Chapter 5 showed you how to use JAX-RS annotations to inject specific information about an HTTP request into your Java methods and fields. This chapter implements most of the injection scenarios introduced in Chapter 5 so that you can see these things in action.

Example ex05_1: Injecting URI Information

This example illustrates the injection annotations that are focused around pulling in information from the incoming request URI. Specifically, it shows how to use `@PathParam`, `@MatrixParam`, and `@QueryParam`. Parallel examples are also shown using `javax.ws.rs.core.UriInfo` to obtain the same data.

The Server Code

The first thing you should look at on the server side is `CarResource`. This class pulls the various examples in Chapter 5 together to illustrate using `@MatrixParam` and `@PathParam` with the `javax.ws.rs.core.PathSegment` class:

src/main/java/com/restfully/shop/services/CarResource.java

```java
@Path("/cars")
public class CarResource
{
   public static enum Color
   {
      red,
      white,
      blue,
      black
   }

   @GET
   @Path("/matrix/{make}/{model}/{year}")
   @Produces("text/plain")
```

```
    public String getFromMatrixParam(
                        @PathParam("make") String make,
                        @PathParam("model") PathSegment car,
                        @MatrixParam("color") Color color,
                        @PathParam("year") String year)
    {
        return "A " + color + " " + year + " "
                + make + " " + car.getPath();
    }
```

The getFromMatrixParam() method uses the @MatrixParam annotation to inject the matrix parameter color. An example of a URI it could process is /cars/matrix/mercedes/ e55;color=black/2006. Notice that it automatically converts the matrix parameter into the Java *enum* Color:

```
@GET
@Path("/segment/{make}/{model}/{year}")
@Produces("text/plain")
public String getFromPathSegment(@PathParam("make") String make,
                        @PathParam("model") PathSegment car,
                        @PathParam("year") String year)
{
    String carColor = car.getMatrixParameters().getFirst("color");
    return "A " + carColor + " " + year + " "
                        + make + " " + car.getPath();
}
```

The getFromPathSegment() method also illustrates how to extract matrix parameter information. Instead of using @MatrixParam, it uses an injected PathSegment instance representing the model path parameter to obtain the matrix parameter information:

```
@GET
@Path("/segments/{make}/{model : .+}/year/{year}")
@Produces("text/plain")
public String getFromMultipleSegments(
                        @PathParam("make") String make,
                        @PathParam("model") List<PathSegment> car,
                        @PathParam("year") String year)
{
    String output = "A " + year + " " + make;
    for (PathSegment segment : car)
    {
        output += " " + segment.getPath();
    }
    return output;
}
```

The getFromMultipleSegments() method illustrates how a path parameter can match multiple segments of a URI. An example of a URI that it could process is /cars/segments/mercedes/e55/amg/year/2006. In this case, e55/amg would match the model path parameter. The example injects the model parameter into a list of PathSegment instances:

```
@GET
@Path("/uriinfo/{make}/{model}/{year}")
@Produces("text/plain")
public String getFromUriInfo(@Context UriInfo info)
{
    String make = info.getPathParameters().getFirst("make");
    String year = info.getPathParameters().getFirst("year");
    PathSegment model = info.getPathSegments().get(3);
    String color = model.getMatrixParameters().getFirst("color");

    return "A " + color + " " + year + " "
                + make + " " + model.getPath();
}
```

The final method, `getFromUriInfo()`, shows how you can obtain the same information using the `UriInfo` interface. As you can see, the matrix parameter information is extracted from `PathSegment` instances.

The next piece of code you should look at on the server is `CustomerResource`. This class shows how `@QueryParam` and `@DefaultValue` can work together to obtain information about the request URI's query parameters. An example using `UriInfo` is also shown so that you can see how this can be done without injection annotations:

src/main/java/com/restfully/shop/services/CustomerResouce.java

```
@Path("/customers")
public class CustomerResource {
...

    @GET
    @Produces("application/xml")
    public StreamingOutput getCustomers(
                final @QueryParam("start") int start,
                final @QueryParam("size") @DefaultValue("2") int size)
    {
      ...
    }
```

The `getCustomers()` method returns a set of customers from the customer database. The `start` parameter defines the start index and the `size` parameter specifies how many customers you want returned. The `@DefaultValue` annotation is used for the case in which a client does not use the query parameters to index into the customer list.

The next implementation of `getCustomers()` uses `UriInfo` instead of injection parameters:

```
@GET
@Produces("application/xml")
@Path("uriinfo")
public StreamingOutput getCustomers(@Context UriInfo info)
{
    int start = 0;
    int size = 2;
    if (info.getQueryParameters().containsKey("start"))
    {
```

```
        start = Integer.valueOf(
                      info.getQueryParameters().getFirst("start"));
    }
    if (info.getQueryParameters().containsKey("size"))
    {
        size = Integer.valueOf(
                      info.getQueryParameters().getFirst("size"));
    }
    return getCustomers(start, size);
}
```

As you can see, the code to access query parameter data programmatically is a bit more verbose than using injection annotations.

The Client Code

The client code for this example lives in the file *src/test/java/com/restfully/shop/test/ InjectionTest.java*. The code is quite boring, so I won't get into too much detail.

The `testCarResource()` method invokes these requests on the server to test the `CarResource` class:

```
GET http://localhost:9095/cars/matrix/mercedes/e55;color=black/2006
GET http://localhost:9095/cars/segment/mercedes/e55;color=black/2006
GET http://localhost:9095/cars/segments/mercedes/e55/amg/year/2006
GET http://localhost:9095/cars/uriinfo/mercedes/e55;color=black/2006
```

The `testCustomerResource()` method invokes these requests on the server to test the `CustomerResource` class:

```
GET http://localhost:9095/customers
GET http://localhost:9095/customers?start=1&size=3
GET http://localhost:9095/customers/uriinfo?start=2&size=2
```

The request without query parameters shows `@DefaultValue` in action.

Build and Run the Example Program

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex05_1* directory of the workbook example code.
2. Make sure your PATH is set up to include both the JDK and Maven, as described in Chapter 15.
3. Perform the build and run the example by typing `maven install`.

Example ex05_2: Forms and Cookies

The *ex05_2* exercise shows examples of injecting form data, cookies, and HTTP headers using the `@FormParam`, `@CookieParam`, and `@HeaderParam` annotations. This example

is a bit different than former examples, as there is no client code. Instead, to see these annotations in action, you will use a browser as your client.

The Server Code

The example starts off with an HTML form defined in *src/main/webapp/index.html*:

```html
<html>
<body>

<form action="/rest/customers" method="post">
    First Name: <input type="text" name="firstname"/><br/>
    Last Name: <input type="text" name="lastname"/><br/>
    <INPUT type="submit" value="Send">
</form>

</body>
</html>
```

It is a simple form for creating a customer using our familiar `CustomerResource` service:

src/main/java/com/restfully/shop/CustomerResource.java

```java
@Path("/customers")
public class CustomerResource {
...
    @POST
    @Produces("text/html")
    public Response createCustomer(
                          @FormParam("firstname") String first,
                          @FormParam("lastname") String last)
    {
```

The HTML form posts data to the `createCustomer()` method of `CustomerResource` when users click the Send button:

```java
        Customer customer = new Customer();
        customer.setId(idCounter.incrementAndGet());
        customer.setFirstName(first);
        customer.setLastName(last);
        customerDB.put(customer.getId(), customer);
        System.out.println("Created customer " + customer.getId());
        String output = "Created customer <a href=\"customers/" +
                customer.getId() + "\">" + customer.getId()
                        + "</a>";
        String lastVisit = DateFormat.getDateTimeInstance(
                DateFormat.SHORT, DateFormat.LONG).format(new Date());
        return Response.created(URI.create("/customers/"
                                            + customer.getId()))
                    .entity(output)
                    .cookie(new NewCookie("last-visit", lastVisit))
                    .build();

    }
```

The `createCustomer()` method does a couple things. First, it uses the form data injected with `@FormParam` to create a `Customer` object and insert it into an in-memory map. It then builds an HTML response that shows text linking to the new customer. Finally, it sets a cookie on the client by calling the `ResponseBuilder.cookie()` method. This cookie, named `last-visit`, holds the current time and date. This cookie will be used so that on subsequent requests, the server knows the last time the client accessed the website:

```
@GET
@Path("{id}")
@Produces("text/plain")
public Response getCustomer(
                    @PathParam("id") int id,
                      @HeaderParam("User-Agent") String userAgent,
                        @CookieParam("last-visit") String date)
{
```

The `getCustomer()` method retrieves a `Customer` object from the in-memory map referenced by the `id` path parameter. The `@HeaderParam` annotation injects the value of the `User-Agent` header. This is a standard HTTP 1.1 header that denotes the type of client that made the request (Safari, Firefox, Internet Explorer, etc.). The `@CookieParam` annotation injects the value of the `last-visit` cookie that the client should be passing along with each request:

```
final Customer customer = customerDB.get(id);
if (customer == null) {
   throw new WebApplicationException(Response.Status.NOT_FOUND);
}
String output = "User-Agent: " + userAgent + "\r\n";
output += "Last visit: " + date + "\r\n\r\n";
output += "Customer: " + customer.getFirstName() + " "
                + customer.getLastName();
String lastVisit = DateFormat.getDateTimeInstance(
        DateFormat.SHORT, DateFormat.LONG).format(new Date());
return Response.ok(output)
        .cookie(new NewCookie("last-visit", lastVisit))
        .build();
}
```

The implementation of this method is very simple. It outputs the `User-Agent` header and `last-visit` cookie as plain text (`text/plain`). It also resets the `last-visit` cookie to the current time and date.

Server Configuration

You will need to change the *web.xml* file a bit to support this example. Previously, we mapped the RESTEasy servlet to service all incoming requests. Since we now have a static HTML file, *index.html*, we need to change the servlet mapping for JAX-RS invocations:

```
<web-app>
    <display-name>Archetype Created Web Application</display-name>
```

```
<context-param>
    <param-name>resteasy.servlet.mapping.prefix</param-name>
    <param-value>/rest</param-value>
</context-param>

<servlet>
    <servlet-name>Resteasy</servlet-name>
    <servlet-class>
  org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
    </servlet-class>
    <init-param>
        <param-name>
            javax.ws.rs.Application
        </param-name>
        <param-value>
            com.restfully.shop.services.ShoppingApplication
        </param-value>
    </init-param>
</servlet>

<servlet-mapping>
    <servlet-name>Resteasy</servlet-name>
    <url-pattern>/rest/*</url-pattern>
</servlet-mapping>

</web-app>
```

The `<url-pattern>` for the RESTEasy `<servlet-mapping>` is now prefixed with `/rest`. If we didn't make this change, a `GET /index.html` request would be routed to RESTEasy and the *index.html* file would never be returned to the client. We need to tell RESTEasy that it must use this prefix to process requests. This is done with the context parameter `resteasy.servlet.mapping.prefix`. This configuration may be different for other JAX-RS implementations.

Build and Run the Example Program

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex05_2* directory of the workbook example code.
2. Make sure your PATH is set up to include both the JDK and Maven, as described in Chapter 15.
3. Perform the build and run the example by typing `maven jetty:run`. This command is a bit different than our previous examples. This script builds the WAR file, but it also starts up the Jetty servlet container.

You test-drive *ex05_2* by using your browser. The first step is to go to *http://localhost: 9095*, as shown in Figure 18-1.

When you click Send, you will see the screen shown in Figure 18-2.

Figure 18-1. Customer creation form
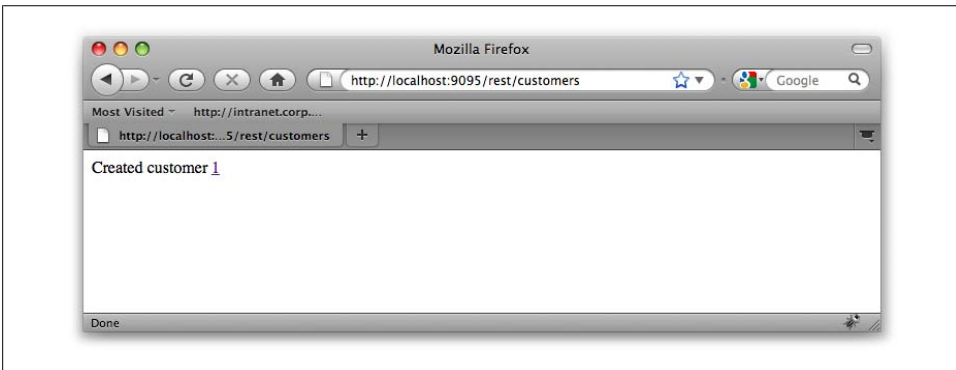
Figure 18-2. Creation response

Clicking the customer link will show you a plain-text representation of the customer, as shown in Figure 18-3.

If you refresh this page, you will see the timestamp of the "Last visit" string increment each time as the `CustomerResource` updates the `last-visit` cookie.
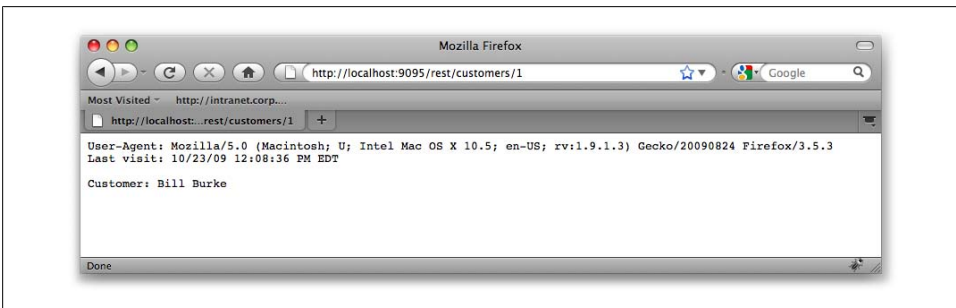
Figure 18-3. Customer output

Examples for Chapter 6

In Chapter 3, you saw a quick overview on how to write a simple JAX-RS service. You might have noticed that we needed a lot of code to process incoming and outgoing XML data. In Chapter 6, you learned that all this handcoded marshalling code is unnecessary. JAX-RS has a number of built-in content handlers that can do the processing for you. You also learned that if these prepackaged providers do not meet your requirements, you can write your own content handler.

There are two examples in this chapter. The first rewrites the *ex03_1* example to use JAXB instead of handcoded XML marshalling. The second example implements a custom content handler that can send serialized Java objects over HTTP.

Example ex06_1: Using JAXB

This example shows how easy it is to use JAXB and JAX-RS to exchange XML documents over HTTP. The `com.restfully.shop.domain.Customer` class is the first interesting piece of the example.

src/main/java/com/restfully/shop/domain/Customer.java

```
@XmlRootElement(name="customer")
public class Customer {
   private int id;
   private String firstName;
   private String lastName;
   private String street;
   private String city;
   private String state;
   private String zip;
   private String country;

   @XmlAttribute
   public int getId() {
      return id;
   }
```

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

        @XmlElement(name="first-name")
        public String getFirstName() {
            return firstName;
        }

        public void setFirstName(String firstName) {
            this.firstName = firstName;
        }

        @XmlElement(name="last-name")
        public String getLastName() {
            return lastName;
        }

        public void setLastName(String lastName) {
            this.lastName = lastName;
        }

        @XmlElement
        public String getStreet() {
            return street;
        }
    ...
    }
```

The JAXB annotations provide a mapping between the `Customer` class and XML.

You don't need to write a lot of code to implement the JAX-RS service because JAX-RS already knows how to handle JAXB annotated classes:

src/main/java/com/restfully/shop/services/CustomerResource.java

```
    @Path("/customers")
    public class CustomerResource {
        private Map<Integer, Customer> customerDB =
                            new ConcurrentHashMap<Integer, Customer>();
        private AtomicInteger idCounter = new AtomicInteger();

        public CustomerResource() {
        }

        @POST
        @Consumes("application/xml")
        public Response createCustomer(Customer customer) {
            customer.setId(idCounter.incrementAndGet());
            customerDB.put(customer.getId(), customer);
            System.out.println("Created customer " + customer.getId());
            return Response.created(URI.create("/customers/" +
                                        customer.getId())).build();

        }
```

```
    @GET
    @Path("{id}")
    @Produces("application/xml")
    public Customer getCustomer(@PathParam("id") int id) {
       Customer customer = customerDB.get(id);
       if (customer == null) {
         throw new WebApplicationException(Response.Status.NOT_FOUND);
       }
       return customer;
    }

    @PUT
    @Path("{id}")
    @Consumes("application/xml")
    public void updateCustomer(@PathParam("id") int id,
                                        Customer update) {
       Customer current = customerDB.get(id);
       if (current == null)
         throw new WebApplicationException(Response.Status.NOT_FOUND);

       current.setFirstName(update.getFirstName());
       current.setLastName(update.getLastName());
       current.setStreet(update.getStreet());
       current.setState(update.getState());
       current.setZip(update.getZip());
       current.setCountry(update.getCountry());
    }
}
```

If you compare this with the `CustomerResource` class in *ex03_1*, you'll see that the code in this example is much more compact. There is no handcoded marshalling code and our methods are dealing with `Customer` objects directly instead of raw strings.

The Client Code

The client code for this example is exactly the same as the code in *ex03_1*, so there's not much to explain here.

Changes to pom.xml

JBoss RESTEasy is broken up into a bunch of smaller jars so that you can pick and choose what features of RESTEasy to use. Because of this, the core RESTEasy JAR file does not have the JAXB content handlers. Therefore, we need to add a new dependency to our *pom.xml* file:

```
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-jaxb-provider</artifactId>
    <version>1.2</version>
</dependency>
```

Adding this dependency will add the JAXB provider to your project. It will also pull in any third-party dependency RESTEasy needs to process XML.

Build and Run the Example Program

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex06_1* directory of the workbook example code.
2. Make sure your PATH is set up to include both the JDK and Maven, as described in Chapter 15.
3. Perform the build and run the example by typing `maven install`.

Example ex06_2: Creating a Content Handler

For this example, we're going to create something entirely new. The Chapter 6 example of a content handler is a reimplementation of JAXB support. It is suitable for that chapter because it illustrates both the writing of a `MessageBodyReader` and `MessageBody` `Writer` and demonstrates how the `ContextResolver` is used. For *ex06_2*, though, we're going to keep things simple.

In *ex06_2*, we're going to rewrite *ex06_1* to exchange Java objects between the client and server instead of XML. Java objects, you ask? Isn't this REST? Well, there's no reason a Java object can't be a valid representation of a resource! If you're exchanging Java objects, you can still realize a lot of the advantages of REST and HTTP. You still can do content negotiation (described in Chapter 8) and HTTP caching (described in Chapter 10).

The Content Handler Code

For our Java object content handler, we're going to write one class that is both a `MessageBodyReader` and a `MessageBodyWriter`:

src/main/java/com/restfully/shop/services/JavaMarshaller.java

```
@Provider
@Produces("application/x-java-serialized-object")
@Consumes("application/x-java-serialized-object")
public class JavaMarshaller
                implements MessageBodyReader, MessageBodyWriter
{
```

The `JavaMarshaller` class is annotated with `@Provider`, `@Produces`, and `@Consumes`, as required by the specification. The media type used by the example to represent a Java object is `application/x-java-serialized-object`:[*]

```
public boolean isReadable(Class type, Type genericType, Annotation[] annotations,
MediaType mediaType)
{
    return Serializable.class.isAssignableFrom(type);
}

public boolean isWriteable(Class type, Type genericType,
                    Annotation[] annotations, MediaType mediaType)
{
    return Serializable.class.isAssignableFrom(type);
}
```

For the `isReadable()` and `isWriteable()` methods, we just check to see if our Java type implements the `java.io.Serializable` interface:

```
public Object readFrom(Class type, Type genericType,
                    Annotation[] annotations, MediaType mediaType,
                        MultivaluedMap httpHeaders,
                            InputStream is)
                    throws IOException, WebApplicationException
{
    ObjectInputStream ois = new ObjectInputStream(is);
    try
    {
        return ois.readObject();
    }
    catch (ClassNotFoundException e)
    {
        throw new RuntimeException(e);
    }
}
```

The `readFrom()` method uses basic Java serialization to read a Java object from the HTTP input stream:

```
public long getSize(Object o, Class type,
                    Type genericType, Annotation[] annotations,
                     MediaType mediaType)
{
    return -1;
}
```

The `getSize()` method returns −1. It is impossible to figure out the exact length of our marshalled Java object without serializing it into a byte buffer and counting the number of bytes. We're better off letting the servlet container figure this out for us:

---

[*] This obscure media type is actually defined in the JDK within the Javadoc of the class `java.awt.datatransfer.DataFlavor`.

```
      public void writeTo(Object o, Class type,
                          Type genericType, Annotation[] annotations,
                           MediaType mediaType,
                            MultivaluedMap httpHeaders, OutputStream os)
                             throws IOException, WebApplicationException
   {
      ObjectOutputStream oos = new ObjectOutputStream(os);
      oos.writeObject(o);
   }
```

Like the readFrom() method, basic Java serialization is used to marshal our Java object into the HTTP response body.

The Resource Class

The CustomerResource class doesn't change much from *ex06_2*:

src/main/java/com/restfully/shop/services/CustomerResource.java

```
   @Path("/customers")
   public class CustomerResource
   {
   ...

   @POST
      @Consumes("application/x-java-serialized-object")
      public Response createCustomer(Customer customer)
      {
         customer.setId(idCounter.incrementAndGet());
         customerDB.put(customer.getId(), customer);
         System.out.println("Created customer " + customer.getId());
         return Response.created(URI.create("/customers/" + customer.getId())).build();

      }
   ...
   }
```

The code is actually exactly the same as that used in *ex06_2*, except that the @Produces and @Consumes annotations use the application/x-java-serialized-object media type.

The Application Class

The ShoppingApplication class needs to change a tiny bit from the previous examples:

src/main/java/com/restfully/shop/services/ShoppingApplication.java

```
   public class ShoppingApplication extends Application {
      private Set<Object> singletons = new HashSet<Object>();
      private Set<Class<?>> classes = new HashSet<Class<?>>();

      public ShoppingApplication() {
         singletons.add(new CustomerResource());
         classes.add(JavaMarshaller.class);
```

```
        }

        @Override
        public Set<Class<?>> getClasses() {
            return classes;
        }

        @Override
        public Set<Object> getSingletons() {
            return singletons;
        }
    }
```

For our `Application` class, we need to register the `JavaMarshaller` class. If we don't, the JAX-RS runtime won't know how to handle the `application/x-java-serialized-object` media type.

The Client Code

The client code needs to change a bit, since we're sending serialized Java objects across the wire instead of XML:

src/test/java/com/restfully/shop/test/CustomerResourceTest.java

```
    public class CustomerResourceTest
    {
        @Test
        public void testCustomerResource() throws Exception
        {
            System.out.println("*** Create a new Customer ***");
            Customer cust = new Customer();
            cust.setFirstName("Bill");
            cust.setLastName("Burke");
            cust.setStreet("256 Clarendon Street");
            cust.setCity("Boston");
            cust.setState("MA");
            cust.setZip("02115");
            cust.setCountry("USA");
```

For our client, we need to instantiate and initialize a `Customer` object instead of creating an XML string:

```
            URL postUrl = new URL("http://localhost:9095/customers");
            HttpURLConnection connection =
                            (HttpURLConnection) postUrl.openConnection();
            connection.setDoOutput(true);
            connection.setInstanceFollowRedirects(false);
            connection.setRequestMethod("POST");
            connection.setRequestProperty("Content-Type",
                                "application/x-java-serialized-object");
            OutputStream os = connection.getOutputStream();
            ObjectOutputStream oos = new ObjectOutputStream(os);
            oos.writeObject(cust);
            oos.flush();
```

We then use Java serialization to write the object to the HTTP output stream. The rest of the client works pretty much the same.

Build and Run the Example Program

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex06_2* directory of the workbook example code.
2. Make sure your PATH is set up to include both the JDK and Maven, as described in Chapter 15.
3. Perform the build and run the example by typing `maven install`.

Examples for Chapter 7

In Chapter 7, you learned how to create complex responses using the `Response` and `ResponseBuilder` classes. You also learned how to map thrown exceptions to a `Response` using a `javax.ws.rs.ext.ExceptionMapper`. Since most of our examples use a `Response` object in one way or another, this chapter focuses only on writing an `ExceptionMapper`.

Example ex07_1: ExceptionMapper

This example is a slight modification from *ex06_1* to show you how you can use `ExceptionMappers`. Let's take a look at the `CustomerResource` class to see what is different:

src/main/java/com/restfully/shop/services/CustomerResource.java

```
@Path("/customers")
public class CustomerResource {
...
    @GET
    @Path("{id}")
    @Produces("application/xml")
    public Customer getCustomer(@PathParam("id") int id)
    {
        Customer customer = customerDB.get(id);
        if (customer == null)
        {
            throw new NotFoundException("Could not find customer "
                                        + id);
        }
        return customer;
    }

    @PUT
    @Path("{id}")
    @Consumes("application/xml")
    public void updateCustomer(@PathParam("id") int id,
                               Customer update)
    {
```

```
        Customer current = customerDB.get(id);
        if (current == null)
          throw new NotFoundException("Could not find customer " + id);

        current.setFirstName(update.getFirstName());
        current.setLastName(update.getLastName());
        current.setStreet(update.getStreet());
        current.setState(update.getState());
        current.setZip(update.getZip());
        current.setCountry(update.getCountry());
      }
   }
```

In *ex06_1*, our `getCustomer()` and `updateCustomer()` methods threw a `javax.ws.rs.WebApplicationException`. We've replaced this exception with our own custom class, `NotFoundException`:

src/main/java/com/restfully/shop/services/NotFoundException.java

```
public class NotFoundException extends RuntimeException
{
   public NotFoundException(String s)
   {
      super(s);
   }
}
```

There's nothing really special about this exception class other than it inherits from `java.lang.RuntimeException`. What we are going to do, though, is map this thrown exception to a `Response` object using an `ExceptionMapper`:

src/main/java/com/restfully/shop/services/NotFoundExceptionMapper.java

```
@Provider
public class NotFoundExceptionMapper
                     implements ExceptionMapper<NotFoundException>
{
   public Response toResponse(NotFoundException exception)
   {
      return Response.status(Response.Status.NOT_FOUND)
                     .entity(exception.getMessage())
                     .type("text/plain").build();
   }
}
```

When a client makes a GET request to a customer URL that does not exist, the `CustomerResource.getCustomer()` method throws a `NotFoundException`. This exception is caught by the JAX-RS runtime, and the `NotFoundExceptionMapper.toResponse()` method is called. This method creates a `Response` object that returns a 404 status code and a plain-text error message.

The last thing we have to do is modify our `Application` class to register the
`ExceptionMapper`:

src/main/java/com/restfully/shop/services/NotFoundExceptionMapper.java

```
public class ShoppingApplication extends Application {
   private Set<Object> singletons = new HashSet<Object>();
   private Set<Class<?>> classes = new HashSet<Class<?>>();

   public ShoppingApplication()
   {
      singletons.add(new CustomerResource());
      classes.add(NotFoundExceptionMapper.class);
   }

   @Override
   public Set<Class<?>> getClasses()
   {
      return classes;
   }

   @Override
   public Set<Object> getSingletons()
   {
      return singletons;
   }
}
```

The Client Code

The client code for this example is very simple. We make a GET request to a customer
resource that doesn't exist:

src/test/java/com/restfully/shop/test/CustomerResourceTest.java

```
public class CustomerResourceTest
{
   @Test
   public void testCustomerResource() throws Exception
   {
      // Show the update
      System.out.println("**** Get Unknown Customer ***");
      URL getUrl = new URL("http://localhost:9095/customers/1");
      HttpURLConnection connection =
                  (HttpURLConnection) getUrl.openConnection();
      connection.setRequestMethod("GET");
      try
      {
         int code = connection.getResponseCode();
      }
      catch (FileNotFoundException e)
      {
         System.out.println("Customer not found.");
      }
      connection.disconnect();
```

```
        }
    }
```

One thing to notice about this client code is that if the server returns a 404 response code, `HttpURLConnection` will throw a `java.io.FileNotFoundException`.

Build and Run the Example Program

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex07_1* directory of the workbook example code.
2. Make sure your PATH is set up to include both the JDK and Maven, as described in Chapter 15.
3. Perform the build and run the example by typing `maven install`.

Examples for Chapter 8

In Chapter 8, you learned that clients can use HTTP Content Negotiation to request different data formats from the same URL using the `Accept` header. You also learned that JAX-RS takes the `Accept` header into account when deciding how to dispatch an HTTP request to a Java method. In this chapter, you'll see two different examples that show how JAX-RS and HTTP conneg can work together.

Example ex08_1: Conneg with JAX-RS

This example is a slight modification from *ex06_1* and shows two different concepts. The first is that the same JAX-RS resource method can process two different media types. Chapter 8 gives the example of a method that returns a JAXB annotated class instance that can be returned as either JSON or XML. We've implemented this in *ex08_1* by slightly changing the `CustomerResource.getCustomer()` method:

src/main/java/com/restfully/shop/services/CustomerResource.java

```
@Path("/customers")
public class CustomerResource {
...
    @GET
    @Path("{id}")
    @Produces({"application/xml", "application/json"})
    public Customer getCustomer(@PathParam("id") int id)
    {
        ...
    }
```

The JAXB provider that comes with RESTEasy can convert JAXB objects to JSON or XML. In this example, we have added the media type `application/json` to `getCustomer()`'s `@Produces` annotation. The JAX-RS runtime will process the `Accept` header and pick the appropriate media type of the response for `getCustomer()`. If the `Accept` header is `application/xml`, XML will be produced. If the `Accept` header is JSON, the Customer object will be outputted as JSON.

Another thing that was mentioned in Chapter 8 is that you can use the `@Produces` annotation to dispatch to different Java methods. To illustrate this, we've added the `getCustomerString()` method that processes the same URL as `getCustomer()` but for a different media type:

```
@GET
@Path("{id}")
@Produces("text/plain")
public Customer getCustomerString(@PathParam("id") int id)
{
    return getCustomer(id).toString();
}
```

The Client Code

The client code for this example executes various HTTP GET requests to retrieve different representations of a `Customer`. Each request sets the `Accept` header a little differently so that it can obtain a different representation. For example:

src/test/java/com/restfully/shop/test/CustomerResourceTest.java

```
public class CustomerResourceTest
{
    @Test
    public void testCustomerResource() throws Exception
    {
        ... initialization code ...

        // Get XML customer
        System.out.println("*** GET Customer as XML **");
        URL getUrl = new URL("http://localhost:9095/customers/1");
        connection = (HttpURLConnection) getUrl.openConnection();
        connection.setRequestMethod("GET");
        connection.setRequestProperty("Accept", "application/xml");
        Assert.assertEquals(HttpURLConnection.HTTP_OK,
                            connection.getResponseCode());
        Assert.assertEquals("application/xml",
                            connection.getContentType());

        BufferedReader reader = new BufferedReader(new
                InputStreamReader(connection.getInputStream()));

        String line = reader.readLine();
        while (line != null)
        {
            System.out.println(line);
            line = reader.readLine();
        }
        connection.disconnect();

        ... the reset of the code ...
    }
}
```

This code fragment is completed again for `application/json` and `text/plain`.

Build and Run the Example Program

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex08_1* directory of the workbook example code.
2. Make sure your PATH is set up to include both the JDK and Maven, as described in Chapter 15.
3. Perform the build and run the example by typing `maven install`.

Example ex08_2: Conneg via URL Patterns

Chapter 8 discussed how some clients, particularly browsers, are unable to use the `Accept` header to request different formats from the same URL. To solve this problem, many JAX-RS implementations allow you to map a filename suffix (*.html*, *.xml*, *.txt*) to a particular media type. JBoss RESTEasy has this capability. We're going to illustrate this using your browser as a client along with a slightly modified version of *ex08_1*.

The Server Code

A few minor things have changed on the server side. First, we add an additional `getCustomerHtml()` method to our `CustomerResource` class:

```
@GET
@Path("{id}")
@Produces("text/html")
public String getCustomerHtml(@PathParam("id") int id)
{
    return "<h1>Customer As HTML</h1><pre>"
                + getCustomer(id).toString() + "</pre>";
}
```

Since you're going to be interacting with this service through your browser, it might be nice if the example outputs HTML in addition to text, XML, and JSON.

The only other change to the server side is configuration for RESTEasy:

src/main/webapp/WEB-INF/web.xml

```
<web-app>

    <context-param>
        <param-name>resteasy.media.type.mappings</param-name>
        <param-value>
                html : text/html,
                txt : text/plain,
                xml : application/xml
        </param-value>
```

```
        </context-param>
    ...
    </web-app>
```

The `resteasy.media.type.mappings` content parameter is added to define a mapping between various file suffixes and the media types they map to. A comma separates each entry. The suffix string makes up the first half of the entry and the colon character delimits the suffix from the media type it maps to. Here, we've defined mappings between `.html` and `text/html`, `.txt` and `text/plain`, and `.xml` and `application/xml`.

Build and Run the Example Program

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex08_2* directory of the workbook example code.
2. Make sure your PATH is set up to include both the JDK and Maven, as described in Chapter 15.
3. Perform the build and run the example by typing `maven jetty:run`.

The `jetty:run` target will run the servlet container so that you can make browser invocations on it. Now, open up your browser and visit this URL:

```
http://localhost:9095/customers/1
```

Visiting this URL will show you which default media type your browser requests. Each browser may be different. For me, Firefox 3.x prefers HTML and Safari prefers XML.

Next, browse each of the following URLs:

```
http://localhost:9095/customers/1.html
http://localhost:9095/customers/1.txt
http://localhost:9095/customers/1.xml
```

You should see a different representation for each of these URLs.

Examples for Chapter 9

In Chapter 9, you learned about many of the concepts of HATEOAS and how to use JAX-RS to add these principles to your RESTful web services. In this chapter, you'll look through two different examples. The first shows you how to introduce Atom links into your XML documents. The second uses `Link` headers to publish state transitions within a RESTful web service application.

Example ex09_1: Atom Links

This example is a slight modification of the *ex06_1* example introduced in Chapter 19. This example expands the `CustomerResource` RESTful web service so that a client can fetch subsets of the customer database. If a client does a `GET /customers` request in our RESTful application, it will receive a subset list of customers in XML. Two Atom links are embedded in this document that allow you to view the next or previous sets of customer data. Example output would be:

```
<customers>
   <customer id="3">
      ...
   </customer>
   <customer id="4">
      ...
   </customer>
   <link rel="next"
         href="http://example.com/customers?start=5&size=2"
         type="application/xml"/>
   <link rel="previous"
         href="http://example.com/customers?start=1&size=2"
         type="application/xml"/>
</customers>
```

The `next` and `previous` links are URLs pointing to the same `/customers` URL, but they contain URI query parameters indexing into the customer database.

The Server Code

The first thing you want to look at is the extensions made to the JAXB model. A new class `Link` is added so that you can define Atom links and embed them within your JAXB objects:

src/main/java/com/restfully/shop/domain/Link.java

```java
@XmlRootElement(name = "link")
public class Link
{
   protected String relationship;
   protected String href;
   protected String type;

   public Link()
   {
   }

   public Link(String relationship, String href, String type)
   {
      this.relationship = relationship;
      this.href = href;
      this.type = type;
   }

   @XmlAttribute(name = "rel")
   public String getRelationship()
   {
      return relationship;
   }

   public void setRelationship(String relationship)
   {
      this.relationship = relationship;
   }

   @XmlAttribute
   public String getHref()
   {
      return href;
   }

   public void setHref(String href)
   {
      this.href = href;
   }

   @XmlAttribute
   public String getType()
   {
      return type;
   }

   public void setType(String type)
```

```
        {
            this.type = type;
        }
    }
```

As you can see, `Link` is a very simple class annotated with JAXB annotations.

The next thing is to define a JAXB class that maps to the `<customers>` element. It must be capable of holding an arbitrary number of `Customer` instances as well as the Atom links for our `next` and `previous` link relationships:

src/main/java/com/restfully/shop/domain/Customers.java

```java
@XmlRootElement(name = "customers")
public class Customers
{
    protected Collection<Customer> customers;
    protected List<Link> links;

    @XmlElementRef
    public Collection<Customer> getCustomers()
    {
        return customers;
    }

    public void setCustomers(Collection<Customer> customers)
    {
        this.customers = customers;
    }

    @XmlElementRef
    public List<Link> getLinks()
    {
        return links;
    }

    public void setLinks(List<Link> links)
    {
        this.links = links;
    }

    @XmlTransient
    public String getNext()
    {
        if (links == null) return null;
        for (Link link : links)
        {
            if ("next".equals(link.getRelationship()))
                return link.getHref();
        }
        return null;
    }

    @XmlTransient
    public String getPrevious()
    {
```

```
         if (links == null) return null;
         for (Link link : links)
         {
            if ("previous".equals(link.getRelationship()))
                   return link.getHref();
         }
         return null;
      }

   }
```

There is no nice way to define a map in JAXB, so all the Atom links are stuffed within a collection property of `Customers`. The convenience methods `getPrevious()`and `get Next()` iterate through this collection to find the `next` and `previous` Atom links embedded within the document if they exist.

The final difference from the *ex06_1* example is the implementation of `GET /customers` handling:

src/main/java/com/restfully/shop/services/CustomerResource.java

```
   @Path("/customers")
   public class CustomerResource
   {
      @GET
      @Produces("application/xml")
      @Formatted
      public Customers getCustomers(@QueryParam("start") int start,
                    @QueryParam("size") @DefaultValue("2") int size,
                    @Context UriInfo uriInfo)
      {
```

The `@org.jboss.resteasy.annotations.providers.jaxb.Formatted` annotation is a RESTEasy-specific plug-in that formats the XML output returned to the client to include indentations and new lines so that the text is easier to read.

The query parameters for the `getCustomers()` method, `start` and `size`, are optional. They represent an index into the customer database and how many customers you want returned by the invocation. The `@DefaultValue` annotation is used to define a default page size of 2.

The `UriInfo` instance injected with `@Context` is used to build the URLs that define `next` and `previous` link relationships:

```
         UriBuilder builder = uriInfo.getAbsolutePathBuilder();
         builder.queryParam("start", "{start}");
         builder.queryParam("size", "{size}");
```

Here, the code defines a URI template by using the `UriBuilder` passed back from `UriInfo.getAbsolutePathBuilder()`. The `start` and `size` query parameters are added to the template. Their values are populated using template parameters later on when the actual links are built:

```
ArrayList<Customer> list = new ArrayList<Customer>();
ArrayList<Link> links = new ArrayList<Link>();
synchronized (customerDB)
{
   int i = 0;
   for (Customer customer : customerDB.values())
   {
      if (i >= start && i < start + size)
                             list.add(customer);
      i++;
   }
}
```

The code then gathers up the `Customer` instances that will be returned to the client based on the `start` and `size` parameters. All this code is done within a `synchronized` block to protect against concurrent access on the `customerDB` map:

```
// next link
if (start + size < customerDB.size())
{
   int next = start + size;
   URI nextUri = builder.clone().build(next, size);
   Link nextLink = new Link("next",
              nextUri.toString(), "application/xml");
   links.add(nextLink);
}
// previous link
if (start > 0)
{
   int previous = start - size;
   if (previous < 0) previous = 0;
   URI previousUri = builder.clone().build(previous, size);
   Link previousLink = new Link("previous",
              previousUri.toString(), "application/xml");
   links.add(previousLink);
}
```

If there are more possible customer instances left to be viewed, a `next` link relationship is calculated using the `UriBuilder` template defined earlier. A similar calculation is done to see if a `previous` link relationship needs to be added to the document:

```
}
Customers customers = new Customers();
customers.setCustomers(list);
customers.setLinks(links);
return customers;
}
```

Finally, a `Customers` instance is created and initialized with the `Customer` instances to be returned to the client and any link relationships.

The Client Code

To have compact, simple client code, the RESTEasy client framework is used to implement this example. It initially gets the XML document from the **/customers** URL. It then loops using the **next** link relationship as the URL to print out all the customers in the database:

```
public class CustomerResourceTest
{
   @Test
   public void testQueryCustomers() throws Exception
   {
      RegisterBuiltin.register(ResteasyProviderFactory.getInstance());
      String url = "http://localhost:9095/customers";
      while (url != null)
      {
         ClientRequest request = new ClientRequest(url);
         String output = request.getTarget(String.class);
         System.out.println("** XML from " + url);
         System.out.println(output);

         Customers customers = request.getTarget(Customers.class);
         url = customers.getNext();
      }
   }
}
```

An interesting thing to note about this is that the server is guiding the client to make state transitions as it browses the customer database. Once the initial URL is invoked, further queries are solely driven by Atom links.

Build and Run the Example Program

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex09_1* directory of the workbook example code.
2. Make sure your PATH is set up to include both the JDK and Maven, as described in Chapter 15.
3. Perform the build and run the example by typing `maven install`.

Example ex09_2: Link Headers

There are two educational goals I want to get across with this example. The first is the use of `Link` headers within a RESTful application. The second is that if your services provide the appropriate links, you only need one published URL to navigate through your system. When you look at the client code for this example, you'll see that only one URL is hardcoded to start the whole process of the example.

To illustrate these techniques, a few more additional JAX-RS services were built beyond the simple customer database example that has been repeated so many times throughout this book. Chapter 2 discussed the design of an e-commerce application. This chapter starts the process of implementing this application by introducing an order-entry RESTful service.

The Server Code

To make it easier to publish `Link` headers, the `Link` class from *ex09_1* is expanded to add a `toString()` and `valueOf()` method:

src/main/java/com/restfully/shop/domain/Link.java

```java
@XmlRootElement(name = "link")
public class Link
{
...
...
   /**
    * To write as link header
    *
    * @return
    */
   public String toString()
   {
      StringBuilder builder = new StringBuilder("<");
      builder.append(href).append(">; rel=").append(relationship);
      if (type != null) builder.append("; type=").append(type);
      return builder.toString();
   }

   private static Pattern parse = Pattern.compile(
                                    "<(.+)>\\s*;\\s*(.+)");

   /**
    * For unmarshalling Link Headers.
    * Its not an efficient or perfect algorithm
    *  and does make a few assumptiosn
    *
    * @param val
    * @return
    */
   public static Link valueOf(String val)
   {
      Matcher matcher = parse.matcher(val);
      if (!matcher.matches())
        throw new RuntimeException("Failed to parse link: " + val);
      Link link = new Link();
      link.href = matcher.group(1);
      String[] props = matcher.group(2).split(";");
      HashMap<String, String> map = new HashMap();
      for (String prop : props)
      {
         String[] split = prop.split("=");
```

```
            map.put(split[0].trim(), split[1].trim());
         }
         if (map.containsKey("rel"))
         {
            link.relationship = map.get("rel");
         }
         if (map.containsKey("type"))
         {
            link.type = map.get("type");
         }
         return link;
      }
   }
```

In Chapter 7, you learned that you can piggyback your own custom response headers using the `Response` and `ResponseBuilder` classes. The `ResponseBuilder.header()` method can take any object as the value of the header. This object's `toString()` method is used to create the value of the header.

On the flipside, any object can be injected using the `@HeaderParam` annotation if it implements a static `valueOf(String)` method. The `Link.valueOf()` method uses Java regular expressions to parse a string-based `Link` header value into a `Link` object.

The `Order` and `LineItem` classes are added to the JAXB domain model. They are used to marshal the XML that represents order entries in the system. They are not that interesting, so I'm not going to get into much detail here.

OrderResource

The `OrderResource` class is used to create, post, and cancel orders in our e-commerce system. The purge operation is also available to destroy any leftover order entries that have been cancelled but not removed from the order entry database. Let's look:

src/main/java/com/restfully/shop/services/OrderResource.java

```
@Path("/orders")
public class OrderResource
{
   private Map<Integer, Order> orderDB =
                            new Hashtable<Integer, Order>();
   private AtomicInteger idCounter = new AtomicInteger();

   @POST
   @Consumes("application/xml")
   public Response createOrder(Order order, @Context UriInfo uriInfo)
   {
      order.setId(idCounter.incrementAndGet());
      orderDB.put(order.getId(), order);
      System.out.println("Created order " + order.getId());
      UriBuilder builder = uriInfo.getAbsolutePathBuilder();
      builder.path(Integer.toString(order.getId()));
      return Response.created(builder.build()).build();

   }
```

The `createOrder()` method handles POST `/orders` requests. It generates new `Order` IDs and adds the posted `Order` instance into the order database (the map). The `UriInfo.getAbsolutePathBuilder()` method generates the URL passed back with the `Location` header returned by the `Response.created()` method. You'll see later that the client uses this URL to further manipulate the created order:

```java
@GET
@Path("{id}")
@Produces("application/xml")
public Response getOrder(@PathParam("id") int id,
                         @Context UriInfo uriInfo)
{
   Order order = orderDB.get(id);
   if (order == null)
   {
     throw new WebApplicationException(Response.Status.NOT_FOUND);
   }
   Response.ResponseBuilder builder = Response.ok(order);
   if (!order.isCancelled()) addCancelHeader(uriInfo, builder);
   return builder.build();
}
```

The `getOrder()` method processes GET `/orders/{id}` requests and retrieves individual orders from the database (the map). If the order has not been cancelled already, a `cancel` `Link` header is added to the `Response` so the client knows if an order can be cancelled and which URL to post a cancel request to:

```java
protected void addCancelHeader(UriInfo uriInfo,
                               Response.ResponseBuilder builder)
{
   UriBuilder absolute = uriInfo.getAbsolutePathBuilder();
   String cancelUrl = absolute.clone()
                        .path("cancel").build().toString();
   builder.header("Link", new Link("cancel", cancelUrl, null));
}
```

The `addCancelHeader()` method creates a `Link` object for the `cancel` relationship using a URL generated from `UriInfo.getAbsolutePathBuilder()`:

```java
@HEAD
@Path("{id}")
@Produces("application/xml")
public Response getOrderHeaders(@PathParam("id") int id,
                               @Context UriInfo uriInfo)
{
   Order order = orderDB.get(id);
   if (order == null)
   {
     throw new WebApplicationException(Response.Status.NOT_FOUND);
   }
   Response.ResponseBuilder builder = Response.ok();
   builder.type("application/xml");
   if (!order.isCancelled()) addCancelHeader(uriInfo, builder);
   return builder.build();
}
```

The getOrderHeaders() method processes HTTP HEAD /orders/{id} requests. This is a convenience operation for HTTP clients that want the link relationships published by the resource but don't want to have to parse an XML document to get this information. Here, the getOrderHeaders() method returns the cancel Link header with an empty response body:

```
@POST
@Path("{id}/cancel")
public void cancelOrder(@PathParam("id") int id)
{
   Order order = orderDB.get(id);
   if (order == null)
   {
      throw new WebApplicationException(Response.Status.NOT_FOUND);
   }
   order.setCancelled(true);
}
```

Users can cancel an order by posting an empty message to /orders/{id}/cancel. The cancelOrder() handles these requests and simply looks up the Order in the database and sets its state to cancelled:

```
@GET
@Produces("application/xml")
@Formatted
public Response getOrders(@QueryParam("start") int start,
                 @QueryParam("size") @DefaultValue("2") int size,
                 @Context UriInfo uriInfo)
{
...
   Orders orders = new Orders();
   orders.setOrders(list);
   orders.setLinks(links);
   Response.ResponseBuilder responseBuilder = Response.ok(orders);
   addPurgeLinkHeader(uriInfo, responseBuilder);
   return responseBuilder.build();
}
```

The getOrders() method is similar to the CustomerResource.getCustomers() method discussed in the *ex09_1* example, so I won't go into a lot of details. One thing it does differently, though, is to publish a purge link relationship through a Link header. Posting to this link allows clients to purge the order entry database of any lingering cancelled orders:

```
protected void addPurgeLinkHeader(UriInfo uriInfo,
                            Response.ResponseBuilder builder)
{
   UriBuilder absolute = uriInfo.getAbsolutePathBuilder();
   String purgeUrl = absolute.clone().path("purge")
                              .build().toString();
   builder.header("Link", new Link("purge", purgeUrl, null));
}
```

The `addPurgeLinkHeader()` method creates a `Link` object for the `purge` relationship using a URL generated from `UriInfo.getAbsolutePathBuilder()`:

```
@HEAD
@Produces("application/xml")
public Response getOrdersHeaders(@QueryParam("start") int start,
                    @QueryParam("size") @DefaultValue("2") int size,
                    @Context UriInfo uriInfo)
{
    Response.ResponseBuilder builder = Response.ok();
    builder.type("application/xml");
    addPurgeLinkHeader(uriInfo, builder);
    return builder.build();
}
```

The `getOrdersHeaders()` method is another convenience method for clients that are only interested in the link relationships provided by the resource:

```
@POST
@Path("purge")
public void purgeOrders()
{
    synchronized (orderDB)
    {
        List<Order> orders = new ArrayList<Order>();
        orders.addAll(orderDB.values());
        for (Order order : orders)
        {
            if (order.isCancelled())
            {
                orderDB.remove(order.getId());
            }
        }
    }
}
```

Finally, the `purgeOrders()` method implements the purging of cancelled orders.

StoreResource

One of the things I want to illustrate with this example is that a client only needs to be aware of one URL to navigate through the entire system. The `StoreResource` class is the base URL of the system and publishes `Link` headers to the relevant services of the application:

src/main/java/com/restfully/shop/services/StoreResource.java

```
@Path("/shop")
public class StoreResource
{
    @HEAD
    public Response head(@Context UriInfo uriInfo)
    {
```

```
        UriBuilder absolute = uriInfo.getBaseUriBuilder();
        String customerUrl = absolute.clone().path("customers")
                                             .build().toString();
        String orderUrl = absolute.clone().path("orders")
                                          .build().toString();

        Response.ResponseBuilder builder = Response.ok();
        builder.header("Link",
                new Link("customers", customerUrl, "application/xml"));
        builder.header("Link",
                    new Link("orders", orderUrl, "application/xml"));
        return builder.build();
    }
}
```

This class accepts HTTP HEAD /shop requests and publishes the customers and
orders link relationships. These links point to the services represented by the
CustomerResource and OrderResource classes.

The Client Code

The client code creates a new customer and order. It then cancels the order, purges it,
and, finally, relists the order entry database. All URLs are accessed by following Link
headers or Atom links. The RESTEasy client framework implements the example client:

```
public class OrderResourceTest
{
    protected Map<String, Link> processLinkHeaders(
                                        ClientResponse response)
    {
        List<String> linkHeaders = (List<String>) response.getHeaders()
                                                    .get("Link");
        Map<String, Link> links = new HashMap<String, Link>();
        for (String header : linkHeaders)
        {
            Link link = Link.valueOf(header);
            links.put(link.getRelationship(), link);
        }
        return links;
    }
```

The processLinkHeaders() method is a utility method for extracting a map of Link
objects from the Link headers embedded within a ClientResponse:

```
    @Test
    public void testCreateCancelPurge() throws Exception
    {
        RegisterBuiltin.register(ResteasyProviderFactory.getInstance());
        String url = "http://localhost:9095/shop";
        ClientRequest request = new ClientRequest(url);
        ClientResponse response = request.head();
        Map<String, Link> shoppingLinks = processLinkHeaders(response);
```

The testCreateCancelPurge() method starts off by doing a HEAD request to /shop to obtain a list of service URLs provided by our application. The shoppingLinks is a map of the link relationships returned by this request:

```
Link customers = shoppingLinks.get("customers");
System.out.println("** Create a customer through this URL: "
                    + customers.getHref());

Customer customer = new Customer();
customer.setFirstName("Bill");
customer.setLastName("Burke");
customer.setStreet("10 Somewhere Street");
customer.setCity("Westford");
customer.setState("MA");
customer.setZip("01711");
customer.setCountry("USA");

request = new ClientRequest(customers.getHref());
request.body("application/xml", customer);
response = request.post();
Assert.assertEquals(201, response.getStatus());
```

A customer is created in the customer database by POSTing an XML representation to the URL referenced in the customers link relationship. This relationship is retrieved from our initial HEAD request to /shop:

```
Link orders = shoppingLinks.get("orders");

Order order = new Order();
order.setTotal("$199.99");
order.setCustomer(customer);
order.setDate(new Date().toString());
LineItem item = new LineItem();
item.setCost("$199.99");
item.setProduct("iPhone");
order.setLineItems(new ArrayList<LineItem>());
order.getLineItems().add(item);

System.out.println();
System.out.println("** Create an order through this URL: "
                    + orders.getHref());
request = new ClientRequest(orders.getHref());
request.body("application/xml", order);
response = request.post();
Assert.assertEquals(201, response.getStatus());
String createdOrderUrl = (String) response.getHeaders()
                            .getFirst("Location");
```

Next, an order entry is created by posting to the orders link relationship. The URL of the created order is extracted from the returned Location header. We will need this later when we want to cancel this order:

```
System.out.println();
System.out.println("** New list of orders");
request = new ClientRequest(orders.getHref());
response = request.get();
System.out.println(response.getEntity(String.class));
Map<String, Link> ordersLinks = processLinkHeaders(response);
```

A `GET /orders` request is initiated to show all the orders posted to the system. The `Link` headers returned by this invocation are processed so that they can be used later when the client wants to purge cancelled orders:

```
request = new ClientRequest(createdOrderUrl);
response = request.head();
Map<String, Link> orderLinks = processLinkHeaders(response);
```

Next, the client cancels the order that was created earlier. A HEAD request is made to the created order's URL to obtain the `cancel` link relationship:

```
Link cancel = orderLinks.get("cancel");
if (cancel != null)
{
    System.out.println("** Canceling the order at URL: "
                                + cancel.getHref());
    request = new ClientRequest(cancel.getHref());
    response = request.post();
    Assert.assertEquals(204, response.getStatus());
}
```

If there is a `cancel` link relationship, the client posts an empty message to this URL to cancel the order:

```
System.out.println();
System.out.println("** New list of orders after cancel: ");
request = new ClientRequest(orders.getHref());
response = request.get();
System.out.println(response.getEntity(String.class));
```

The client does another `GET /orders` to show that the state of our created order was set to cancelled:

```
System.out.println();
Link purge = ordersLinks.get("purge");
System.out.println("** Purge cancelled orders at URL: "
                        + purge.getHref());
request = new ClientRequest(purge.getHref());
response = request.post();
Assert.assertEquals(204, response.getStatus());

System.out.println();
System.out.println("** New list of orders after purge: ");
request = new ClientRequest(orders.getHref());
response = request.get();
System.out.println(response.getEntity(String.class));
}
```

Finally, by posting an empty message to the `purge` link, the client cleans the order entry database of any cancelled orders.

Build and Run the Example Program

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex09_2* directory of the workbook example code.
2. Make sure your PATH is set up to include both the JDK and Maven, as described in Chapter 15.
3. Perform the build and run the example by typing `maven install`.

Examples for Chapter 10

In Chapter 10, you learned about HTTP Caching techniques. Servers can tell HTTP clients if and how long they can cache retrieved resources. Expired caches can be re-validated to avoid resending big messages by issuing conditional GET invocations. Conditional PUT operations can be invoked for safe concurrent updates.

Example ex10_1: Caching and Concurrent Updates

The example in this chapter expands on the `CustomerResource` example repeated throughout this book to support caching, conditional GETs, and conditional PUTs.

The Server Code

The first thing is to add a `hashCode()` method to the `Customer` class:

src/main/java/com/restfully/shop/domain/Customer.java

```
@XmlRootElement(name = "customer")
public class Customer
{
...
   @Override
   public int hashCode()
   {
      int result = id;
      result = 31 * result + (firstName != null
                                 ? firstName.hashCode() : 0);
      result = 31 * result + (lastName != null
                                 ? lastName.hashCode() : 0);
      result = 31 * result + (street != null
                                 ? street.hashCode() : 0);
      result = 31 * result + (city != null ? city.hashCode() : 0);
      result = 31 * result + (state != null ? state.hashCode() : 0);
      result = 31 * result + (zip != null ? zip.hashCode() : 0);
      result = 31 * result + (country != null
                                    ? country.hashCode() : 0);
      return result;
```

```
        }
    }
```

This method is used in the `CustomerResource` class to generate semiunique `ETag` header values. While a hash code calculated in this manner isn't guaranteed to be unique, there is a high probability that it will be. A database application might use an incremented version column to calculate the `ETag` value.

The `CustomerResource` class is expanded to support conditional GETs and PUTs. Let's take a look at the relevant pieces of code:

src/main/java/com/restfully/shop/services/CustomerResource.java

```
    @Path("/customers")
    public class CustomerResource
    {
    ...

        @GET
        @Path("{id}")
        @Produces("application/xml")
        public Response getCustomer(@PathParam("id") int id,
                                    @Context Request request) {
            Customer cust = customerDB.get(id);
            if (cust == null)
            {
                throw new WebApplicationException(Response.Status.NOT_FOUND);
            }

            if (sent == null) System.out.println("No ETag sent by client");

            EntityTag tag = new EntityTag(Integer.toString(cust.hashCode()));

            CacheControl cc = new CacheControl();
            cc.setMaxAge(5);
```

The `getCustomer()` method first starts out by retrieving the current `Customer` object identified by the `id` parameter. A current `ETag` value is created from the hash code of the `Customer` object. A new `Cache-Control` header is instantiated as well:

```
        Response.ResponseBuilder builder =
                    request.evaluatePreconditions(tag);
        if (builder != null) {
            System.out.println(
                    "** revalidation on the server was successful");
            builder.cacheControl(cc);
            return builder.build();
        }
```

Next, `Request.evaluatePreconditions()` is called to perform a conditional GET. If the client has sent an `If-None-Match` header that matches the calculated current `ETag`, the method returns immediately with an empty response body. In this case, a new `Cache-Control` header is sent back to refresh the `max-age` the client will use:

```
            // Preconditions not met!

            cust.setLastViewed(new Date().toString());
            builder = Response.ok(cust, "application/xml");
            builder.cacheControl(cc);
            builder.tag(tag);
            return builder.build();
        }
    }
```

If no `If-None-Match` header was sent or the preconditions were not met, the `Customer` is sent back to the client with an updated `Cache-Control` header:

```
    @Path("{id}")
    @PUT
    @Consumes("application/xml")
    public Response updateCustomer(@PathParam("id") int id,
                                   @Context Request request,
                                    Customer update ) {
        Customer cust = customerDB.get(id);
        if (cust == null)
            throw new WebApplicationException(Response.Status.NOT_FOUND);
        EntityTag tag = new EntityTag(Integer.toString(cust.hashCode()));
```

The `updateCustomer()` method is responsible for updating a customer. It first starts off by finding the current `Customer` with the given `id`. From this queried customer, it generates the up-to-date value of the `ETag` header:

```
        Response.ResponseBuilder builder =
                        request.evaluatePreconditions(tag);

        if (builder != null) {
            // Preconditions not met!
            return builder.build();
        }
```

The current `ETag` header is compared against any `If-Match` header sent by the client. If it does match, the update can be performed:

```
        // Preconditions met, perform update

        cust.setFirstName(update.getFirstName());
        cust.setLastName(update.getLastName());
        cust.setStreet(update.getStreet());
        cust.setState(update.getState());
        cust.setZip(update.getZip());
        cust.setCountry(update.getCountry());


        builder = Response.noContent();
        return builder.build();
    }
}
```

Finally, the update is performed.

The Client Code

For simplicity, the client code is written using the RESTEasy client API. It first performs a conditional GET. It then tries to do a conditional PUT using a bad `ETag` value:

```
public class CustomerResourceTest
{
    @Test
    public void testCustomerResource() throws Exception
    {
        RegisterBuiltin.register(
                            ResteasyProviderFactory.getInstance());
        ClientRequest request =
                new ClientRequest("http://localhost:9095/customers/1");
        ClientResponse<Customer> response =
                                    request.get(Customer.class);
        Assert.assertEquals(200, response.getStatus());
        Customer cust = response.getEntity();

        String etag = response.getHeaders().getFirst("ETag");
```

The `testCustomerResource()` method starts off by fetching a preinitialized `Customer` object. It does this so that it can obtain the current `ETag` of the `Customer` representation:

```
        System.out.println("Doing a conditional GET with ETag: " + etag);
        request.clear();
        request.header("If-None-Match", etag);
        response = request.get(Customer.class);
        Assert.assertEquals(304, response.getStatus());
```

This code is performing a conditional GET. The `If-None-Match` header is set using the previously fetched `ETag` value. The client is expecting that the server return a 304, "Not Modified" response:

```
        // Update and send a bad etag with conditional PUT
        cust.setCity("Bedford");
        request.clear();
        request.header("If-Match", "JUNK");
        request.body("application/xml", cust);
        ClientResponse response2 = request.put();
        Assert.assertEquals(412, response2.getStatus());
    }
}
```

Finally, the code does a conditional PUT with a bad `ETag` value sent with the `If-Match` header. The client is expecting this operation to fail with a 412, "Precondition Failed" response.

Build and Run the Example Program

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex10_1* directory of the workbook example code.
2. Make sure your PATH is set up to include both the JDK and Maven, as described in Chapter 15.
3. Perform the build and run the example by typing `maven install`.

Another interesting thing you might want to try is to start up and leave the application running by doing a `maven jetty:run`. Open your browser to *http://localhost:9095/customers/1*. Continually refresh this URL. You will be able to see if your browser performs a conditional GET request or not by viewing the `<last-viewed>` element of the returned XML. I found that Firefox 3.5.2 does a conditional GET while Safari 4.0.1 does not.

Examples for Chapter 11

In Chapter 11, you learned a bit about how JAX-RS fits in the grander scheme of things like Java EE and Spring. In this chapter, there are two similar examples that define the services illustrated in Chapter 2. The first marries JAX-RS with EJB. The second uses the Spring form to write our JAX-RS services. Instead of using in-memory maps like the earlier examples in the workbook, both examples use Java Persistence (JPA) to map Java objects to a relational database.

Example ex11_1: EJB and JAX-RS

This example shows how you can use JAX-RS with EJB and JPA. It makes use of some of the integration code discussed in Chapter 11.

Project Structure

To implement *ex11_1*, the JBoss 5.1 Application Server is used to deploy the example. JBoss 5.1 is only Java EE 5–compliant, so it is not JAX-RS-aware. Also, because JBoss 5.1 is not a Java EE 6 implementation, I had to create three different Maven modules to comply with the Java EE 5 packaging structure. The directory structure is shown in Figure 24-1.

The main Maven module of the project is *ejb/*. Within this directory is a Maven JAR project. It contains the majority of the Java source code of the example. The JAXB, JPA, and EJB classes are all defined in this project along with the JPA deployment descriptor.

The next Maven module of the project is *war/*. This project configures RESTEasy as shown in all our previous examples. It also defines the `Application` class and a few `ExceptionMappers`.

The *ear/* project builds an EAR file and deploys it to the JBoss Application Server. This project is also where the test code lives. After the EAR file has been deployed, the test code is run.
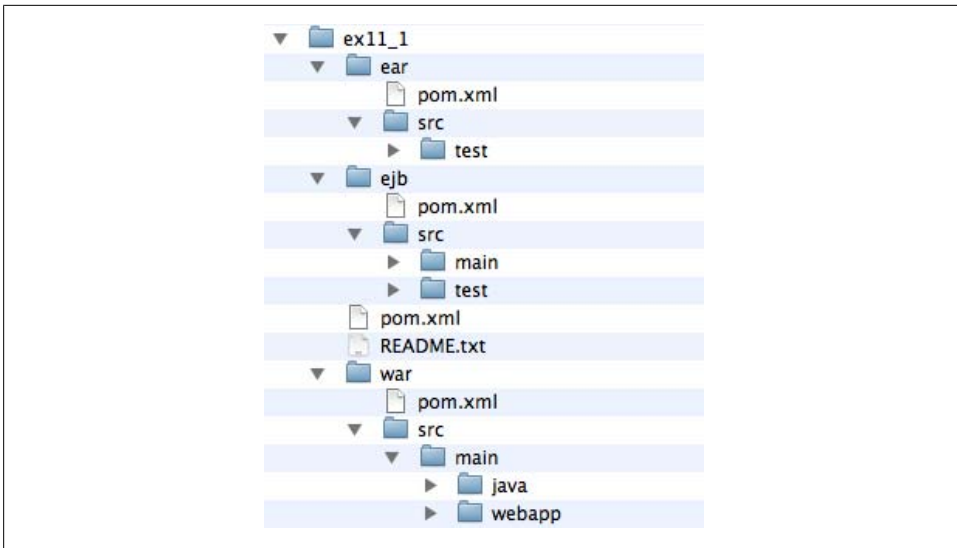
Figure 24-1. Directory structure

The EJB Project

The EJB project contains the bulk of the example's code. It is very similar to *ex09_2* from Chapter 22, except the project has been expanded to save created order entries into a relational database instead of an in-memory map. Like all of our previous examples, the JAXB classes that define our XML data format live in the `com.rest fully.shop.domain` package. A separate parallel Java package, `com.restfully.shop.per sistence`, was created for the example's JPA classes. These JPA classes are almost a carbon copy of the JAXB ones, except they are using JPA annotations to map to a relational database.

You could use JAXB and JPA annotations together within one class hierarchy, but this isn't the best idea, as there are a few problems you might encounter. The first has to do with how JPA works. Objects like the `OrderEntity` have relationships to other classes like `LineItemEntity`, `ProductEntity`, and `CustomerEntity`. In JPA, it is common to lazily load these objects as their object graphs are traversed. This can save on database access time. The problem where JAX-RS is concerned is that the JAX-RS runtime will usually turn the Java object into an XML document outside the scope of an EJB request. This might cause lazy-load exceptions when JAXB tries to traverse the entire object graph.

You can write your code so that it is careful not to introduce lazy-load exceptions, but there is one other major problem you may encounter. You will often want to support older clients that use older versions of the XML format. This can cause a divergence between your XML schema and your database schema. The best way to avoid this problem is to create two separate class hierarchies. That way, your XML and database

mappings can evolve separately from one another. Yeah, it's a little more code for you to write, but it will save you headaches in the long run.

I'm going to skip a lot of the details of this example. You've already seen how JAXB classes work and this book isn't an exercise on learning JPA, so I'll focus on how JAX-RS interacts with EJB. Let's take a look at one of the EJBs:

ejb/src/main/java/com/restfully/shop/services/CustomerResource.java

```java
@Path("/customers")
public interface CustomerResource
{
    @POST
    @Consumes("application/xml")
    Response createCustomer(Customer customer,
                                         @Context UriInfo uriInfo);

    @GET
    @Produces("application/xml")
    @Formatted
    Customers getCustomers(@QueryParam("start") int start,
                   @QueryParam("size") @DefaultValue("2") int size,
                   @QueryParam("firstName") String firstName,
                   @QueryParam("lastName") String lastName,
                   @Context UriInfo uriInfo);

    @GET
    @Path("{id}")
    @Produces("application/xml)
    Customer getCustomer(@PathParam("id") int id);
}
```

For a non-JAX-RS-aware EJB container to work with JAX-RS, you need to define your JAX-RS annotations on the EJB's business interface. The `CustomerResource` interface does just this.

The EJB code

Our EJB business logic is defined within the `CustomerResourceBean` class:

ejb/src/main/java/com/restfully/shop/services/CustomerResourceBean.java

```java
@Stateless
public class CustomerResourceBean implements CustomerResource
{
    @PersistenceContext
    private EntityManager em;
```

Our EJB class is annotated with the `@javax.ejb.Stateless` annotation to mark it as a stateless session EJB. The `CustomerResourceBean` class implements the `CustomerResource` interface.

There is a `javax.persistence.EntityManager` field named em. The annotation `@javax.persistence.PersistenceContext` injects an instance of the `EntityManager` into that field.

The `EntityManager` persists Java objects into a relational database. These are all facilities of EJB and JPA:

```
public Response createCustomer(Customer customer, UriInfo uriInfo)
{
    CustomerEntity entity = new CustomerEntity();
    domain2entity(entity, customer);
    em.persist(entity);
    em.flush();

    System.out.println("Created customer " + entity.getId());
    UriBuilder builder = uriInfo.getAbsolutePathBuilder();
    builder.path(Integer.toString(entity.getId()));
    return Response.created(builder.build()).build();

}
```

The `createCustomer()` method implements the RESTful creation of a `Customer` in the database. The `Customer` object is the unmarshalled representation of the XML document posted through HTTP. The code allocates an instance of `com.restfully.shop.per sistence.CustomerEntity` and copies the data from `Customer` to this instance. The `Enti tyManager` then persists the `CustomerEntity` instance into the database. Finally, the method uses `UriInfo.getAbsolutePathBuilder()` to create a URL that will populate the value of the `Location` header that is sent back with the HTTP response:

```
public Customer getCustomer(int id)
{
    CustomerEntity customer = em.getReference(CustomerEntity.class,
                                              id);
    return entity2domain(customer);
}
```

The `getCustomer()` method services `GET /customers/<id>` requests and retrieves `CustomerEntity` objects from the database using the `EntityManager`. The `entity2domain()` method call converts the `CustomerEntity` instance found in the database into an instance of the JAXB class `Customer`. This `Customer` instance is what is returned to the JAX-RS runtime:

```
public static void domain2entity(CustomerEntity entity,
                                 Customer customer)
{
    entity.setId(customer.getId());
    entity.setFirstName(customer.getFirstName());
    entity.setLastName(customer.getLastName());
    entity.setStreet(customer.getStreet());
    entity.setCity(customer.getCity());
    entity.setState(customer.getState());
    entity.setZip(customer.getZip());
    entity.setCountry(customer.getCountry());
}

public static Customer entity2domain(CustomerEntity entity)
{
```

```
                  Customer cust = new Customer();
                  cust.setId(entity.getId());
                  cust.setFirstName(entity.getFirstName());
                  cust.setLastName(entity.getLastName());
                  cust.setStreet(entity.getStreet());
                  cust.setCity(entity.getCity());
                  cust.setState(entity.getState());
                  cust.setZip(entity.getZip());
                  cust.setCountry(entity.getCountry());
                  return cust;
              }
```

The `domain2entity()` and `entity2domain()` methods simply convert to and from the
JAXB and JPA class hierarchies:

```
      public Customers getCustomers(int start,
                                    int size,
                                    String firstName,
                                    String lastName,
                                    UriInfo uriInfo)
      {
          UriBuilder builder = uriInfo.getAbsolutePathBuilder();
          builder.queryParam("start", "{start}");
          builder.queryParam("size", "{size}");

          ArrayList<Customer> list = new ArrayList<Customer>();
          ArrayList<Link> links = new ArrayList<Link>();
```

The `getCustomers()` method is expanded as compared to previous examples in this
book. The `firstName` and `lastName` query parameters are added. This allows clients to
search for customers in the database with a specific first and last name:

```
      Query query = null;
      if (firstName != null && lastName != null)
      {
          query = em.createQuery(
                  "select c from Customer c where c.firstName=:first
                       and c.lastName=:last");
          query.setParameter("first", firstName);
          query.setParameter("last", lastName);

      }
      else if (lastName != null)
      {
          query = em.createQuery(
                  "select c from Customer c where c.lastName=:last");
          query.setParameter("last", lastName);
      }
      else
      {
          query = em.createQuery("select c from Customer c");
      }
```

The `getCustomers()` method builds a JPA query based on the values of `firstName` and
`lastName`. If these are both set, it searches in the database for all customers with that

first and last name. If only `lastName` is set, it searches only for customers with that last name. Otherwise, it just queries for all customers in the database:

```
List customerEntities = query.setFirstResult(start)
        .setMaxResults(size)
        .getResultList();
```

Next, the code executes the query. You can see that doing paging is a little bit easier with JPA than the in-memory database we used in Chapter 22. The `setMaxResults()` and `Query.setFirstResult()` methods set the index and size of the dataset you want returned:

```
for (Object obj : customerEntities)
{
   CustomerEntity entity = (CustomerEntity) obj;
   list.add(entity2domain(entity));
}
```

Next, the code iterates through all the `CustomerEntity` objects returned by the executed query and creates `Customer` JAXB object instances:

```
// next link
// If the size returned is equal then assume there is a next
if (customerEntities.size() == size)
{
   int next = start + size;
   URI nextUri = builder.clone().build(next, size);
   Link nextLink = new Link("next",
                       nextUri.toString(), "application/xml");
   links.add(nextLink);
}
// previous link
if (start > 0)
{
   int previous = start - size;
   if (previous < 0) previous = 0;
   URI previousUri = builder.clone().build(previous, size);
   Link previousLink = new Link("previous",
                   previousUri.toString(), "application/xml");
   links.add(previousLink);
}
Customers customers = new Customers();
customers.setCustomers(list);
customers.setLinks(links);
return customers;
   }

}
```

Finally, the method calculates whether the `next` and `previous` Atom links should be added to the `Customers` JAXB object returned. This code is very similar to the examples described in Chapter 22.

The other EJB classes defined in the example are pretty much extrapolated from the *ex09_2* example and modified to work with JPA. I don't want to rehash old code, so I won't get into detail on how these work.

The ExceptionMappers

The `EntityManager.getReference()` method is used by various EJBs in this example to locate objects within the database. When this method cannot find an object within the database, it throws a `javax.persistence.EntityNotFoundException`. If we deployed this code as-is, JAX-RS would end up eating this exception and returning a 500, "Internal Server Error" to our clients if they tried to access an unknown object in the database. The 404, "Not Found" error response code makes a lot more sense to return in this scenario. To facilitate this, a few JAX-RS `ExceptionMappers` are used. Let's take a look:

ejb/src/main/java/com/restfully/shop/services/EntityNotFoundExceptionMapper.java

```
@Provider
public class EntityNotFoundExceptionMapper
               implements ExceptionMapper<EntityNotFoundException>
{
   public Response toResponse(EntityNotFoundException exception)
   {
      return Response.status(Response.Status.NOT_FOUND).build();
   }
}
```

This class catches `EntityNotFoundExceptions` and generates a 404 response. Unfortunately, the example is not complete. The EJB specification requires that any `RuntimeException` that is thrown by an EJB method be wrapped within a `javax.ejb.EJBException` and rethrown. Even though our EJB method throws `EntityNotFoundException`, the JAX-RS runtime would instead receive an `EJBException` and would not know how to handle the exception. To make this work, an additional `ExceptionMapper` is used to catch `EJBExceptions` and to look for an `ExceptionMapper` that can handle the cause of the `EJBException`:

ejb/src/main/java/com/restfully/shop/services/EJBExceptionMapper.java

```
@Provider
public class EJBExceptionMapper
                    implements ExceptionMapper<EJBException>
{
   @Context
   private Providers providers;

   public Response toResponse(EJBException exception)
   {
      if (exception.getCausedByException() == null)
      {
         return Response.serverError().build();
      }
      Class cause = exception.getCausedByException().getClass();
      ExceptionMapper mapper = providers.getExceptionMapper(cause);
```

```
                if (mapper == null)
                {
                    return Response.serverError().build();
                }
                else
                {
                    return mapper.toResponse(exception.getCausedByException());
                }
            }
        }
    }
```

The EJBExceptionMapper class has a javax.ws.rs.ext.Providers interface injected into
it. It uses the Providers.getExceptionMapper() method to find an ExceptionMapper that
might handle root exceptions. If one is found, a Response is created from that mapper.
Otherwise a 500, "Internal Service Error" is returned to the client.

The WAR Project

The code in the WAR project configures RESTEasy. It also registers the EJBs defined
by the EJB project with the JAX-RS runtime. Since JBoss 5.1 is not JAX-RS-aware, we
need to follow the conventions discussed in Chapter 11 for doing this.

The first set of changes is to *web.xml*:

war/src/main/webapp/WEB-INF/web.xml

```
<web-app>
...
    <ejb-local-ref>
        <ejb-ref-name>ejb/CustomerResource</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <local>com.restfully.shop.services.CustomerResource</local>
        <ejb-link>
            com.restfull.shop.services.CustomerResourceBean
        </ejb-link>
    </ejb-local-ref>

    <ejb-local-ref>
        <ejb-ref-name>ejb/ProductResource</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <local>com.restfully.shop.services.ProductResource</local>
        <ejb-link>
            com.restfull.shop.services.ProductResourceBean
        </ejb-link>
    </ejb-local-ref>

    <ejb-local-ref>
        <ejb-ref-name>ejb/OrderResource</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <local>com.restfully.shop.services.OrderResource</local>
        <ejb-link>
            com.restfull.shop.services.OrderResourceBean
        </ejb-link>
    </ejb-local-ref>
```

```
        <ejb-local-ref>
            <ejb-ref-name>ejb/StoreResource</ejb-ref-name>
            <ejb-ref-type>Session</ejb-ref-type>
            <local>com.restfully.shop.services.StoreResource</local>
            <ejb-link>
                com.restfull.shop.services.StoreResourceBean
            </ejb-link>
        </ejb-local-ref>
    </web-app>
```

The first part of the *web.xml* file configures RESTEasy and the `Application` class. This is exactly the same as the other examples in the workbook, so I won't go over this configuration. The remainder of the file defines EJB references to the EJBs that were defined in the EJB project.

The `ShoppingApplication` class looks up these EJB references and registers them with the JAX-RS runtime:

war/src/main/java/com/restfully/shop/services/ShoppingApplication.java

```
    public class ShoppingApplication extends Application
    {

        public Set<Class<?>> getClasses()
        {
            Set<Class<?>> classes = new HashSet<Class<?>>();
            classes.add(EntityNotFoundExceptionMapper.class);
            classes.add(EJBExceptionMapper.class);
            return classes;
        }
```

The `getClasses()` method returns the `ExceptionMappers` we defined in the EJB project so that they can be registered with the JAX-RS runtime:

```
        public Set<Object> getSingletons()
        {
            HashSet<Object> set = new HashSet();
            try
            {
                InitialContext ctx = new InitialContext();
                Object obj = ctx.lookup(
                        "java:comp/env/ejb/CustomerResource");
                set.add(obj);

                obj = ctx.lookup(
                        "java:comp/env/ejb/OrderResource");
                set.add(obj);

                obj = ctx.lookup(
                        "java:comp/env/ejb/ProductResource");
                set.add(obj);

                obj = ctx.lookup(
                        "java:comp/env/ejb/StoreResource");
                set.add(obj);
```

```
        }
        catch (Exception ex)
        {
            throw new RuntimeException(ex);
        }
        return set;
    }
}
```

The `getSingletons()` method does a JNDI lookup of all the EJB references that were defined in the *web.xml*. It adds these references to a `Set` and returns them to the JAX-RS runtime so that they can be registered as JAX-RS services.

The EAR Project

If JBoss 5.1 were a Java EE 6–compliant server, the example could have been packaged as a simple WAR file. Java EE 5 does not allow you to define EJBs within a WAR file. EJBs must be packaged within their own JARs and bundled with the WAR file within an EAR. The Maven EAR project accomplishes this task.

The EAR project also contains the test client code. Let's take a look:

ear/src/test/java/com/restfully/shop/test/ShoppingTest.java

```
public class ShoppingTest
{
    @BeforeClass
    public static void init()
    {
        RegisterBuiltin.register(
            RestEasyProviderFactory.getInstance());
    }
```

The `init()` method initializes the RESTEasy client runtime:

```
protected Map<String, Link>
                        processLinkHeaders(ClientResponse response)
{
    List<String> linkHeaders = (List<String>) response
                                .getHeaders().get("Link");
    Map<String, Link> links = new HashMap<String, Link>();
    for (String header : linkHeaders)
    {
        Link link = Link.valueOf(header);
        links.put(link.getRelationship(), link);
    }
    return links;
}
```

The `processLinkHeaders()` is the same method mentioned in *ex09_2* from Chapter 22. It is used to process `Link` headers:

```
@Test
public void testPopulateDB() throws Exception
```

```
        {
            String url = "http://localhost:9095/shop";
            ClientRequest request =
                new ClientRequest("http://localhost:8080/ex11_1-war/shop");
            ClientResponse response = request.head();
            Map<String, Link> shoppingLinks = processLinkHeaders(response);

            System.out.println("** Populate Products");
            request = new ClientRequest(shoppingLinks.get("products")
                                                    .getHref());

            Product product = new Product();
            product.setName("iPhone");
            product.setCost(199.99);
            request.body("application/xml", product);
            response = request.post();
            Assert.assertEquals(201, response.getStatus());

            product = new Product();
            product.setName("MacBook Pro");
            product.setCost(3299.99);
            request.body("application/xml", product);
            response = request.post();
            Assert.assertEquals(201, response.getStatus());

            product = new Product();
            product.setName("iPod");
            product.setCost(49.99);
            request.body("application/xml", product);
            response = request.post();
            Assert.assertEquals(201, response.getStatus());

        }
```

The `testPopulateDB()` method makes HTTP calls on the `ProductResource` JAX-RS service to create a few products in the database:

```
        @Test
        public void testCreateOrder() throws Exception
        {
            String url = "http://localhost:9095/shop";
            ClientRequest request =
                new ClientRequest("http://localhost:8080/ex11_1-war/shop");
            ClientResponse response = request.head();
            Map<String, Link> shoppingLinks = processLinkHeaders(response);
```

Like *ex09_2*, the client interacts with the `StoreResource` JAX-RS service to obtain links to all the services in the system:

```
        System.out.println("** Buy an iPhone for Bill Burke");
        System.out.println();
        System.out.println("** First see if Bill Burke exists
                                                as a customer");
        request = new ClientRequest(shoppingLinks.get("customers")
                        .getHref() + "?firstName=Bill&lastName=Burke");
        Customers customers = request.getTarget(Customers.class);
```

```
Customer customer = null;
if (customers.getCustomers().size() > 0)
{
    System.out.println("- Found a Bill Burke in the database,
                                              using that");
    customer = customers.getCustomers().iterator().next();
}
else
{
    System.out.println("- Cound not find a Bill Burke
                              in the database, creating one.");
    customer = new Customer();
    customer.setFirstName("Bill");
    customer.setLastName("Burke");
    customer.setStreet("222 Dartmouth Street");
    customer.setCity("Boston");
    customer.setState("MA");
    customer.setZip("02115");
    customer.setCountry("USA");
    request = new ClientRequest(shoppingLinks.get("customers")
                                              .getHref());
    request.body("application/xml", customer);
    response = request.post();
    Assert.assertEquals(201, response.getStatus());
    String uri = (String) response.getHeaders()
                              .getFirst("Location");

    request = new ClientRequest(uri);
    customer = request.getTarget(Customer.class);
}
```

The first thing the client code does is to check if customer "Bill Burke" already exists. If that customer doesn't already exist, it is created within the customer database:

```
System.out.println();
System.out.println("Search for iPhone in the Product database");
request = new ClientRequest(shoppingLinks.get("products")
                      .getHref() + "?name=iPhone");
Products products = request.getTarget(Products.class);
Product product = null;
if (products.getProducts().size() > 0)
{
    System.out.println("- Found iPhone in the database.");
    product = products.getProducts().iterator().next();
}
else
{
    throw new RuntimeException(
                "Failed to find an iPhone in the database!");
}
```

The customer wants to buy a product called iPhone, so the client searches the product database for it:

```
System.out.println();
System.out.println("** Create Order for iPhone");
```

```
            LineItem item = new LineItem();
            item.setProduct(product);
            item.setQuantity(1);
            Order order = new Order();
            order.setTotal(product.getCost());
            order.setCustomer(customer);
            order.setDate(new Date().toString());
            order.getLineItems().add(item);
            request = new ClientRequest(shoppingLinks.get("orders")
                                            .getHref());
            request.body("application/xml", order);
            response = request.post();
            Assert.assertEquals(201, response.getStatus());

            System.out.println();
            System.out.println("** Show all orders.");
            request = new ClientRequest(shoppingLinks.get("orders")
                                            .getHref());
            String xml = request.getTarget(String.class);
            System.out.println(xml);


    }
}
```

Finally, an order is created within the database.

Build and Run the Example Program

Perform the following steps:

1. Download JBoss 5.1 Application Server from *http://jboss.org/jbossas/downloads*.
2. Unzip JBoss 5.1 into any directory you want.
3. Open a command prompt or shell terminal and change to *jboss-5.1.0.GA/bin* directory.
4. JBoss must be started manually before you can run the example. To do this, execute *run.sh* or *run.bat*, depending on whether you are using a Unix- or Windows-based system.
5. Open another command prompt or shell terminal and change to the *ex11_1* directory of the workbook example code.
6. Make sure your PATH is set up to include both the JDK and Maven, as described in Chapter 15.
7. Perform the build and run the example by typing `maven install`.

The *pom.xml* file within the EAR project uses a special JBoss plug-in so that it can deploy the EAR file from the example to the application server. After the EAR is deployed, the client test code will be executed. Following the execution of the test, the EAR will be undeployed from JBoss by Maven.

Example ex11_2: Spring and JAX-RS

There isn't much difference between the code of *ex11_1* and *ex11_2*. The Java classes are basically the same, except the `@Stateless` annotations were removed from the JAX-RS resource classes because the example is using Spring instead of EJB for its component model. For the same reason, the `EJBExceptionMapper` class isn't needed anymore.

Besides the removal of EJB metadata, the differences between the two projects are mainly packaging and configuration. If you look through the *ex11_2* directory, you'll see that we're back to using embedded Jetty and a single WAR project. The *web.xml* file is a bit different than the EJB example, so let's take a look at that first:

src/main/webapp/WEB-INF/web.xml

```
<web-app>
...

    <env-entry>
        <env-entry-name>spring-beans-file</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>
            META-INF/applicationContext.xml
        </env-entry-value>
    </env-entry>


    </web-app>
```

This example follows the Spring integration conventions discussed in Chapter 11. The *web.xml* file adds an `<env-entry>` to point to the Spring XML file that holds all of the example's Spring configuration. Let's look at this Spring XML file:

src/main/resources/applicationContext.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd"
       default-autowire="byName">

    <bean id="entityManagerFactory"
       class="org.springframework.orm.jpa.
                    LocalContainerEntityManagerFactoryBean">
        <property name="jpaVendorAdapter">
            <bean class="org.springframework.orm.jpa.vendor
                                .HibernateJpaVendorAdapter">
                <property name="showSql" value="false"/>
                <property name="generateDdl" value="true"/>
                <property name="databasePlatform"
```

```
                                     value="org.hibernate.dialect.HSQLDialect"/>
                 </bean>
             </property>
         </bean>

         <bean id="dataSource"
               class="org.apache.commons.dbcp.BasicDataSource"
               destroy-method="close">
             <property name="driverClassName"
                         value="org.hsqldb.jdbcDriver"/>
             <property name="url" value="jdbc:hsqldb:test/db/myDB"/>
             <property name="username" value="sa"/>
             <property name="password" value=""/>
         </bean>

         <bean id="transactionManager"
               class="org.springframework.orm.jpa.JpaTransactionManager"/>

         <tx:annotation-driven/>
```

The first part of the Spring configuration file is the configuration required to get JPA and Spring to work together. While the package structure for the Spring example is simpler than the EJB one, you can see that the configuration is a bit more complex:

```
         <bean class="org.springframework.orm
                 .jpa.support.PersistenceAnnotationBeanPostProcessor"/>

         <bean id="customer" class="com.restfully.shop.services
                                             .CustomerResourceBean"/>
         <bean id="product" class="com.restfully.shop.services
                                             .ProductResourceBean"/>
         <bean id="order" class="com.restfully.shop.services
                                             .OrderResourceBean"/>
         <bean id="store" class="com.restfully.shop.services
                                             .StoreResourceBean"/>
     </beans>
```

The rest of the Spring XML file defines all of the JAX-RS resource beans.

The Spring XML file is loaded and registered with the JAX-RS runtime by the ShoppingApplication class:

src/main/java/com/restfully/shop/services/ShoppingApplication.java

```
     public class ShoppingApplication extends Application
     {
         private Set<Class<?>> classes = new HashSet<Class<?>>();

         public ShoppingApplication()
         {
             classes.add(EntityNotFoundExceptionMapper.class);
         }

         public Set<Class<?>> getClasses()
         {
             return classes;
```

```
    }

    protected ApplicationContext springContext;

    public Set<Object> getSingletons()
    {
        try
        {
            InitialContext ctx = new InitialContext();
            String xmlFile = (String) ctx.lookup(
                    "java:comp/env/spring-beans-file");
            springContext = new ClassPathXmlApplicationContext(xmlFile);


        }
        catch (Exception ex)
        {
            ex.printStackTrace();
            throw new RuntimeException(ex);
        }
        HashSet<Object> set = new HashSet();
        set.add(springContext.getBean("customer"));
        set.add(springContext.getBean("order"));
        set.add(springContext.getBean("product"));
        set.add(springContext.getBean("store"));
        return set;
    }

}
```

The `getSingletons()` method is responsible for initializing Spring and registering any JAX-RS resource beans created by Spring with the JAX-RS runtime. It first looks up the name of the Spring XML configuration file within JNDI. It then initializes a Spring `ApplicationContext` from that file. Finally, it looks up each JAX-RS bean within the project and registers it with the JAX-RS runtime.

Build and Run the Example Program

Perform the following steps:

1. Open a command prompt or shell terminal and change to the *ex11_2* directory of the workbook example code.

2. Make sure your PATH is set up to include both the JDK and Maven, as described in Chapter 15.

3. Perform the build and run the example by typing `maven install`.

Index

getCustomer method, 31, 260
getEntity method, 97
getMatrixParameters method, 58
getMetadata method, 97
getOrderHeaders method, 252
getOrders method, 252
getPath method, 58, 59
getPathParameters method, 59
getSingletons method, 38, 142
getSize method, 86, 88
getStatus method, 97
getUserPrincipal method, 164
GlassFish, 179
GZIP compression, RESTEasy support of, 192

H

hashCode method, 259
HATEOAS, 4, 11, 119–125
    advantages of use with web services, 121–
        124
        decoupling interaction details, 121
        location transparency, 121
        reduced state transition errors, 122
        W3C standardization, 124
    Atom links, 120
    engine of application state, 12
    JAX-RS and, 125–130
        UriBuilder class, 125
        UriInfo, 127–130
    link headers versus Atom links, 124
    web services and, 120–125
    workbook examples
        Atom links, 243–248
        link headers, 248–257
HEAD method, 8
    JAX-RS annotation for, 43
href attribute, 120
hreflang attribute, 121
HTML forms, 119
HTTP, 4
    binding of methods, 43
    caching, 132
    content negotiation (see content
        negotiation)
    method extensions, 44
    methods, 7
HTTP methods
    complex responses, 97–102
        GenericEntity class, 101

        returning cookies, 99
        Status enum, 100
    default response codes, 95–97
        errors, 96
        successful responses, 96
    exception handling, 102–104
        ExceptionMapper, 103
    workbook example (see PATCH method
        example)
HttpHeaders interface
    conneg information for incoming requests,
        109
HTTPS protocol, 155
    authentication, usage in, 158
HttpURLConnection class, 165
    authentication, 167
    caching, 167
    Client Certificate authentication, 168
hyperlinks, 119
Hypermedia As The Engine Of Application
    State (see HATEOAS)

I

implementations (JAX-RS), 179
    Apache CXF, 185–189
    Jersey, 179–184
        client API, 180
        component integration, 184
        data formats, 182
        embeddable Jersey, 179
        MVC support, 183
        WADL, 181
    RESTEasy, 189–194
informational codes, 101
injection, 55
    @Encoded, 68
    @CookieParam, 63–64
    @DefaultValue, 67
    @FormParam, 61
    @HeaderParam, 62
    @MatrixParam, 60
    @Path Param, 56–59
    @QueryParam, 60
    automatic Java type conversion, 65–68
        collections, 67
        conversion failures, 67
        Java object conversion, 66
        primitive type conversion, 65
    common annotations for, 55

About the Author

Bill Burke is a Fellow at the JBoss division of Red Hat. A longtime JBoss contributor and architect, he has founded many projects, including JBoss Clustering, AOP, and EJB 3.0. He was a JCP Expert Group member for the Java EE 5, EJB 3.0, and JAX-RS specifications. Bill is also co-author of O'Reilly's fourth and fifth editions of *Enterprise JavaBeans*. His current projects are RESTEasy, a JAX-RS implementation, and REST-*.org, an organization dedicated to defining specifications for RESTful enterprise middleware.

Colophon

The animal on the cover of *RESTful Java with JAX-RS* is an Australian bee-eater (*Merops ornatus*). It is commonly referred to as a rainbow bee-eater because of the vibrant colored feathers that adorn its body. Its bronze crown and nape, blue rump, and green and bronze wings make it easily distinguishable. Its red eye sits inside of a black stripe, outlined in blue, that extends from its bill to its ears. Females and males look alike and are only differentiated by the female's shorter and thicker tail streamers.

Distributed throughout Australia, Papua New Guinea, and eastern Indonesia, the Australian bee-eater usually lives in cleared areas and often uses quarries or mines to build its nesting tunnels. Of course, tunnels in such places are subject to destruction as a result of human activity. Other threats to the bee-eater's survival include foxes and wild dogs that dig up its nesting tunnels.

It is believed that Australian bee-eaters are monogamous. The female builds the nesting tunnels, while her male partner catches food for both of them. To dig the tunnel, the female balances on her wings and feet, using her bill to dig and her feet to move loose soil backward. On average, she can dig about three inches per day.

Although the nesting tunnels are very narrow, bee-eaters have been known to share tunnels with other bee-eaters and sometimes even other bird species. The female can lay as many as seven eggs at a time. Both parents are responsible for incubating them (for about 24 days) and feeding them once they hatch. Often older birds that never found a mate or whose mate has died will help feed others' young as well.

Not surprisingly, the Australian bee-eater preys on bees, and though it is unaffected by the bee's sting, it is very careful to rub the bee on its perch to remove its stinger before consuming it. The bird always takes care to close its eye to prevent any poison from the bee's broken poison sac getting in it. The Australian bee-eater can consume several bees in the course of a single day and thus beekeepers generally aren't fans of the bird. Its diet consists of other insects as well, including dragonflies, beetles, butterflies, and moths.

The cover image is from *Cassell's Natural History*, Vol. III. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSansMonoCondensed.