From Technologies to Solutions

RESTful Java Web Services

Master core REST concepts and create RESTful web services in Java

Jose Sandoval

PACKT PUBLISHING

RESTful Java Web Services

Master core REST concepts and create RESTful web services in Java

Jose Sandoval

BIRMINGHAM - MUMBAI

RESTful Java Web Services

First published: November 2009

Production Reference: 1051109

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK

ISBN 978-1-847196-46-0

www.packtpub.com

Cover Image by Duraid Fatouhi (duraidfatouhi@yahoo.com)

Credits

Author
Jose Sandoval

Reviewers
Atanas Roussev
Richard Wallace

Acquisition Editor
Sarah Cullington

Development Editor
Dhiraj Chandiramani

Technical Editor
Ishita Dhabalia

Copy Editor
Sanchari Mukherjee

Indexer
Rekha Nair

Editorial Team Leader
Gagandeep Singh

Project Team Leader
Lata Basantani

Project Coordinator
Srimoyee Ghoshal

Proofreader
Lynda Silwoski

Graphics
Nilesh R. Mohite

Production Coordinator
Dolly Dasilva

Cover Work
Dolly Dasilva

About the Author

Jose Sandoval is a software developer based in Canada. He has played and worked with web technologies since the Mosaic web browser days. For the last 12 years he's worked or consulted for various financial institutions and software companies in North America, concentrating on large-scale Java web applications. He holds a Bachelor of Mathematics from the University of Waterloo and an MBA from Wilfrid Laurier University.

Aside from coding and writing, he enjoys watching a good soccer match and coaching his son's soccer team. You can learn more about his interests at his website `www.josesandoval.com` or his consulting firm's website `www.sandoval.ca`. Or you can reach him directly at `jose@josesandoval.com`.

I would like to thank Renee and Gabriel, for being the center and compass of my adventures; my family, for supporting me unconditionally; my friends and colleagues, for challenging me at every opportunity; my clients, for trusting me with their projects; and the entire Packt Publishing team, for helping me throughout the writing of this book.

About the Reviewers

Atanas Roussev is a father, an entrepreneur, and a software engineer. A certified Sun and Oracle developer, his work can be found at EA, Morgan Stanley, and many startups. His latest activities are in Java, GWT, mobile programming, and building HTTP4e (Eclipse add-on for HTTP and REST).

In the last decade he moved from Bulgaria to Vancouver, British Columbia, learning new words such as "timbits" and "double-double". He enjoys any offline time in the Rockies and he loves challenging his three kids at Guitar Hero and math.

You can find him at `www.roussev.org` or just e-mail him at `atanas@roussev.org`.

Richard Wallace is a software developer, currently working for Atlassian. He has been developing Java software for over seven years and has been a strong advocate for RESTful web services at Atlassian since starting there two years ago. He enjoys reading a good sci-fi book and spending time with his family.

I'd like to thank my wonderful wife and kids for making life an exciting adventure everyday.

Table of Contents

Preface

If you're already familiar with REST theory, but are new to RESTful Java web services, and want to use the Java technology stack together with Java RESTful frameworks to create robust web services, this is the book for you.

This book is a guide for developing RESTful web services using Java and the most popular RESTful Java frameworks available today. This book covers the theory of REST, practical coding examples for RESTful clients, a practical outline of the RESTful design, and a complete implementation of a non-trivial web service using the frameworks Jersey's JAX-RS, Restlet's Lightweight REST, JBoss's JAX-RS RESTEasy, and Struts 2 with the REST plugin.

We cover each framework in detail so you can compare their strengths and weaknesses. This coverage will also provide you with enough knowledge to begin developing your own web services after the first reading. What's more, all the source code is included for you to study and modify. Finally, we discuss performance issues faced by web service developers and cover practical solutions for securing your web services.

What this book covers

Chapter 1, *RESTful Architectures*, introduces you to the REST software architectural style and discusses the constraints, main components, and abstractions that make a software system RESTful. It also elaborates on the details of HTTP requests and responses between clients and servers, and the use of RESTful web services in the context of Service-Oriented Architectures (SOA).

Chapter 2, *Accessing RESTful Services – Part 1*, teaches you to code four different RESTful Java clients that connect and consume RESTful web services, using the messaging API provided by Twitter.

Chapter 3, *Accessing RESTful Services – Part 2*, shows you how to develop a mashup application that uses RESTful web services that connect to Google, Yahoo!, Twitter, and TextWise's SemanticHacker API. It also covers in detail what it takes to consume JSON objects using JavaScript.

Chapter 4, *RESTful Web Services Design*, demonstrates how to design a micro-blogging web service (similar to Twitter), where users create accounts and then post entries. It also outlines a set of steps that can be used to design any software system that needs to be deployed as a RESTful web service.

Chapter 5*, Jersey: JAX-RS*, implements the micro-blogging web service specified in Chapter 4 using Jersey, the reference implementation of Sun's Java API for RESTful Web Services.

Chapter 6, *The Restlet Framework*, implements the micro-blogging web service specified in Chapter 4 using the Restlet framework, using two of its latest versions, 1.1 and 2.0.

Chapter 7, *RESTEasy: JAX-RS*, implements the micro-blogging web service specified in Chapter 4 using JBoss's RESTEasy framework.

Chapter 8, *Struts 2 and the REST Plugin*, implements the micro-blogging web service specified in Chapter 4 using Struts 2 framework (version 2.1.6) together with the REST plugin. This chapter covers configuration of Struts 2 and the REST plugin, mapping of URIs to Struts 2 action classes, and handling of HTTP requests using the REST plugin.

Chapter 9, *Restlet Clients and Servers*, extends coverage of the Restlet framework. This chapter looks at the client connector library and the standalone server library.

Chapter 10, *Security and Performance*, explores how to secure web services using HTTP Basic Authentication, and covers the OAuth authentication protocol. This chapter also covers the topics of availability and scalability and how they relate to implementing high performing web services.

What you need for this book

At the beginning of each chapter, you're given a list of the tools you will need to code and to compile the sample applications presented. However, the main software tools needed are the latest Java JDK and the latest Tomcat web server—these tools are available for any modern operating system.

Who this book is for

This book is for developers who want to code RESTful web services using the Java technology stack together with any of the frameworks Jersey's JAX-RS, Restlet's Lightweight REST, JBoss's JAX-RS RESTEasy, and Struts 2 with the REST plugin.

You don't need to know REST, as we cover the theory behind it all; however, you should be familiar with the Java language and have some understanding of Java web applications.

For each framework, we develop the same web service outlined in Chapter 4, *RESTful Web Services Design*. This is a practical guide and a greater part of the book is about coding RESTful web services, and not just about the theory of REST.

Conventions

In this book, you'll find a number of different styles of text that differentiate between sections in every chapter. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows (note the keyword `true`): "Without this directive set to `true`, our application will not identify resource classes to handle HTTP requests."

A block of code is set as follows:

```
if (acceptHeader == null || acceptHeader.isEmpty()
    || acceptHeader.equals("application/xml")) {
    representation = UserBO.getAllXML();
} else if (acceptHeader.equals("application/json")) {
    representation = UserBO.getAllJSON();
    type = "json";
}
```

When we wish to draw your attention to a particular portion of a code block, the relevant lines or items are set in bold as follows:

```
@GET
@Produces("application/xml")
public String getXML() {
return UserBO.getAllXML();
}
```

Any command-line input or output is written as follows:

```
javac -classpath "/apache-tomcat-6.0.16/lib/servlet-api.jar;commons-
logging-1.1.1.jar;commons-codec-1.3.jar;commons-httpclient-3.1.jar"
*.java
```

New terms and **important words** are shown in bold.

Words that you see on the screen in menus or dialog boxes appear in the text bolded, for example, "Note the **Response** in the right pane of the Swing application."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcomed. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important to us and helps us develop titles that offer you the most value for your money.

To send us general feedback, simply send an email to `feedback@packtpub.com`, mentioning the book's title in the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and are interested in writing a book about it or in contributing to one, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code for the book

Visit `http://www.packtpub.com/files/code/6460_Code.zip`.

Errata

Although we have taken every opportunity to ensure the accuracy of our content, mistakes do happen. If you find mistakes in one of our books—in the text or the code samples—we would be grateful if you report them to us. Reporting errors or inaccuracies will improve subsequent versions of this book.

If you find any errors, please report them by visiting `http://www.packtpub.com/support`, selecting the title of this book, clicking on the **let us know** link, and entering the details of the error in the provided form. Once your submission is verified, it will be added to the existing errata. Any existing errata can be viewed at `http://www.packtpub.com/support`.

Piracy

Piracy of copyrighted materials on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyrighted materials and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the website's address and name and we'll take immediate action. Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com`, if you are having a problem with any aspect of the book, and we will do our best to address it.

1
RESTful Architectures

In this chapter, we cover the REST software architectural style, as described in Roy Fielding's PhD dissertation. We discuss the set of constraints, the main components, and the abstractions that make a software system RESTful. We also look in detail at how data transfers take place between clients and servers. Finally, we look at how RESTful web services are used in the context of large **Service-Oriented Architectures** (**SOA**).

This chapter distills the theory of REST to its main core. No previous knowledge about the subject is necessary, but I assume you are familiar with web technologies and the basics of the HTTP protocol.

What is REST?

The term **REST** comes from Roy Fielding's PhD dissertation, published in 2000, and it stands for **REpresentational State Transfer**. REST by itself is not an architecture; REST is a set of constraints that, when applied to the design of a system, creates a software architectural style. If we implement all the REST guidelines outlined in Fielding's work, we end up with a system that has specific roles for data, components, hyperlinks, communication protocols, and data consumers.

Fielding arrived at REST by evaluating all networking resources and technologies available for creating distributed applications. Without any constraints, anything and everything goes, leading us to develop applications that are hard to maintain and extend. With this in mind, he looks at document distributed application architectural styles beginning with what he calls the *null space*—which represents the availability of every technology and every style of application development with no rules or limits—and ends with the following constraints that define a RESTful system:

- It must be a client-server system
- It has to be stateless—there should be no need for the service to keep users' sessions; in other words, each request should be independent of others

- It has to support a caching system—the network infrastructure should support cache at different levels
- It has to be uniformly accessible—each resource must have a unique address and a valid point of access
- It has to be layered—it must support scalability
- It should provide code on demand—although this is an optional constraint, applications can be extendable at runtime by allowing the downloading of code on demand, for example, Java Applets

These constraints don't dictate what kind of technology to use; they only define how data is transferred between components and what are the benefits of following the guidelines. Therefore, a RESTful system can be implemented in any networking architecture available. More important, there is no need for us to invent new technologies or networking protocols: we can use existing networking infrastructures such as the Web to create RESTful architectures. Consequently, a RESTful architecture is one that is maintainable, extendable, and distributed.

Before all REST constraints were formalized, we already had a working example of a RESTful system: the Web. We can ask, then, why introduce these RESTful requirements to web application development when it's agreed that the Web is already RESTful.

We need to first qualify here what it's meant for the Web to be RESTful. On the one hand, the static web is RESTful, because static websites follow Fielding's definition of a RESTful architecture. For instance, the existing web infrastructure provides caching systems, stateless connection, and unique hyperlinks to resources, where resources are all of the documents available on every website and the representations of these documents are already set by files being browser readable (HTML files, for example). Therefore, the static web is a system built on the REST-like architectural style.

On the other hand, traditional dynamic web applications haven't always been RESTful, because they typically break some of the outlined constraints. For instance, most dynamic applications are not stateless, as servers require tracking users through container sessions or client-side cookie schemes. Therefore, we conclude that the dynamic web is not normally built on the REST-like architectural style.

We can now look at the abstractions that make a RESTful system, namely resources, representations, URIs, and the HTTP request types that make up the uniform interface used for client/server data transfers.

Resources

A RESTful resource is anything that is addressable over the Web. By addressable, we mean resources that can be accessed and transferred between clients and servers. Subsequently, a resource is a logical, temporal mapping to a concept in the problem domain for which we are implementing a solution.

These are some examples of REST resources:

- A news story
- The temperature in NY at 4:00 p.m. EST
- A tax return stored in IRS databases
- A list of code revisions history in a repository like SVN or CVS
- A student in some classroom in some school
- A search result for a particular item in a web index, such as Google

Even though a resource's mapping is unique, different requests for a resource can return the same underlying binary representation stored in the server. For example, let's say we have a resource within the context of a publishing system. Then, a request for "the latest revision published" and the request for "revision number 12" will at some point in time return the same representation of the resource: the last revision is revision 12. However, when the latest revision published is increased to version 13, a request to the latest revision will return version 13, and a request for revision 12 will continue returning version 12. As resources in a RESTful architecture, each of these resources can be accessed directly and independently, but different requests could point to the same data.

Because we are using HTTP to communicate, we can transfer any kind of information that can be passed between clients and servers. For example, if we request a text file from CNN, our browser receives a text file. If we request a Flash movie from YouTube, our browser receives a Flash movie. The data is streamed in both cases over TCP/IP and the browser knows how to interpret the binary streams because of the HTTP protocol response header *Content-Type*. Consequently, in a RESTful system, the representation of a resource depends on the caller's desired type (MIME type), which is specified within the communication protocol's request.

Representation

The representation of resources is what is sent back and forth between clients and servers. A representation is a temporal state of the actual data located in some storage device at the time of a request. In general terms, it's a binary stream together with its metadata that describes how the stream is to be consumed by either the client or the server (metadata can also contain extra information about the resource, for example, validation, encryption information, or extra code to be executed at runtime).

Throughout the life of a web service there may be a variety of clients requesting resources. Different clients are able to consume different representations of the same resource. Therefore, a representation can take various forms, such as an image, a text file, or an XML stream or a JSON stream, but has to be available through the same URI.

For human-generated requests through a web browser, a representation is typically in the form of an HTML page. For automated requests from other web services, readability is not as important and a more efficient representation can be used such as XML.

URI

A **Uniform Resource Identifier**, or **URI**, in a RESTful web service is a hyperlink to a resource, and it's the only means for clients and servers to exchange representations.

The set of RESTful constraints don't dictate that URIs must be hyperlinks. We only talk about RESTful URIs being hyperlinks, because we are using the Web to create web services. If we were using a different set of supporting technologies, a RESTful URI would look completely different. However, the core idea of addressability would still remain.

In a RESTful system, the URI is not meant to change over time, as the architecture's implementation is what manages the services, locates the resources, negotiates the representations, and then sends back responses with the requested resources. More important, if we were to change the structure of the storage device at the server level (swapping database servers, for example), our URIs will remain the same and be valid for as long the web service is online or the context of a resource is not changed.

Without REST constraints, resources are accessed by location: typical web addresses are fixed URIs. For instance, if we rename a file on a web server, the URI will be different; if we move a file to a different directory tree in a web server, the URI will change. Note that we could modify our web servers to execute redirects at runtime to maintain addressability, but if we were to do this for every file change, our rules would become unmanageable.

Uniform interfaces through HTTP requests

In previous sections, we introduced the concepts of resources and representations. We said that resources are mappings of actual entity states that are exchanged between clients and servers. Furthermore, we discussed that representations are negotiated between clients and servers through the communication protocol at runtime—through HTTP. In this section, we look in detail at what it means to exchange these representations, and what it means for clients and servers to take actions on these resources.

Developing RESTful web services is similar to what we've been doing up to this point with our web applications. However, the fundamental difference between modern and traditional web application development is how we think of the actions taken on our data abstractions. Specifically, modern development is rooted in the concept of nouns (exchange of resources); legacy development is rooted in the concept of verbs (remote actions taken on data). With the former, we are implementing a RESTful web service; with the latter, we are implementing an **RPC**-like service (**Remote Procedure Call**). What's more, a RESTful service modifies the state of the data through the representation of resources; an RPC service, on the other hand, hides the data representation and instead sends commands to modify the state of the data at the server level (we never know what the data looks like). Finally, in modern web application development we limit design and implementation ambiguity, because we have four specific actions that we can take upon resources—**Create**, **Retrieve**, **Update**, and **Delete** (**CRUD**). On the other hand, in traditional web application development, we can have countless actions with no naming or implementation standards.

Therefore, with the delineated roles for resources and representations, we can now map our CRUD actions to the HTTP methods `POST`, `GET`, `PUT`, and `DELETE` as follows:

Data action	HTTP protocol equivalent
CREATE	`POST`
RETRIEVE	`GET`
UPDATE	`PUT`
DELETE	`DELETE`

In their simplest form, RESTful web services are networked applications that manipulate the state of resources. In this context, resource manipulation means resource creation, retrieval, update, and deletion. However, RESTful web services are not limited to just these four basic data manipulation concepts. On the contrary, RESTful web services can execute logic at the server level, but remembering that every result must be a resource representation of the domain at hand.

A uniform interface brings all the aforementioned abstractions into focus. Consequently, putting together all these concepts we can describe RESTful development with one short sentence: we use URIs to connect clients and servers to exchange resources in the form of representations.

Let's now look at the four HTTP request types in detail and see how each of them is used to exchange representations to modify the state of resources.

GET/RETRIEVE

The method GET is used to RETRIEVE resources.

Before digging into the actual mechanics of the HTTP GET request, first, we need to determine what a resource is in the context of our web service and what type of representation we're exchanging.

For the rest of this section, we'll use the artificial example of a web service handling students in some classroom, with a location of `http://restfuljava.com/`. For this service, we assume an XML representation of a student to look as follows:

```
<student>
    <name>Jane</name>
    <age>10</age>
    <link>/students/Jane</link>
</student>
```

And a list of students to look like:

```
<students>
    <student>
        <name>Jane</name>
        <age>10</age>
        <link>/students/Jane</link>
    </student>
    <student>
        <name>John</name>
        <age>11</age>
```

```
            <link>/students/John</link>
        </student>
        <link>/students</link>
    </students>
```

With our representations defined, we now assume URIs of the form `http://restfuljava.com/students` to access a list of students, and `http://restfuljava.com/students/{name}` to access a specific student that has the unique identifier of value `name`.

We can now begin making requests to our web service. For instance, if we wanted the record for a student with the name *Jane*, we make a request to the URI `http://restfuljava.com/students/Jane`. A representation of `Jane`, at the time of the request, may look like:

```
<student>
    <name>Jane</name>
    <age>10</age>
    <link>/students/Jane</link>
</student>
```

Subsequently, we can access a list of students through the URI `http://restfuljava.com/students`. The response from the service will contain the representation of all students and may look like (assuming there are two students available):

```
<students>
    <student>
        <name>Jane</name>
        <age>10</age>
        <link>/students/Jane</link>
    </student>
    <student>
        <name>John</name>
        <age>11</age>
        <link>/students/John</link>
    </student>
    <link>/students</link>
</students>
```