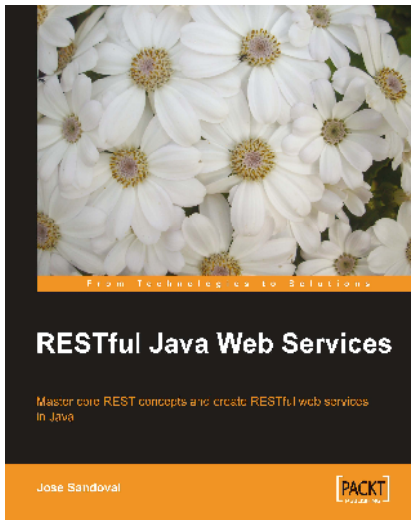
RESTful Java Web Services

Jose Sandoval



Chapter No. 4
"RESTful Web Services Design"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.4 "RESTful Web Services Design"

A synopsis of the book’s content

Information on where to buy this book

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.

RESTful Java Web Services

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.

4
RESTful Web Services Design

The RESTful development process follows traditional development paradigms. However, with RESTful web services, we need to analyze the resource requirements first, design the representation for our resources second, identify the URIs third, and, lastly, worry about implementation technologies.

Throughout this book we've talked about creating web services that are noun dependent as opposed to verb dependent. In this chapter we'll look at what that means in terms of the design process. For this, we'll design a blogging application, but we'll leave the implementation for later chapters. Our sample application is a micro-blogging web service (similar to Twitter), where users create accounts and then post entries.

Finally, while designing our application, we'll define a set of steps that can be applied to designing any software system that needs to be deployed as a RESTful web service.

Designing a RESTful web service

Designing RESTful web services is not different from designing traditional web applications. We still have business requirements, we still have users who want to do things with data, and we still have hardware constraints and software architectures to deal with. The main difference, however, is that we look at the requirements to tease out resources and forget about specific actions to be taken on these resources.

We can think of RESTful web service design as being similar to Object Oriented Design (OOD). In OOD, we try to identify objects from the data we want to represent together with the actions that an object can have. But the similarities end at the data structure definition, because with RESTful web services we already have specific calls that are part of the protocol itself.

The underlying RESTful web service design principles can be summarized in the following four steps:

1. **Requirements gathering**—this step is similar to traditional software requirement gathering practices.
2. **Resource identification**—this step is similar to OOD where we identify objects, but we don't worry about messaging between objects.
3. **Resource representation definition**—because we exchange representation between clients and servers, we should define what kind of representation we need to use. Typically, we use XML, but JSON has gained popularity. That's not to say that we can't use any other form of resource representation—on the contrary, we could use XHTML or any other form of binary representation, though we let the requirements guide our choices.
4. **URI definition**—with resources in place, we need to define the API, which consists of URIs for clients and servers to exchange resources' representations.

This design process is not static. These are iterative steps that gravitate around resources. Let's say that during the *URI definition* step we discover that one of the URI's responses is not covered in one of the resources we have identified. Then we go back to define a suitable resource. In most cases, however, we find that the resources that we already have cover most of our needs, and we just have to combine existing resources into a meta-resource to take care of the new requirement.

Requirements of sample web service

The RESTful web service we design in this chapter is a social networking web application similar to Twitter.

Throughout this book, we follow an OOD process mixed with an agile philosophy for designing and coding our applications. This means that we create just enough documentation to be useful, but not so much that we spend an inordinate amount of time deciphering it during our implementation phase.

As with any application, we begin by listing the main business requirements, for which we have the following use cases (these are the main functions of our application):

- A web user creates an account with a username and a password (creating an account means that the user is now registered).
- Registered users post blog entries to their accounts. We limit messages to 140 characters.
- Registered and non-registered users view all blog entries.
- Registered and non-registered users view user profiles.
- Registered users update their user profiles, for example, users update their password.
- Registered and non-registered users search for terms in all blog entries.

However simple this example may be, social networking sites work on these same principles: users sign up for accounts to post personal updates or information. Our intention here, though, is not to fully replicate Twitter or to fully create a social networking application. What we are trying to outline is a set of requirements that will test our understanding of RESTful web services design and implementation.

The core value of social networking sites lies in the ability to connect to multiple users who connect with us, and the value is derived from what the connections mean within the community, because of the tendency of users following people with similar interests. For example, the connections between users create targeted distribution networks.

The connections between users create random graphs in the *graph theory* sense, where nodes are users and edges are connections between users. This is what is referred to as the *social graph*.

Resource identification

Out of the use cases listed above, we now need to define the service's resources. From reading the requirements we see that we need users and messages. Users appear in two ways: a single user and a list of users. Additionally, users have the ability to post blog entries in the form of messages of no more than 140 characters. This means that we need resources for a single message and a list of messages. In sum, we identify the following resources:

- User
- List of users
- Message
- List of messages

Representation definition

As we discussed in our introduction to RESTful web services, a representation is a temporal mapping of a resource at the time of a request. Furthermore, a representation is transmitted between clients and servers over HTTP. Because of HTTP's flexibility, any binary stream can be transferred. Nevertheless, we don't recommend choosing just any type of binary representation that requires special code or libraries to consume. We recommend using primarily XML and JSON structures, remembering, of course, that the requirements of the problem we're solving dictate what representation types we must provide.

A well-designed RESTful web service needs to provide multiple resource representations. We can't assume that only web browsers will be accessing our public APIs or that only the one type of client we identified in our *requirement gathering* process will use our services.

What are the options available, then?

Again, arriving at the ideal representation format is a matter of the design process. We need to take into account what the service is doing and what clients will be using the resources for. The safest representation format is therefore XML. This is what web services are known for: transferring XML streams over web transport protocols. More important, most programming languages already have libraries available to parse XML streams.

Finally, we need to account for linkability of representations. Linkability of representation means that the web services provide for resource discoverability, such that resources link to other resources (what's currently being referred to as **HATEOS** or **Hypermedia As The Engine Of State transfer**). For example, our URI for a list of users returns a structure of users with each element in the list having a direct URI to each element in the service (a link to a user).

XML representations

From our analysis, we identified two types of resources: users and messages. As part of the heuristics we outlined earlier, we need to define what our representation will look like. The following representations are the structures that we will have to implement when we actually code the web service.

Users

We first define a user representation as follows:

```
<user>
  <username></username>
  <password></password>
  <link></link>
</user>
```

As part of a user resource, we store only a username and a password. The username is unique in the context of our system and is used to uniquely identify each user. The `link` element, which points back to the web service, is either assigned when the resource is created or a representation is built for transport (for our sample service, we let the backend create the link).

We now define a list of users as follows:

```
<users>
<count></count>
<link></link>
    <user>
        <username></username>
        <password></password>
        <link></link>
    </user>
    ...
    <user>
        <username></username>
        <password></password>
        <link></link>
    </user>
</users>
```

This XML structure declares a list of users stored in an XML element `<users>`. We use ellipses or . . . to show that we can have more than one user in the list.

We can see here the linkability concept at play: with a list of users we can drill down to individual users using the `link` element's value.

Messages

We first define a single blog entry or message as follows:

```
<message>
    <messageID></messageID>
    <content></content>
    <link></link>
    <user>
        <username></username>
        <password></password>
        <link></link>
    </user>
</message>
```

A message needs a message id, the body of the message (the content element), and the user who posted the message. Note that depending on what we are doing with the message, we don't pass all the resource's information back and forth. For example, when we are creating a message at the client layer, we don't know what the value for `messageID` is. Therefore, we still need to pass the message structure to the service, but our web service will know that any `messageID` value needs to be ignored, because, in our case, it will be created by the storage layer.

Finally, we define a list of messages as follows:

```
<messages>
    <count></count>
    <link></link>
    <message>
        <messageID></messageID>
        <content></content>
        <link></link>
        <user>
            <username></username>
            <password></password>
            <link></link>
        </user>
    </message>
    ...
    <message>
        <messageID></messageID>
        <content></content>
        <link></link>
        <user>
            <username></username>
            <password></password>
            <link></link>
        </user>
    </message>
</messages>
```

This XML structure holds a collection of messages, and each message holds the user who posted the message.

We use the XML representation type for input and output. Input in this case means that we send a resource representation to create and update the resources at the web service layer in the form of an XML object. Output means that a client requests an XML representation of a resource.

JSON representations

We use the same key names for our JSON representation, and we still have only two types of resources: users and messages. Again, these structures are our specification of what we need to return for each request.

Users

We define a user representation as follows:

```
{"user":{"username":"john", "password":"password", "link":"/users/
john"}}
```

And we define a list of users as follows (we use the `...` characters to show that there is more than one user in the array):

```
{"users-result":{"count":"6", "users":[{"username":"john",
"password":"password", "link":"/users/john"}, ...,{"username":"jane",
"password":"password", "link":"/users/jane"}]}}
```

The array for all users as a JSON structure, looks as follows:

```
"users":[{"username":"john", "password":"password", "link":"/users/
john"}, ...,{"username":"jane", "password":"password", "link":"/users/
jane"}]
```

Once the JSON response has been evaluated with the JavaScript `eval()` function, similar to what we did in Chapters 2 and 3, we can then access any of the values in the structure. For example, if we need the user name of the first element on the array, we use `users-result.users[0].username`.

Messages

We now define a message representation as follows:

```
{"message":{"messageID":"some-id", "content":"some content",
"link":"/messages/some-id", "user":{"user":{"username":"john",
"password":"password", "link":"/users/john"}}}
```

And a list of messages as follows:

```
{"messages-result":{"count":"6", "link":"/messages",
"messages":[{"messageID":"some-id", "content":"some content",
"link":"/messages/some-id",  "user":{"username":"john",
"password":"password", "link":"/users/john"}}, ...,{"messageID":"some-
id2", "content":"some content", "link":"/messages/some-id2",
"user":{"username":"jane", "password":"password", "link":"/users/
jane"}}]}}
```

Each message element in the array has a user structure embedded as follows:

```
{"messageID":"some-id", "content":"some content",
"user":{"username":"john", "password":"password", "link":"/users/
john"}}
```

Once the JSON response has been evaluated with the JavaScript `eval()` function, we can then access the first element on the list with `messages-result.messages[0].content`; if we want to get the user name of the user who posted the message, we access the value with `messages-result.messages[0].user.username`.

Note that during implementation we'll use JSON representations only as response streams. This means that we won't use JSON structures to create or update resources at the web service layer. Although we could use XML and JSON to update resources at the service layer, we'll omit JSON for the sake of brevity.

URI definition

The next step involves the definition of URIs. This is a crucial step, as the URIs define our API and it's likely that we want to make our web service public.

We strive to make our APIs logical, hierarchical, and as permanent as we can. We must emphasize these three tenets, as we may have many developers depending on the services we make available. Therefore, a good API is one that doesn't change too often and is unambiguous to use. Furthermore, the idea of RESTful APIs is that we maintain URI uniqueness and reliability of service. (For a complete discussion about this topic, see `http://www.w3.org/Provider/Style/URI`.)

The first thing we need is a web address. In our case, we assume that we're using our development machine running on `http://localhost:8080/`.

It's important to distinguish between a web service and a web application: we use web services to implement web applications, and web services can be, and are recommended to be, independent of web applications. What's more, web applications are meant to be consumed by humans, as opposed to web services that are intended for machine consumption.

For example, a RESTful API could live under `http://api.restfuljava.com/` and a web application using the API could live under `http://restfuljava.com/`. Both the API and the web application should be running on independent hardware for performance reasons, but when we get to implement our sample service we run everything on the same server.

The nomenclature of RESTful URIs falls under the topic of URI templates and the following conventions are widely used. First, for items or identifiers that don't change, we find the keyword to be part of the actual URI—for instance, we use `users` to be the URI for a list of all users. Second, we use keys or dynamic keywords to be enclosed in { and }. Applying these conventions, our URI list for users looks as follows:

- `http://localhost:8080/users`—with the `GET` method, this URI returns a list of all users; with the `POST` method, we create a new user and the payload is a user's XML representation; we don't support the `PUT` method; finally, we don't support the `DELETE` method for an entire list of users
- `http://localhost:8080/users/{username}`—with the `GET` method, this URI returns a representation of a user with a unique identifier `username`; with the `PUT` method, it updates a user; and, with the `DELETE` method, it deletes a user.

And for messages, our URI list looks as follows:

- `http://localhost:8080/messages`—with the `GET` method, this URI returns a list of all messages from all users; with the `POST` method, it creates a new message, with the message's XML representation as the payload of the request
- `http://localhost:8080/messages/{messageID}`—with the `GET` method, this URI returns a representation for a message with the unique identifier `messageID`; with the `DELETE` method, it deletes a message; and we don't support the `POST` or `PUT` methods
- `http://localhost:8080/messages/users/{username}`—with the `GET` method, this URI returns a list of all message for a user with identifier `username`; no `POST`, `PUT`, or `DELETE` methods are supported

At the time of this writing, the URI Template specification is still under review. For more information, see `http://tools.ietf.org/html/draft-gregorio-uritemplate-03`.

Executing logic with RESTful URIs

A question arises when designing RESTful web services that has to do with executing code at the server level. Specifically, how do we execute logic if we limit our client/server interactions to only four CRUD-like calls (`POST`, `GET`, `PUT`, and `DELETE`)? For this we need to introduce URIs that execute logic on the server, remembering that responses must be in the form of resource representations. In other words, we avoid any RPC style calls and concentrate on the resources only.

For our web service, we only offer the ability of searching for a term or a phrase in the blog entries. For example, any user can search for the term "programming" or "software development" using the following URI (note that the URI pattern is arbitrary and you can choose whatever makes sense for the service you are developing):

```
http://localhost:8080/messages/search/{search_item}
```

This URI returns a list of messages that contain the word or words `search_item`—this is strictly a `GET` method call and no `POST`, `PUT`, or `DELETE` method is supported

Using URIs to request representation types

A RESTful web service is one that adheres to all the constraints we outlined in Chapter 1, *RESTful Architectures*. However, we have encountered APIs that don't strictly adhere to every constraint. For example, requesting representation types via URIs is something we saw with Twitter's API. We requested three different types of representations with the URI `http://twitter.com/statuses/public_timeline.{xml, json, rss}`.

We said that this is not technically a RESTful web service request, because we don't use the communication protocol—HTTP headers—to tell the service what kind of representation to get. Even though this is not a RESTful web service, it still works. Nevertheless, the API may be open to interpretation. For example, what does it mean to send an HTTP GET request to the URI `http://twitter.com/statuses/public_timeline.json` with an HTTP *Accept* header value of `application/xml`? Do we get a JSON representation or an XML representation? A properly designed RESTful web service has to adhere to *all* REST constraints, and using the protocol to negotiate representations is part of being RESTful.

Creating a properly defined RESTful web service, however, ensures that there are no misunderstandings on how to use such services. For example, setting the HTTP method type to GET with an appropriate *Accept* header value makes it clear that we are requesting a resource of a specific type. In the end, you, as a developer or software architect, need to make a decision as to which style will benefit your users the most.

Using this representation request style is a design decision that facilitates—it can be argued—the writing of clients. For instance, the majority of public APIs are read only, so using a straight HTTP GET request with the type of representation embedded in the URI is easier than instantiating a full HTTP request and modifying the HTTP header *Accept* each time. Note that neither is hard to implement; however, with the former, we save a couple of lines of code at the client layer. Again, it's a matter of choice.

Our sample web service is a canonical application, thus we don't deviate from any of the constraints. This means that we use the HTTP protocol to request a preferred resource representation and not the URI style described in this section.

Summary

As we've seen, the RESTful design process is resource centric. In addition, we have no arbitrary actions executing on the data—we have HTTP method calls that exchange representations using clearly defined URIs.

The steps to arrive at a web service that adheres to all REST constraints are similar to traditional web application design. So we can still use all our traditional requirement gathering techniques, but tweak the process to account for proper design of usable URIs and consumable representations.

Now that we have our social networking web service specification defined together with a set of RESTful design principles, the next step is implementation. We have four different REST frameworks in the menu, so in the next chapter we begin with Jersey, which is the reference implementation of the Java API for RESTful Web Services Specification, more commonly known as JAX-RS.

Where to buy this book

You can buy RESTful Java Web Services from the Packt Publishing website:
`http://www.packtpub.com/restful-java-web-services/book`

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com