Community Experience Distilled

Building a RESTful Web Service with Spring

A hands-on guide to building an enterprise-grade, scalable RESTful web service using the Spring Framework

Ludovic Dewailly

[PACKT] PUBLISHING open source* community experience distilled

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 **'The First Endpoint'**
- A synopsis of the book's content
- More information on **Building a RESTful Web Service with Spring**

About the Author

Ludovic Dewailly is a senior, hands-on software engineer and development manager with over 12 years of experience in designing and building software solutions on platforms ranging from resource-constrained mobile devices to cloud-computing systems. He is currently helping FancyGiving.com (a social shopping, wishing, and gifting platform) design and build their system. Ludovic's interests lie in software architecture and tackling the challenges of web scale.

Preface

In today's connected world, APIs have taken a central role on the Web. They provide the fabric on which systems interact with each other. And REST has become synonymous with APIs. REpresentational State Transfer, or REST, is an architectural style that lends itself well to tackling the challenges of building scalable web services.

In the Java ecosystem, the Spring Framework is the application framework of choice. It provides a comprehensive programming and configuration model that takes away the "plumbing" of enterprise applications.

It will, therefore, come as no surprise that Spring provides an ideal framework for building RESTful web services. In this book, we will take a hands-on look at how to build an enterprise-grade RESTful web service with the Spring Framework. As an underlying theme, we will illustrate the concepts in each chapter with the implementation of a sample web service that deals with managing rooms in a hotel.

By the end of this book, readers will be equipped with the necessary techniques to create a RESTful web service and sufficient knowledge to scale and secure their web service to meet production readiness requirements.

What this book covers

Chapter 1, *A Few Basics*, discusses the REST architecture approach and its underlying principles.

Chapter 2, *Let's Get Started*, enables us to put the scaffolding together, before building a RESTful web service.

Chapter 3, *Building RESTful Web Services with Maven and Gradle*, looks at the building blocks of creating RESTful endpoints.

Chapter 4, *Data Representation*, discusses how to manage data representation before we proceed further with building more endpoints. This chapter also offers advice on creating common response formats and error handling.

Chapter 5, *CRUD Operations in REST*, expands on the previous chapters and takes a look at how you can map CRUD operations to RESTful endpoints.

Chapter 6, *Performance*, explains that for a web service to be production-ready, it needs to be performant. This chapter discusses performance optimization techniques.

Chapter 7, *Dealing with Security*, looks at how to ensure a web service is secure by delving into steps that designers need to take. This chapter looks at how to deal with authentication and authorization, as well as input validation techniques.

Chapter 8, *Testing Restful Web Services*, looks at how to guarantee that a web service delivers the expected functionality, and the testing strategies that designers need to consider. This chapter offers readers the approaches for creating comprehensive test plans.

Chapter 9, *Building a REST Client*, tells us how for a web service to be of any use, it must be consumed. This penultimate chapter focuses on how to build a client for RESTful web services.

Chapter 10, *Scaling a Restful Web Service*, explains that scalability is a vast topic and encompasses many aspects. In this last chapter, we discuss what API designers can put in place to help the scaling of their service.

3
The First Endpoint

In the previous chapter, we looked at how to set up the scaffolding required to build a RESTful web service. We can now start implementing our first endpoint. As introduced in *Chapter 1*, *A Few Basics*, Spring Web MVC provides the necessary tools to create controllers in a RESTful manner. To illustrate their use, we will lean on our sample web service. More specifically, we will start implementing the `Inventory` component of our service.

This chapter will go over the following topics:

- The `Inventory` component of our sample web service
- Spring constructs to build RESTful endpoints
- Running our first endpoint
- A brief discussion on data presentation

The Inventory service

At the heart of our property management service lies rooms that are the representations of physical rooms that guests can reserve. They are organized in categories. A room category is a logical grouping of similar rooms. For example, we could have a *Double Rooms* category for all rooms with double beds. Rooms exhibit properties as per the code snippet that follows:

```
@Entity(name = "rooms")
public class Room {

  private long id;
  private RoomCategory roomCategory;
  private String name;
  private String description;
```

```
    @Id
    @GeneratedValue
    public long getId() {
      return id;
    }

    @ManyToOne(cascade = {CascadeType.PERSIST,
      CascadeType.REFRESH}, fetch = FetchType.EAGER)
    public RoomCategory getRoomCategory() {
      return roomCategory;
    }

    @Column(name = "name", unique = true, nullable = false,
      length = 128)
    public String getName() {
      return name;
    }

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

With the use of the **Java Persistence API** (**JPA**) and Hibernate, rooms have an auto-generated identifier (thanks to the `@javax.persistence.Id` and `@javax.persistence.GeneratedValue` annotations): a unique and mandatory name along with an optional description.

JPA is not in the scope of this book. Information on it can be found at `http://en.wikipedia.org/wiki/Java_Persistence_API`

Additionally, a one-to-many relationship is established between categories and rooms with the use of the `@javax.persistence.ManyToOne` annotation.

The `Inventory` service provides access to rooms and their categories. Several operations are made available by this component. However, we are only interested in one of the operations in this chapter. Therefore, let's consider the following Java interface:

```
public interface InventoryService {

  public Room getRoom(long roomId);

  // other methods omitted for clarity
}
```

This service interface gives us the ability to look up a room by its identifier. Assuming that we have an implementation for this interface, the next step is to expose this operation through our RESTful interface. The following section describes how to go about doing just that.

REST and the MVC pattern

The Spring Web MVC module provides an implementation of the traditional **Model View Controller** pattern. While REST does not mandate the use of any specific pattern, using the MVC pattern is quite a natural fit whereby the RESTful resource or model is exposed through a controller. The view in our case will be a JSON representation of the model.

Without further ado, let's take a look at our first endpoint:

```
@RestController
@RequestMapping("/rooms")
public class RoomsResource {

  private final InventoryService inventoryService;

  public RoomsResource(InventoryService inventoryService) {
    this.inventoryService = inventoryService;
  }

  @RequestMapping(value = "/{roomId}", method = RequestMethod.GET)
  public RoomDTO getRoom(@PathVariable("roomId") String roomId) {
    RoomDTO room = ...
    // omitted for sake of clarity
    return room;
  }
}
```

With the use of `@org.springframework.web.bind.annotation.RestController`, we instruct Spring that `RoomsResource` is a controller.

Traditionally, one would expect this controller class to be called `RoomController`. However, in the RESTful architectural style, the core concept revolves around resources. Therefore, using the *Resource* suffix embodies the REST principles more appropriately.

The other annotation of note in this code is `@org.springframework.web.bind.annotation.RequestMapping`. This is discussed in the next section.

The **Data Transfer Object** (**DTO**) pattern is referenced here (`RoomDTO`), but we will look at it in more detail in *Chapter 4*, *Data Representation*. It provides a useful decoupling between the persistence and presentation layers.

Request mapping

The `@org.springframework.web.bind.annotation.RequestMapping` annotation provides the glue for mapping incoming requests to classes and methods within the code.

Path mapping

Typically, at the class level, the `@org.springframework.web.bind.annotation.RequestMapping` annotation allows routing requests for a specific path to a resource (or controller) class. For example, in the previous code extract, the `RoomsResource` class declares the top-level request mapping, `@RequestMapping("/rooms")`. With this annotation, the class will handle all requests to the path, `rooms`.

HTTP method mapping

It is possible to map specific HTTP methods to classes or Java methods. Our `RoomsResource` class exposes a method for retrieving a `Room` by identifier. It is declared as follows:

```
@RequestMapping(value = "/{roomId}", method = RequestMethod.GET)
```

In combination with the class-level annotation, `GET` requests with the following `/rooms/{roomId}` value will be mapped to `RoomsResource.getRoom()`.

Requests with any other methods (say, `PUT` or `DELETE`) will not be mapped to this Java method.

Request parameter mapping

Besides path parameters, request parameters can also be referenced in a `RestController` class. The `@org.springframework.web.bind.annotation.RequestParam` parameter provides the means to map an HTTP query parameter to a Java method attribute. To illustrate parameter mapping, let's add a new lookup method to our resource:

```
@RequestMapping(method = RequestMethod.GET)
public List<RoomDTO> getRoomsInCategory(@RequestParam(
  "categoryId") long categoryId) {
  RoomCategory category = inventoryService.getRoomCategory(
    categoryId);
  return inventoryService.getAllRoomsWithCategory(category)
  .stream().map(RoomDTO::new).collect(Collectors.toList());
}
```

Requests to URLs, such as `http://localhost:8080/rooms?categoryId=1`, will be handled by this method, and the `categoryId` method attribute will be set to `1`. The method will return a list of rooms for the given category in the JSON format, as follows:

```
[
  {
    "id": 1,
    "name": "Room 1",
    "roomCategoryId": 1,
    "description": "Nice, spacious double bed room with usual
      amenities"
    }
]
```

Service designers could also declare this endpoint without using a parameter. For example, the URL pattern could be `/rooms/categories/{categoryId}`. This approach has the added benefit of improving caching, since not all browsers and proxies cache query parameters. Refer to *Chapter 6*, *Performance*, for more details on caching techniques.

As we continue implementing our property management system in the rest of the book, we will put into practice these mapping constructs. *Chapter 5*, *CRUD Operations in REST*, will especially highlight how to map CRUD (Create, Read, Update, and Delete) operations in REST with these annotations.

The preceding code snippet makes use of Java's new Stream API, which is some functional programming goodness introduced in Java 8. Read more about it at `https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html`.

Running the service

Building upon *Chapter 2*, *Building RESTful Web Services with Maven and Gradle*, we can quickly get our service up and running by using Spring Boot. For this purpose, we need to create a main class, as follows:

```
package com.packtpub.springrest.inventory;
// imports omitted
@SpringBootApplication
public class WebApplication {

  public static void main(String[] args) {
    SpringApplication.run(new Object[]{WebApplication.class,
      "inventory.xml"}, args);
    InventoryService inventoryService = context.getBean(
      InventoryService.class);
  }
}
```

Running this class in your favorite IDE will start an embedded **Tomcat** instance and expose your resources. The service will be accessible at `http://localhost:8080`. For example, accessing `http://localhost:8080/rooms/1` will return the following:

```
{
  "id": 1,
  "name": "Room 1",
  "roomCategoryId": 1,
  "description": "Nice, spacious double bed room with usual
    amenities"
}
```

We depart from *Chapter 2*, *Building RESTful Web Services with Maven and Gradle*, in this example by mixing auto detection and XML-based strategies. By passing both the application class and the name of our XML Spring wiring (`inventory.xml`), Spring Boot will load all the annotated beans it can find, along with the ones declared in XML.

A few words on data representation

Without us doing anything, our DTO objects are magically converted to JSON by Spring. This is thanks to Spring's support for data binding with **Jackson**. The Jackson JSON processor provides a fast and lightweight library for mapping Java objects to JSON objects.

Summary

In this chapter, we discussed the `Inventory` component of our sample RESTful web service and worked on our first endpoint. We explored how to quickly run and access this endpoint, and we even threw in a bonus endpoint to list rooms by categories.

In the next chapter, we will explore in more detail how to control the JSON format of responses, along with the DTO pattern and why it is useful for decoupling the different layers of a system.

Where to buy this book