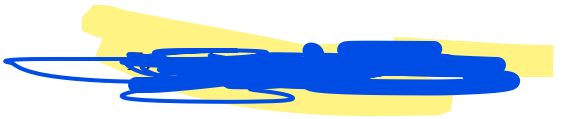

- Spring Events and JMS with ActiveMQ and WebSockets

CHAPTER 1

Messaging

Communication is a concept that has been around forever. Everything needs to communicate by exchanging some kind of information, and yes, you read that right, I mean *everything*. If you think about it, even the ads that you find on the side of a bus or in the grocery store are trying to tell or sell you something, right?

In the computer world, devices (mouse, monitor, keyboard, etc.) communicate with each other by exchanging bits of information. If we consider applications, then we are talking about components, functions, classes, etc., that need to communicate with each other or expose some functionality that can be reached through communication. We typically call this *messaging*.

Messaging

This section introduces some of the main concepts described in detail later, starting with messaging and how it fits in the daily development cycle. Take a look at Figure 1-1.

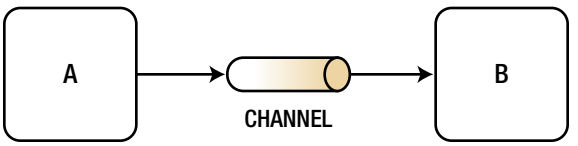

Figure 1-1. *Messaging*

Figure 1-1 shows a simple messaging process, whereby it communicates from point A to point B through whatever media is possible. In this case, Figure 1-1 uses a channel, which could be a simple function call, a socket connection, or an HTTP request. The main idea is to produce/send a message to the consumer/receiver.

Messaging Use Cases

This section covers some of the most common messaging use cases. These uses cases are key to understanding why messaging is so important:

- *Delivery Guaranteed*
    - Developers need to make sure that the message they are sending reaches its destination. If you are using a *broker* (a system that handles connections, messages, and message delivery, synched or asynched),

F. Gutierrez, *Spring Boot Messaging*, DOI 10.1007/978-1-4842-1224-0_1

then the producer needs to know if the broker receives the message, by some kind of acknowledgement. The consumer must do the same, by acknowledging to the broker that the message was received. So this particular use case is commonly used when critical messages are being delivered, such as payments, stocks, or any other important information.

- *Decoupled*
  - When dealing with software architecture, developers look for decoupled components for easy integration, extensibility, and simple operation and maintenance. But how can you achieve decoupling? Messaging is part of the solution, because it allows you to think in your own business domain, which is a bounded context. The information you produce/send is your main concern, regardless of how the consumer/receiver will implement its own business logic.
- *Scalable and High Available*
  - Every time a system experiences high-request demands, it needs to be scalable and not have a single point of failure. For these particular scenarios, messaging is the solution, because multiple consumers/receivers can keep up with the load and you can replicate the messages across multiple system instances or brokers. That way, if one of your instances/brokers is down, you are still in control.
- *Asynchronous*
  - Applications must be very quick and be able to respond to a request as soon as possible. In this case, time is the key factor. How can you achieve this kind of speed when you know that processing the request will take a lot of time and you have multiple clients? You can solve this issue by using the previous use case solution (scalability and/or high availability), but it will reach a point where it's blocking requests. The solution is asynchronous messaging—a fire and forget—where your producer/sender sends messages and gets to their own business logic, leaving the consumers/receivers to process the message on their own time.
- *Interoperability*
  - An important factor when creating messaging systems is the ability to produce/send a message and be able to understand that message when consuming/receiving (maybe it's a plain/text JSON or in XML format or a serialized object). There have been many attempts over the past decades to create interoperable systems. Nowadays, interoperability is possible with new brokers like the one that implements the AMQP protocol or even with simple RESTful APIs or WebSockets that, regardless of the implementation, enable the producer and consumer to interoperate seamlessly.

These use cases have been evolving into very well known messaging models and design patterns, which are discussed in the next section.

Messaging Models and Messaging Patterns

Some messaging models were established when messaging became a part of all systems. These models, in my opinion, evolved into the creation of messaging design patterns.

Point-to-Point

A point-to-point model is a way to send a message to a queue (First In, First Out), whereby only one receiver gets the message. See Figure 1-2.

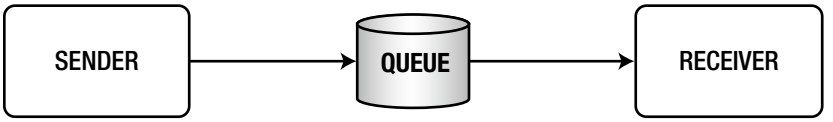

Figure 1-2. *Point-to-point model*

The point-to-point model is also used as a message channel pattern, whereby instead of a queue, you have a channel (which is a way to transport your message). It still guarantees that only one receiver gets the message in the order it was sent.

You'll see some examples later in the book about these models and patterns.

Publish-Subscribe

The publish-subscribe model describes a publisher that sends a message (a topic) to multiple subscribers of that topic. Each subscriber will receive only one copy of the message. See Figure 1-3.

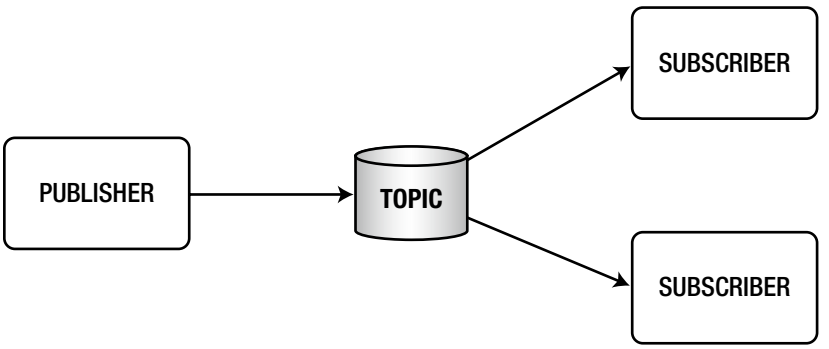

Figure 1-3. *Publish–subscribe model*

The publish-subscribe model is also used as a message channel pattern, whereby instead of a topic, you have a channel that delivers a copy of the message to its subscribers.

These models are more related to the JMS (Java Messaging System) and are important to understand because they form the basics of all enterprise systems.

Messaging Patterns

A design pattern is a solution to a commonly known problem in the software design. By the same token, messaging patterns attempt to solve problems with messaging designs.

You will learn about the implementation of the following patterns during the course of this book, so I want to list them here with simple definitions to introduce them:

- *Message type patterns*: Describe different forms of messaging, such as string (maybe plain text, JSON and/or XML), byte array, object, etc.
- *Message channel patterns*: Determine what kind of a transport (channel) will be used to send a message and what kind of attributes it will have. The idea here is that the producer and consumer know how to connect to the transport (channel) and can send and receive the message. Possible attributes of this transport include a request-reply feature and a unidirectional channel, which you will learn about very soon. One example of this pattern is the point-to-point channel.
- *Routing patterns*: Describe a way to send messages between producer and consumers by providing a routing mechanism (filtering that's dependent on a set of conditions) in an integrated solution. That can be accomplished by programming, or in some cases, the messaging system (the broker) can have these capabilities (as with RabbitMQ).
- *Service consumer patterns*: Describe how the consumers will behave when messages arrive, such as adding a transactional approach when processing the message. There are frameworks that allow you to initiate this kind of behavior (like the Spring Framework, which you do by adding the `@Transactional`, a transaction-based abstraction).
- *Contract patterns*: Contracts between the producer and consumer to have simple communications, such as when you do some REST calls, where you call a JSON or XML message with some fields.
- *Message construction patterns*: Describe how a message is created so it can travel within the messaging system. For example, you can create an "envelope" that can have a body (the actual message) and some headers (with a correlation ID or a sequence or maybe a reply address). With a simple web request, you can add parameters or headers and the actual message becomes the body of the request, making the whole request part of the construction pattern. The HTTP protocol allows for that kind of communication (messaging).
- *Transformation patterns*: Describe how to change the content of the message within the messaging system. Think about a message that requires some processing and needs to be enhanced on the fly, such as a content enricher.

As you can see, these patterns not only describe the messaging process but some of them describe how to handle some of the common use cases you saw earlier. Of course, there are a lot more messaging patterns, and these are just a few that we are going to explore in this book.

If you want more information, I recommend that you visit the Enterprise Integration Patterns web site, http://www.enterpriseintegrationpatterns.com/. Also check out this must-read book—*Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* by Gregor Hohpe and Bobby Woolf, from Addison-Wesley.

In the book, I cover some of these patterns using the Spring Integration module and various messaging systems (brokers).

Messaging with Spring Framework

The Spring Framework is one of the most commonly used frameworks by the Java community. The Spring Framework has enabled a simple and easy way to do messaging, by implementing a template design pattern that can be used with any messaging system. It supports the JMS API with the JmsTemplate, the AMQP with the RabbitTemplate, STOMP, and internal messaging with events and listeners. Don't worry, we will cover all of these later in book.

It's worth mentioning that the Spring Integration project has been well accepted by the community, and the core of the Spring Integration (messaging, channels, and other interfaces) has been part of the Spring Core since version 4.0.

There are many good messaging frameworks out there. Some of them use pure Java, and others depend on the Spring Framework. The Spring team has been working very hard to make it even easier for its developers to use all the features, including concurrency, transactions, retries, etc. To use other libraries, you have to implement them by hand using the same logic.

■ **Note** You can download the book's source code from the Apress site or you can clone it from this GitHub repository: https://github.com/felipeg48/spring-messaging.

Summary

This chapter introduced messaging and the messaging systems. You read about the basic concept of messaging, whereby developers need to send information from one point to another.

The chapter reviewed some use cases, models, and patterns of messaging. The book will cover them in more detail and use them via some of the best implementations out there, including the Spring Framework and some of their modules, including Spring Integration, Spring AMQP, and Spring Cloud Stream.

The next chapter includes a tour of Spring Boot, which is the next generation of creating enterprise-ready Spring applications. Spring Boot is the base for all the modules you use in this book.

CHAPTER 2

Spring Boot

I'll start with a phrase that I wrote in another book: *Spring Boot is a new chapter in creating enterprise-ready applications with the Spring Framework.*

Spring Boot does not replace the Spring Framework; you can see it as a new way to create awesome applications with the framework used by the Java community.

In this chapter, I show you what Spring Boot is and how it works behind the scenes. You will also see the power of Spring Boot with a small example. So why do we need to look at Spring Boot? First of all, all the technologies that we are going to use, such as Spring Integration, Spring AMQP, Spring Cloud Stream, and others, use Spring Boot as their base. All the examples in this book use Spring Boot to create Spring apps. And of course, Spring Boot makes messaging even easier.

What Is Spring Boot?

To begin with, Spring Boot is an *opinionated* technology. What does that mean?

Spring Boot looks at the classpath and tries to determine what kind of application you are trying to run. For example, if you have the `spring-mvc` modules in your classpath, Spring Boot will wire up the `WebApplicationInitializer` and `DispatcherServlet` classes inside the Spring container and will set up an embedded container (Tomcat by default), so you can run your application without having to copy or deploy your application to a servlet container.

Spring Boot's Features

Let's look at the most important Spring Boot features:

- It can create standalone Spring applications. Based on its Maven or Gradle plugin, you can create executable JARs or WARs.
- It has an opinionated technology based on the *starter* `poms`.
- Includes auto-configuration, which will configure your Spring application without any XML or Java config classes.
- Includes embedded servlet containers for web apps (Tomcat, Jetty, or Undertow).
- Includes production-ready features (non-functional requirements) that are ready to use, such as metrics and health checks.

F. Gutierrez, *Spring Boot Messaging*, DOI 10.1007/978-1-4842-1224-0_2

- Spring Boot *is not* a plugin or a code generator (this means that Spring Boot doesn't create a file to be compiled).
- If you have an application or servlet container, you can deploy Spring Boot as a WAR without making any changes.
- You can access all the Spring application events and listeners (something that we will discuss in the next chapter).

This list is just a few features of what Spring Boot can offer; of course there are more. I recommend my other book—*Pro Spring Boot* from Apress—if you need to know more about it. That book includes more comprehensive and detailed sections about how Spring Boot works internally.

Restful API with Spring Boot

One of the important features of Spring Boot is that you can create executable JARs and run them by just executing `java -jar yourapp.jar`. You can also create executable WARs (web archive) that can run standalone (using the embedded container within the WAR) or deploy them to a servlet container without chaining anything in your code.

This section shows you a Restful API that lists the currency and the rates based on the country code. This project is a Spring Boot web application.

■ **Note** You can find a download link for all the source code at the http://www.apress.com/9781484212257.

I'll show you some code snippets of the application, so you have an idea of what Spring Boot can do with very minimal code and no configuration files.

I'm assuming that you already have the code. I suggest that you use the STS (Spring Tool Suite), which you can get from https://spring.io/tools/sts/all. I use this IDE because it has very nice features for running Spring Boot, and all the figures that you see are based on this IDE, but you can choose any IDE you like.

The rest-api-demo Project

This project is a Spring Boot web application that will expose Restful endpoints that will show a country currency and the rates of other countries. This project is in the folder named `ch02`.

This project also uses the JPA (Java Persistence API) with an in-memory database (using the H2 engine), AOP (Aspect-Oriented Programming). The exposed endpoints are listed in Table 2-1.

Table 2-1. *Restful Endpoints*

Method	Path	Description
GET	/currency/latest[?base=<code>]	Shows the latest rates in JSON format. USD is the default base.
GET	/currency/{date}[?base=<code>]	Shows the rates based on the date: yyyy-MM-dd in JSON format.
GET	/currency/{amount}/{base}/to/{code}	Shows a conversion based on the amount, base, and code.
POST	/currency/new	Adds new rates by date. The body should be in JSON format.

Listing 2-1 shows an example of the JSON response. You can use a `base` parameter for some of the endpoints as well.

Listing 2-1. JSON Currency Response - /currency/latest

```
{
    "base": "USD",
    "date": "2016-09-22",
    "rates": [
        {
            "code": "EUR",
            "rate": 0.88857
        },
        {
            "code": "JPY",
            "rate": 102.17
        },
        {
            "code": "MXN",
            "rate": 19.232
        },
        {
            "code": "GBP",
            "rate": 0.75705
        }
    ]
}
```

Listing 2-1 shows you the result response by accessing the `/currency/latest` endpoint. It shows the base currency, in this case the U.S. Dollar, and all the rates (currency) that you can get from the other countries with 1 USD.

For a conversion, you request the endpoint `/currency/{amount}/{base}/to/{code}`. Imagine you want to know how many Japanese Yens are equal to 10 USD. The request can be done like this: `/currency/10/usd/to/jpy`. You should get the result shown in Listing 2-2.

Listing 2-2. Conversion Response - /currency/10/usd/to/jpy

```
{
    "base": "USD",
    "code": "JPY",
    "amount": 10.0,
    "total": 1021.69995
}
```

Listing 2-2 shows you the result of the conversion endpoint. Let's review the other files.

The pom.xml File

Open the `pom.xml` file and review its contents. Let's review the tags and their meanings:

- The `<packaging/>` tag has the WAR value. When packaging this application, it will generate a WAR that will be executable and deployable to any servlet container.

  ```
  <packaging>war</packaging>
  ```

- The `<parent/>` tag is the key for Spring Boot to work, because it has all the dependencies and versions that you need in your application. It is based on Maven's BOM (Bill of Materials) feature. So, it is important that this particular tag always be in your Spring Boot application.

  ```
  <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>1.4.0.RELEASE</version>
      <relativePath /> <!-- lookup parent from repository -->
  </parent>
  ```

- The `<dependencies/>` tag holds all the dependencies in the Spring Boot application. When you choose the web dependency (using the command line or the Spring starter), the `spring-boot-web-starter` dependency is added to this `pom`. This is similar to when you selected the project as a WAR type, whereby the `spring-boot-starter-tomcat` dependency was added. By default, you always have the `spring-boot-starter-test` dependency.

  ```
  <dependencies>
      <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-web</artifactId>
      </dependency>
  ```

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-aop</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

Let's take a moment to look at the `<dependencies/>` tag. If you are familiar with the Maven way to create a project, you will figure this out by now, that there is a missing tag in the `<dependency/>` tag. That's right, the `<version/>` tag is missing. This version is not used any more because the `<parent/>` tag definition includes all the versions and dependencies that you will use in your web application.

Notice that `spring-boot-starter-tomcat` has the scope as provided, which means you can create a runnable WAR (that you can run using `java -jar <war-name>.war`) and deploy it to any servlet container.

Notice the naming convention that you use to create Spring Boot apps—every `<groupId/>` tag is `org.springframework.boot` and the `<artifactId/>` tag is `spring-boot-starter-<technology>`.

Why is this important? Remember that Spring Boot is an opinionated technology, so based on this `pom` and the `spring-boot-starter`, it will recognize which application you are trying to run.

The Rate.java Class

This will be the domain class. It includes an annotation that defines it as an entity. This is because this class will be persisted into the H2 engine (in-memory database). See Listing 2-3.

Listing 2-3. src/main/java/com/apress/messaging/domain/Rate.java

```
@Entity
public class Rate {

        @Id
        private String code;
        private Float rate;

        @JsonIgnore
        @Temporal(TemporalType.DATE)
        private Date date;

        // Setters and Getters omitted.

}
```

Listing 2-3 shows you the domain class—here the `@Entity` and `@Id` annotations are being used—and this is related to the JPA technology (Spring Data JPA). As you can see, it's very basic. Nothing too complicated.

The RateRepository.java Class

Next is the `RateRepository` interface, which is based on the Spring Data JPA technology, where you need to extend from the `JpaRepository<E,ID>` and add the entity (domain) and the ID class type (`String`, a serializable class) that will be used for persistence. See Listing 2-4.

Listing 2-4. src/main/java/com/apress/messaging/repository/RateRepository.java

```
@Repository
public interface RateRepository  extends JpaRepository<Rate,String>{
        List<Rate> findByDate(Date date);
        Rate findByDateAndCode(Date date,String code);
}
```

Listing 2-4 shows you the `RateRepository` interface. You can see the `@Repository` annotation (this is just a marker for the Spring container); the important part is that you need to extend from the `JpaRepository` interface. This interface will implement all the *CRUD* (Create, Read, Update and Delete) actions for you, so you don't have to implement them. Two query methods are defined—`findByDate` (looks for the rates that have that date) and `findByDateAndCode` (looks for a specific date and code). What makes this interface special is that you can create a SQL query by using the properties of the domain class. If you want to learn more about Spring Data JPA, go to http://docs.spring.io/spring-data/jpa/docs/current/reference/html/.

The CurrencyController.java Class

This class defines the REST endpoints. It uses the `RateRepository` interface through a service. See Listing 2-5.

Listing 2-5. src/main/java/com/apress/messaging/controller/CurrencyController.java

```
@RestController
@RequestMapping("/currency")
public class CurrencyController {

    @Autowired
    CurrencyConversionService service;

    @RequestMapping("/latest")
    public ResponseEntity<CurrencyExchange> getLatest(@RequestParam(name="base",
    defaultValue=CurrencyExchange.BASE_CODE)String base) throws Exception{
                //...
        }

    @RequestMapping("/{amount}/{base}/to/{code}")
    public ResponseEntity<CurrencyConversion> conversion(@PathVariable("amount")
    Float amount,@PathVariable("base")String base,@PathVariable("code")String code)
    throws Exception{
                //...
        }

        //More methods here...

}
```

Listing 2-5 shows you a snippet of the `CurrencyController` class, which uses the Spring MVC `@RestController`, `@RequestMapping`, `@RequestParam`, and `@PathVariable` annotations. Every response is based on the `ResponseEntity` class.

This is pure Spring MVC (not too much Spring Boot). If you want to know more about Spring MVC, visit http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html.

Review this class and experiment with it.

The RestApiDemoApplication.java Class

This class is the main entry point of the web application. See Listing 2-6.

Listing 2-6. src/main/java/com/apress/messaging/RestApiDemoApplication.java

```
@SpringBootApplication
public class RestApiDemoApplication {

   public static void main(String[] args) {
     SpringApplication.run(RestApiDemoApplication.class, args);
   }

  @Bean
  public CommandLineRunner data(RateRepository repository) {
    return (args) -> {
        repository.save(new Rate("EUR",0.88857F,new Date()));
        repository.save(new Rate("JPY",102.17F,new Date()));
        repository.save(new Rate("MXN",19.232F,new Date()));
        repository.save(new Rate("GBP",0.75705F,new Date()));
      };
   }
}
```

Listing 2-6 shows you the main class, where Spring Boot will bootstrap the application. `@SpringBootApplication` is a required annotation for every Spring Boot app, because it will trigger all the magic behind Spring Boot. This annotation will use the *auto-configuration* feature to determine your app and the best configuration for it. In this case, because you have the `spring-boot-starter-web` dependency (in your classpath through your `pom.xml` file), it will set up all the necessary beans (like the `DispatcherServlet`) and the embedded servlet container (Tomcat by default) to configure your web app.

Notice the `@Bean`, which returns a `CommandLineRunner` interface. This will be executed after the Spring container is ready with all your beans, and it will use the `RateRepository` interface to save the rate. You can see this as a way to initialize your database.

Other Files…

Of course, there are more classes, but the idea here is for you to experiment with them and see what they do and how they are used in the project. Look at the `CurrencyConversionService` class, which is used in the web controller.

If you want to see how the AOP is being used in this project, look at the `CurrencyCodeAudit` class. It's defining an `around` advice that will be executed only when an `@ToUpper` annotation is found in a method. (You can get the code from this annotation in the book's source code.) Why do you need to use this aspect? Well, if you look at the conversion endpoint, you are expecting a capital case (for the base and the code, `/150/USD/to/JPY`), but with this approach, you can ignore it. Removing `@ToUpper` from the `CurrencyConversionService` methods, such as `/150/usd/to/jpy`, doesn't work, because it's evaluating just the capital letters for the base and code. You instead need to add extra code to support it.

Running the Spring Boot Currency Web App

There are different ways to run a project like this. If you use the STS and import the project, you can simply right-click the project's name and choose Run As --> Spring Boot App, and that's it!

If you want to run it using the command line, make sure you have Maven installed and execute:

```
$ mvn spring-boot:run
```

Then you can go to your browser and type `http://localhost:8080/currency/latest` and you should get the same result as shown in Listing 2-1. You can also use the `curl` command:

```
$ curl http://localhost:8080/currency/latest
```

As you can see, you don't need to create a WAR and deploy it to a container, because Spring Boot comes with an embedded Tomcat container. This allows you to have portability.

Deploying the Spring Boot Currency Web App

If you want to deploy this code to a servlet container, you need to package this project. If you are using the STS, you can right-click over the project and select Run As --> Maven Build. This will bring up a dialog box. In the Goals field, enter Package and then check the Skip Test check box. You can then click Run. This will create an executable and deployable way to the target directory.

If you are using the command line, you can execute:

```
$ mvn package -DskipTests=true
```

This will create the executable and deployable WAR in the target directory. You can execute the WAR by running the following command:

```
$ java -jar target/rest-api-demo-0.0.1-SNAPSHOT.war
```

If you already have, for example, a Tomcat container, you can just drop the WAR file in the `webapps/` folder and run your container. Try to rename the war when you are copying/moving it.

```
$ cp target/rest-api-demo-0.0.1-SNAPSHOT.war /opt/tomcat/webapps/rest-api-demo.war
$ /opt/tomcat/bin/startup.sh
```

Then you can open your browser and go to `http://localhost:8080/rest-api-demo/currency/latest` to get the same results.

More About Spring Boot

If you want to learn more about Spring Boot and its other useful features, I recommend my other book, *Pro Spring Boot* from Apress. It takes a more in-depth approach to this technology, from using the Spring Boot CLI through deploying it into the cloud.

Summary

This chapter introduced Spring Boot, discussed some of its features, and explained how it works with Spring MVC by creating a simple currency Restful application.

In the next chapter, you are going to start with the Spring Application events and listeners, which is a way to do messaging by emitting and consuming messages through the Observer pattern.

CHAPTER 3

Application Events

This chapter covers how to use the *observer pattern (behavioral pattern)* as a way to send messages to whoever needs them or whoever is listening. The chapter also shows how the Spring Framework implements this pattern through its application events, which can be declared as a simple interface implementation or by using specialized annotations.

The Observer Pattern

This particular pattern defines one-to-many dependencies between objects, so when one object (the subject) change its state, it needs to inform the others (observers) about this change. Then they can react to the change, as shown in Figure 3-1.

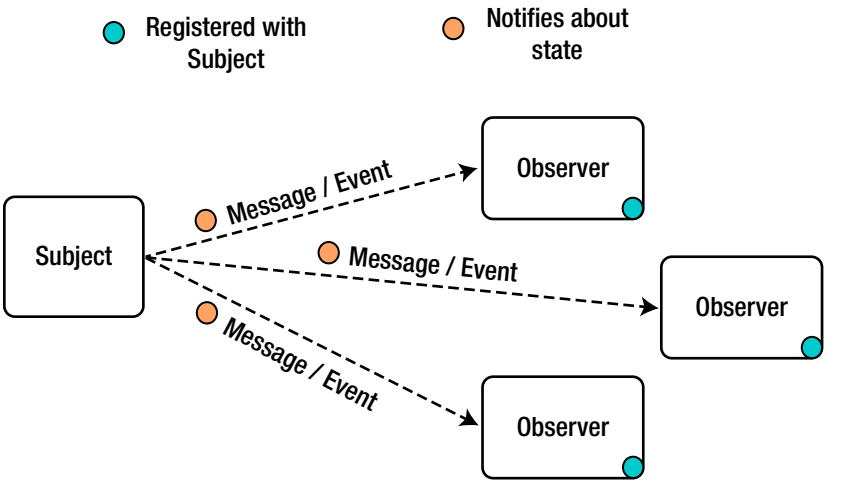

Figure 3-1. *The observer pattern*

There are many ways to implement this design pattern. The Java SDK includes the `java.util.Observable` class, which will set the change and notify the observers, as well as the `java.util.Observer` interface, which will receive the notification through its `update` method.

F. Gutierrez, *Spring Boot Messaging*, DOI 10.1007/978-1-4842-1224-0_3

The Spring Framework has a sophisticated way to use this observer pattern and the recent versions (4.x) of Spring include even more improvements. They not only include this pattern but also have multiple events that allow you to determine more about the internals of the Spring container and to create your own events.

Remember that even though we are talking about the observer pattern and events, this is simply a way to communicate between components using messaging through events.

The Spring ApplicationEvent

You will find that the Spring Framework exposes the abstract `org.springframework.context.ApplicationEvent` class, which extends `java.util.EventObject`. The `EventObject` has an object where the event initially occurred. See Figure 3-2.

Figure 3-2. *The Spring ApplicationEvent hierarchy*

Figure 3-2 shows you the `ApplicationEvent` hierarchy, and as you can see, this class is extended by several events. It's worth mentioning at least two important events:

- `ApplicationContextEvent` (`belongs to the Spring Framework`): This is an abstract class where you have full access to the main central interface, which provides the entire configuration for your application. This class is also extended by the `ContextClosed`, `ContextStartedEvent`, `ContextRefreshedEvent`, and `ContextStoppedEvent` events to give you more details of the lifecycle of the Spring container.
- `SpringApplicationEvent` (`belongs to Spring Boot`): This is also an abstract class that contains all the information about the Spring Boot application through the `SpringApplication` class. The `SpringApplication` class is used to bootstrap and launch a Spring Boot application from a Java main method.

I chose these events because the project you are doing, the Rest API for currency exchanges, uses them. More about this later in this chapter.

Spring ApplicationListener

Now, for every event, you should have a way to receive the message. The Spring Framework has a main event listener. The `org.springframework.context.ApplicationListener <E extends ApplicationEvent>` interface extends from the `java.util.EventListener` (which is just a marker). The `ApplicationListener` is the main event listener for all the Spring `ApplicationContext` events, which you read about in the previous section. See Figure 3-3.

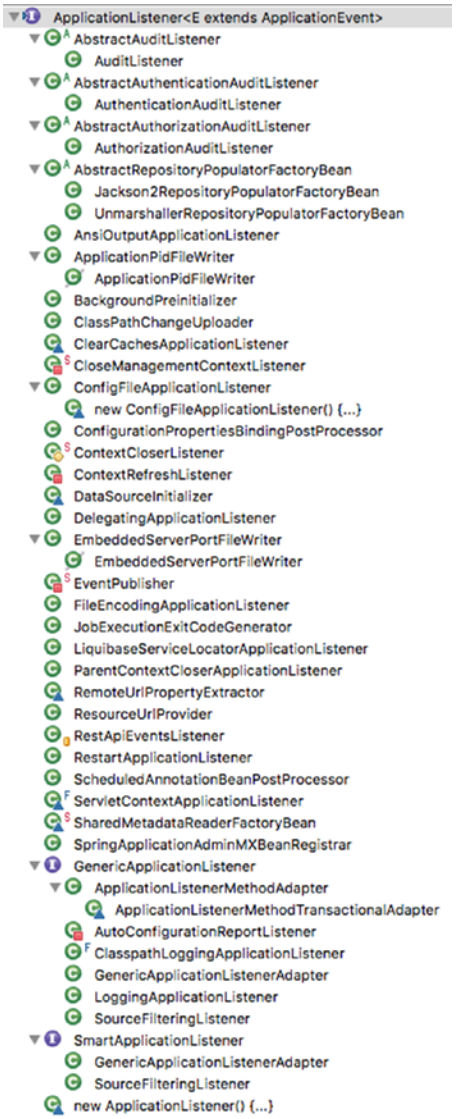

Figure 3-3. *ApplicationListener interface and hierarchy*

Figure 3-3 shows the ApplicationListener hierarchy and all the event listeners you can use. The Spring Framework will send the appropriate events on start, during runtime, and even upon normal shutdown in your Spring application. They will be filtered when the listener is invoked to match the event objects.

There are many use cases whereby you can use the ApplicationEvent or some of its implementations to listen (using the ApplicationListener) for incoming messages and then act on them. For example, in the Rest API currency project, discussed again in the next section, you determine when a user hits some of the Rest endpoints, and you then have a statistical way to see the traffic and recognize which endpoint is used more often.

Rest API Currency Project

Let's go back to the project and apply this type of design pattern to it. By implementing only the ApplicationListener of the ApplicationEvent type, the application will start listening for every event that happens during the Spring container initialization and part of the Beans lifecycle.

Take a look a Listing 3-1, which shows that the RestApiEventsListener class is a Spring component.

Listing 3-1. com.apress.messaging.listener.RestApiEventsListener.java

```
@Component
public class RestApiEventsListener implements ApplicationListener<ApplicationEvent>{

        public void onApplicationEvent(ApplicationEvent event) {

        }
}
```

Listing 3-1 shows you just part of the code where the RestApiEventsListener class implements the ApplicationListener of the ApplicationEvent event. You must implement the onApplicationEvent method that receives the ApplicationEvent as a parameter.

Again, one use case would be to determine how many times a Rest endpoint is being accessed. Look at Figure 3-2, where the ApplicationEvent hierarchy is—you will notice that there is a RequestHandledEvent and a ServletRequestHandledEvent that extends it. With the ServletRequestHandledEvent class, you can get the URL (endpoint) that is being accessed and create a counter for it. See Listing 3-2.

Listing 3-2. com.apress.messaging.listener.RestApiEventsListener.java

```
@Component
public class RestApiEventsListener implements ApplicationListener<ApplicationEvent>{

    private static final String LATEST = "/currency/latest";

    @Autowired
    private CounterService counterService;

    @Log(printParamsValues=true)
    public void onApplicationEvent(ApplicationEvent event) {
```

```
        if(event instanceof ServletRequestHandledEvent){
            if(((ServletRequestHandledEvent)event)
                        .getRequestUrl().equals(LATEST)){
                        counterService
                        .increment("url.currency.latest.hits");
                        }
                }
        }
}
```

Listing 3-2 shows a little more of the code. Let's take a look at it:

- `ServletRequestHandledEvent`: This event is being published when there is a request to an endpoint. This is part of the web framework. This event contains all the web context information, which is why you can get the information about the URL being accessed.
- `CounterService`: This interface belongs to the `spring-boot-actuator` module, which allows you to have a metric by incrementing or decrementing a tag/property. In this case, the tag is `url.currency.latest.hits`.
- `@Log(printParamsValues=true)`: This is a custom annotation that will be used as part of the before advice that logs all the information about the method being called. You can see the code in the `com.apress.messaging.aop.CodeLogger.java` class.

If you run the application, you will get some output generated from the `@Log` annotation (see Figure 3-4), but if you visit the `/currency/latest` endpoint a few times (see Figure 3-5), the application will start counting this endpoint as being accessed by using the `CounterService` instance. Then you can access this metric from the `/metrics` endpoint and see that the `url.currency.latest.hits` is being shown with the number of hits of that endpoint (see Figure 3-6).

```
2016-11-05 09:16:48.825  INFO 36482 --- [  restartedMain] o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/env/{name:.*}],methods=[GET],produces=[applica
2016-11-05 09:16:48.825  INFO 36482 --- [  restartedMain] o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/env || /env.json],methods=[GET],produces=[appl
2016-11-05 09:16:48.827  INFO 36482 --- [  restartedMain] o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/dump || /dump.json],methods=[GET],produces=[ap
2016-11-05 09:16:48.828  INFO 36482 --- [  restartedMain] o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/info || /info.json],methods=[GET],produces=[ap
2016-11-05 09:16:48.828  INFO 36482 --- [  restartedMain] o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/trace || /trace.json],methods=[GET],produces=[
2016-11-05 09:16:49.088  INFO 36482 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
2016-11-05 09:16:49.140  INFO 36482 --- [  restartedMain] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
2016-11-05 09:16:49.151  INFO 36482 --- [  restartedMain] o.s.c.support.DefaultLifecycleProcessor  : Starting beans in phase 0
2016-11-05 09:16:49.214  INFO 36482 --- [  restartedMain] com.apress.messaging.aop.CodeLogger      :
=================================
 Class: RestApiEventsListener
Method: onApplicationEvent

 Param: ContextRefreshedEvent
 Value: org.springframework.context.event.ContextRefreshedEvent[source=org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext
=================================
2016-11-05 09:16:49.278  INFO 36482 --- [  restartedMain] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2016-11-05 09:16:49.280  INFO 36482 --- [  restartedMain] com.apress.messaging.aop.CodeLogger      :
=================================
 Class: RestApiEventsListener
Method: onApplicationEvent

 Param: EmbeddedServletContainerInitializedEvent
 Value: org.springframework.boot.context.embedded.EmbeddedServletContainerInitializedEvent[source=org.springframework.boot.context.embedded.tomcat.TomcatEmbed
=================================
2016-11-05 09:16:49.323  INFO 36482 --- [  restartedMain] com.apress.messaging.aop.CodeLogger      :
=================================
 Class: RestApiEventsListener
Method: onApplicationEvent

 Param: CurrencyEvent
 Value: com.apress.messaging.event.CurrencyEvent[source=com.apress.messaging.service.CurrencyService@118d3152]
---------------------------------
```

Figure 3-4. *Console logs after running the rest-api-events project*

Figure 3-4 shows you some of the log output that you will see when running your application. These logs are the messages that the Spring Framework sends as `ApplicationEvent` events. Remember that this output is generated by the `@Log` annotation (a before AOP advice).

Figure 3-5 shows you some of the logs after accessing the `/currency/latest` endpoint a few times. You can see that the `ApplicationEvent` is an instance of the `ServletRequestedEvent`, which contains the requested URL.

```
2016-11-06 08:49:59.859  INFO 44519 --- [nio-8080-exec-7] com.apress.messaging.aop.CodeLogger      :
==================================
 Class: RestApiEventsListener
Method: onApplicationEvent

 Param: ServletRequestHandledEvent
 Value: ServletRequestHandledEvent: url=[/currency/latest]; client=[0:0:0:0:0:0:0:1]; method=[GET]; servlet=[dispatcherServlet]; session=[null]; user=[null]; time
==================================
2016-11-06 08:50:01.051  INFO 44519 --- [nio-8080-exec-8] com.apress.messaging.aop.CodeLogger      :
==================================
 Class: RestApiEventsListener
Method: onApplicationEvent

 Param: ServletRequestHandledEvent
 Value: ServletRequestHandledEvent: url=[/currency/latest]; client=[0:0:0:0:0:0:0:1]; method=[GET]; servlet=[dispatcherServlet]; session=[null]; user=[null]; time
==================================
```

Figure 3-5. *Console logs after visiting the /currency/latest endpoint*

Figure 3-6 shows you the `/metrics` endpoint (provided by the `spring-boot-actuator` dependency). At the bottom of the figure, you can see the `"counter.url.currency.latest.hits":4` metric, which is updated by the `CounterService` instance (see Listing 3-2).

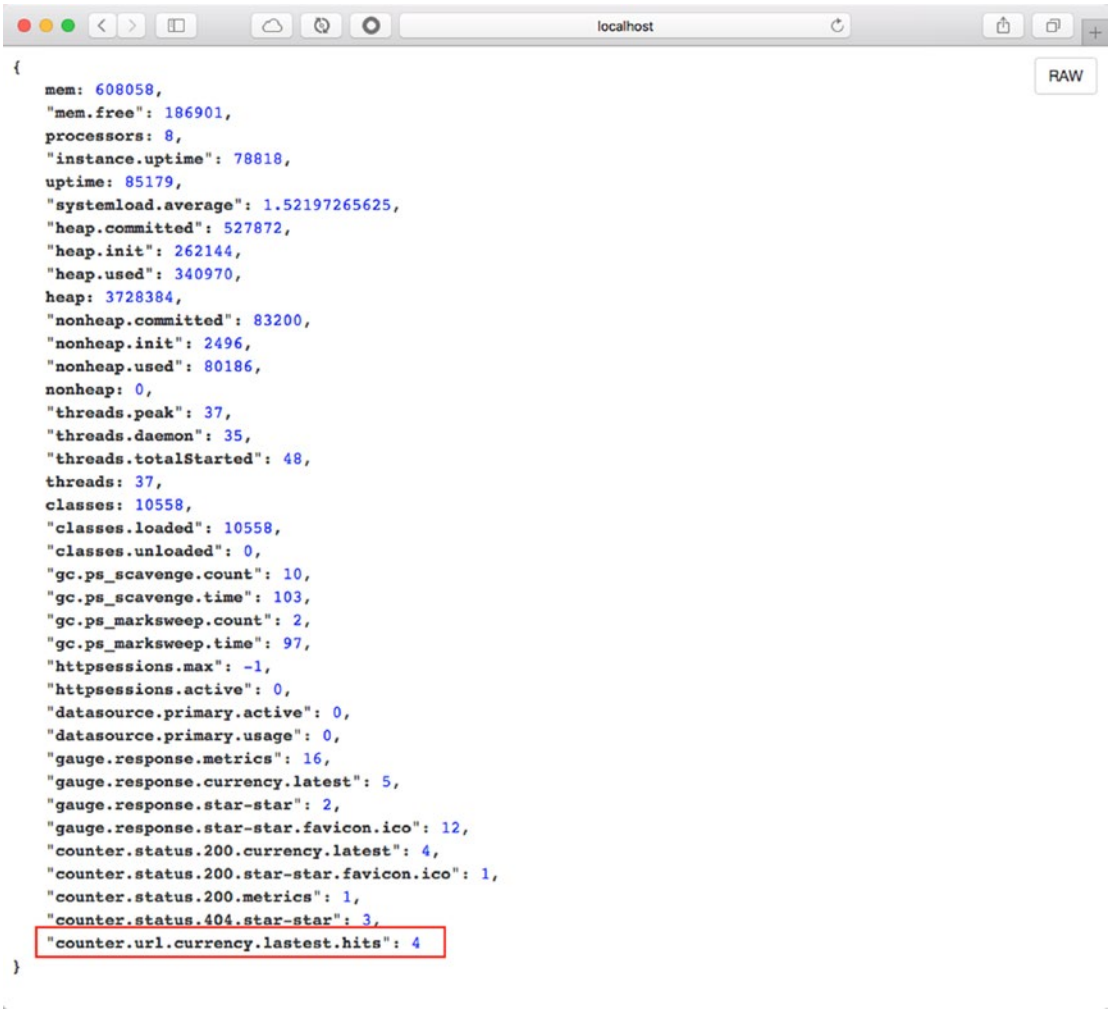

Figure 3-6. *http://localhost:8080/metrics*

Custom Events

So far, you have seen a way to create listeners for any `ApplicationEvent`, but what about when using custom events, ones that contain information about your domain object? This section shows you how to create a custom event.

To create a custom event, you must extend from `ApplicationEvent` so it's easy to publish it later.

Using the current project, imagine that you will send an event when there's an error (unchecked exception) during the call of any currency conversion. Maybe this will simply log the cause and show the object that caused the exception. Let's start by reviewing the `CurrencyConversionEvent` class, as shown in Listing 3-3.

Listing 3-3. com.apress.messaging.event.CurrencyConversionEvent.java

```
package com.apress.messaging.event;

import org.springframework.context.ApplicationEvent;
import com.apress.messaging.domain.CurrencyConversion;

public class CurrencyConversionEvent extends ApplicationEvent {

    private static final long serialVersionUID = -4481493963350551884L;
    private CurrencyConversion conversion;
    private String message;

    public CurrencyConversionEvent(Object source, CurrencyConversion conversion) {
        super(source);
        this.conversion = conversion;
    }

    public CurrencyConversionEvent(Object source, String message, CurrencyConversion
    conversion) {
        super(source);
        this.message = message;
        this.conversion = conversion;
    }

    public CurrencyConversion getConversion(){
        return conversion;
    }

    public String getMessage(){
        return message;
    }
}
```

Listing 3-3 shows you a basic class that extends from `ApplicationEvent` and has two constructors. Each constructor will call its parent (`ApplicationEvent`) to set the source, set the current `CurrencyConversion` instance, and determine the message. In other words, you have the information needed to determine the source of errors in any currency conversion call.

Next, let's review the event listener that will receive the `CurrencyConversionEvent`. See Listing 3-4.

Listing 3-4. com.apress.messaging.listener.CurrencyConversionEventListener.java

```
package com.apress.messaging.listener;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.ApplicationListener;
import org.springframework.stereotype.Component;
import com.apress.messaging.event.CurrencyConversionEvent;
```

```
@Component
public class CurrencyConversionEventListener implements ApplicationListener<Currency
ConversionEvent> {

    private static final String DASH_LINE = "==================================";
    private static final String NEXT_LINE = "\n";
    private static final Logger log = LoggerFactory.getLogger(CurrencyConversionEvent
    Listener.class);

    @Override
    public void onApplicationEvent(CurrencyConversionEvent event) {
        Object obj = event.getSource();
        StringBuilder str = new StringBuilder(NEXT_LINE);
        str.append(DASH_LINE);
        str.append(NEXT_LINE);
        str.append("  Class: " + obj.getClass().getSimpleName());
        str.append(NEXT_LINE);
        str.append("Message: " + event.getMessage());
        str.append(NEXT_LINE);
        str.append("  Value: " + event.getConversion());
        str.append(NEXT_LINE);
        str.append(DASH_LINE);
        log.error(str.toString());
    }

}
```

Listing 3-4 shows you the listener that will receive all the `CurrencyConversionEvent` events. This class implements the `ApplicationListener` of type `CurrencyConversionEvent`, and it needs to implement the `onApplicationEvent`. As you can see, it is just logging the class, the message, and the `CurrencyConversion` domain object.

Next, let's see which class will be publishing the event when an error occurs. Let's review our `CurrencyConversionService` and the `convertFromTo` method, which has logic to get the currency codes. See Listing 3-5.

Listing 3-5. com.apress.messaging.service.CurrencyConversionService.java

```
public CurrencyConversion convertFromTo(@ToUpper String base, @ToUpper String
code,Float amount) {
        Rate baseRate = new Rate(CurrencyExchange.BASE_CODE,1.0F,new Date());
        Rate codeRate = new Rate(CurrencyExchange.BASE_CODE,1.0F,new Date());

        if(!CurrencyExchange.BASE_CODE.equals(base))
                baseRate = repository.findByDateAndCode(new Date(), base);

        if(!CurrencyExchange.BASE_CODE.equals(code))
                codeRate = repository.findByDateAndCode(new Date(), code);
```

```
        if(null == codeRate || null == baseRate)
                throw new BadCodeRuntimeException("Bad Code Base, unknown code:
                " + base, new CurrencyConversion(base,code,amount,-1F));

        return new CurrencyConversion(base,code,amount,(codeRate.getRate()/baseRate.
        getRate()) * amount);
}
```

Listing 3-5 shows you the `convertFromTo` method, which is doing the conversion by finding the rates based on the `base` and `code` variables. The `if` statement will throw a `BadCodeRuntimeException`, which has a constructor that accepts a string and a `CurrencyConversion` object. `BadCodeRuntimeException` is the unchecked exception that extends from `RuntimeException` (you can review that in the source code).

When the exception is thrown, we must publish the `CurrencyConversionEvent` and the error message, but adding this logic would mess up the code and soon we would have tangled, messy code. We can instead create an AOP that uses this exception when it's thrown. Listing 3-6 uses an AOP to publish the `CurrencyConversionEvent` event after throwing the exception.

Listing 3-6. com.apress.messaging.aop.CurrencyConversionAudit.java

```
@Aspect
@Component
public class CurrencyConversionAudit {

    private ApplicationEventPublisher publisher;

    @Autowired
    public CurrencyConversionAudit(
                        ApplicationEventPublisher publisher){
            this.publisher = publisher;
    }

    @Pointcut("execution(* com.apress.messaging.service.*Service.*(..))")
    public void exceptionPointcut() {}

    @AfterThrowing(pointcut="exceptionPointcut()",
                                throwing="ex")
    public void badCodeException(JoinPoint jp,
                                BadCodeRuntimeException ex){

        if(ex.getConversion()!=null){
            publisher.publishEvent(
                new CurrencyConversionEvent(
                                jp.getTarget(),
                                ex.getMessage(),
                                ex.getConversion()));
                }
        }

}
```

Listing 3-6 shows you `@AfterThrowing`, which knows about the `BadCodeRuntimeException` when it's thrown, and then executes the code inside the method. As you can see, it uses the `ApplicationEventPublisher` instance (which is being injected by the Spring Framework in the class constructor) and the `publishEvent` method, which sends the information about the class where the exception occurred, the message, and the `CurrencyConversion` object.

If you run the project and access `/{amount}/{base}/to/{code}` like this `/1.0/usdx/to/mx`, you will get something similar to Figure 3-7.

```
==================================
2016-11-06 18:24:24.004 ERROR 47845 --- [nio-8080-exec-1] c.a.m.l.CurrencyConversionEventListener  :
==================================
  Class: CurrencyConversionService
Message: Bad Code Base, unknown code: USDX
  Value: CurrencyConversion [base=USDX, code=MXN, amount=1.0, total=-1.0]
==================================
```

Figure 3-7. *Log error in the CurrencyConversionService*

As you can see, creating custom events is really easy. Remember these simple rules to use a custom event:

- Create an event class that extends from `ApplicationEvent`.
- Create an event listener class that implements the `ApplicationListener` of your custom event and implement the `onApplication` method.
- Use the `ApplicationEventPublisher` class to publish your custom event.

Using Event Listeners with Annotations

So far we have seen how to use and implement the `ApplicationListener` of type `ApplicationEvent`. This section shows you how to use some of the Spring-provided annotations as an easy way to listen for events.

@EventListener

The `@EventListener` annotation is a helpful annotation that you can use directly in the method that will handle the event. This means that there is no need to implement `ApplicationListener` anymore.

Listing 3-7 shows how you can use it.

Listing 3-7. com.apress.messaging.listener.RestAppListener.java

```
@Component
public class RestAppEventListener {

    @EventListener
    @Log(printParamsValues=true)
    public void restAppHandler(
            SpringApplicationEvent springApp){
        }
}
```

Listing 3-7 shows you the `@EventListener` annotation, which is applied to the `restAppHandler` method. The Spring Framework will wire everything up so this listener receives all the `SpringApplicationEvent` events. The `SpringApplicationEvent` is another event that extends from the `ApplicationEvent` abstract class but contains information about the Spring Boot application, like the arguments used in a command line, the banner, the resource loader, etc.

If you run the project, you will see something similar to Figure 3-8.

```
2016-11-06 20:43:28.676  INFO 48750 --- [  restartedMain] com.apress.messaging.aop.CodeLogger      :
================================
 Class: RestAppEventListener
Method: restAppHandler

 Param: ApplicationReadyEvent
 Value: org.springframework.boot.context.event.ApplicationReadyEvent[source=org.springframework.boot.SpringApplication@14860744]
================================
```

Figure 3-8. *Logs of the RestAppEventListener*

As you can see, the `@EventListener` annotation is simple to use. This annotation has even more features:

- It supports conditions, so it can be executed only if the expression given is true. For example:

  ```
  @EventListener(condition = "#springApp.args.length > 1")
  ```

  This snippet tells the listener to only use the events if the argument's length is greater than 1. If you replace the previous listener in Listing 3-7, you won't see the `RestAppEventListener` logs.

- You can listen for many events by passing an array of the event classes as the default value. For example:

  ```
  @EventListener({CurrencyEvent.class,
                  CurrencyConversionEvent.class})
  @Log(printParamsValues=true)
  public void restAppHandler(ApplicationEvent appEvent){ }
  ```

  This snippet listens for the `CurrencyEvent` and `CurrencyConversion` events in the same method, which now gets an `ApplicationEvent` instance. Also you can have no arguments and still listen for multiple events.

- When you have multiple events to listen to, you might want to prioritize them. You can add the `@Order` annotation to the method to do so. For example:

  ```
  @EventListener
  @Order(Ordered.HIGHEST_PRECEDENCE)
  @Log(printParamsValues=true)
  public void restAppHandler(SpringApplicationEvent springApp){
  }
  ```

  This snippet will be processed in the order of highest precedence.

- You can also process your event listeners in an asynchronous way, by adding the `@Async` annotation. For example:

```
@EventListener
@Async
@Log(printParamsValues=true)
public void restAppHandler(SpringApplicationEvent springApp){
}
```

@TransactionalEventListener

The Spring Framework 4.2.x and above versions introduce an additional annotation that allows you to listen for the transaction phase, such as a database transaction or any other transactions, including messaging events.

Let's start by using this annotation in the currency project. Look at the `RateEventListener` class shown in Listing 3-8.

Listing 3-8. com.apress.messaging.listener.RateEventListener.java

```
@Component
public class RateEventListener {

    @TransactionalEventListener
    @Log(printParamsValues=true,
             callMethodWithNoParamsToString="getRate")
    public void processEvent(CurrencyEvent event){ }
}
```

Listing 3-8 shows you the `RateEventListener` class, which uses the `@TransactionalEventListener` and processes the custom `CurrencyEvent` event (you can look at the code in the `com.apress.messaging.event` package). The `@TransactionalEventListener` will receive the events when a transactional channel is being established, either programmatically or by using the `@Transactional` annotation.

If you look at the `com.apress.messaging.service.CurrencyService.java` class, you will see the following code:

```
@Transactional
public void saveRate(Rate rate){
        repository.save(new
                Rate(rate.getCode(),
                     rate.getRate(),
                    rate.getDate()));
        publisher.publishEvent(new CurrencyEvent(this,rate));
}
```

This snippet shows the `saveRate` method, which is marked with the `@Transactional` annotation. It will publish a `CurrencyEvent` when it's done saving.

If you run the project you will see the logs about the `RateEventListener` several times. Take a peek at the main app (`RestApiEventsApplication.java`); you will see rates being saved using the `CurrencyService` instance and the logs of each transaction (after they are committed) by the `RateEventListener` listener. See Figure 3-9.

```
2016-11-06 21:12:07.511  INFO 48750 --- [  restartedMain] com.apress.messaging.aop.CodeLogger      :
==================================
 Class: RateEventListener
Method: processEvent

 Param: CurrencyEvent
 Value: Rate [code=EUR, rate=0.88857, date=2016-11-06]
```

Figure 3-9. *RateEventListener logs*

`@TransactionalEventListener` can listen for specific transaction phases. If you need to listen for events during a phase, you can use it like so:

```
@TransactionalEventListener(
        phase = TransactionPhase.BEFORE_COMMIT)
```

you can have: `BEFORE_COMMIT`, `AFTER_COMMIT` (default), `AFTER_ROLLBACK` and `AFTER_COMPLETION`.

■ **Note** Remember that you can get all the code from the Apress site or directly from the GitHub Repository: http://www.apress.com/9781484212257

Summary

This chapter explained how the observer pattern works. It also covered the way the Spring Framework uses this pattern to expose application events.

It showed you some use cases whereby you can listen for application events and use them to count how many times an endpoint is being accessed by `ServletRequestHandledEvent`. It showed you how to create your own custom events by extending the `ApplicationEvent` abstract class.

You learned about an easy way to listen for events by using annotations such as the `@EventListener`. You also saw how to use the `@TransactionalEventListener` that is being triggered when a transaction happens, during, before, or after commit and after a rollback.

The next chapter covers the Java Message Service API and how you can do messaging using it.

CHAPTER 4

JMS with Spring Boot

The Java Message Service (JMS) was announced in June 2001 with version 1.0.2b. It's another solution for sending messaging between two or more clients. It was considered part of a Message Oriented Middleware (MOM) group of technologies at that time. The idea was to provide an API for a recurrent problem, a producer-consumer use case that allowed loosely coupled, reliable, and asynchronous components in a distributed environment.

This chapter starts with a simple project that will help you understand how the JMS clients work and how to configure it with Spring Boot. Then we are going to use this knowledge to build on the previous project, the currency REST API that now will be a receiver to save new rates. So, let's get started.

JMS

The JMS API provides two messaging models—*point-to-point* and *publish-subscribe*. Point-to-point is where messages are delivered to a receiver, and the delivery is guaranteed to only one consumer that is connected to a queue (see Figure 4-1).

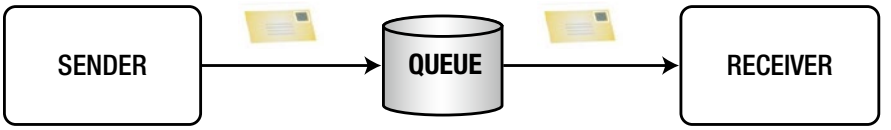

Figure 4-1. *The point-to-point messaging model*

The publish-subscribe model is where a message is delivered to zero or more consumers (normally called subscribers). The publisher creates a message topic for all the clients that want to subscribe to it (see Figure 4-2).

F. Gutierrez, *Spring Boot Messaging*, DOI 10.1007/978-1-4842-1224-0_4

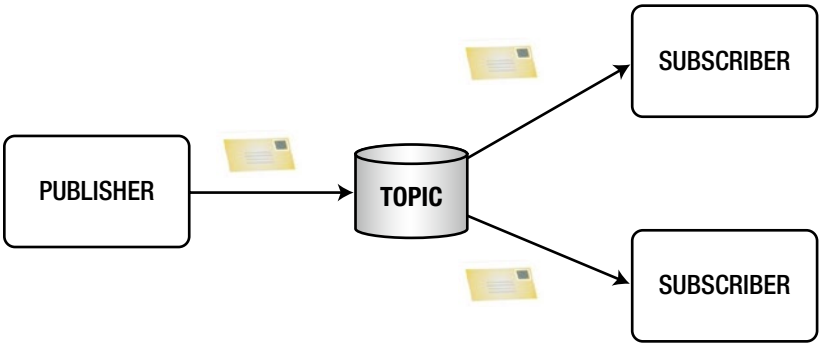

Figure 4-2. *The publish-subscribe messaging model*

JMS is a required API that needs to be implemented. In order to use or create JMS applications, you need to choose a *provider* (often called JMS server or broker) that will connect and decouple your senders/publishers from your receivers/subscribers, a *client* that will produce/send or receive/subscribe messages, a JMS *message* that contains the actual message (payload), and a JMS *queue* for a point-to-point messaging or a *topic* for a publish-subscriber scenario. We first talk more about the client.

JMS with Java Skipping to Spring Boot Part

Let's first see how can you create a point-to-point sender client in Java; see Listing 4-1.

Listing 4-1. Point-to-Point Sender Client Snippet Code

```
//Step 1. Create the Connection
InitialContext ctx = new InitialContext();
QueueConnectionFactory factory = (QueueConnectionFactory)ctx.
lookup("connectionFactory");
QueueConnection connection = factory.createQueueConnection();
connection.start();

//Step 2. Create a Queue Session
QueueSession session = connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);

//Step 3. Get the Queue object
Queue queue =( Queue)ctx.lookup("myQueue");

//Step 4. Create the Sender
QueueSender sender = session.createSender(queue);

//Step 5. Create the Message
TextMessage msg = session.createTextMessage();
msg.setText("Hello World");

//Step 6. Send the Message
sender.send(msg);
```

As you can see in Listing 4-1, this process is very straightforward. There are only six steps to send a text message. In Step 1, you need to know which connection process to use. Normally you need to include the `jndi.properties` file in your code with some information about your JMS provider; for example, if you use the Apache ActiveMQ, you need to specify its properties, as shown in Listing 4-2 (`jndi.properties`).

Listing 4-2. jndi.properties for Apache ActiveMQ

```
# Initial Context for the Apache ActiveMQ
java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory

# This property must be the same as the one declared in the ctx.lookup statement.
# by default is: connectionFactory or ConnectionFactory
# connectionFactoryNames = connectionFactory, queueConnectionFactory,
queueConnectionFactory

#   Memory Broker = vm://localhost
# External Broker = tcp://hostname:61616
java.naming.provider.url=vm://localhost

# Queue naming rules:
# queue.[jndiName] = [physicalName]
queue.myQueue = apress.MyQueue

# Topic naming rules:
# topic.[jndiName] = [physicalName]
topic.myTopic = apress.MyTopic
```

Listing 4-2 shows the `jndi.properties` file that you need to include in every JMS application. In this case, it's using the Apache ActiveMQ settings and naming convention for the queues and topics.

Now, let's take a look at the receiver, shown in Listing 4-3.

Listing 4-3. Point-to-Point Receiver Client Snippet Code

```
// Step 1. Create Connection
InitialContext ctx = new InitialContext();
QueueConnectionFactory factory = (QueueConnectionFactory)ctx.lookup("connectionFactory");
QueueConnection connection = factory.createQueueConnection();
connection.start();

// Step 2. Create Session
QueueSession session = connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);

// Step 3. Get the Queue
Queue queue=(Queue)ctx.lookup("myQueue");

// Step 4. Create the Receiver
QueueReceiver receiver = session.createReceiver(queue);
```

```
// Step 5. Create the Listener
MessageListener listener = new MessageListener() {

                        @Override
                        public void onMessage(Message message) {
                                //Process the message here
                        }
                };

// Step 6. Register the Listener
receiver.setMessageListener(listener);
```

Listing 4-3 shows you the six steps needed to create a consumer for a point-to-point message. Of course, if you have this code in a separate project, you need to include `jndi.properties` as well (see Listing 4-2).

If you want to use the publisher-subscriber messaging model, you can create your publisher as shown in Listing 4-4.

Listing 4-4. Publisher-Subscriber Publisher Client

```
//Step 1. Create the Connection
InitialContext ctx = new InitialContext();
TopicConnectionFactory factory =(TopicConnectionFactory)ctx.lookup("connectionFactory");
TopicConnection connection=f.createTopicConnection();
connection.start();

//Step 2. Create a Topic Session
TopicSession session = connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);

//Step 3. Get the Topic object
Topic topic = (Topic)ctx.lookup("myTopic");

//Step 4. Create the Sender
TopicPublisher publisher = session.createPublisher(topic);

//Step 5. Create the Message
TextMessage msg = session.createTextMessage();
msg.setText("Hello World");

//Step 6. Send the Message
publisher.publish(msg);
```

Listing 4-4 shows you the publisher code—a publisher-subscriber message model—that will publish a simple text message to a topic named `myTopic`. Not too different from the point-to-point model. How about the subscriber? See Listing 4-5.

Listing 4-5. Publisher-Subscriber Subscriber Client

```
// Step 1. Create Connection
InitialContext ctx = new InitialContext();
TopicConnectionFactory factory = (TopicConnectionFactory)ctx.lookup("connectionFactory");
TopicConnection connection = factory.createTopicConnection();
connection.start();

// Step 2. Create Session
TopicSession session = connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);

// Step 3. Get the Topic
Topic topic = (Topic)ctx.lookup("myTopic");

// Step 4. Create the Receiver
TopicSubscriber subscriber = session.createSubscriber(topic);

// Step 5. Create the Listener
MessageListener listener = new MessageListener() {

                        @Override
                        public void onMessage(Message message) {
                                //Process the message here
                        }
                };

// Step 6. Register the Listener
subscriber.setMessageListener(listener);
```

Listing 4-5 shows the publisher-subscriber messaging client; this client subscribes to the topic named `myTopic`. You can run multiple clients with this code and each one will receive the message from the publisher; again not too different from the point-to-point model. Of course, you still need `jndi.properties` for these clients.

As you can see, both messaging model implementations are very straightforward. If you take a closer look to any of the code, you will notice that in every case, the message sent is just a text message, but what happens if you need to send something else? JMS supports different message types:

- `StreamMessage` is a serialized stream object.
- `MapMessage` consists of a name/value pairs, like a hash table.
- `TextMessage` is a string.
- `ObjectMessage` is a serializable object.
- `ByteMessage` is a raw stream of bytes.

As homework, try to create some clients (point-to-point or publisher-subscribe models) using these code snippets. The idea is to familiarize yourself on what you can do with this kind of messaging.

I've just given you a sneak peek on what you normally do with JMS without using an external framework. I think there are a lot of steps just to do something simple.

JMS with Spring Boot

Chapter 2 showed you how Spring Boot knows about the application you are trying to run because it's an opinionated technology. Just by adding a `spring-boot-starter` pom, you tell Spring Boot how to configure everything, right?

We are going to use Apache ActiveMQ in this example (you are welcome to use any other broker; the code will be the same), which means that we can include the `spring-boot-starter-activemq` dependency. By adding this dependency, Spring Boot will bring all the JMS and ActiveMQ (JAR files) that we will need for the applications and it will auto-configure all the necessary properties and extra configuration needed for the JMS client.

Remember all the steps that you needed to do in Java? These are not necessary with Spring Boot.

■ **Note** We are going to use more dependencies like AOP to take care of some logging concerns. Take a look at the `pom.xml` file in the projects of these chapters.

We are going to work with the **`jms-sender`** project, which you can find in the book's source code. This project has a lot of the classes with some code commented out, and just by uncommenting, it should work. Don't worry, though, as I'll guide you through the following sections by explaining each snippet of code.

■ **Note** Remember that you can get all the code from the Apress site or directly from the GitHub Repository at `http://www.apress.com/9781484212257`

Producer

Let's start by reviewing the producer using Spring Boot. Open the `com.apress.messaging.jms.SimpleSender.java` class. See Listing 4-6.

Listing 4-6. com.apress.messaging.jms.SimpleSender.java

```
@Component
public class SimpleSender {

    private JmsTemplate jmsTemplate;

    @Autowired
    public SimpleSender(JmsTemplate jmsTemplate){
        this.jmsTemplate = jmsTemplate;
    }
```

```
    public void sendMessage(String destination,
                            String message){
        this.jmsTemplate.convertAndSend(destination, message);
    }
}
```

Listing 4-6 shows you the simplest and easiest way to create a producer. Let's analyze each part of the code:

- `@Component`: This annotation, as you probably already know, marks the class as a Spring bean, making it available at runtime.
- `JmsTemplate`: This is the most important piece of the client, because this class will send a message (among other things) to the JMS provider/broker.
- @Autowired: This annotation is used in the class constructor to inject the `JmsTemplate` bean (described before). You can even omit this annotation and Spring will figure out that you need this dependency here.
- `convertAndSend`: The `JmsTemplate` instance has this method and it converts the message. Because this is a string, it's converted automatically into a `javax.jms.TextMessage` and is sent to the destination, which is normally a queue.

Before you run this part of the code, let's examine the main application. Open the `com.apress.messaging.JmsSenderApplication.java` class, as shown in Listing 4-7.

Listing 4-7. com.apress.messaging.JmsSenderApplication.java

```
@SpringBootApplication
public class JmsSenderApplication {

    public static void main(String[] args) {
        SpringApplication.run(JmsSenderApplication.class, args);
    }

    @Bean
    CommandLineRunner simple(JMSProperties props,
                             SimpleSender sender){
        return args -> {
           sender.sendMessage(props.getQueue(), "Hello World");
        };
    }
}
```

Listing 4-7 shows you the main application. Let's consider each part:

- `@SpringBootApplication`: You already know this annotation. It's the way that Spring Boot identifying what type of application you are trying to run.
- `simple(JMSProperties,SimpleSender)`: This method is executed when the Spring container is ready to be used and will inject a `JMSProperties` and the `SimpleSender` beans for its use.
- `JMSProperties`: This class is used as a properties holder that reads the `application.properties` file and looks for the `apress.jms.queue` property (in this case set to `jms-sender`). If you want to know more, you can read more about it in the "Externalized Configuration" section in the Spring Boot reference.

Now you are ready to run this. (You can run it using the command line with Maven: `$ mvn spring-boot:run` or if you imported it into the STS, you can run it within the Boot Dashboard.) Once you run it, you will see something similar to Figure 4-3.

```
2016-11-16 20:14:39.036  INFO 31017 --- [  restartedMain] JMSAudit
==================================
[BEFORE]
Method: sendMessage
Params:
> arg0: jms-demo
> arg1: Hello World
==================================
```

Figure 4-3. *JmsSenderApplication logs*

Figure 4-3 shows you the logs once the application is running. The code is actually sending the message `"Hello World"` to the `jms-demo` queue. This means that you will see the `JMSAudit` text with some extra information. Where did I actually print this? Review the `com.apress.messaging.aop.JMSAudit.java` class. This class is an Around advice that is normally used for logging purposes. I know that for this example it's too much, but it gives me more ways to explore AOP.

You can see that this client is sending a message, but where is it? You know that we are using the `spring-boot-starter-activemq` dependency, but there doesn't seem to be any broker running. Where is it?

Remember that Spring Boot makes opinions based on your classpath dependencies, so by knowing that you have the `spring-boot-starter-activemq` dependency, it will see if you already have declared beans of type `connectionFactory`, `session`, `sender`, etc., so it can use them. If Spring Boot doesn't find anything, it will create all these by default and it will use the *in-memory* provider (the URL is `vm://localhost`). That's why there is no error at runtime and you can see that the message is sent.

Consumer

Next, let's look at the consumer. First I'll show you the consumer that uses the `javax.jms.MessageListener` interface (it's the same as before, in the JMS with Java section) to see what is needed to configure it. See Listing 4-8.

Listing 4-8. com.apress.messaging.jms.QueueListener.java

```
@Component
public class QueueListener implements MessageListener {

        public void onMessage(Message message) {

        }
}
```

Listing 4-8 shows you the receiver that will listen for any messages in the queue (`jms-demo`). Let's review the code:

- `@Component`: If this is commented out, remove the `//` and add the right imports. I suggest you use the STS and press Cmd+Shift+O on a Mac or Ctrl+Shift+O on Windows. This annotation will mark the class as a Spring bean, so it can be used in the configuration.
- `MessageListener`: This interface is necessary to receive JMS messages, and it's necessary to implement the `onMessage` method.
- `onMessage(Message)`: This is necessary to implement the method, and it has the actual message that is consumed from the queue.

As you can see, it's very simple, but if you try it to run it, you will see the same result as before. You'll see just the log about sending the message. Why is this? Well, you must tell Spring Boot how to use this listener, so take a look at Listing 4-9.

Listing 4-9. com.apress.messaging.config.JMSConfig.java

```
@Configuration
@EnableConfigurationProperties(JMSProperties.class)
public class JMSConfig {

    @Bean
    public DefaultMessageListenerContainer
        customMessageListenerContainer(
                ConnectionFactory connectionFactory,
                   MessageListener queueListener,
            @Value("${apress.jms.queue}") final
                            String destinationName){
        DefaultMessageListenerContainer listener = new
                   DefaultMessageListenerContainer();
        listener.setConnectionFactory(connectionFactory);
        listener.setDestinationName(destinationName);
        listener.setMessageListener(queueListener);
        return listener;
    }
}
```

Listing 4-9 shows you the configuration that you need to enable the `QueueListener` class. Let's review this class:

- `@Configuration`: This is a marker for the class to be taken as a Java config for the Spring container, so everything here will be used to set up Spring.
- `@EnableConfigurationProperties`: Remember that we were using the `application.properties` file to set the queue's name? This particular annotation will use the provided class (`JMSProperties.class`) as the property holder, so you can set some properties in the `application.properties` file. Later, you can get its value either by using `@Value` or by using the JMSProperties instance bean and the getters.
- `@Bean`: This is a marker to create a bean of type `DefaultMessageListenerContainer` in the Spring container.
- `DefaultMessageListenerContainer`: In this case, this is a return type that will be taken as a Spring bean. It has all the necessary information to determine the `QueueListener` class and what queue to consume from (`destinationName`).
- `ConnectionFactory`: Spring will inject this instance and it will auto-configure it using the defaults (unless you provide a custom `ConnectionFactory`). In this case, it will use the in-memory provider/broker.
- `MessageListener`: Spring will inject the `QueueListener` class (the receiver from Listing 4-8) so it can be used to set up the `DefaultMessageListenerContainer` instance.
- `@Value("${apress.jms.queue}")`: This annotation will inject the value `"jms-demo"` from the `apress.jms.queue` property that is in the `application.properties` file into the `destinationName` parameter.

Then, the actual method will create the `DefaultMessageListenerContainer` and set all its properties (`connectionFactory`, `queueListener`, and `destinationName`).

Now, if you run the application, you should get output similar to Figure 4-4.

```
2016-11-17 07:58:08.337  INFO 34103 --- [  restartedMain] JMSAudit                                 :
===================================
[BEFORE]
Method: sendMessage
Params:
> arg0: jms-demo
> arg1: Hello World
===================================
2016-11-17 07:58:08.366  INFO 34103 --- [enerContainer-1] JMSAudit                                 :
===================================
[BEFORE]
Method: onMessage
Params:
> arg0: ActiveMQTextMessage {commandId = 5, responseRequired = true, messageId = ID:pivotal-es.local-64812-1479394688145-4:2:1:1:1, originalDestination = null,
===================================
```

Figure 4-4. *Logs*

Figure 4-4 shows the onMessage method being called by consuming the message from the queue (remember that these logs are produced by the AOP aspect). If you take a closer look at the actual message, you should see something similar to this:

```
ActiveMQTextMessage {
    commandId = 5,
    responseRequired = true,
    messageId = ID:pivotal-es.local-64812,
    originalDestination = null,
    originalTransactionId = null,
    producerId = ID:pivotal-es.local-64812,
    destination = queue://jms-demo,
    transactionId = null,
    expiration = 0,
    timestamp = 1479394688357,
    arrival = 0,
    brokerInTime = 1479394688357,
    brokerOutTime = 1479394688362,
    correlationId = null,
    replyTo = null,
    persistent = true,
    type = null,
    priority = 4,
   groupID = null,
   groupSequence = 0,
   targetConsumerId = null,
   compressed = false,
   userID = null,
   content = null,
   marshalledProperties = null,
   dataStructure = null,
   redeliveryCounter = 0,
   size = 1046,
   properties = null,
   readOnlyProperties = true,
   readOnlyBody = true,
   droppable = false, j
   msXGroupFirstForConsumer = false,
   text = Hello World
}
```

As you can see, we are receiving an ActiveMQTextMessage that is an implementation around the javax.jms.TextMessage interface. It's important to know the destination property that points to the jms-demo queue, the payload, and the text property with its Hello World value.

You might wonder if there is a simpler way to configure the listener. How can you determine what to configure? Well, Spring Boot makes this even easier. You learn about this process in the next section.

Consumer with Annotations

The Spring Framework provides useful annotations for consuming messages, very similar to `ApplicationEvents` and `Streams`. Spring Boot helps auto-configure these annotations, thereby making it easier for the developer.

Let's first review the `com.apress.messagin.jms.AnnotatedReceiver.java` class. See Listing 4-10.

Listing 4-10. com.apress.messagin.jms.AnnotatedReceiver.java

```
@Component
public class AnnotatedReceiver {

    @JmsListener(destination = "${apress.jms.queue}")
    public void processMessage(String content) {

    }

}
```

Listing 4-10 shows you the `AnnotatedReceiver` class that includes the `@JmsListener` annotation:

- `@Component`: Remember, this is a marker for Spring to enable this bean in the Spring container.
- `@JmsListener`: This annotation is configured to create a message listener using the destination specified by the *SpEL* (Spring Expression Language) expression. In this case, it's the `apress.jms.queue` property that has the `jms-demo` value.

That's it! Spring Boot will configure everything for you, so no more beans to declare a message listener container.

■ **Note** Before you run this receiver, first comment out the bean definition (`DefaultMessageListenerContainer`) in the `JMSConfig` class and the `@Component` in the `QueueListener` class.

It's now time to run this receiver (just remember to comment out the bean and listener, because you don't need them anymore). Once you run the application, you should have something similar to Figure 4-5.

```
2016-11-17 10:53:55.835  INFO 36359 --- [  restartedMain] JMSAudit
==================================
[BEFORE]
Method: sendMessage
Params:
> arg0: jms-demo
> arg1: Hello World
==================================
2016-11-17 10:53:55.870  INFO 36359 --- [enerContainer-1] JMSAudit
==================================
[BEFORE]
Method: processMessage
Params:
> arg0: Hello World
==================================
```

Figure 4-5. *@JmsListener logs*

Figure 4-5 shows you the logs after running the application. You can see that the method that's called is `processMessage`; this is the same message that was annotated by the `@JmsListener` annotation.

Currency Project

Let's talk about the currency project again. Imagine that you have a customer who wants to send a more accurate rate, but can only send rate messages using JMS. This means that your currency project needs to be a receiver, but the customer will need some kind of acknowledgement that you received the rate message.

Let's start by creating the sender client using the same `jms-sender` project. Review the `com.apress.messaging.jms.RateSender.java` class shown in Listing 4-11.

Listing 4-11. com.apress.messaging.jms.RateSender.java

```
@Component
public class RateSender {

    private JmsTemplate jmsTemplate;

    @Autowired
    public RateSender(JmsTemplate jmsTemplate){
        this.jmsTemplate = jmsTemplate;
    }

    public void sendCurrency(String destination, Rate rate){
        this.jmsTemplate.convertAndSend(destination,rate);
    }
}
```

Listing 4-11 shows you the class you are going to use to send new Rate objects. As you can see, it's very similar to the SimpleSender class in Listing 4-6. The only difference is that instead of sending a text (String) message, it's now sending a Rate object. Take a look at the com.apress.messaging.domain.Rate.java class in Listing 4-12.

Listing 4-12. com.apress.messaging.domain.Rate.java

```
public class Rate {

    private String code;
    private Float rate;
    private Date date;

    public Rate() { }

    public Rate(String base, Float rate, Date date) {
        super();
        this.code = base;
        this.rate = rate;
        this.date = date;
    }

    //Setters and Getter omitted.

}
```

Listing 4-12 shows you the Rate domain class that you have already seen in the currency project, but remember that in previous chapter, we annotated it with @Entity and @Id as part of the JPA persistence. This time it will just be simple, because there is no need to persist the rate.

If you try to run the app at this point, you will get an error that says something like:

```
Cannot convert object of type [com.apress.messaging.domain.Rate] to JMS message.
Supported message payloads are: String, byte array, Map<String,?>, Serializable object.
```

We can implement Serializable into the Rate class, but the currency project doesn't have that. Now, if you remember, the idea was to create a REST API that accepts the JSON format, so let's see how can we use JSON here.

Open the com.apress.messaging.config.JMSConfig.java class. See Listing 4-13.

Listing 4-13. com.apress.messaging.config.JMSConfig.java

```
@Configuration
@EnableConfigurationProperties(JMSProperties.class)
public class JMSConfig {
```

```
    @Bean
    public MessageConverter jacksonJmsMessageConverter() {
        MappingJackson2MessageConverter converter = new
                    MappingJackson2MessageConverter();
        converter.setTargetType(MessageType.TEXT);
        converter.setTypeIdPropertyName("_class_");
        return converter;
    }
}
```

Listing 4-13 shows you the configuration needed to expose the message in the JSON format. Let's review it:

- `MessageConverter`: This is an interface that specifies a converter between Java objects and JMS messages. It exposes the `toMessage` and `fromMessage` methods. The Spring Boot auto-configuration will wire everything up to use this particular message converter.
- `MappingJackson2MessageConverter`: This class implements the `MessageConverter` interface and adds more methods to help with the conversion from/to JSON/object.
- `setTargetType`: This method is necessary to help the converter identify what type is needed to convert from/to. In this case, we are sending a JSON in a String format, which means that it will create a `TextMessage` object behind the scenes.
- `setTypeIdPropertyName`: This is an *important* setting, because it will drive the way the mapping is done behind the scenes. This can be any value you want. It's just a simple identifier that will hold the qualified class name that is being mapped.

That's enough to run the application again, but first let's see how the main app will look. See Listing 4-14.

Listing 4-14. com.apress.messaging.JmsSenderApplication.java

```
@SpringBootApplication
public class JmsSenderApplication {

    public static void main(String[] args) {
        SpringApplication.run(JmsSenderApplication.class, args);
    }

    @Bean
    CommandLineRunner process(JMSProperties props,
                              RateSender sender){
        return args -> {
                sender.sendCurrency(props.getRateQueue(),
                     new Rate("EUR",0.88857F,new Date()));
                sender.sendCurrency(props.getRateQueue(),
                     new Rate("JPY",102.17F,new Date()));
```

```
                sender.sendCurrency(props.getRateQueue(),
                     new Rate("MXN",19.232F,new Date()));
                sender.sendCurrency(props.getRateQueue(),
                    new Rate("GBP",0.75705F,new Date()));
        };
    }
}
```

Listing 4-14 shows you the main app (just remember to comment out the previous code). As you can see, it's very simple. We are just sending the new `Rate` objects. If you run the application, you should get the output shown in Figure 4-6.

```
2016-11-17 18:49:07.521  INFO 43138 --- [  restartedMain] JMSAudit
==================================
[BEFORE]
Method: sendCurrency
Params:
> arg0: rates
> arg1: Rate [code=MXN, rate=19.232, date=2016-11-17]
==================================
```

Figure 4-6. *Logs when JMS sends a rate object*

Figure 4-6 shows you the logs without any errors. It's saying that everything went well and some rates were sent successfully. But this is not enough, because how can we guarantee that actually is a JSON format?

Using a Remote Apache ActiveMQ Broker

Let's use ActiveMQ as a remote broker. It will help us determine if the messages are actually going over to the broker instead of using the in-memory provider.

Make sure you have ActiveMQ up and running (you need to download it from `http://activemq.apache.org/` and follow the installation instructions for your system). I used ActiveMQ version 5.14.0, but you can use any version. Before you run the application again, you need to make sure that it will be using the ActiveMQ broker that is running. Open the `src/main/resources /application.properties` file and uncomment out all the `spring.activemq.*` and the `apress.jms.*` properties. The result should look like Listing 4-15.

Listing 4-15. src/main/resources/application.properties

```
spring.activemq.broker-url=tcp://localhost:61616
spring.activemq.user=admin
spring.activemq.password=admin

#Apress Configuration
apress.jms.queue=jms-demo
apress.jms.rate-queue=rates
```

Listing 4-15 shows the `application.properties` file where the remote ActiveMQ is declared (in this case, it's the local system), the default port is `61616`, and the username and password are set to `admin`. With these properties in place, Spring Boot will configure the `connectionFactory` that will connect to the remote broker.

So, if you have the ActiveMQ broker running, you should be able to go to the `http://localhost:8161/admin` URL in your browser and see the web page shown in Figure 4-7.

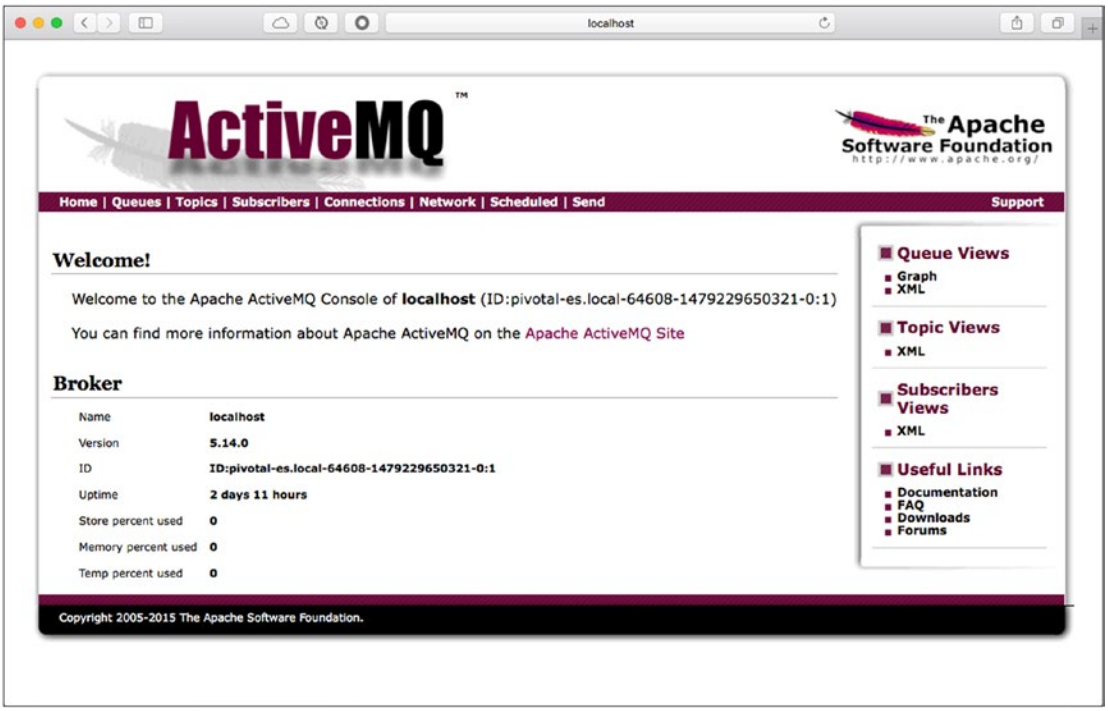

Figure 4-7. *ActiveMQ web console: http://localhost:8161/admin*

Then you can execute the `jms-demo` application and select the queue from the web console to see that the queue called `rates` has been created (this name was based on the `apress.jms.rate-queue` property value). See Figure 4-8.

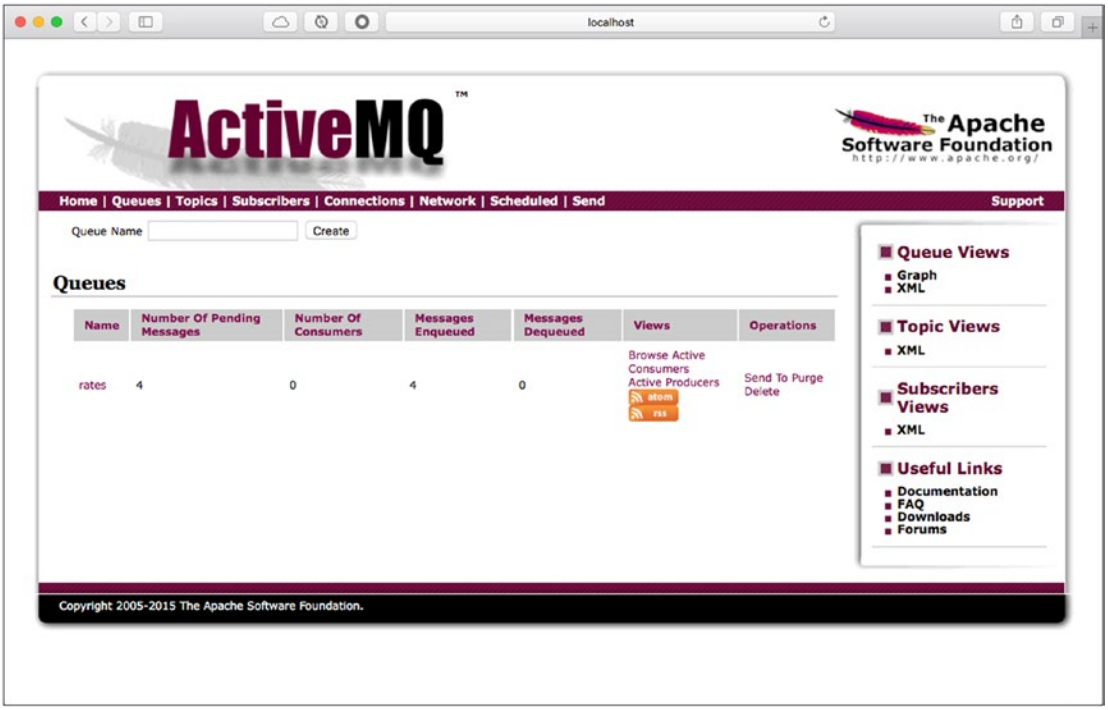

Figure 4-8. *ActiveMQ Queues tab*

If you click the queue called `rates`, you should see the four messages that were sent to the broker. See Figure 4-9.

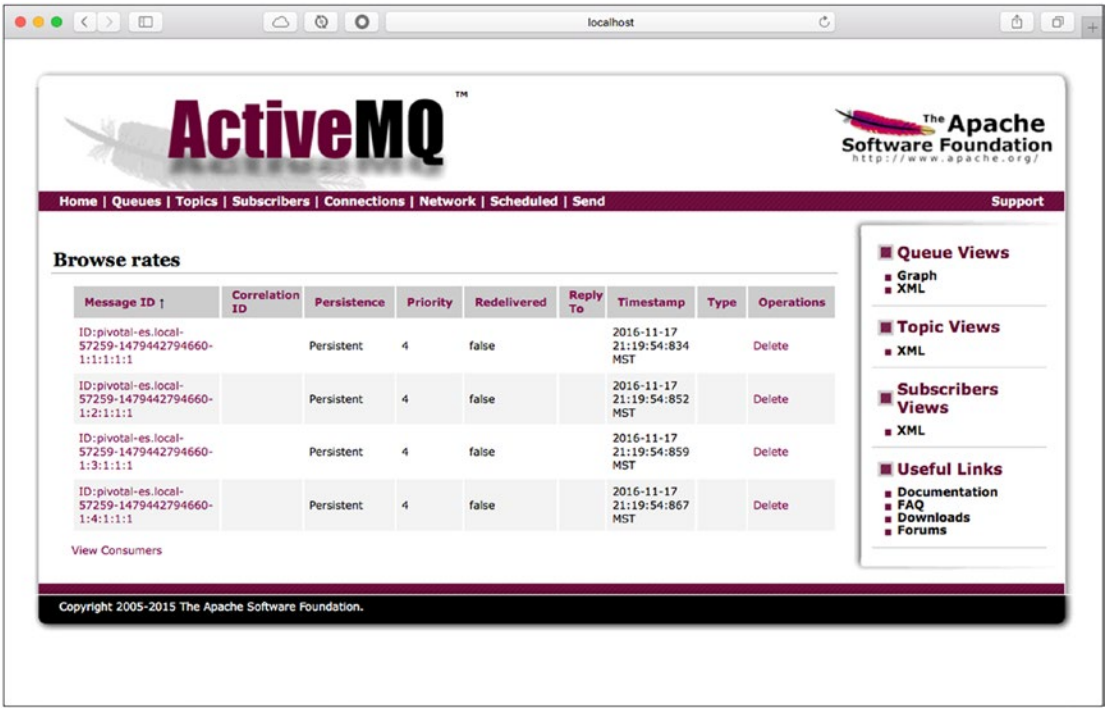

Figure 4-9. *ActiveMQ queue rate messages*

Click in any of the messages to see the content. See Figure 4-10.

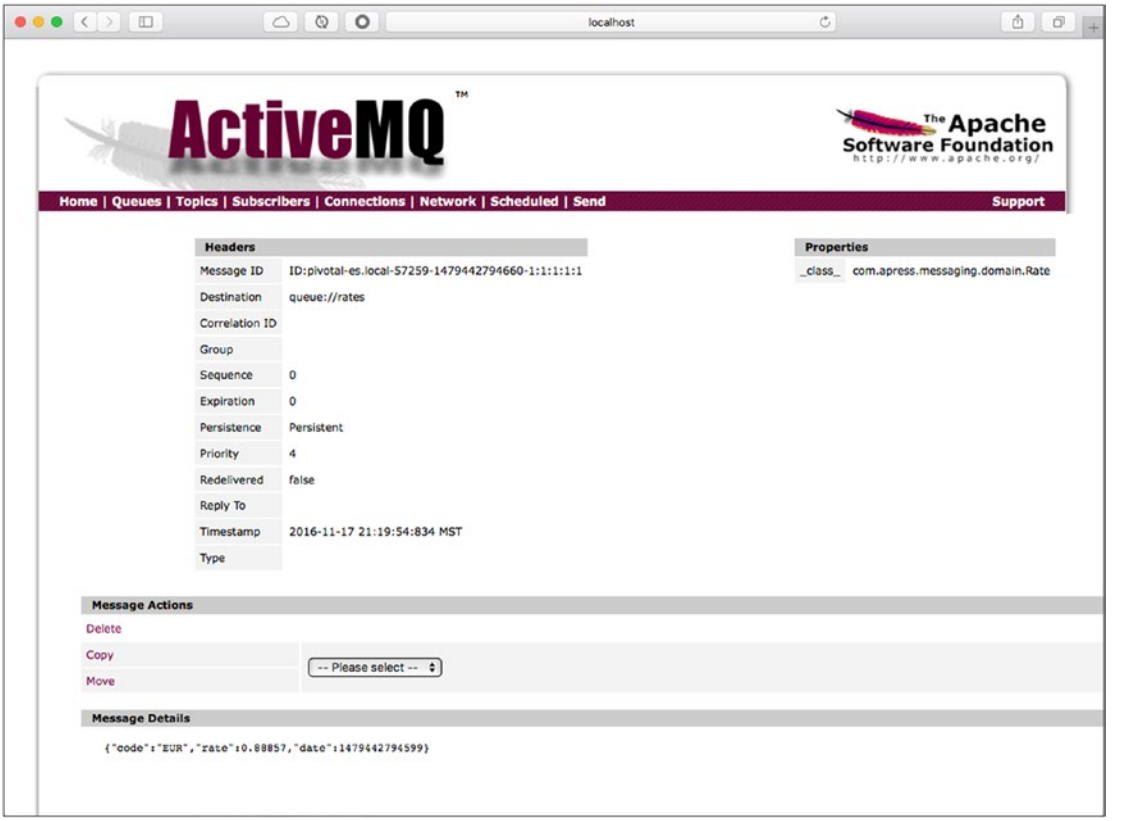

Figure 4-10. *ActiveMQ Queue --> Rates --> Message*

Figure 4-10 shows you the actual message received by the broker. Take a look at the message details; you will see something similar to this:

```
{"code":"EUR","rate":0.88857,"date":1479442794599}
```

The rate is sent by the `jms-sender` app. On the right, you can see a Properties legend where the
`_class_` property ID has the value of the `com.apress.messaging.domain.Rate` class (remember that this is important for the receiver as well, so it can convert back from JSON to object).

Now you know how to send objects that can be converted into JSON. Next, you need to receive this message, right? The sender needs to receive some acknowledgement from the receiver (it was part of the requirements) as well.

Reply-To

Spring JMS provides a way to reply to another queue, kind of like having a response-request/RPC model. You use it along with the listener, the `@SendTo` annotation.

In order to see this in action, we are going to work with the same `jms-sender` project. Remember to disable some of the components. You do this by commenting out the `@Component` annotations in all the listeners we have been working on.

Open the `com.apress.message.jms.RateReplyReceiver.java` class. See Listing 4-16.

Listing 4-16. com.apress.message.jms.RateReplyReceiver.java

```
@Component
public class RateReplyReceiver {

    @JmsListener(destination = "${apress.jms.rate-queue}")
    @SendTo("${apress.jms.rate-reply-queue}")
    public Message<String> processRate(Rate rate){
        //Process the Rate and return any significant value
        return MessageBuilder
                .withPayload("PROCCESSED")
                .setHeader("CODE", rate.getCode())
                .setHeader("RATE", rate.getRate())
                .setHeader("ID", UUID.randomUUID().toString())
                .setHeader("DATE",
                       new SimpleDateFormat("yyyy-MM-dd")
                               .format(new Date()))
                .build();
    }

}
```

Listing 4-16 shows you the `RateReplyReceiver.java` class and the `@SendTo` annotation, which needs a value that will correspond to the `reply-queue` where the result message will be sent. Let's review it:

- `@SendTo`: This annotation will be the key to a reply. Make sure you still have the `@JmsListener` annotation, meaning that this method will act as a receiver and a sender. The method must have a return type. In this case, it will use the `apress.jms.rate-reply-queue` value to set the reply queue. The annotation value can be omitted if the main message has the `JMSReplyTo` header set.
- `Message<T>`: This is an interface that is built on the `Generics` type and provides useful getters, like `getPayload` and `getHeaders`. This is the preferred way to send a message.
- `MessageBuilder`: This is a helper class that allows you to enhance the message by allowing you to add more headers.

Next let's look again at the RateSender class. It not only will send the rate, but also will be listening to the reply-queue. Remember that the requirement was to receive some kind of acknowledgement from the receiver. See Listing 4-17.

Listing 4-17. com.apress.message.jms.RateSender.java

```
@Component
public class RateSender {

    private JmsTemplate jmsTemplate;

    @Autowired
    public RateSender(JmsTemplate jmsTemplate){
        this.jmsTemplate = jmsTemplate;
    }

    public void sendCurrency(String destination, Rate rate){
        this.jmsTemplate.convertAndSend(destination,rate);
    }

    @JmsListener(destination="${apress.jms.rate-reply-queue}")
    public void process(String body,@Header("CODE") String code){

    }
}
```

Listing 4-17 shows you the new RateSender class. As you can see, we are reusing the @JmsListener and and new @Header annotation. Let's review it:

- @JmsListener: You already know this annotation. The only difference is to include the correct name of the reply-queue. In this case, it's the apress.jms.rate-reply-queue property value.
- @Header: This annotation will provide you with direct access to the header of the Message<T>, and in this case it will be the rate code that was processed. Spring JMS has more options: @Headers brings a java.util.Map object, @Payload brings the actual payload, and @Valid turns on validation for your payload.

Before you run this, make sure you have the ActiveMQ broker up and running.

If you run the application you should have not only the sender, but also the receiver and the reply logs. See Figure 4-11.

```
2016-11-18 11:00:42.474  INFO 49565 --- [enerContainer-1] JMSAudit
==================================
[BEFORE]
Method: processRate
Params:
> arg0: Rate [code=JPY, rate=102.17, date=2016-11-18]
==================================
2016-11-18 11:00:42.478  INFO 49565 --- [enerContainer-1] JMSAudit
==================================
[BEFORE]
Method: process
Params:
> arg0: PROCCESSED
> arg1: EUR
==================================
```

Figure 4-11. *Logs with the reply-to queue*

You can take a look at the ActiveMQ web console and see that the `reply-rate` queue was created. See Figure 4-12.

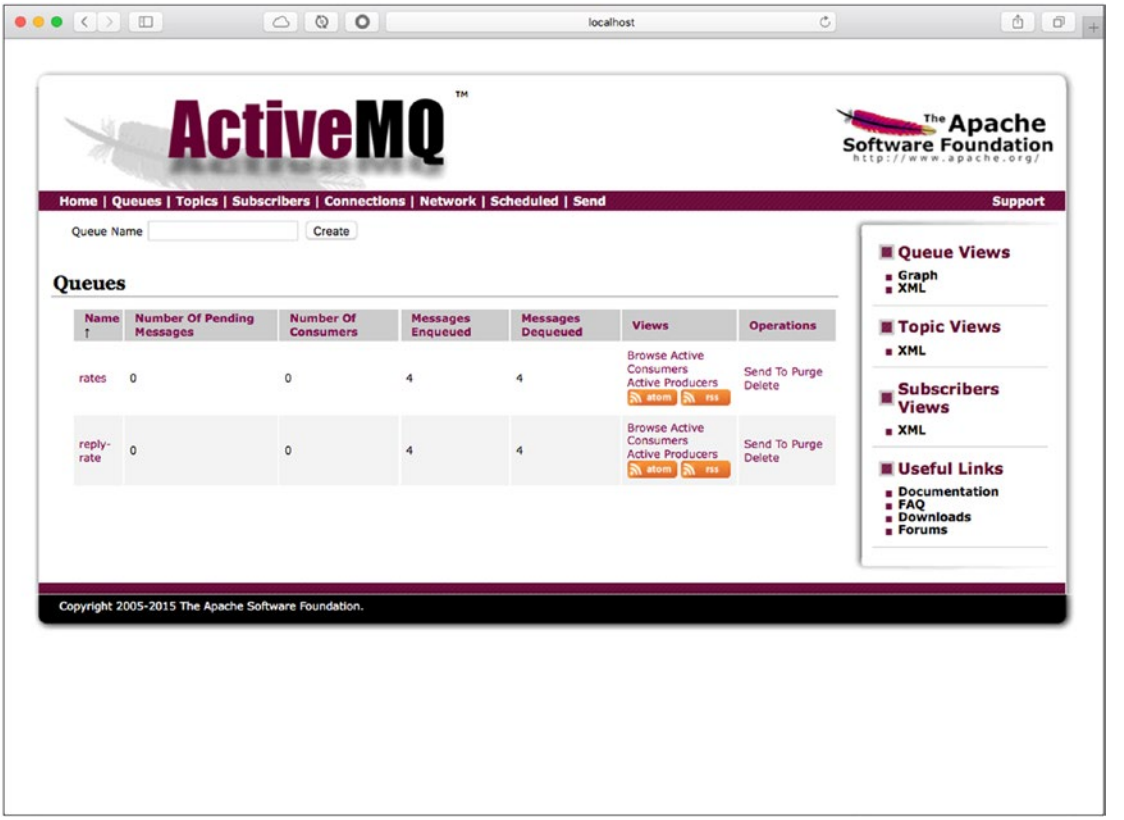

Figure 4-12. *Queues showing the reply-rate queue*

Now you know how to create a `reply-to` and have a kind of RPC mechanism for your applications.

Topics

This section discusses the next JMS messaging model, the publisher-subscriber or topics. This model has publishers that send messages to a topic and these topics can be from zero for multiple subscribers that will get a copy of the message. You need to think of this like a newspaper or a magazine subscription. You subscribe (to a particular interest—a topic) to receive a newspaper or magazine from the publisher.

This section continues with the `jms-sender` project, which will be the publisher. It also opens a new project within this chapter, the `jms-topic-subscriber` project. It's similar in structure to `jms-sender`.

In `jms-sender`, we are going to use the same `RateSender.java` and the main entry point where we send the rates, but there is one small change. Open the `src/main/resources/application.properties` file. It should look like Listing 4-18.

Listing 4-18. src/main/resources/application.properties

```
# Spring Web
spring.main.web-environment=false

#Default ActiveMQ properties
spring.activemq.broker-url=tcp://localhost:61616
spring.activemq.user=admin
spring.activemq.password=admin

#Apress Configuration
apress.jms.queue=jms-demo
apress.jms.rate-queue=rates
apress.jms.rate-reply-queue=reply-rate

#Enable Topic Messaging
spring.jms.pub-sub-domain=true

#Apress Topic Configuration
apress.jms.topic=rate-topic
```

Listing 4-18 shows the `application.properties` file, which has one particular property, the `spring.jms.pub-sub-domain`. This property by default is `false`, thereby making your producer send messages to a queue. When it's set to `true`, the producer (publisher) will send the message to a topic. This also applies to the listeners. If you set this property to `true`, the listeners will become subscribers of the topic.

Look at the last property as well. We are simply defining the name of the topic where all the subscribers will be listening.

You can now open and review the `com.apress.messaging.jms.RateTopicReceiver.java` class from the `jms-topic-subscriber` project and see that it's the same code (except for the name of the destination). In this project, you need to have the same `application.properties` file with the `spring.jms.pub-sub-domain` set to `true` (just make sure it has this property).

Now it's time to run the `jms-topic-subscriber` project. Before you run the `jms-demo` project, look at the Topics section of the ActiveMQ web console, as shown in Figure 4-13.

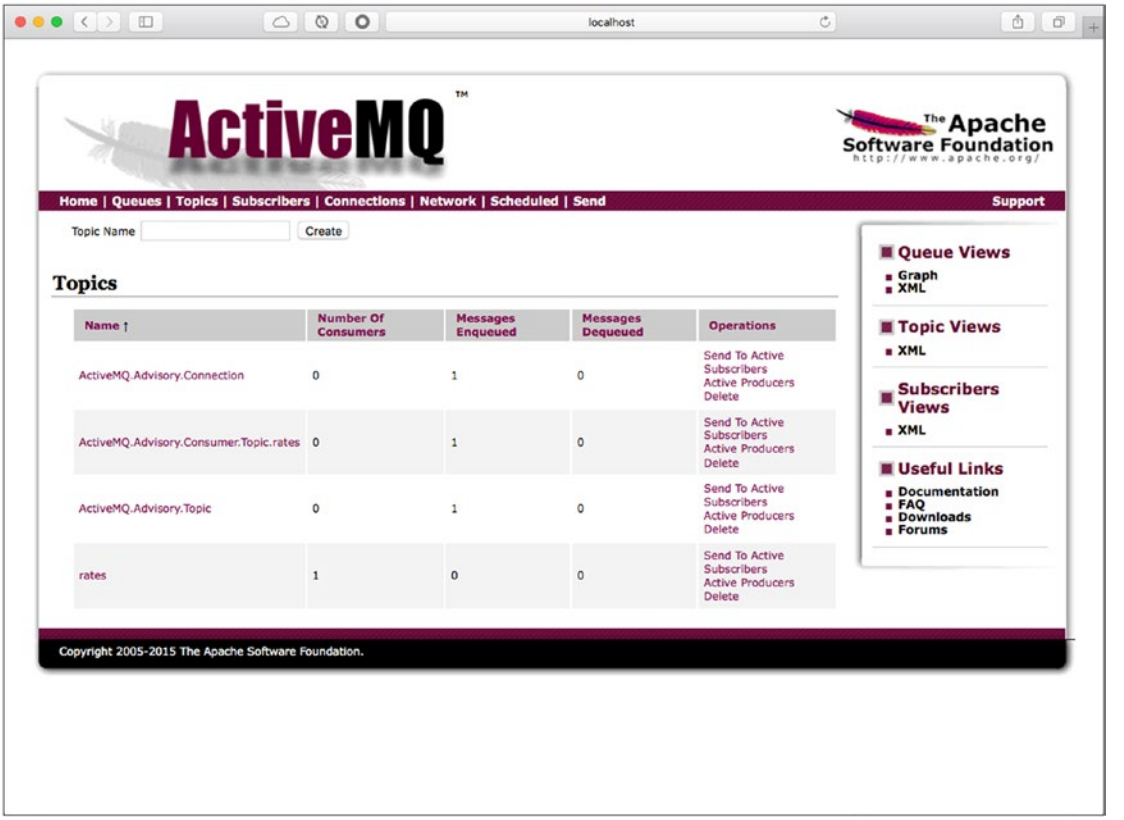

Figure 4-13. *The Topics section of the ActiveMQ web console*

Figure 4-13 shows that a topic called `rates` was created and there is a one consumer. Next, run the `jms-sender` project to see whether the rates are sent to the topic. Take a look at the `jms-topic-subscriber` project logs; you should see that you are consuming from the `rates` topic.

As an experiment, you can run multiple instances of the `jms-topic-subscriber` project (this is easy if you are using the STS and the Boot Dashboard) and verify that each one of them gets a copy of the rate. See Figure 4-14.

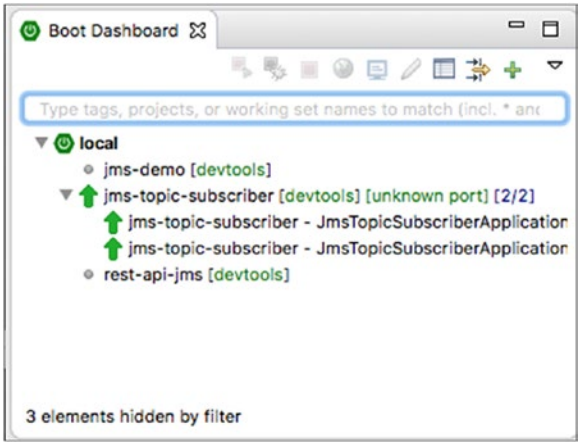

Figure 4-14. *STS Boot Dashboard running two instances of the jms-topic-subscriber project*

Currency Project

What do we need to do in order to use the currency project and start listening for new rates from other clients? The solution is already in the `rest-api-jms` project. This is what you need to do:

- Make sure you have the `application.properties`, `spring.activemq.*`, and `rate.jms.*` properties enabled.
- Review the `RateJmsReceiver` class and comment/uncomment out the listener you want to use (we have the simple listener and the listener with the `reply-to`).
- Review the `RateJmsConfiguration` class that has the JSON converter.

As homework, try to make it run. Remember that you will need to have the ActiveMQ broker up and running. Also, as an extra step, try to make this project topic-aware.

Summary

This chapter showed you how to use Spring Boot to send and receive messages using the JMS technology.

You saw the different JMS messaging models and what developers need to do to send or receive messages.

With Spring Boot, you saw how easy is to set up a JMS client and how, with simple annotation, you can have a functional application that uses JMS as a messaging system. Even though you just saw how to use Apache ActiveMQ as the broker, the same programming can be applied to HornetQ, IBM MQ, etc., just by providing the correct properties (connection factories and message listeners) in the `application.properties` file.

The next chapter discusses a different way to do messaging, using the Advance Messaging Queuing Protocol—AMQP.

CHAPTER 7

Web Messaging

This chapter covers WebSockets with Spring Boot and describes how this technology can help you implement messaging across apps or even across multiple instances of the same application.

When talking about web applications, we can say that REST is another way to do messaging, and it is. In this chapter, we are going to focus on a stateful way of communicating, which is what WebSockets brings to the table.

WebSockets

WebSockets is a protocol that enables two-way communication, and it's normally used in web browsers. This protocol starts by using a handshake (normally a HTTP request) and then sends a basic message frame (a protocol switch) over TCP. The idea of the WebSockets is to avoid multiple HTTP connections like the AJAX (`XMLHttpRequest`) or the iframe and the long polling. See Figure 7-1.

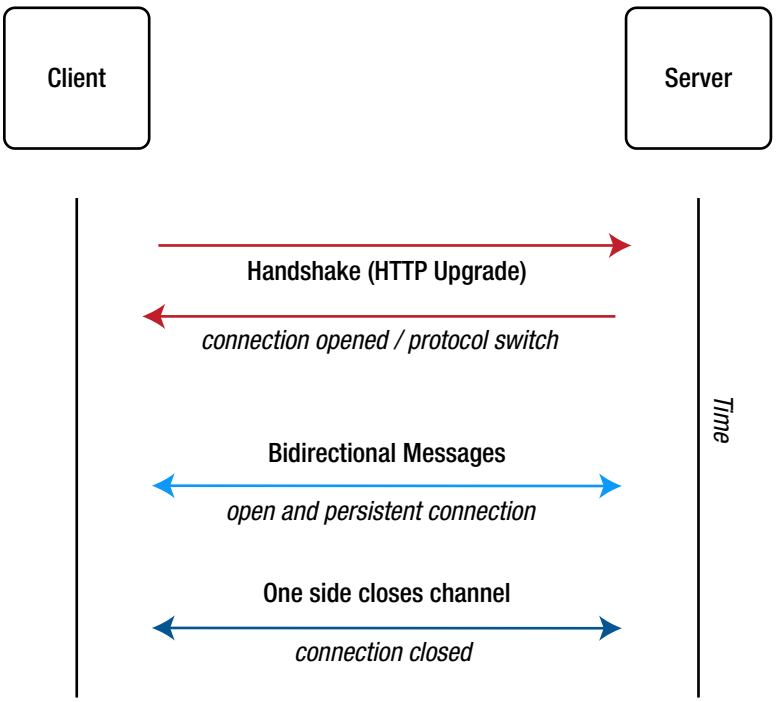

Figure 7-1. *TCP/WebSockets*

F. Gutierrez, *Spring Boot Messaging*, DOI 10.1007/978-1-4842-1224-0_7

Using WebSockets with Spring

Before we get into how to use WebSockets with Spring Boot, it's important to know that not all the browsers support this technology. Visit `http://caniuse.com/websockets` to find out which browsers are WebSockets-ready.

The Spring Framework version 4 includes a new `spring-websocket` module that supports WebSockets; it's also compatible with the Java specification JSR-356. This module also has fallback options that simulate the WebSockets API when necessary (remember that not all the browsers support WebSockets). It uses the SockJS protocol for this task. You can get more information about it at `https://github.com/sockjs/sockjs-protocol`.

It's also worth mentioning that after the initial handshake (the HTTP, where we use SockJS), the communications switch to a TCP connection (meaning that you are sending just a stream of bytes—either text or binary). Therefore, you can use any type of messaging architecture, like async or event-driven messaging. At this level, you can use sub-protocols like STOMP (Simple/Streaming Text Oriented Message Protocol), which allows you to have a better messaging format that the client and server can understand. See Figure 7-2.

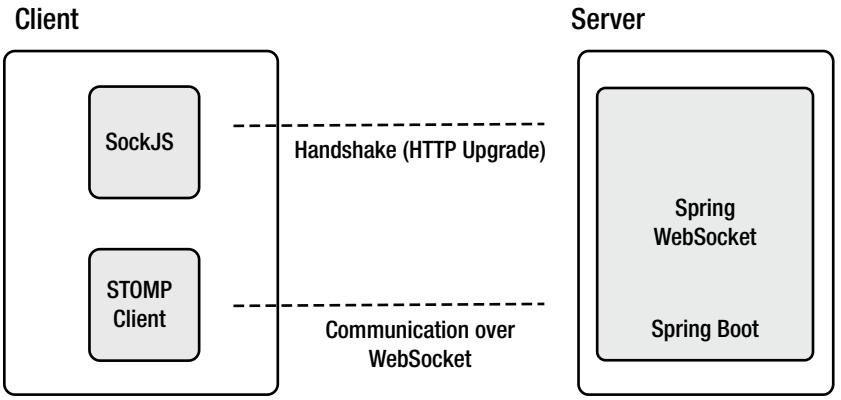

Figure 7-2. *WebSockets with Spring*

Figure 7-2 shows how to implement WebSockets using Spring and the components needed on the client side.

Low-Level WebSockets

Before we jump into the fallback options (SockJS) and sub-protocols (STOMP), let's see how can we use Spring Boot with a low-level WebSockets.

In the book's source code, go to the Chapter 7 and open the two projects (you can import them into your favorite IDE or use any text editor). In the following section, we are going to work with the `websocket-demo` project.

Let's start by analyzing the configuration. In order to use low-level WebSockets communications, we need to implement the `org.springframework.web.socket.config.annotation.WebSocketConfigurer` interface. Open the `com.apress.messaging.config.LlWebSocketConfig` class, as shown in Listing 7-1.

Listing 7-1. com.apress.messaging.config.LlWebSocketConfig.java

```
@Configuration
@EnableWebSocket
public class LlWebSocketConfig implements WebSocketConfigurer{

    LlWebSocketHandler handler;

    public LlWebSocketConfig(LlWebSocketHandler handler){
        this.handler = handler;
    }

    @Override
    public void registerWebSocketHandlers(
                            WebSocketHandlerRegistry registry) {
                registry.addHandler(this.handler, "/llws");
    }

}
```

Listing 7-1 shows you the configuration needed to enable WebSockets in Spring. Remember, the way we are configuring this class is to have a low-level WebSockets communication. Let's review the components that are used:

- `@EnableWebSocket`: This is necessary to enable and configure WebSockets requests.
- `WebSocketConfigurer`: This is an interface that defines callback methods to configure the WebSockets request handling. Normally you are required to implement the `registerWebSocketHandlers` method.
- `registerWebSocketHandlers`: This method needs to be implemented by adding the handlers that will be used for the WebSockets processing request. In this method, we are registering a handler (`LlWebSocketHandler`) instance and passing the endpoint used for the handshake and communication.
- `WebSocketHandlerRegistry`: This an interface used to register a `WebSocketHandler` implementation. We are going to use a `TextWebSocketHandler` implementation and look at its code in the next section.

As you can see, it's very simple to configure a low-level WebSockets using Spring. Next, let's open the `com.apress.messaging.web.socket.LlWebSocketHandler` class. See Listing 7-2.

Listing 7-2. com.apress.messaging.web.socket.LlWebSocketHandler.java

```
@Component
public class LlWebSocketHandler extends TextWebSocketHandler{

    @Override
    public void afterConnectionEstablished(
                 WebSocketSession session) throws Exception {
        super.afterConnectionEstablished(session);
    }

    @Override
    protected void handleTextMessage(
              WebSocketSession session, TextMessage message)
                                              throws Exception {
                System.out.println(">>>> " + message);
        }
}
```

Listing 7-2 shows you the WebSockets handler we are going to use to receive messages from the client. Let's examine this class:

- `TextWebSocketHandler`: This is a concrete class that implements the `WebSocketHandler` interface through the `AbstractWebSocketHandler` class. This implementation is for processing text messages only. We are going to override two methods: `afterConnectionEstablished` and `handleTextMessage`.
- `afterConnectionEstablished`: This method is called when the client successfully connects using the WebSockets protocol. In this method, you can use the WebSocketSession instance to send or receive messages. For now, we are going to use its default behavior, but we will see more in the logs (through the AOP `WebSocketsAudit` class).
- `handledTextMessage`: This method receives a `WebSocketSession` (that we are going to use later) and a `TextMessage` instance. The `TextMessage` instance manages the stream of bytes and converts them into a string.

So far, we have implemented the server side, but what about the client? Normally, you will have a web page to do the client's job.

Open the `src/main/resources/static/llws.html` file, as shown in Listing 7-3.

Listing 7-3. Snippet of llws.html

```
<script>

$(function(){

        var connection = new
                WebSocket('ws://localhost:8080/llws');
```

```
        connection.onopen = function () {
                console.log('Connected...');
        };

        connection.onmessage = function(event){
                console.log('>>>>> ' + event.data);
                var json = JSON.parse(event.data);
                $("#output").append("<span><strong>"
                       + json.user
                       + "</strong>: <em>"
                       + json.message
                       + "</em></span><br/>");
       };

       connection.onclose = function(event){
              $("#output").append("CONNECTION: CLOSED");
       };

       $("#send").click(function(){
                var message = {}
                message["user"] = $("#user").val();
                message["message"] = $("#message").val();

       connection.send(JSON.stringify(message));
       });

});

</script>
```

Listing 7-3 shows just the JavaScript snippet of the important code, which I explain next:

- `WebSocket`: This is part of the JavaScript engine and it will connect to the specified URI. Note that the schema is `ws` and that we are using the `/llws` endpoint, which we specified in the configuration class (see Listing 7-1).
- `onopen`: This is a callback function that is executed when the connection is established with the server. Note that we are only logging a string to the console.
- `onmessage`: This is a callback function that is executed when a message is received from the server. In this case, we are parsing an `event.data` into a JSON object.
- `onclose`: This is a callback function that is executed when the connection is closed or is lost with the server.
- `$.click/send`: This is a callback function that is attached to the Send button, and it's called when the button is clicked. Here we are using the `send` method, which will send a JSON string of the message object.

Next, let's run the `websocket-demo` project; after it starts, open a browser and go to `http://localhost:8080/llws.html`. You should see something similar to Figure 7-3.

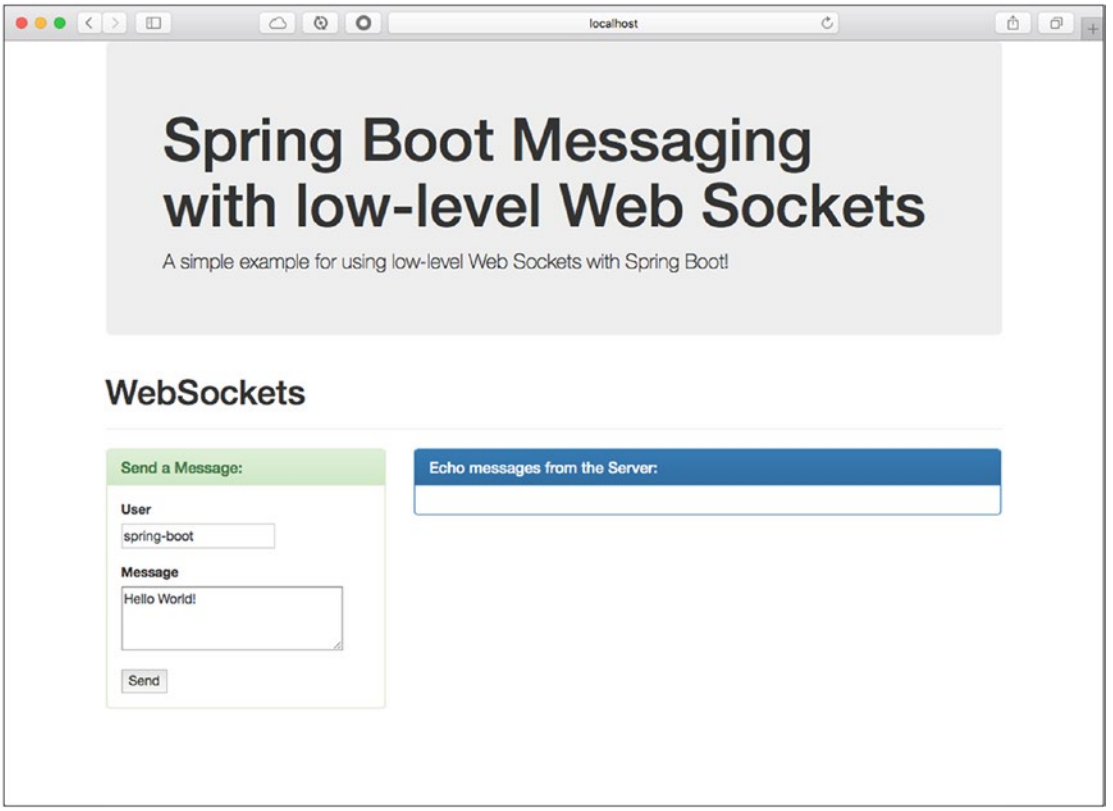

Figure 7-3. *http://localhost:8080/llws.html*

After you go to `llws.html` in your browser, take a look at the application logs. You should see something similar to Figure 7-4.

```
===================================
2017-02-17 13:05:48.002  INFO 73029 --- [nio-8080-exec-5] c.apress.messaging.aop.WebSocketsAudit   :
===================================
[BEFORE]
 Class: com.apress.messaging.web.socket.LlWebSocketHandler
Method: afterConnectionEstablished
Params:
> arg0: StandardWebSocketSession[id=1, uri=/llws]

[AFTER]
Return: void/null
===================================
```

Figure 7-4. *The websocket-demo project logs*

Figure 7-4 shows you the logs, whereby the `afterConnectionEstablished` is being called from the `LlWebSocketHandler` class. This means that the client was successfully connected to the server. You can also take a look at the browser's developer console to see the logs displaying the string `Connected...`. See Figure 7-5.

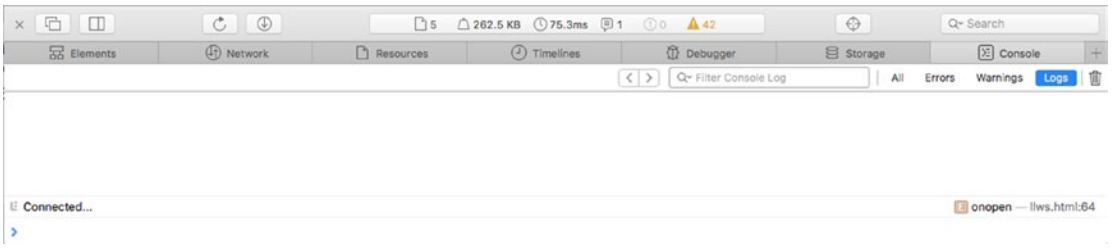

Figure 7-5. *Browser's console*

Next, you can send a message by modifying the user and message inputs from the `llws.html` page and clicking the Send button. After clicking the Send button, you should see something similar to Figure 7-6.

```
>>>> TextMessage payload=[{"user":"s..], byteCount=43, last=true]
```

Figure 7-6. *websocket-demo project logs after clicking Send*

Figure 7-6 shows you the logs after sending the message as well as the print out from the `handleTextMessage` method.

This example shows you how to send a message from the client to the server. Now, let's do a response, an echo, from the server. You need to add something to `handleTextMessage` to handle the reply.

Modify `handleTextMessage` to look like the following:

```
@Override
public void handleTextMessage(
                WebSocketSession session,
                TextMessage message) throws Exception {

        System.out.println(">>>> " + message);

        session.sendMessage(message);
}
```

As you can see, we are using the `session` instance to call `sendMessage`, which means that we are going to reply using the same *session* of the connected client.

Restart the `websocket-demo` project and refresh the `llws.html` page. Then send a message by filling out the user and message inputs. You should see the response in the Echo Messages from the Server panel, as shown in Figure 7-7. Note that if you are using the STS, you only need to wait until the project restarts itself; this is possible thanks to the `spring-boot-devtools`.

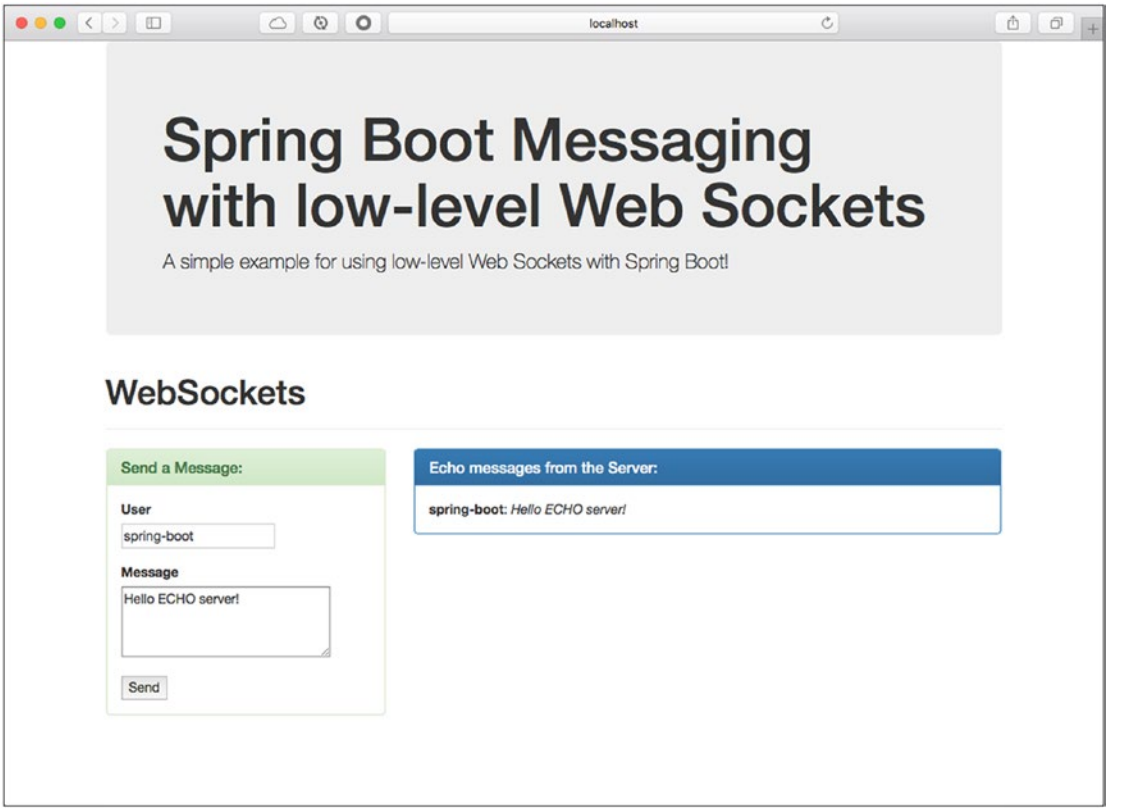

Figure 7-7. *Echo server response*

If you stop the application, you will see something like Figure 7-8 in the Echo Messages from the Server panel.

Figure 7-8. *Closed connection*

As you can see, it's very simple to create low-level WebSockets applications. What happens if you need to send a message from a Spring application? In other words, your app needs to be the client. You can add the following code to your application (see Listing 7-4).

Listing 7-4. WebSockets Client—Snippet of the WebSocketDemoApplication Class

```
StandardWebSocketClient client = new StandardWebSocketClient();
ListenableFuture<WebSocketSession> future =
                client.doHandshake(handler,
                       new WebSocketHttpHeaders(),
                       new URI("ws://localhost:8080/ws-server"));
WebSocketSession session = future.get();
WebSocketMessage<String> message =
                new TextMessage("Hello there...");
session.sendMessage(message);
```

Listing 7-4 shows a snippet of what you need to add if you want to create a WebSockets client (instead of HTML web pages). Here we are using `StandardWebSocketClient` (a low-level WebSockets protocol) and it's manually doing the handshake. It passes the handler some headers and the URI where you will connect. (Note that you can use the previous handler or create your own, and you then can implement the `afterConnectionEstablished` to check if you were successfully connected.) Then you get a `WebSocketSession` instance and can send the message.

You can see that using the WebSockets client with Spring classes is a straightforward implementation. Next, let's start using the fallback options with SockJS and the STOMP sub-protocol.

Using SockJS and STOMP

Why do we need to use SockJS and STOMP? Remember that not all the browsers support WebSockets and normally the client and the server must agree on how they will handle messages. Of course, that is not the right way to message, because we want to have a decoupling scenario, where the client is not tied to the server.

SockJS helps emulate WebSockets and performs the initial handshake. Then, by using STOMP, we can reply in an interoperable wire format that allows us to use multiple brokers that support this protocol.

Chat Room Application

We are going to continue using the `websocket-demo` project, but we are going to work with different files and classes. This example creates a chat room, which is a very common use of the WebSockets technology.

First, let's see how to configure the project will use SockJS and STOMP. Open the `com.apress.messaging.config.WebSocketConfig` class. See Listing 7-5.

Listing 7-5. com.apress.messaging.config.WebSocketConfig.java

```
@Configuration
@EnableWebSocketMessageBroker
@EnableConfigurationProperties(SimpleWebSocketsProperties.class)
public class WebSocketsConfig extends
                AbstractWebSocketMessageBrokerConfigurer {

    SimpleWebSocketsProperties props;

    public WebSocketsConfig(SimpleWebSocketsProperties props){
        this.props = props;
    }

    @Override
    public void registerStompEndpoints(
                            StompEndpointRegistry registry) {
                registry.addEndpoint(props.getEndpoint()).withSockJS();
    }

    @Override
    public void configureMessageBroker(
                            MessageBrokerRegistry config) {
        config.enableSimpleBroker(props.getTopic());
        config.setApplicationDestinationPrefixes(
                props.getAppDestinationPrefix());
    }
}
```

Listing 7-5 shows you the configuration needed to use WebSockets with SocksJS and STOMP. Let's review it:

- `@EnableWebSocketMessageBroker`: This annotation is required to enable broker-backend messaging over WebSockets using a higher-level messaging sub-protocols (SockJS/STOMP).
- `AbstractWebSocketMessageBrokerConfigurer`: This class implements the `WebSocketMessageBrokerConfigurer` to configure message handling with simple messaging protocols like STOMP from WebSockets clients.
- `registerStompEndpoints`: This method is called to register STOMP endpoints, mapping each to a specific URL and configuring the SockJS fallback options.
- `StompEndpointRegistry`: This is a contract for registering STOMP over WebSockets endpoints. It provides a fluent API to build the registry.
- `withSockJS`: This method will enable the SockJS fallback options required for the handshake.

- `configureMessageBroker`: This method is called to configure the message broker options. In this case we are using the `MessageBrokerRegistry`.
- `MessageBrokerRegistry`: This instance will help configure all the broker options. We are using `enableSimpleBroker`, which accepts one or more prefixes to filter destinations targeting the broker. This will be used together with the annotated methods that have the `@SendTo` annotation. We are also using the `setApplicationDestinationPrefixes` to configure one or more prefixes to filter destination targeting the application annotated methods. In other words, it will look for methods that are annotated with `@MessageMapping`.

You can see that we need to add an endpoint, a broker, and prefix paths to configure WebSockets using SockJS and STOMP. Take a look at Figure 7-9.

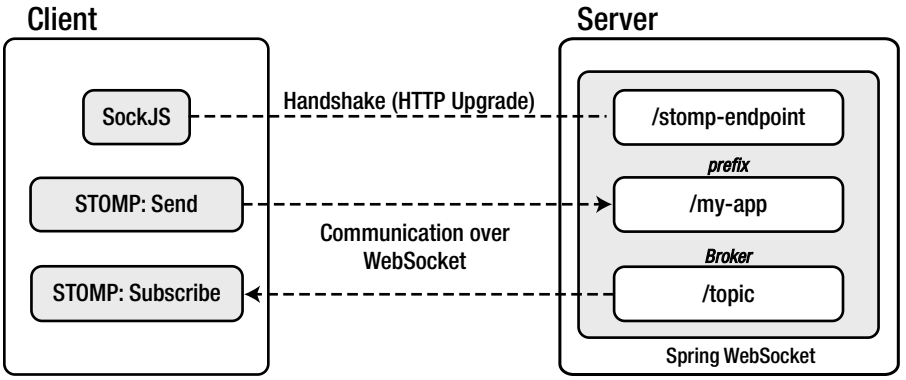

Figure 7-9. *Client/server WebSockets using SockJS and STOMP*

Figure 7-9 shows you a general picture of what the communication will be between the client and the server. Next, open the `com.apress.messaging.controller.SimpleController` class. See Listing 7-6.

Listing 7-6. com.apress.messaging.controller.SimpleController.java

```
@Controller
public class SimpleController {

    @MessageMapping("${apress.ws.mapping}")
    @SendTo("/topic/chat-room")
    public ChatMessage chatRoom(ChatMessage message) {
        return message;
    }
}
```

Listing 7-6 shows you the receiver and sender by using new annotations. Remember that the project now will run a chat room, so different clients can connect and receive message from other users:

- `@MessageMapping`: This annotation has to do with the application prefixes, meaning that the client needs to send a message to `prefix + mapping`. In our case, this is `/my-app/chat-room`. This annotation is supported by methods of the `@Controller` classes, and the value can be treated as an ant-style, slash-separated, and path patterns. With this annotation, you have other annotations that can be used as method parameters, including `@Payload`, `@Header`, `@Headers`, `@DestinationVariable`, and `java.security.Principal`.
- `@SendTo`: You already know this annotation; it's the same one we used in the JMS and RabbitMQ chapters. In this case, this annotation is used to send a message to any other destination.

Even though we didn't use the `@SubscribeMapping`, you can use it in a `@Controller` annotated class, in a method that you want to use to handle incoming messages. This will normally be useful when you need to get a copy of the response by the `@SendTo`.

Another utility class that we didn't use is `SimpleMessagingTemplate`. It can be used to send messages. For example:

```
@Controller
public class AnotherController {

    private SimpMessagingTemplate template;

    @Autowired
    public AnotherController(SimpMessagingTemplate template) {
        this.template = template;
    }

    @RequestMapping(path="/rate/new", method=POST)
    public void newRates(Rate rate) {
        this.template.convertAndSend("/topic/new-rate", rate);
    }

}
```

As you can see, it's very simple to implement the subscriber/publisher model using the WebSockets technology. `AnotherController` acts as a client as well by sending a message (`ChatMessage`).

Next, let's see another client. Open the `src/main/resources/static/sockjs-stomp.html` page. See Listing 7-7.

Listing 7-7. Snippet of sockjs-stomp.html

```
$(function(){
    var socket =
        new SockJS('http://localhost:8080/stomp-endpoint');
    var stompClient = Stomp.over(socket);

    stompClient.connect({}, function (frame) {
        console.log('Connected: ' + frame);

        stompClient.subscribe('/topic/chat-room',
               function (data) {
                    console.log('>>>>> ' + data);
                    var json = JSON.parse(data.body);
                    $("#output")
                       .append("<span><strong>"
                               + json.user
                               + "</strong>: <em>"
                               + json.message
                               + "</em></span><br/>");
                });

    });

    $("#send").click(function(){
        var chatMessage = {}
         chatMessage["user"] = $("#user").val();
         chatMessage["message"] = $("#message").val();

         stompClient.send(
                               "/my-app/chat-room",
                               {},
                               JSON.stringify(chatMessage));
                     });
    });
```

Listing 7-7 shows you the JavaScript client. Let's analyze it:

- `SockJS`: This is a JavaScript library that emulates the WebSockets protocol. This object is connecting to the `/stomp` endpoint that was specified on the server side. For more information about this library, visit http://sockjs.org/.
- `STOMP`: This is a JavaScript library that uses the WebSockets protocol and normally will require the SockJS object. For more information, visit http://jmesnil.net/stomp-websocket/doc/.
- `connect`: This is a callback that will be called when the connection is established with the server.

- `subscribe`: This is a callback that will be called when there is a message at the subscribed destination.
- `send`: This method will send a message to the destination provided.

This is a very simple way to use SockJS and STOMP in a JavaScript client. Now you are ready to run the `websockets-demo` project.

■ **Note** Before you run the `websocket-demo` project in this section, make sure that the `LlWebSocketConfig` class is disabled (by commenting it out of the `@Configuration` and `@EnableWebSocket` annotations).

Run the project and open a browser. Go to the `http://localhost:8080/sockjs-stomp.html` web page. You will see something similar to the previous example. If you are curious, you can go to the browser's developer console and see the console logs. See Figure 7-10.

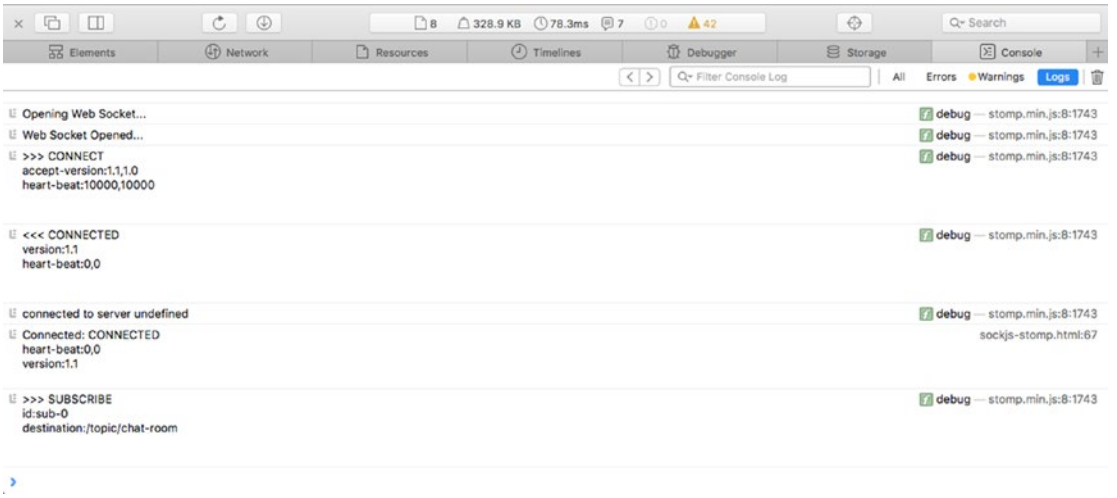

Figure 7-10. *Browser's console log*

Figure 7-10 shows you the connection logs. Now open a second browser window and visit the same URL. The idea is to emulate two clients communicating. Next, send some messages and look at the results. See Figures 7-11 and 7-12.

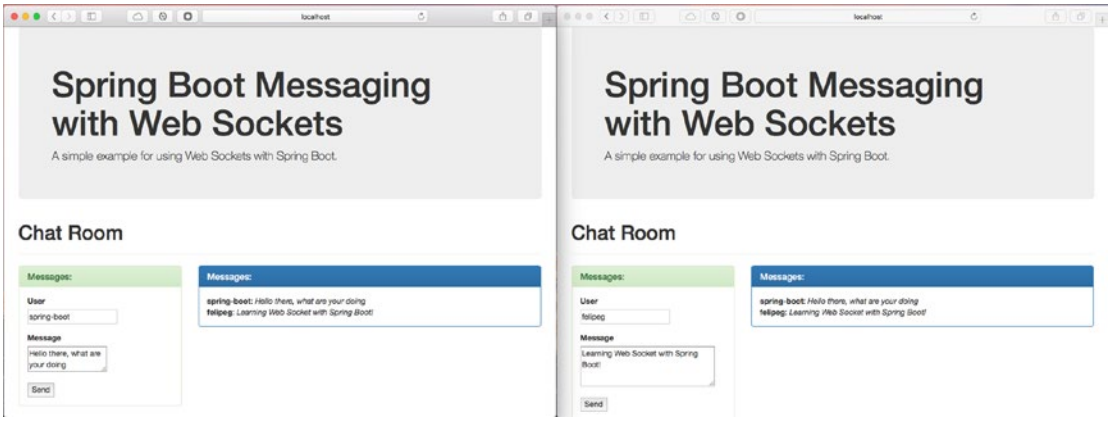

Figure 7-11. *Two browser clients*

```
2017-02-17 16:56:13.765  INFO 80485 --- [boundChannel-22] c.apress.messaging.aop.WebSocketsAudit   :
===================================
[BEFORE]
 Class: com.apress.messaging.controller.SimpleController
Method: chatRoom
Params:
> arg0: ChatMessage [user=spring-boot, message=Hello there, what are your doing, sent=2017-02-17 16:56:13]

[AFTER]
Return: ChatMessage [user=spring-boot, message=Hello there, what are your doing, sent=2017-02-17 16:56:13]
===================================
2017-02-17 16:56:20.820  INFO 80485 --- [boundChannel-25] c.apress.messaging.aop.WebSocketsAudit   :
===================================
[BEFORE]
 Class: com.apress.messaging.controller.SimpleController
Method: chatRoom
Params:
> arg0: ChatMessage [user=felipeg, message=Learning Web Socket with Spring Boot!, sent=2017-02-17 16:56:20]

[AFTER]
Return: ChatMessage [user=felipeg, message=Learning Web Socket with Spring Boot!, sent=2017-02-17 16:56:20]
===================================
```

Figure 7-12. *Application logs*

Figure 7-11 shows two clients sending messages through WebSockets using SockJS and STOMP.

Figure 7-12 shows the `SimpleController` logs and the `ChatMessage` is being handled by the `chatRoom` method (this happened because of the `@MessageMapping` and `@SendTo` annotations).

That's how you create a chat room very easily with Spring Boot and the `spring-websocket` module.

Question: How can you create a SockJS client using Spring? Imagine that you need to send a message to a remote WebSockets connection using STOMP. Look at the code in the `WebSocketsDemoApplication` class, which is commented out. You will see how to use the `SockJSClient` and `WebSocketsStompClient` classes there.

Using RabbitMQ as a STOMP Broker Relay

Have you wondered what would happen if your application needed more support, being more scalable? Well, one solution is to add several Spring Boot apps that have the WebSockets broker enabled and then add a load balancer in front of them. The problem is getting high availability, because you need to add the logic and behavior for the apps, so that, if one is down, the others keep responding to the clients.

The good news is that the `spring-websocket` module has a way to use an external relay: RabbitMQ. RabbitMQ includes the STOMP protocol as a plugin. It also includes real full high availability and an easy way to set up a cluster.

Follow these steps to add RabbitMQ as a STOMP relay to your application:

1. Make sure to enable the RabbitMQ STOMP plugin:

```
$ rabbitmq-plugins enable rabbitmq_stomp
$ rabbitmq-plugins enable rabbitmq_web_stomp
```

2. Add the following dependencies to your `pom.xml` file.

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
</dependency>
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-net</artifactId>
</dependency>
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-all</artifactId>
    <version>4.1.8.Final</version>
</dependency>
```

3. Configure the `WebSocketsConfig` class similar to the following code:

```
@Override
public void configureMessageBroker(
                        MessageBrokerRegistry config) {

        config.setApplicationDestinationPrefixes(
                                props.getAppDestinationPrefix());

        config.enableStompBrokerRelay(
               "/topic", "/queue").setRelayPort(61613);

}
```

What is different from the previous version is that now in the `configureMessageBroker` method you are configuring the `enableStompBrokerRelay` (using `/topic` and `/queue`) and adding the STOMP port with the `setRelayPort` method (with the value of `61613`, the RabbitMQ's STOMP port).

And that's it. You can now use RabbitMQ as a broker relay. Before you run the project, make sure the RabbitMQ broker is up and running. Then you can run the project and use the same `sockjs-stomp.html` web page. The important part here is to keep an eye on the RabbitMQ console to see connections and queues.

Currency Project

Take a look at the `rest-api-websockets` project. You will find the `RateWebSocketsConfig` class, which is very similar to the other project. The idea is that the currency project has a simple WebSockets broker, which will accept any client connection through the WebSockets protocol.

Every time there is a new rate posted, it will send a message to the client's subscribe to the `/rate/new` endpoint. Take a look at this:

- `RateWebSocketsConfig`: This class has the configuration needed for WebSockets messaging.
- `CurrencyController`: This class in the `addNewRates` method has the following statement:

  ```
  webSocket.convertAndSend("/rate/new", currencyExchange);
  ```

- `src/main/resources/public/index-ws.html`: This web page has the code that defines the client that connects to the server. Take a look; it's very straightforward.

To test it, you just add a simple post to the command line:

```
$ curl -i -X POST -H "Content-Type: application/json" -H "Accept: application/json"
-d '{"base":"USD","date":"2017-02-15","rates":[{"code":"EUR","rate":0.82857,
"date":"2017-02-15"},{"code":"JPY","rate":105.17,"date":"2017-02-15"},
{"code":"MXN","rate":22.232,"date":"2017-02-15"},
{"code":"GBP","rate":0.75705,"date":"2017-02-16"}]}' localhost:8080/currency/new
```

You can also use any other REST client, such as `POSTMAN`: `https://www.getpostman.com/`. After posting the new rate, you will see the Rates in the panel, as shown in Figure 7-13.

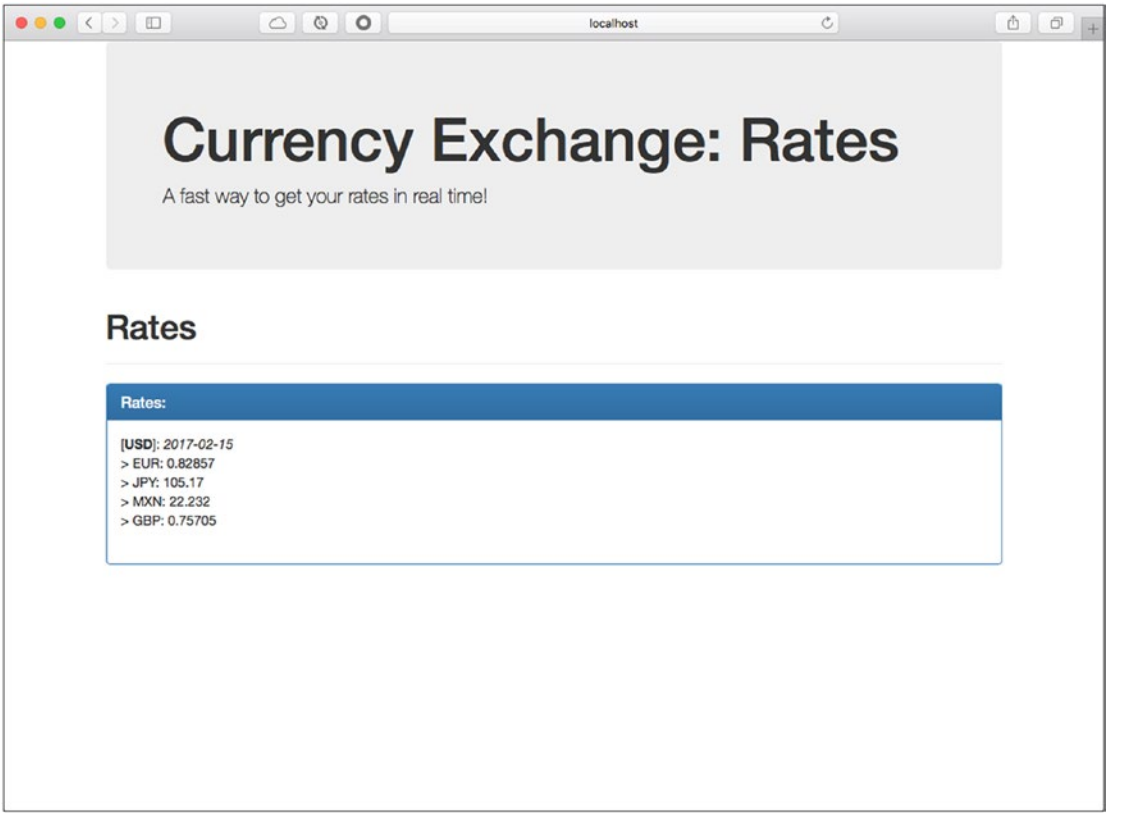

Figure 7-13. *Currency exchange through WebSockets*

Summary

This chapter discussed WebSockets messaging using the `spring-websocket` module and Spring Boot.

You learned how the WebSockets uses an HTTP handshake and then switches to TCP connections. You saw examples of how to create a low-level server/client.

You saw how to use the SockJS and STOMP to facilitate communication for doing async or event-driven messaging. You also learned how to configure RabbitMQ and use it as a STOMP relay. You saw some code in the currency exchange project that sends new rates for any client that connects to this service.

The next chapter shows you how to integrate your code with multiple technologies using the Spring Integration module.

Index

F. Gutierrez, *Spring Boot Messaging*, DOI 10.1007/978-1-4842-1224-0

H

I

J, K, L

M, N

O

P, Q

R

S

T

U, V

W

X, Y, Z