Part 2 Implementation

7 Understanding Spring Boot and Spring MVC

**7 Understanding Spring Boot and Spring MVC**

To implement web apps, we’ll use a project in the Spring ecosystem named Spring Boot.

**7.1 What is a web app?**

Any app you access through your web browser is a **web app**. Accessing an app in a browser makes it more comfortable to use. You don’t have to install anything, and you can use it from any device that has access to the internet, such as a tablet or smartphone.

**7.1.1 A general overview of a web app**

A web app is composed of 2 parts:

* *The client side* is what the user directly interacts with. A web browser represents the client side of a web app. The browser sends requests to a web server, receives responses from it, and provides a way for the user to interact with the app. We also refer to the client side of a web app as the *frontend*.
* *The server side* receives requests from the client and sends back data in response. The server side implements logic that processes and sometimes stores the client requested data before sending a response. We also refer to the server side of a web app as the *backend*.

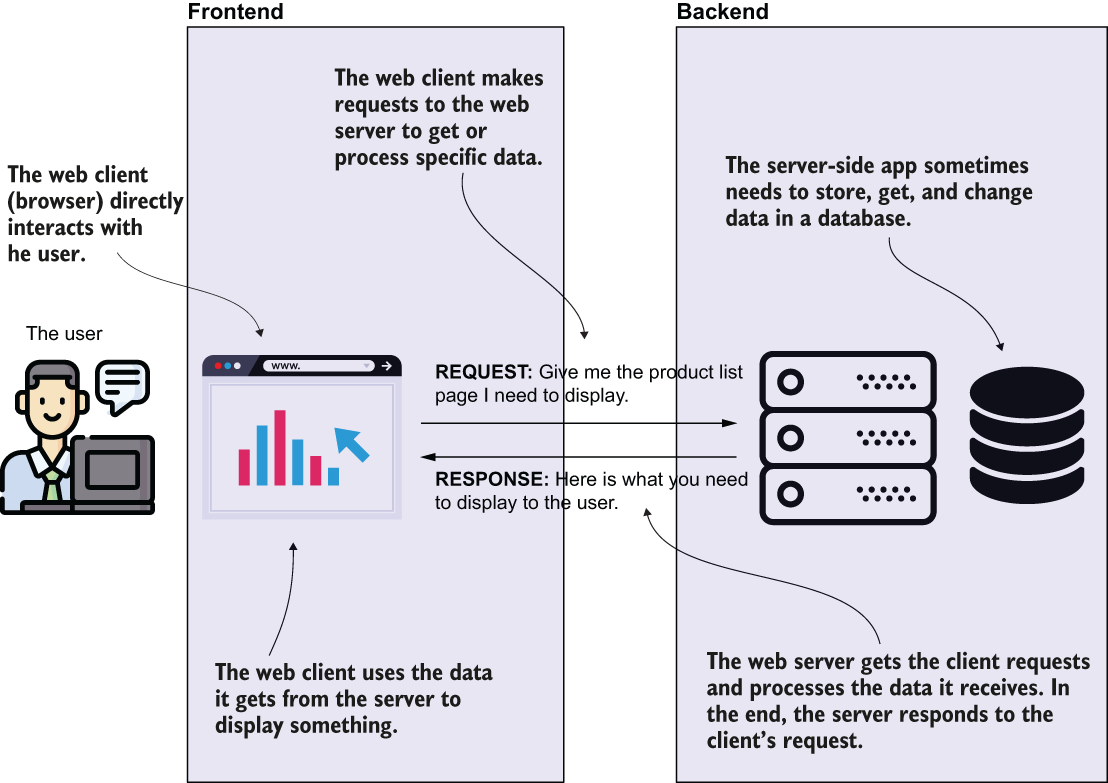
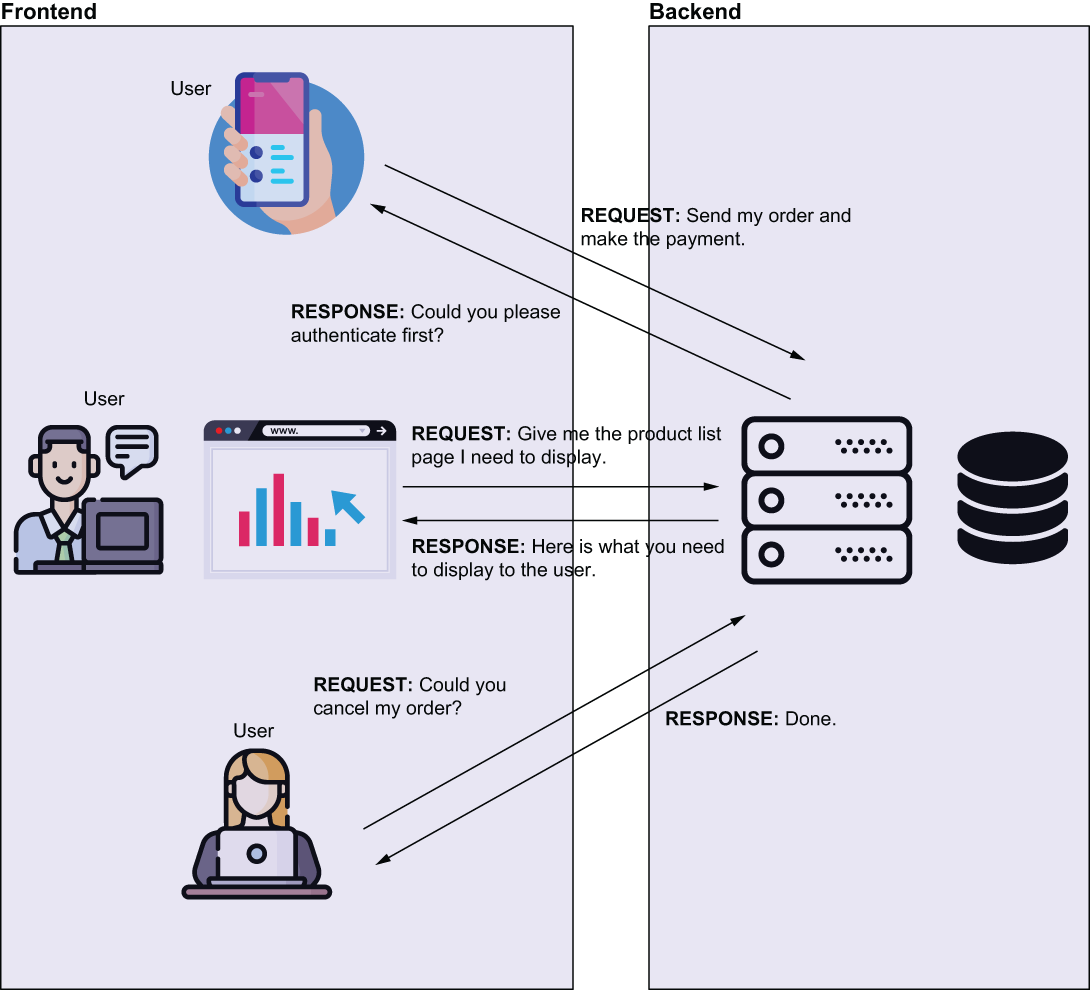


Figure 7.2 The big picture of a web app. The user interacts with the app through its frontend. The frontend communicates with the backend to execute logic at the user’s request and get the data to display. The backend executes business logic and sometimes persists data in a database or communicates with other external services.

When discussing web apps, we usually refer to a client and a server, but it’s important to keep in mind that the backend serves multiple clients concurrently. Numerous people may use the same app at the same time on different platforms.



Each user makes their own requests on specific actions they need to execute. This means that some operations on the backend execute concurrently. If you write code that accesses and changes the same resource, your app might wrongly behave because of race condition scenarios.

**7.1.2 Different fashions of implementing a web app with Spring**

In this section, we discuss the 2 main designs you can use to implement a web application. We classify the approaches of creating a web app as the following:

1. *Apps where the backend provides the fully prepared view in response to a client’s request*. The browser directly interprets the data received from the backend and displays this information to the user in these apps.
2. *Apps using frontend-backend separation.* For these apps, the backend only serves raw data. The browser doesn’t display the data in the backend’s response directly. The browser runs a separate frontend app that gets the backend responses, processes the data, and instructs the browser what to display.

Figure 7.4 presents the 1st approach in which the app doesn’t use a frontend-backend separation. For these apps, almost everything happens on the backend side. The backend gets requests representing user actions and executes some logic. In the end, the server responds with what the browser needs to display. The backend responds with the data in formats that the browser can interpret and display, such as HTML, CSS, images, and so on. It can also send scripts written in languages that the browser can understand and execute (such as JavaScript).

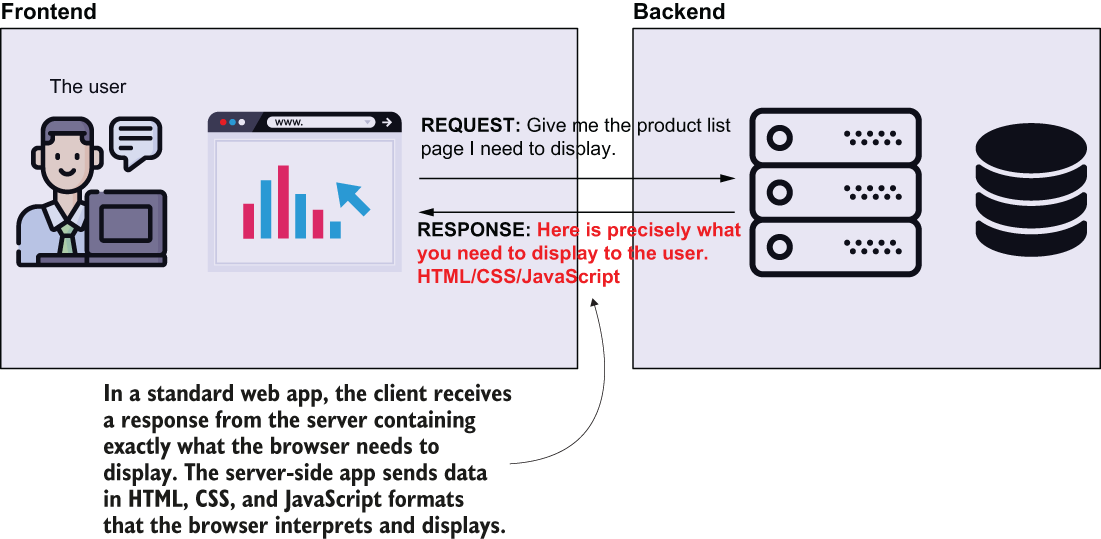


Figure 7.4 When a web app doesn’t provide frontend-backend separation, the browser precisely displays what it gets from the server. The server gets requests from the browser, executes some logic, and then responds. In response, the backend provides content formatted as HTML, CSS, and other fashions that the browser interprets to display.

Figure 7.5 shows an app using frontend-backend separation. Compare the server’s response in figure 7.5 with the response the server sends back in figure 7.4. Instead of telling the browser precisely what to display, the server now only sends raw data. The browser runs an independent frontend app it loads at an initial request from the server. This frontend app takes the server’s raw response, interprets it, and decides how the information is displayed.

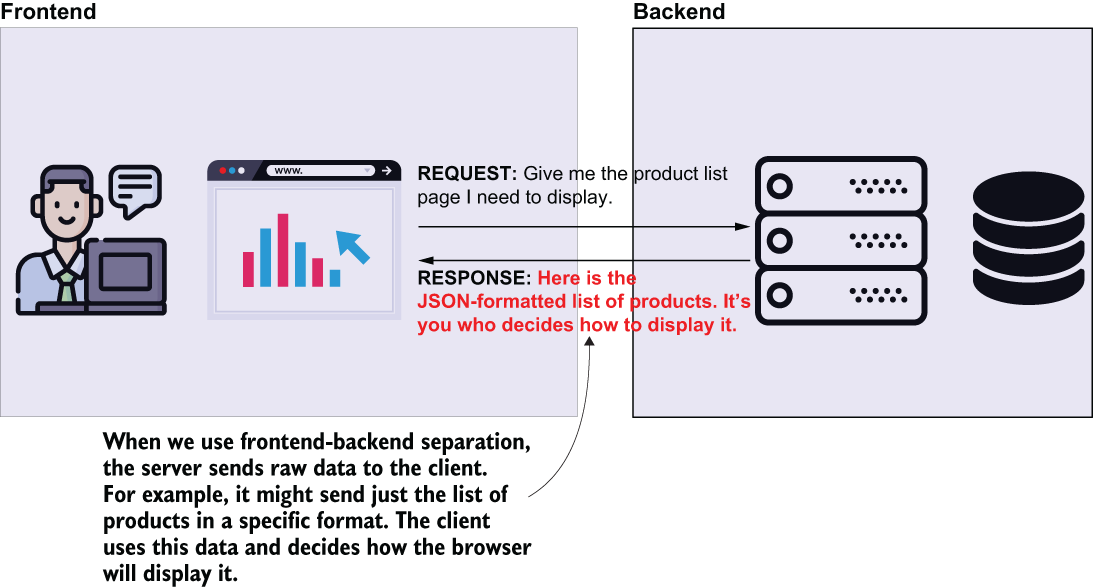


Figure 7.5 Using frontend-backend separation. The server doesn’t respond with the exact data that needs to be displayed by the browser. The backend sends the data to the client but doesn’t tell the browser how to display this data or what to do with it. The backend now only sends raw data (usually in an easily parsable format like JSON or XML). The browser executes a frontend app that takes the server’s raw response and processes it to display the data.

You will find both these approaches in production apps. Sometimes developers refer to the frontend-backend separation approach as being a modern approach. The separation of front- and backend helps in making the development easier to manage for larger apps. Different teams take the responsibility of implementing the back- and frontend, allowing more developers to collaborate to develop the apps. Also, the deployment of the front- and the backend can be independently managed. For a larger app, this flexibility is also a nice benefit.

The other approach that doesn’t use frontend-backend separation is mostly for small apps.

**7.1.3 Using a servlet container in web app development**

One of the most important things to consider is the communication between the client and the server. A web browser uses a protocol named Hypertext Transfer Protocol (HTTP) to communicate with the server over the network. This protocol accurately describes how the client and the server exchange data over the network. The web app components use this protocol to exchange data in a request-response fashion. The client sends a request to the server, and the server responds. The client waits for the response after every request it sends.

But does that mean your app needs to know how to process the HTTP messages? Well, you can implement this capability if you wish, but unless you want to have some fun writing low-level functionalities, you’ll use a component already designed to understand HTTP.

In fact, what you need is not only something that understands HTTP, but something that can translate the HTTP request and response to a Java app. This something is a **servlet container** (sometimes referred to as a web server): a translator of the HTTP messages for your Java app. This way, your Java app doesn’t need to take care of implementing the communication layer. One of the most appreciated servlet container implementations is Tomcat, which is also the dependency we’ll use.

NOTE You can use its alternatives for your Spring app. Among the list of solutions used in real-world, you find *Jetty* (<https://www.eclipse.org/jetty/> ), *JBoss* (<https://www.jboss.org/> ), and *Payara* (<https://www.payara.fish/>).

Figure 7.6 is a visual representation of a servlet container (Tomcat) in our app’s architecture.

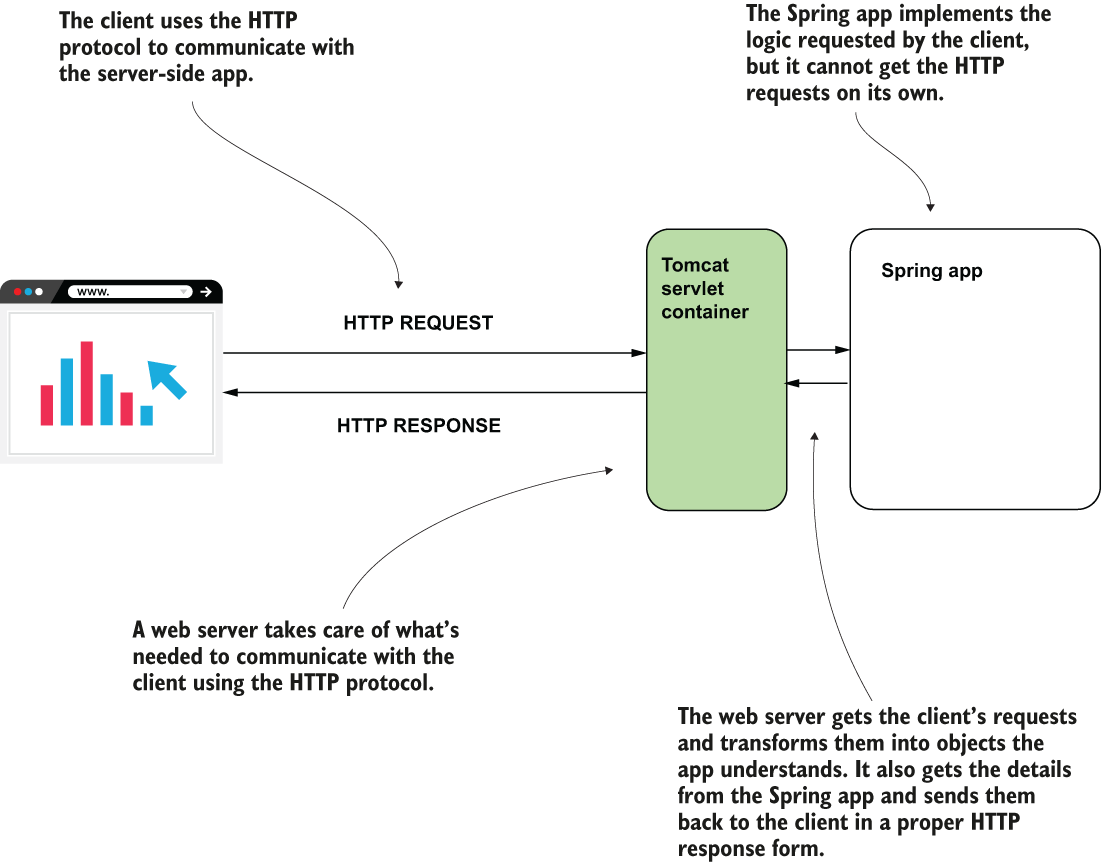


Figure 7.6 A servlet container (e.g., Tomcat) speaks HTTP. It translates the HTTP request to our Spring app and the app’s response into an HTTP response. This way, we don’t need to care about the protocol used for communication on the network, as we simply write everything as Java objects and methods.

What is a **servlet**? A servlet is nothing more than a Java object that directly interacts with the servlet container. When the servlet container gets an HTTP request, it calls a servlet object’s method and provides the request as a parameter. The same method also gets a parameter representing the HTTP response used by the servlet to set the response sent back to the client that made the request.

Some time ago, the servlet was the most critical component of a backend web app from the developer’s point of view. Suppose a developer had to implement a new page accessible at a specific path in the URL (e.g., /home/profile/edit, etc.) for a web app. The developer needed to create a new servlet instance, configure it in the servlet container, and assign it to a specific path (figure 7.7). The servlet contained the logic associated with the user’s request and the ability to prepare a response, including info for the browser on how to display the response. For any path the web client could call, the developer needed to add the instance in the servlet container and configure it. Because such a component manages servlet instances you add into its context, we name it a **servlet container**. It basically has a context of servlet instances it controls, just as Spring does with its beans. For this reason, we call a component such as Tomcat a servlet container.

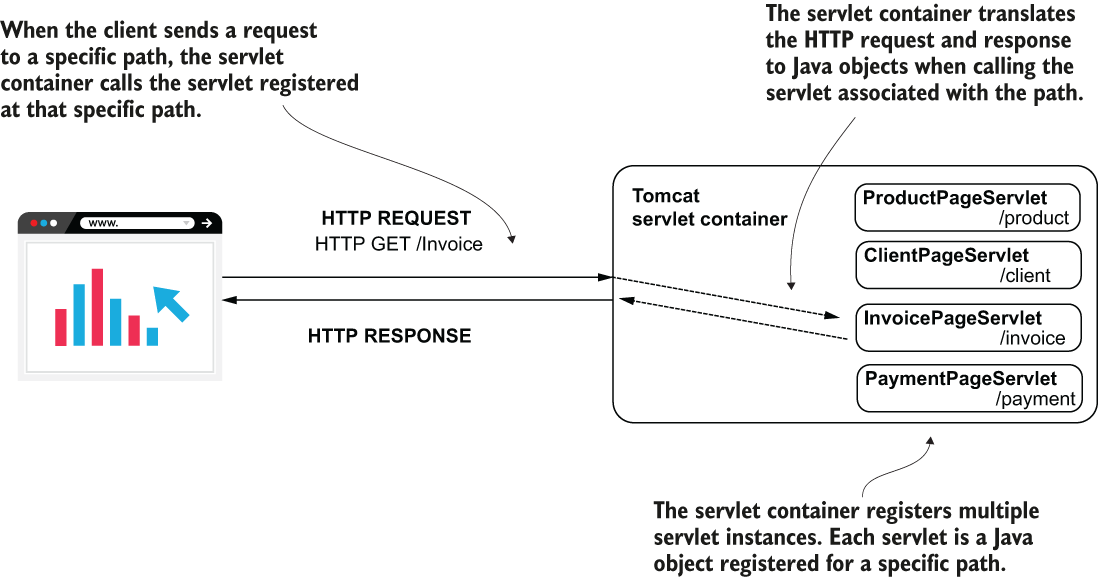


Figure 7.7 The servlet container (Tomcat) registers multiple servlet instances. Each servlet is associated with a path. When the client sends a request, Tomcat calls a method of the servlet associated with the path the client requested. The servlet gets the values on the request and builds the response that Tomcat sends back to the client.

We don’t typically create servlet instances. We’ll use a servlet with the Spring apps we develop with Spring, but you won’t need to write this yourself, so you don’t have to focus on learning to implement servlets. But you do need to remember the servlet is the entry point to your app’s logic. It’s the component the servlet container (Tomcat, in our case) directly interacts with. It’s how the request data enters your app and how the response goes through Tomcat back to the client (figure 7.8).

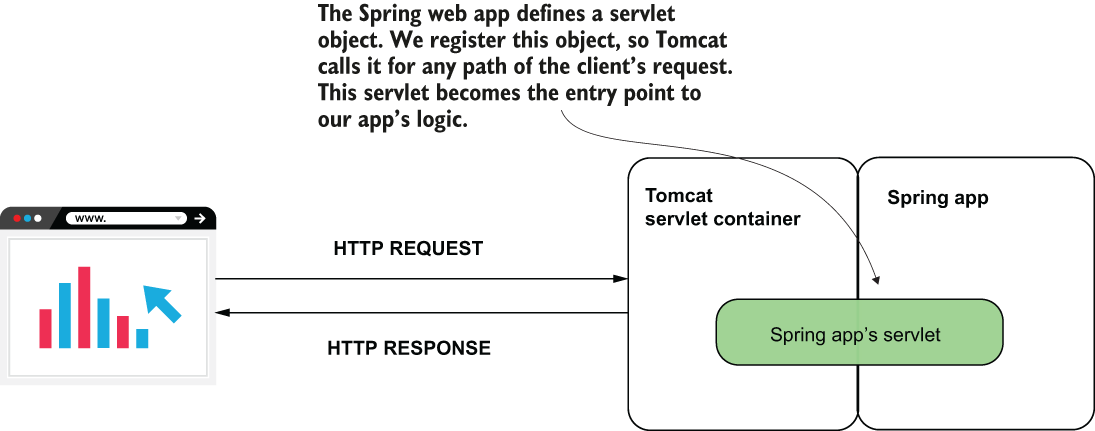


Figure 7.8 The Spring app defines a servlet object and registers it into the servlet container. Now both Spring and the servlet container know this object and can manage it. The servlet container calls this object for any client request, allowing the servlet to manage the request and the response.

**7.2 The magic of Spring Boot**

To create a Spring web app, we need to configure a servlet container, create a servlet instance, and then make sure we correctly configure this servlet instance such that Tomcat calls it for any client request. Nowadays you don’t have to write so many configurations.

Spring Boot is a tool for implementing modern Spring apps. Spring Boot is now one of the most appreciated projects in the Spring ecosystem. It helps you create Spring apps more efficiently and focus on the business code you write by eliminating a huge part of the code you used to write for configurations. Especially in a world of service-oriented architectures (SOA) and microservices, where you create apps more often.

Listed here are what I consider the most critical Spring Boot features, and what they offer:

* *Simplified project creation*—You can use a project initialization service to get an empty but configured skeleton app.
* *Dependency starters*—Spring Boot groups certain dependencies used for a specific purpose with dependency starters. You don’t need to figure out all the must-have dependencies you need to add to your project for one particular purpose nor which versions you should use for compatibility.
* *Autoconfiguration based on dependencies*—Based on the dependencies you added to your project, Spring Boot defines some default configurations. Instead of writing all the configurations yourself, you only need to change the ones provided by Spring Boot that don’t match what you need. Changing the configs likely requires less code (if any).

## **PROJECT 01 – create a Spring Boot project with Spring Initializr – return static page**

**7.2.1 Using a project initialization service to** **create a Spring Boot project**

Here we discuss using a project initialization service to create a Spring Boot project.

Some IDEs integrate directly with a project initialization service, and some don’t. For example, in IntelliJ Ultimate or STS, you’ll find this feature available when creating a new project (figure 7.9)—but if you use IntelliJ Community, you don’t.

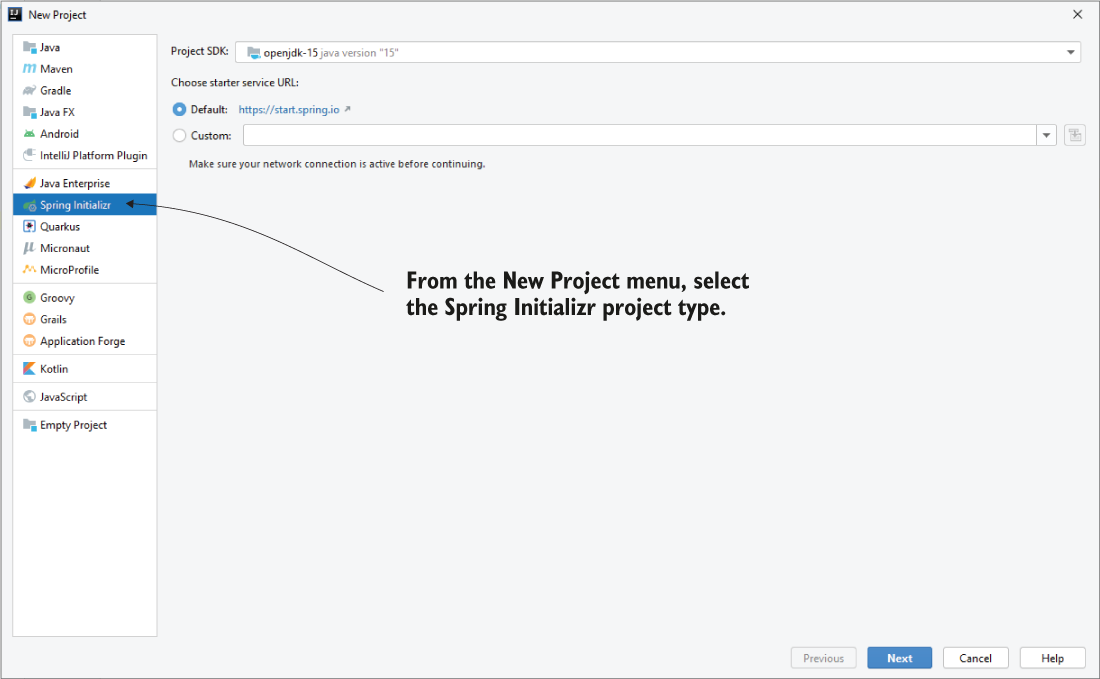


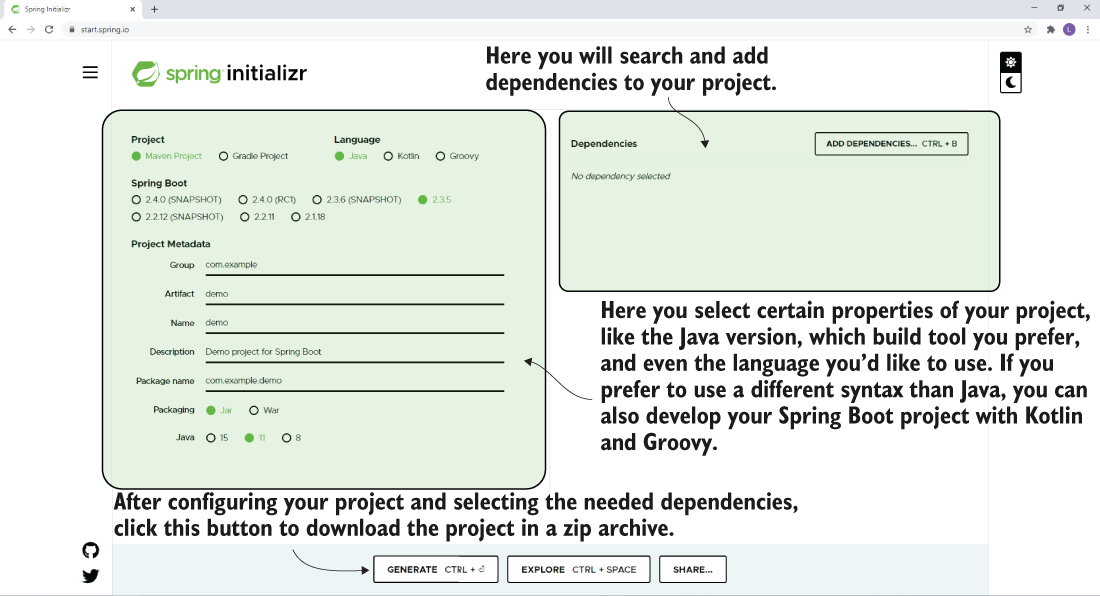
Figure 7.9 Some IDEs integrate with a project initializer service directly. For example, in IntelliJ Ultimate, you can select Spring Initializr from the New Project menu to create a Spring Boot app with a project initializer service.

If your IDE supports this feature, you’ll probably find it named Spring Initializr in your project creation menu. But if your IDE doesn’t support direct integration with a Spring Boot project initialization service, you can use this feature by accessing <http://start.spring.io> directly in your browser. This service will help you create a project you can import into any IDE. Let’s use this approach to create our 1st project.

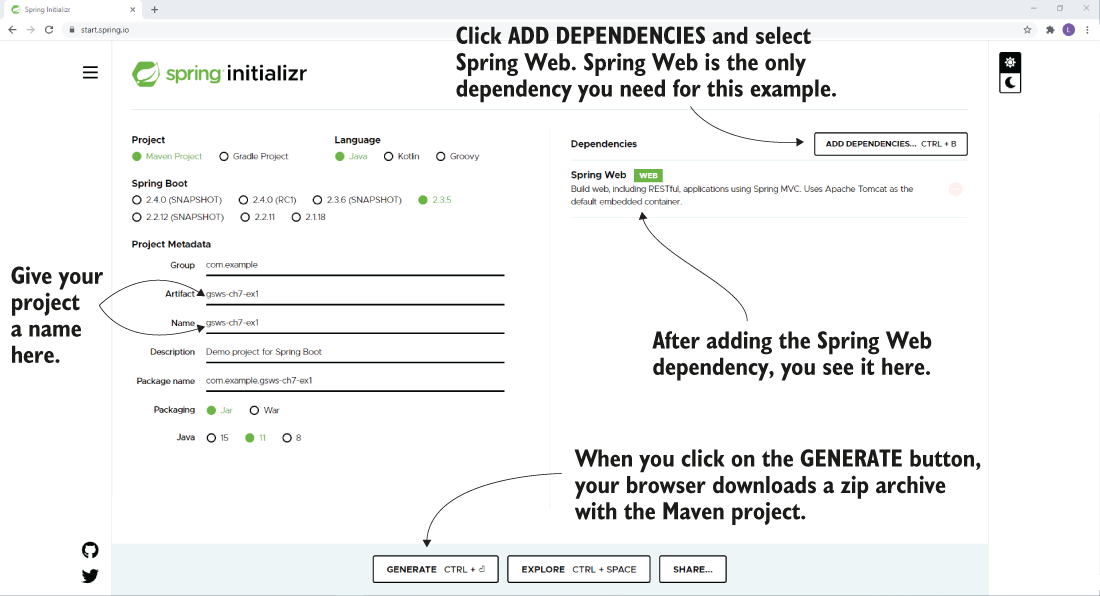
The following list summarizes the steps we’ll take to create the Spring Boot project using <https://start.spring.io/> (figure 7.10):

1. Access start.spring.io in a web browser.
2. Select the project properties (language, the version, the build tool, and so on).
3. Select the needed dependencies you want to add to your project.
4. Use the Generate button to download the archived project.
5. Unarchive the project and open it in your IDE.

Once you access start.spring.io in a web browser, you’ll find an interface similar to the one below. You have to specify some project properties, like the build tool you prefer between Maven and Gradle and the Java version you want to use. Spring Boot even offers you the possibility to change the syntax of your app to Kotlin or Groovy.



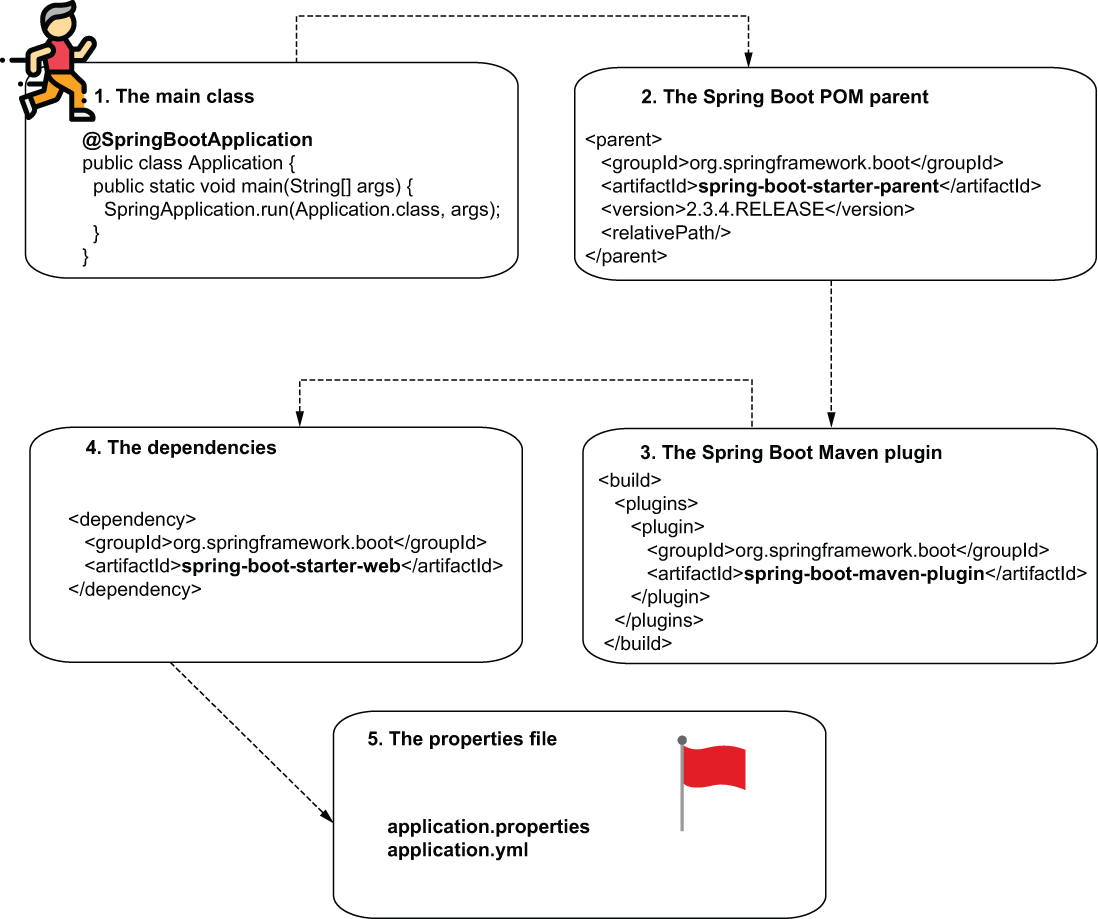
We’ll continue using Maven and Java 11. Figure 7.12 shows you an example of filling the fields for generating a new Spring Boot project. In this example, we only need to add a dependency named Spring Web. This dependency adds everything our project needs to become a Spring web app.



When you click the Generate button, the browser downloads a zip archive containing a Spring Boot project.

Now we discuss the main things Spring Initializr configured into your Maven project (figure 7.13):

1. The Spring app main class
2. The Spring Boot POM parent
3. The dependencies
4. The Spring Boot Maven plugin
5. The properties file



We’ll discuss each configuration:

THE APP’S MAIN CLASS CREATED BY START.SPRING.IO

The 1st thing to look at is the application’s main class. Unarchive the downloaded file and open it in your IDE. You can observe that Spring Initializr added the **Main** class to your app and also some configurations in the pom.xml file. The Main class of a Spring Boot app is annotated with the @SpringBootApplication annotation:

@SpringBootApplication ❶

public class Main {

public static void main(String[] args) {

SpringApplication.run(Main.class, args);

}

}

❶ This annotation defines the Main class of a Spring Boot app.

To understand how Spring Boot apps work in detail, read:

Spring Boot in Action (Manning, 2015) - Craig Walls

Spring Boot: Up and Running (O’Reilly Media, 2021) - Mark Heckler

THE SPRING BOOT MAVEN PARENT CONFIGURED BY START.SPRING.IO

2nd, we look at the **pom.xm**l file. If you open your project’s pom.xml file, you’ll find added some details here. One of the most important details you’ll find is the Spring Boot parent node:

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.3.4.RELEASE</version>

<relativePath/>

</parent>

One of the essential things this parent does is provide you with compatible versions for the dependencies you’ll add to your project. You’ll observe that we don’t specify a version for a dependency we use in most cases. We let (and it’s recommended) Spring Boot choose the version of a dependency to make sure we don’t run into incompatibilities.

THE SPRING BOOT MAVEN PLUGIN CONFIGURED BY START.SPRING.IO

Next we look at the Spring Boot Maven plugin configured by start.spring.io when creating the project. You find this plugin also configured in the pom.xml file. The plugin declaration you usually find at the end of the pom.xml file inside the <build> <plugins> ... </plugins></build> tags. This plugin is responsible for adding part of the default configurations you’ll observe in your project:

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

THE MAVEN DEPENDENCIES ADDED BY START.SPRING.IO WHEN CREATING THE PROJECT

Also in the pom.xml file, you find the dependency you added when creating the project in start.spring.io, Spring Web. You’ll find this dependency provided. It is a dependency starter named spring-boot-starter-web. For now, know that it doesn’t specify a version.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

For all the examples we have written, we also specified a version for each dependency. The reason you don’t specify one now is to let Spring Boot choose the right one for you. This is why we need the Spring Boot parent to the pom.xml file.

THE APPLICATION PROPERTIES FILE

The last essential thing Spring Initializr added to your project is a file named “application.properties.” You find this file in the resources folder of your Maven project. Initially, this file is empty, and for this first example we’ll keep it this way.

**7.2.2 Using dependency starters to simplify the dependency management**

Let’s focus on the 2nd essential advantage Spring Boot offers: **dependency starters**. Dependency starters save you plenty of time, and they’re an invaluable feature Spring Boot offers.

A dependency starter is a group of dependencies you add to configure your app for a specific purpose. In your project’s pom.xml file, the starter looks like a normal dependency. Observe the name of the dependency: A starter name usually starts with “spring-boot-starter-” followed by a relevant name that describes the capabilities it added to the app:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

Say you want to add web capabilities to your app. In the past, to configure a Spring web app you had to add all the needed dependencies to your pom.xml file yourself and make sure their versions were compatible one with the other. Configuring all the dependencies you need is not an easy job. Taking care of the version compatibility is even more complicated.

With dependency starters, we don’t request dependencies directly. We request capabilities (figure 7.14). You add a dependency starter for a particular capability you need, say web functionalities, a database, or security. Spring Boot makes sure to add the right dependencies to your app with the proper compatible version for your requested capability. We can say that dependency starters are capability-oriented groups of compatible dependencies.

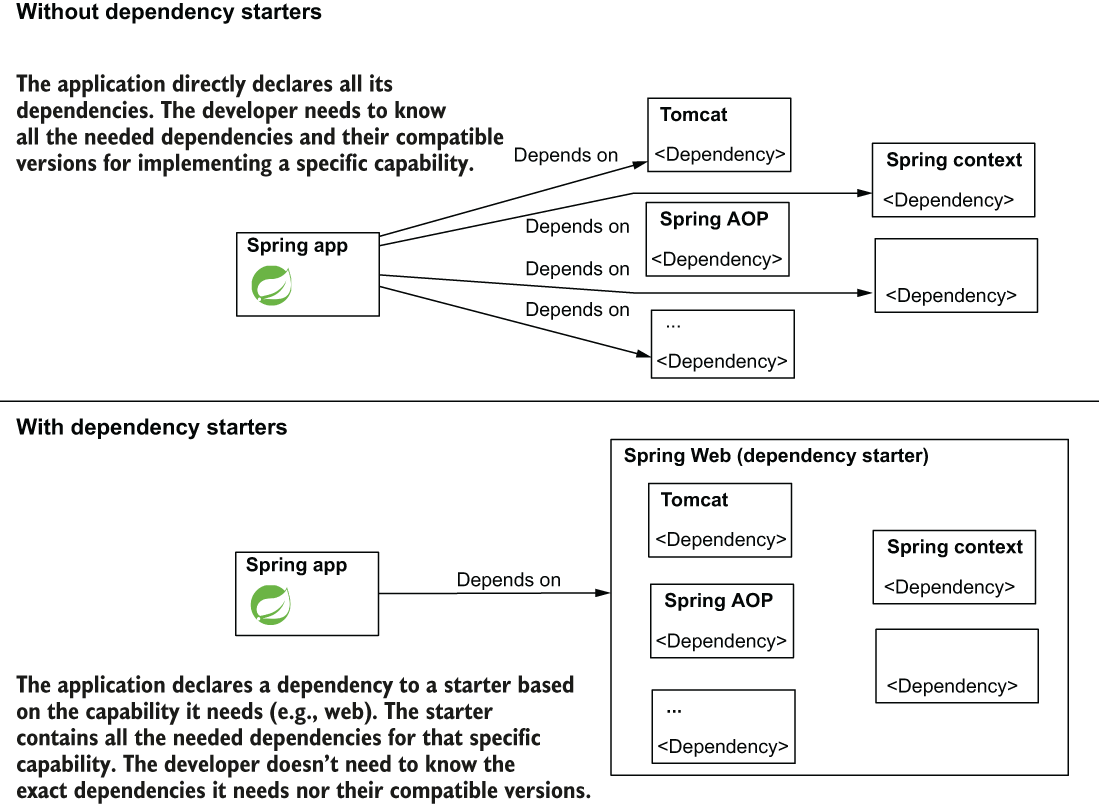


Figure 7.14 Using dependency starters. Instead of individually referring to specific dependencies, the app now depends on only a starter. The starter contains all the needed dependencies for implementing a specific capability. The starter also makes sure these dependencies are compatible with one another.

Look at your pom.xml file. You only added the spring-boot-starter-web dependency, no Spring context, no AOP, no Tomcat! But, if you look in the “External Libraries” folder of your app, you’ll find JAR archives for all these. Spring Boot knew you would need them and downloaded them with specific versions it knows are compatible.

**7.2.3 Using autoconfiguration by convention based on dependencies**

Spring Boot also provides **autoconfiguration** for your application. We say that it applies the convention-over-configuration principle. Of all the previous Spring Boot features discussed in this chapter, the autoconfiguration is probably the most appreciated and the most known.

You didn’t even write anything yet—only downloaded the project and opened it in your IDE. But you can start the app, and you’ll find your app boots a Tomcat instance by default accessible on port 8080. In your console, you find something similar to the next snippet:

Tomcat started on port(s): 8080 (http) with context path '' ❶

Started Main in 1.684 seconds (JVM running for 2.306)

❶ Spring Boot configured Tomcat and starts it by default on port 8080.

Based on the dependencies you added, Spring Boot realizes what you expect from your app and provides you some default configurations. Spring Boot gives you the configurations, which are generally used for the capabilities you requested when adding the dependencies.

For example, Spring knows when you added the web dependency you need for a servlet container and configures you a Tomcat instance because, in most cases, developers use this implementation. For Spring Boot, Tomcat is the convention for a servlet container.

The convention represents the most-used way to configure the app for a specific purpose. Spring Boot configures the app by convention such that you now only need to change those places where your app needs a more particular configuration. With this approach, you’ll write less code for configuration (if any).

**7.3 Implementing a web app with Spring MVC**

We’ll implement our first web page in a Spring web app. It’s true we already have a Spring Boot project with the default configurations, but this app only starts a Tomcat server. These configurations don’t make our app a web app yet! We still have to implement the pages that someone can access using a web browser. With these changes, you’ll learn to implement a web page and how your Spring app works behind the scenes.

To add a web page to your app, you follow 2 steps (figure 7.15):

1. Write an HTML document with the content you want to be displayed by the browser.
2. Write a controller with an action for the web page created at point 1.

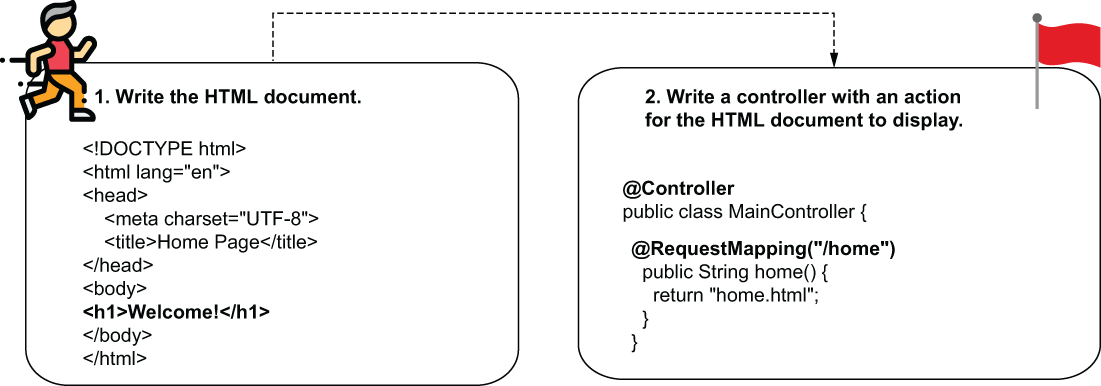


Figure 7.15 The steps for adding a static web page to your application. Add the HTML document containing the information the browser will display and then write a controller with an action assigned to it.

We first start adding a static web page with the content we want to display in the browser. This web page is just an HTML document, and for our example the page only displays a short text in a heading. You need to add the file in the *“resources/static”* folder of your Maven project. This folder is the default place where the Spring Boot app expects to find the pages to render.

Listing 7.1 The content of the HTML file

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Home Page</title>

</head>

<body>

<h1>Welcome!</h1> ❶

</body>

</html>

❶ In a standard HTML document, we display a heading text.

The second step you take is writing a controller with a method that links the HTTP request to the page you want your app to provide in response. The **controller** is a component of the web app that contains methods (often named actions) executed for a specific HTTP request. In the end, the controller’s action returns a reference to the web page the app returns in response. We’ll keep our first example simple, and we won’t make the controller execute any specific logic for the request for now. We’ll just configure an action to return in response to the content of the home.html document we created and stored in the “resources/static” folder in the first step.

To mark a class as a controller, you only need to use the @Controller annotation, a stereotype annotation (like @Component and @Service). This means that Spring will also add a bean of this class to its context to manage it. Inside this class, you can define controller actions, which are methods associated with specific HTTP requests.

Say you want the browser to display this page’s content when the user accesses the /home path. To achieve this result, you annotate the action method with the @RequestMapping annotation specifying the path as a value of the annotation: @RequestMapping("/home"). The method needs to return, as a string, the name of the document you want the app to send as a response.

Listing 7.2 The definition of the controller class

@Controller ❶

public class MainController {

@RequestMapping("/home") ❷

public String home() {

"home.html"; ❸

}

}

❶ We annotate the class with the @Controller stereotype annotation.

❷ We use the @RequestMapping annotation to associate the action with an HTTP request path.

❸ We return the HTML document name that contains the details we want the browser to display.

First, you need to know how Spring manages the request and calls this controller action we implemented.

When starting the app, you will see the log. It tells you Tomcat started and the port it uses in the app console. If you use the default, Tomcat uses port 8080.

Tomcat started on port(s): 8080 (http) with context path ''

Open a browser window on the same computer where you run the app and write in the address bar: <http://localhost:8080/home> . Do not forget to write the path /home you mapped with the controller’s action; otherwise, you’ll get an error and an HTTP response with the status “404 Not Found.”

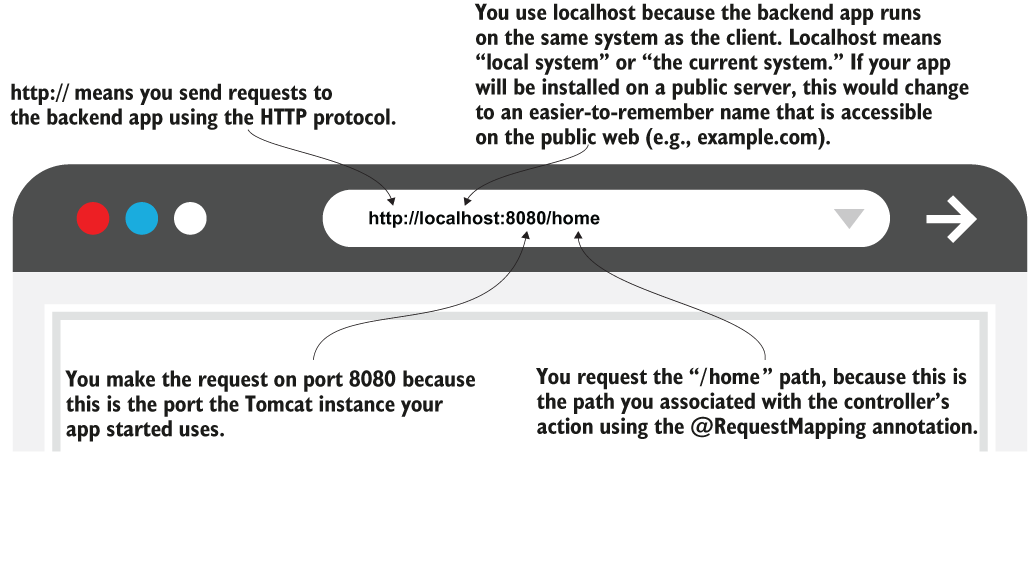


Figure 7.16 Testing the implementation. Using a browser, send a request to the backend app. You need to use the port Tomcat opened and the path you specified with the @RequestMapping annotation.

Figure 7.17 shows you the result of accessing the web page in a browser.

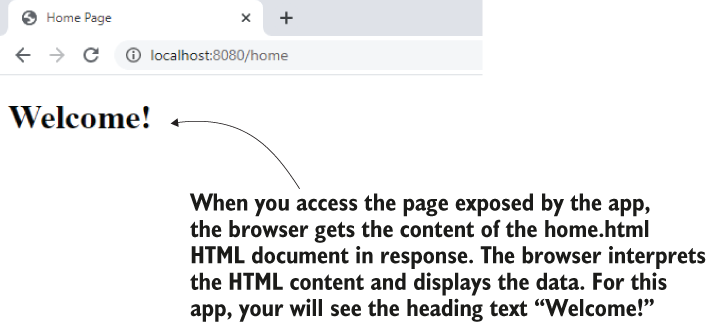


Figure 7.17 Accessing the page in a browser, you’ll see the heading text “Welcome!” The browser interprets and displays the HTML received in response from the backend.

Now that you’ve seen the app’s behavior, let’s discuss the mechanism behind it. Spring has a set of components that interact with each other to get the result you observed. Figure 7.18 presents these components and the flow in which they manage an HTTP request.

1. The client makes an HTTP request.
2. Tomcat gets the client’s HTTP request. Tomcat has to call a servlet component for the HTTP request. In the case of Spring MVC, Tomcat calls a servlet Spring Boot configured. We name this servlet *dispatcher servlet*.
3. The dispatcher servlet is the entry point of the Spring web app. (It’s that servlet discussed in figure 7.8 earlier; also figure 7.18.) Tomcat calls the dispatcher servlet for any HTTP request it gets. Its responsibility is to manage the request further inside the Spring app. It has to find what controller action to call for the request and what to send back in response to the client. This servlet is also referred to as a “*front controller*.”
4. The first thing the dispatcher servlet needs to do is find a controller action to call for the request. To find out which controller action to call, the dispatcher servlet delegates to a component named handler mapping. The *handler mapping* finds the controller action you associated with the request with the @RequestMapping annotation.
5. After finding out which controller action to call, the dispatcher servlet calls that specific controller action. If the handler mapping couldn’t find any action associated with the request, the app responds to the client with an HTTP “404 Not Found” status. The controller returns the page name it needs to render for the response to the dispatcher servlet. We refer to this HTML page also as “the view.”
6. At this moment, the dispatcher servlet needs to find the view with the name received from the controller to get its content and send it as response. The dispatcher servlet delegates the responsibility of getting the view content to a component named “View Resolver.”
7. The dispatcher servlet returns the rendered view in the HTTP response.

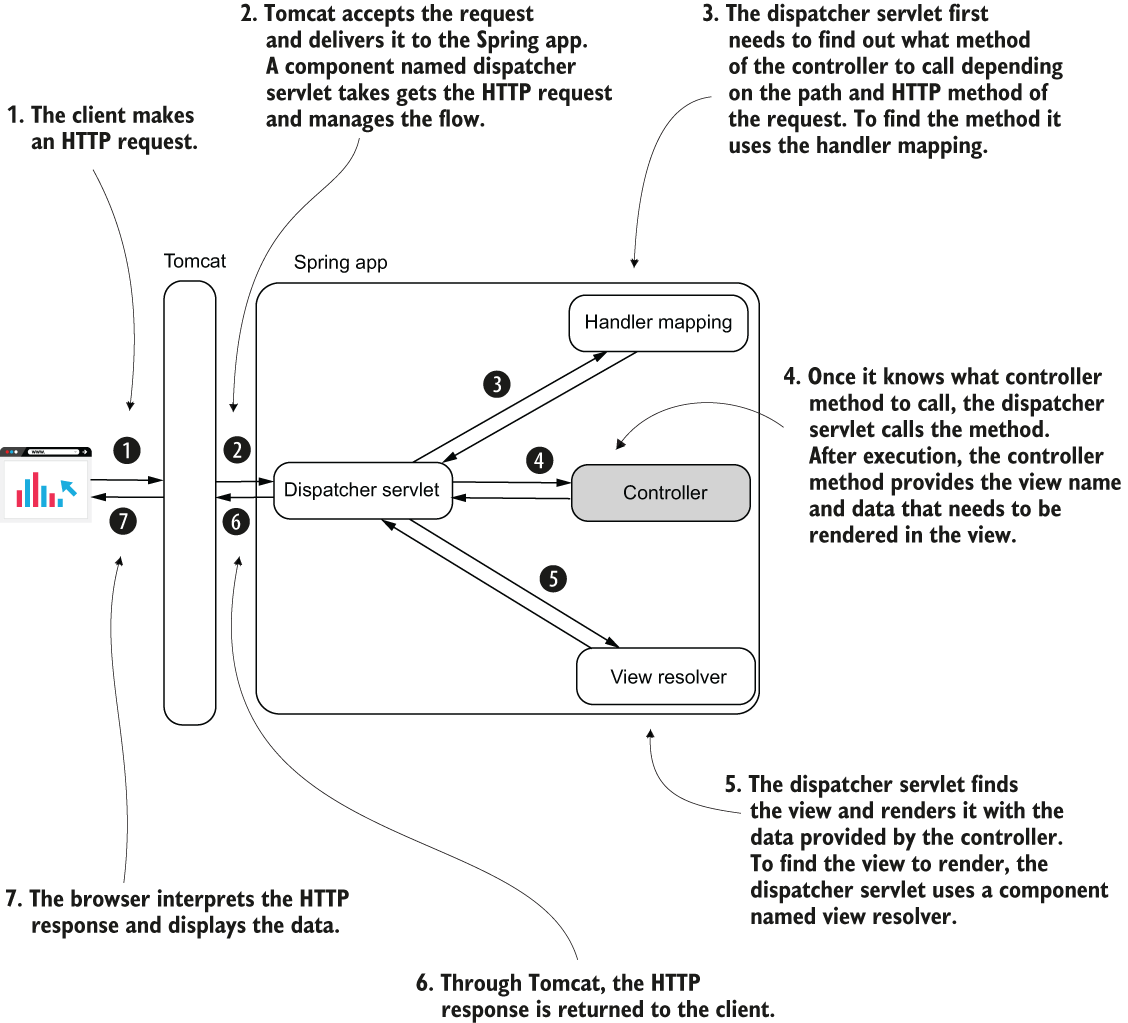


Figure 7.18 The Spring MVC architecture. In the diagram, you find the main components of Spring MVC. These components and the way they collaborate are responsible for a web app’s behavior. The controller (shaded differently) is the only component you implement. Spring Boot configures the other components.

NOTE In this chapter, I described the handler mapping as the component that finds the controller action by the HTTP request path. The handler mapping also searches by something named the HTTP method.

Spring (with Spring Boot) considerably simplifies the development of a web app by arranging this setup. You only need to write controller actions and map them to requests using annotations. A large part of the logic is hidden in the framework, and this helps you write the apps faster and cleaner.

Real-world apps are often more complex than just returning the content of a static HTML page. In most cases, the page displays dynamic details processed by the app before rendering the HTTP response.

**Summary**

* People use web apps more often today than they use desktop apps. For this reason, you must understand how web apps work and learn to implement them.
* A web app is an application the user interacts with using a web browser. A web app has a client side and a server side where the data is processed and stored. The client side (frontend) sends requests to the server side (backend). The backend executes the action requested by the frontend and responds back.
* Spring offers you the capability of implementing web apps. To avoid writing many configurations, you can use Spring Boot: a Spring ecosystem project that applies the convention-over-configuration principle providing you default configurations for the capabilities your app needs.
* Spring Boot also helps you more easily configure your dependencies through the dependency starters it offers. A dependency starter is a group of dependencies with comparable versions to offer your app a specific capability.
* To get the HTTP requests and deliver the responses, a Java backend web app needs a servlet container (e.g., Tomcat): software with the capability to translate HTTP requests and responses to the Java app. With servlet containers, you don’t need to implement the communication over the network using the HTTP protocol.
* You can easily create your web app project as a Spring Boot project, which autoconfigures a servlet container and comes with the capabilities you need to write the use cases of your web app. Spring Boot also configures a set of components that intercept and manage HTTP requests. These components are part of a class design we name Spring MVC.
* Because Spring Boot autoconfigures the Spring MVC components and the servlet container, you only need to write the HTML document containing the data the app sends as a response and a controller class for a minimal HTTP request-response flow action.
* You use annotations to configure your controller and the controller’s actions. To mark a class as a Spring MVC controller, use the @Controller stereotype annotation. To assign a controller action to a specific HTTP request, use the @RequestMapping annotation.

8 Implementing web apps with Spring Boot and Spring MVC

In this chapter, we start with implementing pages whose content changes according to how your app processes the data for specific requests. Today we rarely see static pages on websites. There is a way to decide what content to add on the pages before delivering the HTTP response back to the browser.

We’ll implement dynamic views using **template engines**. A template engine is a dependency that allows you to easily get and display variable data the controller sends.

In section 8.2, you’ll learn how to send data from the client to the server through the HTTP request. We’ll use that data in the controller’s method and create the dynamic content on the view.

In section 8.3, we discuss HTTP methods, and you’ll learn that the request path isn’t enough to identify a client’s request. Together with the request path, the client uses an HTTP method represented with a verb (GET, POST, PUT, DELETE, PATCH, etc.), which expresses the client’s intention. In our example, we’ll implement an HTML form that someone can use to send values the backend has to process.

**8.1 Implementing web apps with a dynamic view**

Suppose you implement the cart page of an online shop. This page shouldn’t display the same data for everyone. It also doesn’t even show the same information every time for the same user. This page shows precisely the products a particular user has added to their cart. In figure 8.1, you find an example of a dynamic view presented with the Manning website’s cart functionality. Observe how requests to the same page manning.com/ cart receive different data in response. The information displayed is different, even if the page is the same. The page has dynamic content!

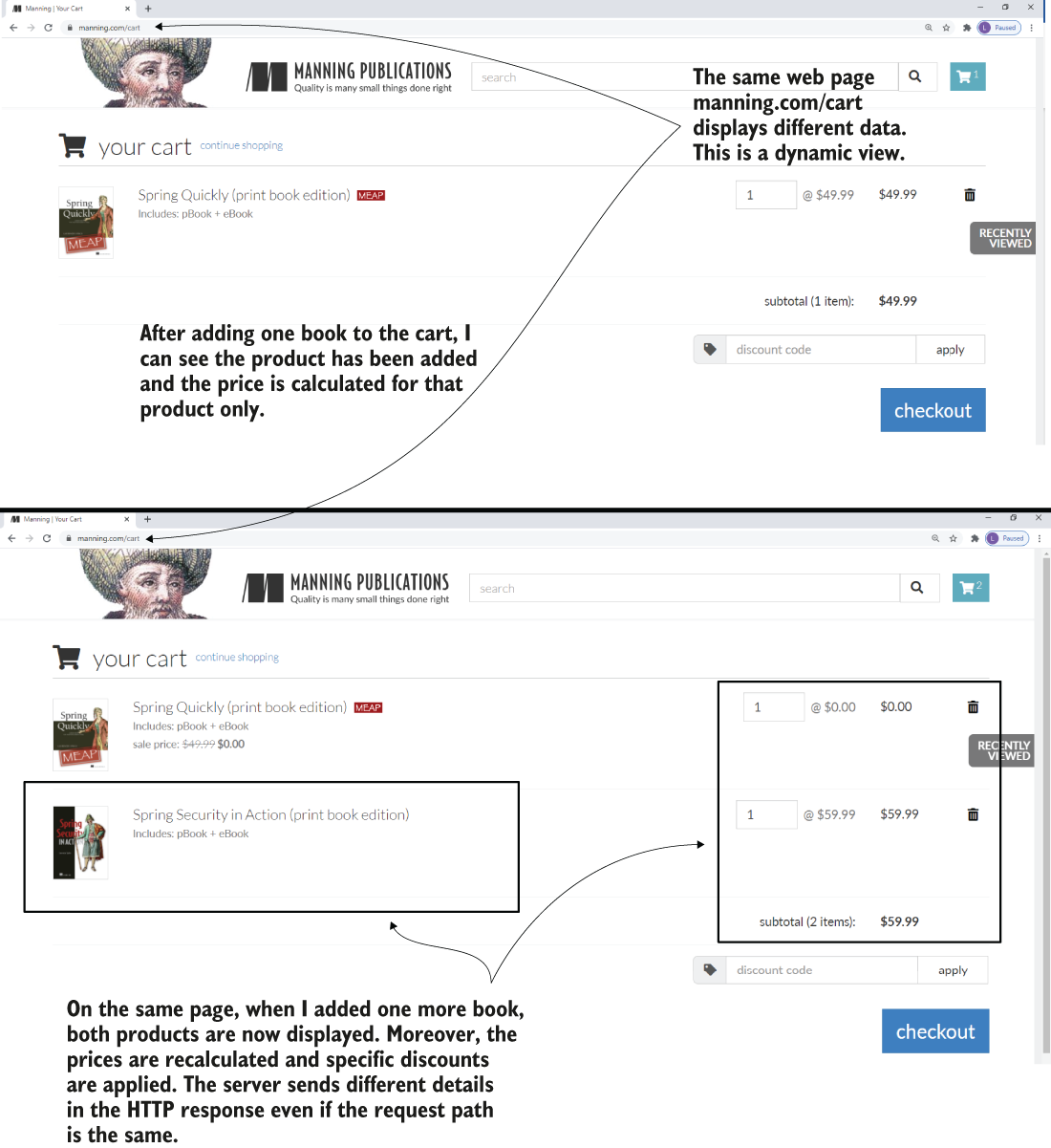


Figure 8.1 A dynamic view presented with the Manning shopping cart functionality. Even if the requested page is the same, the content of the page is different. The backend sent different data in the response before and after adding one more product to the cart.

In this section, we implement a web app with a dynamic view. Most apps today need to display dynamic data to the user. Now, for a user’s request expressed through an HTTP request sent by the browser, the web app receives some data, processes it, and then sends back an HTTP response that the browser needs to display (figure 8.2). We’ll review the Spring MVC flow and then work on an example to demonstrate how the view can get dynamic values from the controller.

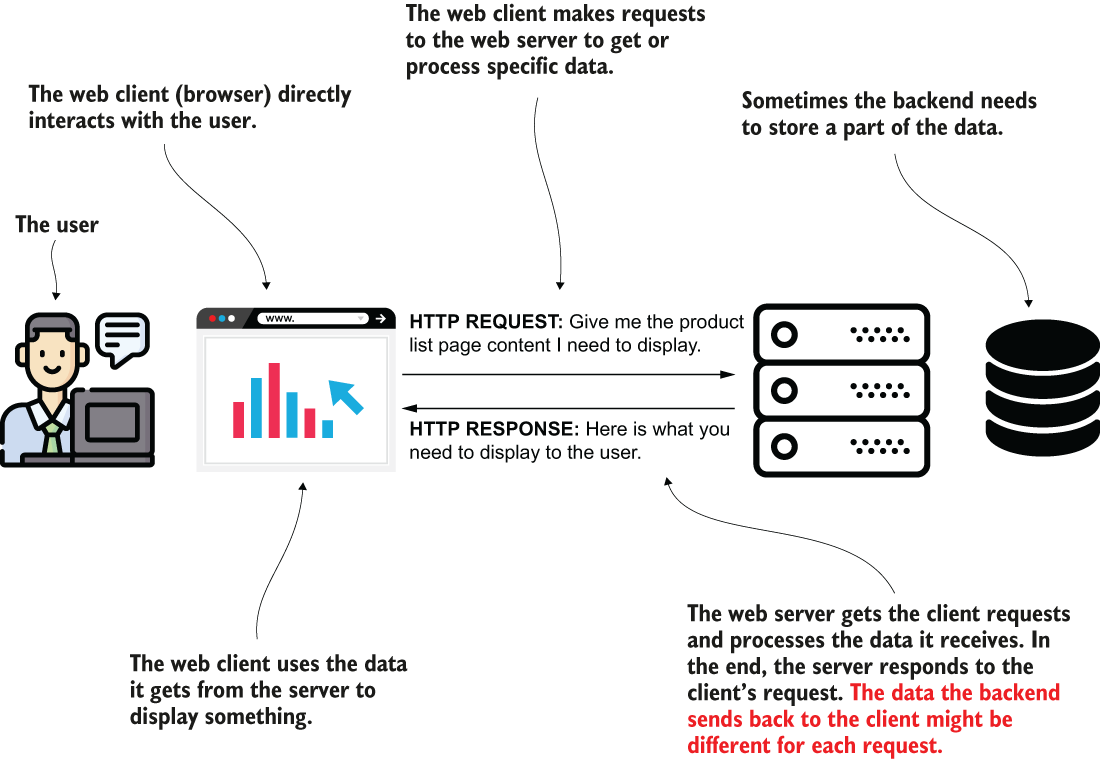


Figure 8.2 A client sends data through the HTTP request. The backend processes this data and builds a response to send back to the client. Depending on how the backend processed the data, different requests may result in other data displayed to the user.

In the previous example, the browser’s content was the same for every HTTP request for our page. Remember the Spring MVC flow (figure 8.3):

1. The client sends an HTTP request to the web server.
2. The dispatcher servlet uses the handler mapping to find out what controller action to call.
3. The dispatcher servlet calls the controller’s action.
4. After executing the action associated with the HTTP request, the controller returns the view name the dispatcher servlet needs to render into the HTTP response.
5. The response is sent back to the client.

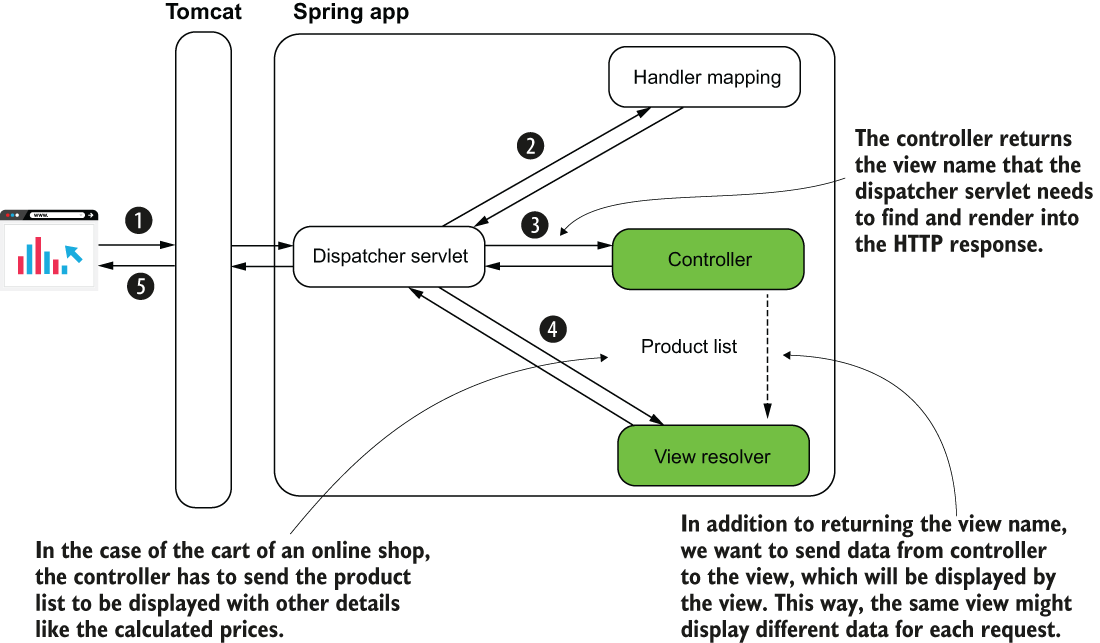


Figure 8.3 The Spring MVC flow. To define a dynamic view, the controller needs to send data to the view. The data the controller sends can be different for each request. For example, in an online shop’s cart functionality, the controller initially sends a list of one product to the view. After the user adds more products, the list the controller sends contains all the products in the cart. The same view shows different information for these requests.

Number 4 is where we need to make a change. We want the controller not only to return the view name but somehow also send data to the view. The view will incorporate this data to define the HTTP response. This way, if the server sends a list of one product, and the page displays the list, the page will display one product. If the controller sends two products for the same view, now the displayed data will be different because the page will show two products (the behavior you observed in figure 8.1).

## **PROJECT 02 – Thymeleaf – dinamic view - send data from the Controller to the View**

This example is simple, but you can use this approach to send any data from the controller to the view.

Let’s assume for now we want to send a name and print it with a specific color. In a real-world scenario, you’d maybe need to print the name of the user somewhere on the page.

We’ll create a Spring Boot project and add a template engine **Thymeleaf** to the dependencies in the pom.xml file. The template engine is a dependency that allows us to easily send data from the controller to the view and display this data in a specific way. I chose Thymeleaf because it’s less complex than others, and I find it easier to understand and learn. The templates used with Thymleaf are simple HTML static files.

The dependency you need to add to the pom.xml file:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId> ❶

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId> ❷

</dependency>

❶ The dependency starter that needs to be added to use Thymeleaf as a template engine

❷ Though you’re building a web app, you still need to add the dependency starter for web apps.

In the definition of the controller we annotate the method to map the action to a specific request path using @RequestMapping. We now also define a parameter to the method. This parameter of type **Model** stores the data we want the controller to send to the view. In this Model instance, we add the values we want to send to the view and identify each of them with a unique name (also referred to as key). To add a new value that the controller sends to the view, we call the **addAttribute()** method. The first parameter of the addAttribute() method is the key; the second parameter is the value you send to the view.

Listing 8.1 The controller class defines the page action

@Controller ❶

public class MainController {

@RequestMapping("/home") ❷

public String home(Model page) { ❸

page.addAttribute("username", "Katy"); ❹

page.addAttribute("color", "red"); ❹

return "home.html"; ❺

}}

❶ The @Controller stereotype annotation marks this class as Spring MVC controller and adds a bean of this type to the Spring context.

❷ We assign the controller’s action to an HTTP request path.

❸ The action method defines a parameter of type Model that stores the data the controller sends to the view.

❹ We add the data we want the controller to send to the view.

❺ The controller’s action returns the view to be rendered into the HTTP response.

We get an error if we directly add to the browser’s address bar “localhost:8080” without a path like “/home.” The error is a default page you see displayed by a Spring Boot app when you get an HTTP 404 (Not Found) response status. When you call directly “localhost:8080” you refer to the path “/.” Because you didn’t assign any controller action to this path, it’s normal to get an HTTP 404. If you wish to see something else instead, assign a controller action to this path as well using the @RequestMapping annotation.

To define the view, you need to add a new “home.html” file to your Spring Boot project’s “resources/templates” folder. The small difference: beforehand, we added the HTML file in the “resources/static” folder because we created a static view. Now that we’re using a template engine to create a dynamic view, you need to add the HTML file to the “resources/templates” folder instead.

In the content of the “home.html” file I added to the project, the first important thing to notice in the file’s content is the <html> tag where I added the attribute xmlns:th="http://www.thymeleaf.org". This definition is equivalent to an import in Java. It allows us further to use the prefix “th” to refer to specific features provided by Thymeleaf in the view.

A little bit further in the view, you find 2 places where I used this “th” prefix to refer to the controller’s data to the view. With the ${attribute\_key} syntax, you refer to any of the attributes you send from the controller using the Model instance. For example, I used the ${username} to get the value of the “username” attribute and ${color} to get the value of the “color” attribute.

Listing 8.2 The home.html file representing the dynamic view of the app

<!DOCTYPE html>

<html lang="en" xmlns:th="http://www.thymeleaf.org"> ❶

<head>

<meta charset="UTF-8">

<title>Home Page</title>

</head>

<body>

<h1>Welcome

<span th:style="'color:' + ${color}" ❷

th:text="${username}"></span>!</h1> ❷

</body>

</html>

❶ Defines the Thymeleaf “th” prefix

❷ Uses the “th” prefix to use the values sent by the controller

Start the application and access the web page in a browser. Your page will look like the one in figure 8.4.

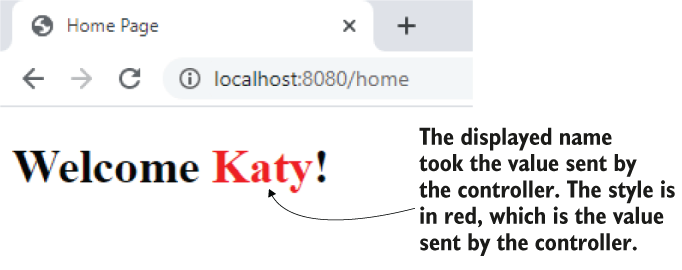


Figure 8.4 The result. Running the app and accessing the page in a browser, you observe the view that uses the values the controller sent.

Now, whatever your controller sends, the view uses.

## **PROJECT 03 – Thymeleaf – @RequestParam to request parameters – send data from the client to the server**

**8.1.1 Getting data on the HTTP request**

In this section, we discuss how the client sends data to the server through HTTP requests. In apps, we often need to give the client the ability to send information to the server. This data gets processed and then displayed on the view. Here are some examples of use cases where the client has to send data to the server:

* You implement the order functionality of an online shop. The client needs to send to the server the products that the user orders. Further, the server takes care of the order.
* You implement a web forum where you allow users to add and edit new posts. The client sends the post details to the server, which stores or changes the details in a database.
* You implement the login functionality of an app. Users write their credentials, which need to be validated. The client sends the credentials to the server, and the server validates these credentials.
* You implement the contact page of a web app. The page displays a form where the user can write a message subject and body. These details need to be sent in an email to a specific address. The client sends these values to the server, and the server takes care of processing them and sending an email to the desired email address.

In most cases, to send data through the HTTP request you use one of the following ways:

* An *HTTP request parameter* represents a simple way to send values from client to server in a key-value(s) pair format. To send HTTP request parameters, you append them to the URI in a request query expression. They are also called *query parameters*. You should use this approach only for sending a small quantity of data.
* An *HTTP request header* is similar to the request parameters in that the request headers are sent through the HTTP header. The big difference is that they don’t appear in the URI, but you still cannot send large quantities of data using HTTP headers.
* A *path variable* sends data through the request path itself. It is the same as for the request parameter approach: you use a path variable to send a small quantity of data. But we should use path variables when the value you send is mandatory.
* The *HTTP request body* is mainly used to send a larger quantity of data (formatted as a string, but sometimes even binary data such as a file).

**8.1.2 Using request parameters to send data from client to server**

Here we implement an example to demonstrate the use of HTTP request parameters—simple ways to send data from the client to the backend. You often encounter this approach in production apps. You use request parameters in the following scenarios:

* *The quantity of data you send is not large*. You set the request parameters using query variables. This approach limits you to about 2,000 characters.
* *You need to send optional data*. A request parameter is a clean way to deal with a value the client might not send. The server can expect to not get a value for specific request parameters.

An often-encountered use case for request parameters used is defining some search and filtering criteria (figure 8.5). Say your app displays product details in a table. Each product is identified by a name, a price, and a brand. You want to allow the user to search for products by any of these. The user might decide to search by price or by name and brand. Any combination is possible. For such a scenario, request parameters are the right choice for implementation. The app sends each of these values (name, price, and brand) in optional request parameters. The client only needs to send the values by which the user decides to search.

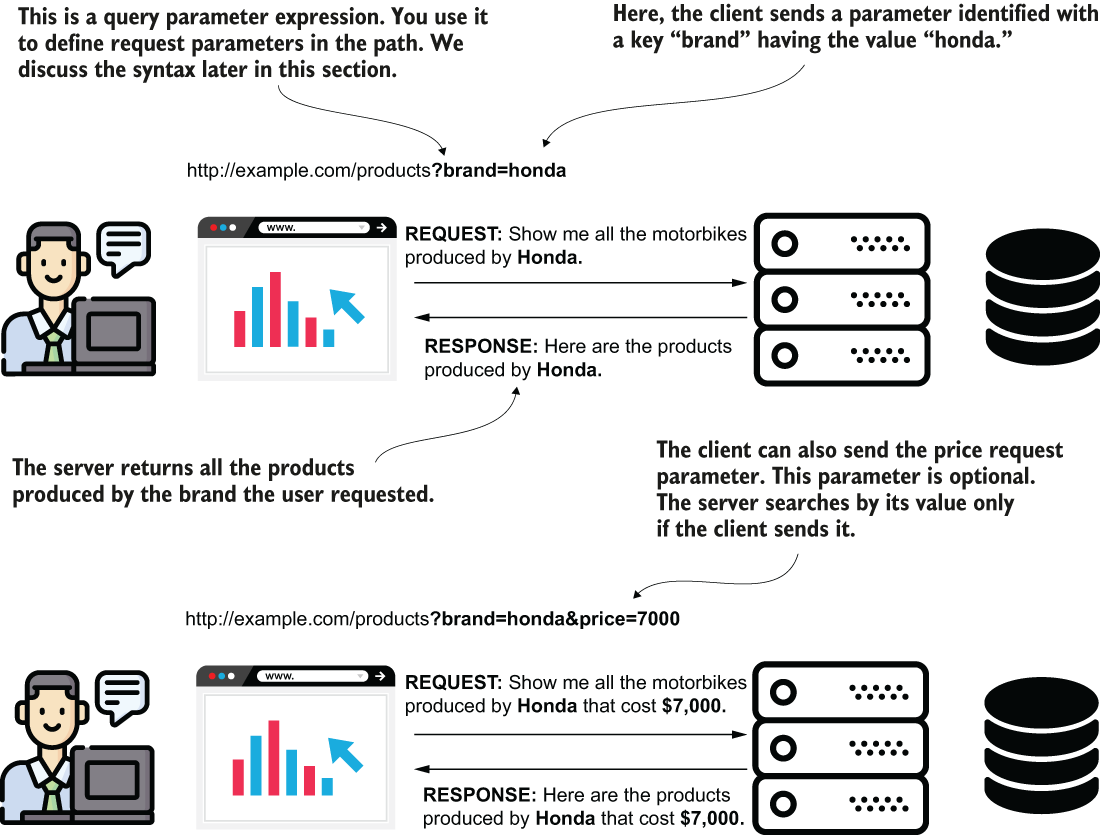


Figure 8.5 A request parameter can be optional. A common scenario for using request parameters is implementing a search functionality where the search criteria are optional. The client sends only some of the request parameters, and the server knows to use only the values it receives. You implement the server to consider it might not get values for some of the parameters.

Let’s use a request parameter by changing the example we discussed to get the color in which the username is displayed from the client. To get the value from a request parameter, you need to add one more parameter to the controller’s action method and annotate that parameter with the **@RequestParam** annotation. The @RequestParam annotation tells Spring it needs to get the value from the HTTP request parameter with the same name as the method’s parameter name.

Listing 8.3 Getting a value through a request parameter

@Controller

public class MainController {

@RequestMapping("/home")

public String home(

@RequestParam String color, ❶

Model page) { ❷

page.addAttribute("username", "Katy");

page.addAttribute("color", color); ❸

return "home.html";

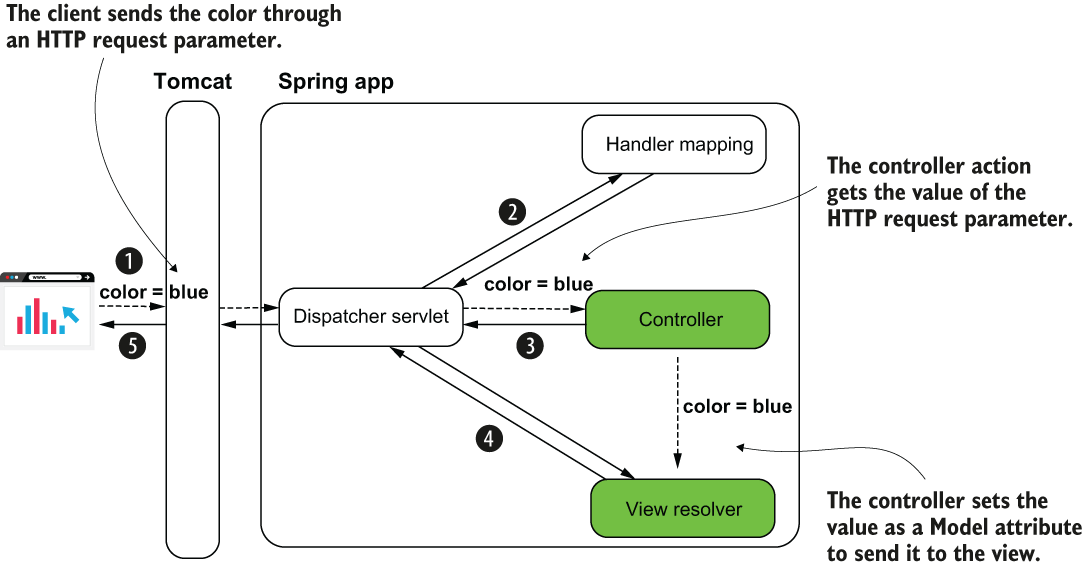
}}

❶ We define a new parameter for the controller’s action method and annotate it with @RequestParam.

❷ We also add the Model parameter that we use to send data from the controller to the view.

❸ The controller passes the color sent by the client to the view.

Figure 8.6 shows how the color parameter value travels from the client to the controller’s action on the backend to be used by the view.



Run the application and access the /home path. To set the request parameter’s value, you need to use the next snippet’s syntax:

<http://localhost:8080/home?color=blue>

When setting HTTP request parameters, you extend the path with a ? symbol followed by pairs of key=value parameters separated by the & symbol. For example, if I want to also send the name as a request parameter, I write:

<http://localhost:8080/home?color=blue&name=Jane>

## **PROJECT 04 – Thymeleaf – optional @RequestParam**

You can add a new parameter to the controller’s action to get this parameter as well.

@Controller

public class MainController {

@RequestMapping("/home")

public String home(

@RequestParam(required = false) String name, ❶

@RequestParam(required = false) String color,

Model page) {

page.addAttribute("username", name); ❷

page.addAttribute("color", color);

return "home.html";

}}

❶ Gets the new request parameter “name”

❷ Sends the “name” parameter’s value to the view

In the group key=value (for example, color=blue), “key” is the name of the request parameter, and its value is written right after the = symbol.

Figure 8.7 visually summarizes the syntax for request parameters.

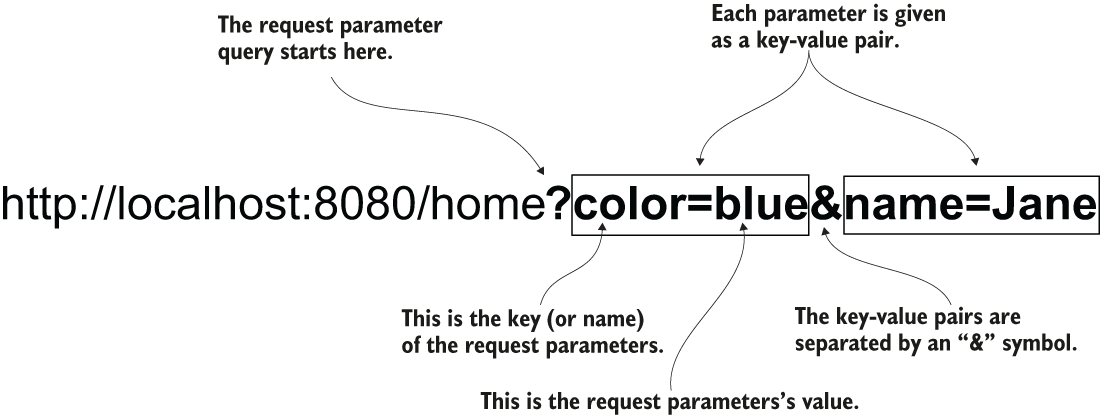


Figure 8.7 Sending data through request parameters. Each request parameter is a key-value pair. You provide the request parameters with the path in a query starting with the question mark symbol. If you set more than one request parameter, you separate each key-value pair with the “and” (&) symbol.

! A request parameter is mandatory by default. If the client doesn’t provide a value for it, the server sends back a response with the status HTTP “400 Bad Request.” If you wish the value to be optional, you need to explicitly specify this on the annotation using the optional attribute: **@RequestParam(optional=true)** (or may be @RequestParam(required = false)).

<http://localhost:8080/home?name=Toshko&color=blue>

## **PROJECT 05 – Thymeleaf – @PathVariable – parameter in the URI**

**8.1.3 Using path variables to send data from client to server**

Using path variables is also a way of sending data from client to server. But instead of using the HTTP request parameters, you directly set variable values in the path.

*Using request parameters:*

<http://localhost:8080/home?color=blue>

*Using path variables:*

<http://localhost:8080/home/blue>

You don’t identify the value with a key anymore. You just take that value from a precise position in the path. On the server side, you extract that value from the path from the specific position. You may have more than one value provided as a path variable, but it’s generally better to avoid using more than a couple. You’ll observe that the path becomes more challenging to read if you go with more than 2 path variables. I prefer using request parameters for more than 2 values instead of path variables. Also, you shouldn’t use path variables for optional values. I recommend you use path variables only for mandatory parameters. If you have optional values to send in the HTTP request, you should use request parameters.

Table 8.1 A quick comparison of the request parameters and path variables approaches

|  |  |
| --- | --- |
| Request parameters | Path variables |
| 1. Can be used with optional values. 2. Avoid a large number of parameters. If you need to use more than 3, I recommend you use the request body. Avoid sending more than 3 query parameters for readability. 3. Some developers consider the query expression more difficult to read than the path expression. | 1. Should not be used with optional values.      1. Always avoid sending more than 3 path variables. It’s even better if you keep a max of 2. 2. Easier to read than a query expression. For a publicly exposed website, it’s also easier for search engines (e.g., Google) to index the pages. This advantage might make the website easier to find through a search engine. |

When the page you write depends on only 1 or 2 values that are the core of the end result, it’s better to write them directly in the path to make the request easier to read. The URL is also easier to find when you bookmark it in your browser and easier to index with a search engine (if it matters for your app).

To reference a path variable in the controller’s action, you simply give it a name and add it to the path between curly braces. You then use the **@PathVariable** annotation to mark the controller’s action parameter to get the path variable’s value.

Listing 8.4 Using path variables to get values from the client

@Controller

public class MainController {

@RequestMapping("/home/{color}") ❶

public String home(

@PathVariable String color, ❷

Model page) {

page.addAttribute("username", "Katy");

page.addAttribute("color", color);

return "home.html";

}}

❶ To define a path variable, you assign it a name and put it in the path between curly braces.

❷ You mark the parameter where you want to get the path variable value with the @PathVariable annotation. The name of the parameter must be the same as the name of the variable in the path.

Run the app and access the page in your browser with different values for the color.

http://localhost:8080/home/blue

http://localhost:8080/home/red

<http://localhost:8080/home/green>

Figure 8.8 visually represents the link between the code and the request path.

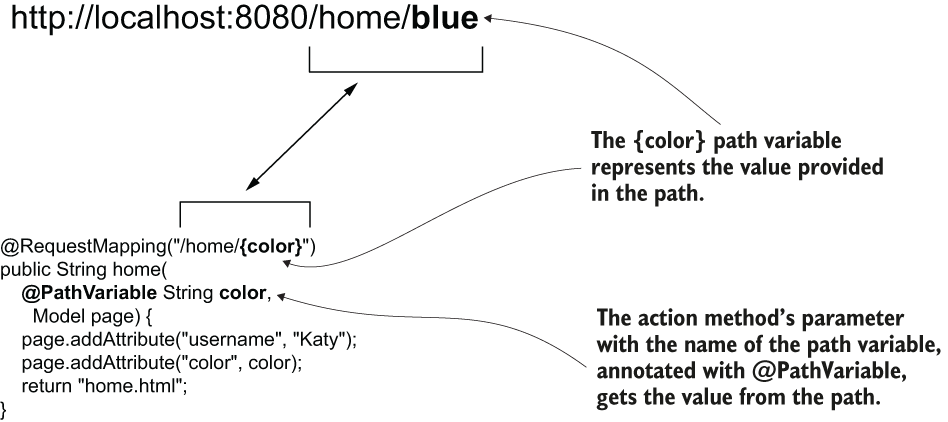


Figure 8.8 Using path variables. To get a value from a path variable, you give the variable a name between curly braces when defining the path on the controller action. You use a parameter annotated with @PathVariable to get the value of the path variable.

## **PROJECT 06 – Thymeleaf – GET and POST products**

**8.2 Using the GET and POST HTTP methods**

In this section, we discuss **HTTP methods** and how the client uses them to express what action (create, change, retrieve, delete) it will apply to the requested resource. A path and a verb identify an HTTP request. Thus far we have only referred to the path, and, without noticing, we used the HTTP GET method. Its purpose is to define what action the client requests. For example, by using GET, we represent an action that only retrieves data. It’s a way for the client to say it wants to obtain something from the server, but the call won’t change data. But you’ll need more than this. An app also needs to change, add, or delete data.

Be careful! You can use an HTTP method against its designed purpose, but this is incorrect. For example, you could use HTTP GET and implement a functionality that changes data. Technically, this is possible, but it’s a bad choice. Never use an HTTP method against its designed purpose.

We’ve relied on the request path to reach a specific action of the controller, but in a more complex scenario you can assign the same path to multiple actions of the controller as long as you use different HTTP methods. We’ll work on an example to apply such a case.

The HTTP method is defined by a verb and represents the client’s intention. If the client’s request only retrieves data, we implement the endpoint with HTTP GET. But if the client’s request somehow changes data on the server side, we use other verbs to represent the client’s intention clearly.

Table 8.2 Basic HTTP methods you’ll often encounter in web apps

|  |  |
| --- | --- |
| **HTTP method** | **Description** |
| GET | The client’s request only **retrieves** data. |
| POST | The client’s request sends **new data** to be **added** by the server. |
| PUT | The client’s request **changes** a data record on the server side. |
| PATCH | The client’s request **partially changes** a data record on the server side. |
| DELETE | The client’s request **deletes** data on the server side. |

Figure 8.9 visually presents the essential HTTP methods to help you remember them.

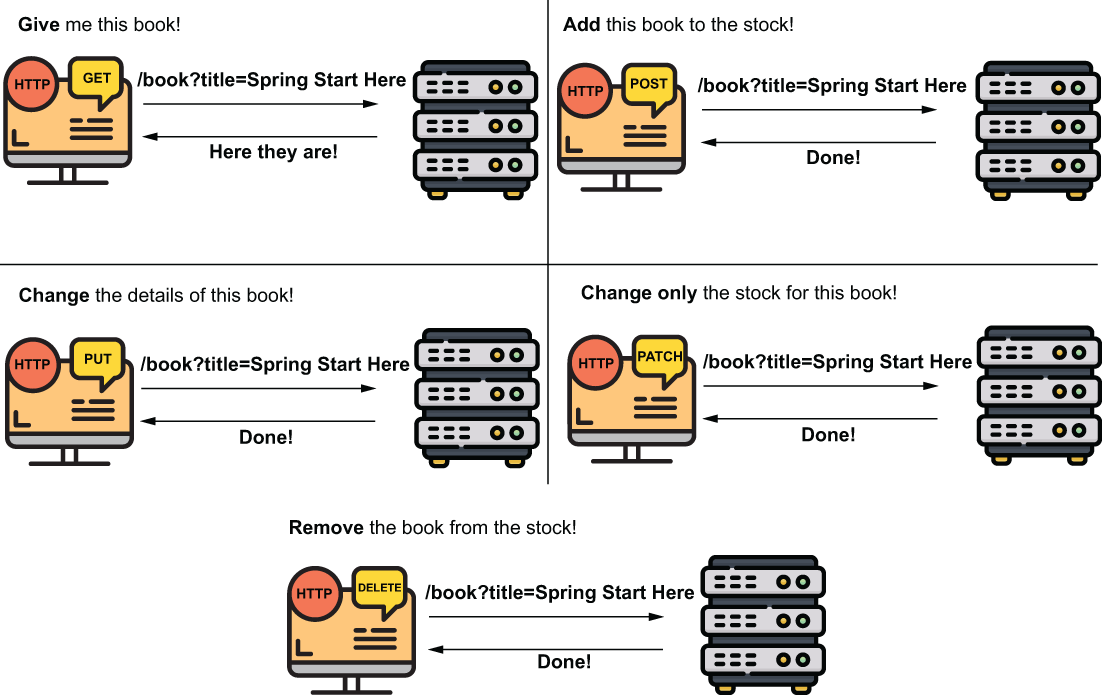


Figure 8.9 The basic HTTP methods. You use GET for retrieving data, POST for adding data, PUT for changing a record, PATCH for changing a part of the record, and DELETE to remove data. The client must use the appropriate HTTP method to express the action executed by a specific request.

! Even if it’s a good practice to make a distinction between entirely replacing a record (PUT) and changing only a part of it (PATCH) in production apps, this distinction is not always made.

We have to create an app that stores a list of products. Each product has a name and a price. The web app displays a list of all products and allows the user to add one more product through an HTML form.

Observe the 2 use cases described by the scenario. The user needs to do the following:

* View all products in the list; here, we’ll continue using HTTP GET.
* Add products to the list; here, we’ll use HTTP POST.

We create a new project, with the dependencies (in the pom.xml file) for web and Thymeleaf:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

In the project, we create a **Product** class to describe a product with its name and price attributes. The Product class is a model class, so we’ll create it in a package named “model”.

Listing 8.5 The Product class describes a product with name and price as attributes

public class Product {

private String name;

private double price;

// Omitted getters and setters

}

Now that we have a way to represent a product, let’s create the list where the app stores the products. The web app will display the product in this list on a web page, and in this list the user can add more products. We will implement the 2 use cases (getting the list of products to display and adding a new product) as methods in a service class. Let’s create a new service class named ProductService in a package named “service.”

The service class instantiates a list and defines 2 methods for adding a new product and getting it.

Listing 8.6 The ProductService class implements the app’s use cases

@Service

public class ProductService {

private List<Product> products = new ArrayList<>();

public void addProduct(Product p) {

products.add(p);

}

public List<Product> findAll() {

return products;

}}

! This design is a simplification. Remember that a Spring bean’s scope by default is singleton, and a web application implies multiple threads (one for each request). Changing a list defined as an attribute of the bean would cause race condition situations in a real-world app where more clients add products simultaneously. For now, we’ll keep our simplification, because in the next chapters we’ll replace the list with a database, so this problem will no longer occur. But keep in mind this is a vicious approach, and, you shouldn’t use something similar in a production-ready app. Singleton beans aren’t thread-safe!

A controller will call the use cases implemented by the service. The controller gets data about a new product from the client and adds it to the list by calling the service, and the controller gets the list of products and sends it to the view. First, let’s create a ProductController class in a package named “controllers” and allow this controller to inject the service bean.

Listing 8.7 The ProductController class uses the service to call the use cases

@Controller

public class ProductsController {

private final ProductService productService;

public ProductsController(ProductService productService) { ❶

this.productService = productService;

}}

❶ We use DI through the controller’s constructor parameters to get the service bean from the Spring context.

Now we expose the 1st use case: displaying the product list on a page. We use a Model parameter to send the data from the controller to the view.

Listing 8.8 Sending the list of products to the view

@Controller

public class ProductsController {

private final ProductService productService;

public ProductsController(ProductService productService) {

this.productService = productService;

}

@RequestMapping("/products") ❶

public String viewProducts(Model model) { ❷

var products = productService.findAll(); ❸

model.addAttribute("products", products); ❹

return "products.html"; ❺

}}

❶ We map the controller action to the /products path. The @RequestMapping annotation, by default, uses the HTTP GET method.

❷ We define a Model parameter that we use to send the data to the view.

❸ We get the product list from the service.

❹ We send the product list to the view.

❺ We return the view name, which will be taken and rendered by the dispatcher servlet.

To display the products in the view, we define the products.html page in the “resources/templates” folder of the project, as you learned in section 8.1. The following listing shows you the content of the “products.html” file, which takes the list of products the controller sends and displays it in an HTML table.

Listing 8.9 Displaying the products on the page

<!DOCTYPE html>

<html lang="en" xmlns:th="http://www.thymeleaf.org"> ❶

<head>

<meta charset="UTF-8">

<title>Home Page</title>

</head>

<body>

<h1>Products</h1>

<h2>View products</h2>

<table>

<tr> ❷

<th>PRODUCT NAME</th> ❷

<th>PRODUCT PRICE</th> ❷

</tr> ❷

<tr th:each="p: ${products}" > ❸

<td th:text="${p.name}"></td> ❹

<td th:text="${p.price}"></td> ❹

</tr>

</table>

</body>

</html>

❶ We define the “th” prefix to use the Thymeleaf capabilities.

❷ We define a static header for our table.

❸ We use the th:each feature from Thymeleaf to iterate on the collection and display a table row for each product in the list.

❹ We display the name and the price of each product on one row.

Figure 8.10 presents the flow for calling the /products path with HTTP GET on the Spring MVC diagram:

1. The client sends an HTTP request for the /products path.
2. The dispatcher servlet uses the handler mapping to find the controller’s action to call for the /products path.
3. The dispatcher servlet calls the controller’s action.
4. The controller requests the product list from the service and sends it to be rendered with the view.
5. The view is rendered into an HTTP response.
6. The HTTP response is sent back to the client.

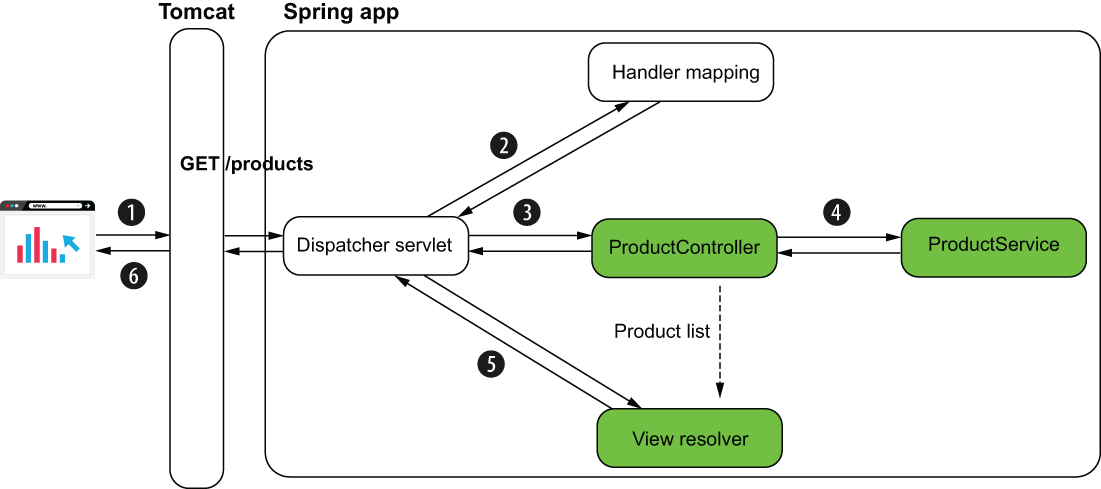


Figure 8.10 When calling /products with HTTP GET, the controller gets the product list from the service and sends it to the view. The HTTP response contains the HTML table with the products from the list.

We still need to implement the 2nd use case. We’ll see nothing more than an empty table if we don’t have an option to add a product to the list. Let’s change the controller and add an action to allow adding a product to the product list.

Listing 8.10 Implementing the action method for adding a product

@Controller

public class ProductsController {

// Omitted code

@RequestMapping(path = "/products", method = RequestMethod.POST) ❶

public String addProduct(

@RequestParam String name, ❷

@RequestParam double price, ❷

Model model

) {

Product p = new Product(); ❸

p.setName(name); ❸

p.setPrice(price); ❸

productService.addProduct(p); ❸

var products = productService.findAll(); ❹

model.addAttribute("products", products); ❹

return "products.html"; ❺

}}

❶ We map the controller action to the /products path. We use the method attribute of the @RequestMapping annotation to change the HTTP method to POST.

❷ We get the name and the price for the product to add using request parameters.

❸ We build a new Product instance and add it to the list by calling the service use case method.

❹ We get the list of products and send it to the view.

❺ We return the name of the view to be rendered.

We used the attribute method of the @RequestMapping annotation to specify the HTTP method. If you don’t set a method, *by default @RequestMapping uses HTTP GET*. But because both the path and the method are essential for any HTTP call, we want to always confirm both. For this reason, developers usually use dedicated annotations for each HTTP method instead of @RequestMapping. For apps, you’ll often find developers using **@GetMapping** to map a GET request to an action, **@PostMapping** for a request using HTTP POST, and so on. We’ll also change our example to use these dedicated annotations for HTTP methods.

Listing 8.11 The ProductController class

@Controller

public class ProductsController {

private final ProductService productService;

public ProductsController(ProductService productService) {

this.productService = productService;

}

@GetMapping("/products") ❶

public String viewProducts(Model model) {

var products = productService.findAll();

model.addAttribute("products", products);

return "products.html";

}

@PostMapping("/products") ❷

public String addProduct(

@RequestParam String name,

@RequestParam double price,

Model model

) {

Product p = new Product();

p.setName(name);

p.setPrice(price);

productService.addProduct(p);

var products = productService.findAll();

model.addAttribute("products", products);

return "products.html";

}}

❶ @GetMapping maps the HTTP GET request with a specific path to the controller’s action.

❷ @PostMapping maps the HTTP POST request with a specific path to the controller’s action.

We can also change the view to allow the user to call the controller’s HTTP POST action and add a product to the list. We’ll use an HTML form to make this HTTP request. The following listing presents the changes we need to make on the products.html page (our view) to add the HTML form.

Listing 8.12 Adding an HTML form to the view for adding a product to the list

<!DOCTYPE html>

<html lang="en" xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="UTF-8">

<title>Home Page</title>

</head>

<body>

<!-- Omitted code -->

<h2>Add a product</h2>

<form action="/products" method="post"> ❶

Name: <input

type="text"

name="name"><br /> ❷

Price: <input

type="number"

step="any"

name="price"><br /> ❸

<button type="submit">Add product</button> ❹

</form>

</body>

</html>

❶ When submitted, the HTML form makes a POST request for path /products.

❷ An input component allows the user to set the name of the product. The value in the component is sent as a request parameter with the key “name.”

❸ An input component allows the user to set the price of the product. The value in the component is sent as a request parameter with the key “price.”

❹ The user uses a submit button to submit the form.

Run and test the app. You access the page in your browser on <http://localhost:8080/products> , and you should be able to add new products and see those already added. Figure 8.11 shows the result.

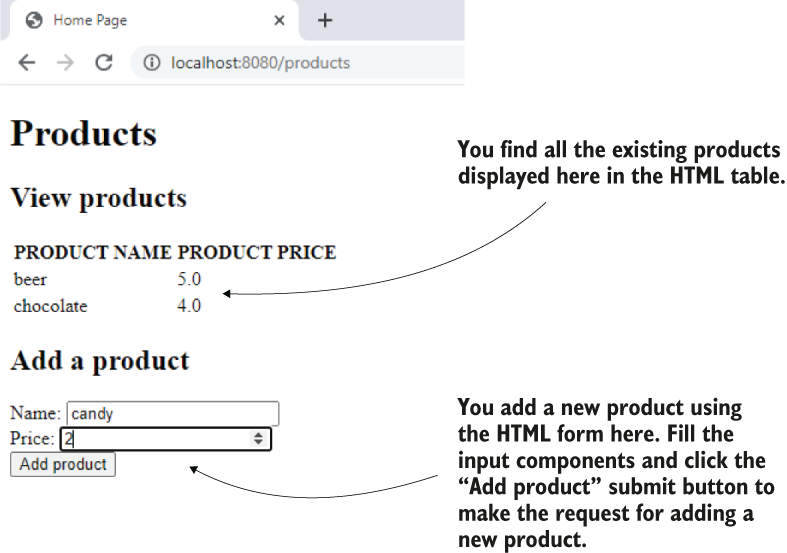


Figure 8.11 The final result. A user sees the products in the HTML table on the page and can add a new product through the HTML form.

## **PROJECT 07 – Thymeleaf – model is direct parameter in the controller**

In our example, I used the @RequestParam annotation. I used this annotation here to make it clear how the client sends the data. But sometimes Spring allows you to omit code. For example, you could use a Product as a parameter of the controller’s action directly. Because the request parameters’ names are the same as the Product class attributes’ names, Spring knows to match them and automatically creates the object. For someone who already knows Spring, this is excellent because it spares you from writing code lines. But beginners might get confused by all these details. Say you find an example in an article that uses this syntax. It might be unclear where the Product instance comes from. If you’ve just begun learning Spring and find yourself in such a situation, my advice is to be aware that Spring tends to have plenty of syntaxes to hide as much code as possible. Whenever you find a syntax you don’t clearly understand in an example or article, try finding the framework specification details.

Listing 8.13 Directly using the model as a parameter of the controller’s action

@Controller

public class ProductsController {

// Omitted code

"/products")

public String addProduct(

Product p, ❶

Model model

) {

productService.addProduct(p);

var products = productService.findAll();

model.addAttribute("products", products);

return "products.html";

}}

❶ You can use the model class as a parameter of the controller’s action directly. Spring knows to create the instance based on the request attributes. *The model class needs to have a default constructor* to allow Spring to create the instance before calling the action method.

**Summary**

* Today’s web apps have dynamic pages (also called dynamic views). A dynamic page might display different content for different requests.
* To know what to display, a dynamic view gets the variable data from the controller.
* An easy way to implement dynamic pages in Spring apps is using a template engine such as Thymeleaf. Alternatives to Thymeleaf are Mustache, FreeMarker, and Java Server Pages (JSP).
* A template engine is a dependency that provides your app with the capability to easily get the data the controller sends and display it on the view.
* The client can send data to the server through request parameters or path variables. A controller’s action gets the details the client sends in parameters annotated with @RequestParam or @PathVariable.
* A request parameter can be optional.
* You should only use path variables for mandatory data the client sends.
* A path and an HTTP method identify an HTTP request. The HTTP method is represented by a verb that identifies the client’s intention. The essential HTTP methods you’ll often find in production apps are GET, POST, PUT, PATCH, and DELETE.
  + GET expresses the client’s intention to retrieve data without changing data on the backend.
  + POST expresses the client’s intention to add new data on the server side.
  + PUT expresses the client’s intention to change a data record on the backend entirely.
  + PATCH expresses the client’s intention to change a part of a data record on the backend.
  + DELETE expresses the client’s intention to remove data on the backend.
* Through a browser’s HTML form process directly, you can use only HTTP GET and HTTP POST. To use other HTTP methods such as DELETE or PUT, you need to implement the call using a client language such as JavaScript.

9 Using the Spring web scopes

In any Spring app, you can choose to declare a bean as one of the following:

* *Singleton*—The default bean scope in Spring, for which the framework uniquely identifies each instance with a name in the context
* *Prototype*—The bean scope in Spring, for which the framework only manages the type and creates a new instance of that class every time someone requests it (directly from the context or through wiring or auto-wiring).

In this chapter, you’ll learn that in web apps you can use other bean scopes that are relevant only to web applications. We call them web scopes:

* *Request scope*—Spring creates an instance of the bean class for every HTTP request. The instance exists only for that specific HTTP request.
* *Session scope*—Spring creates an instance and keeps the instance in the server’s memory for the full HTTP session. Spring links the instance in the context with the client’s session.
* *Application scope* —The instance is unique in the app’s context, and it’s available while the app is running.

We’ll work on an example in which we implement a login functionality.

In section 9.1, we’ll use a request-scoped bean to take the user’s credentials for login and make sure the app uses them only for the login request. Then, in section 9.2, we’ll use a session-scoped bean to store all the relevant details we need to keep for the logged-in user as long as the user remains logged in. In section 9.3, we’ll use the application-scoped bean to add a capability to count logins. Figure 9.1 shows you the steps we take to implement this app.

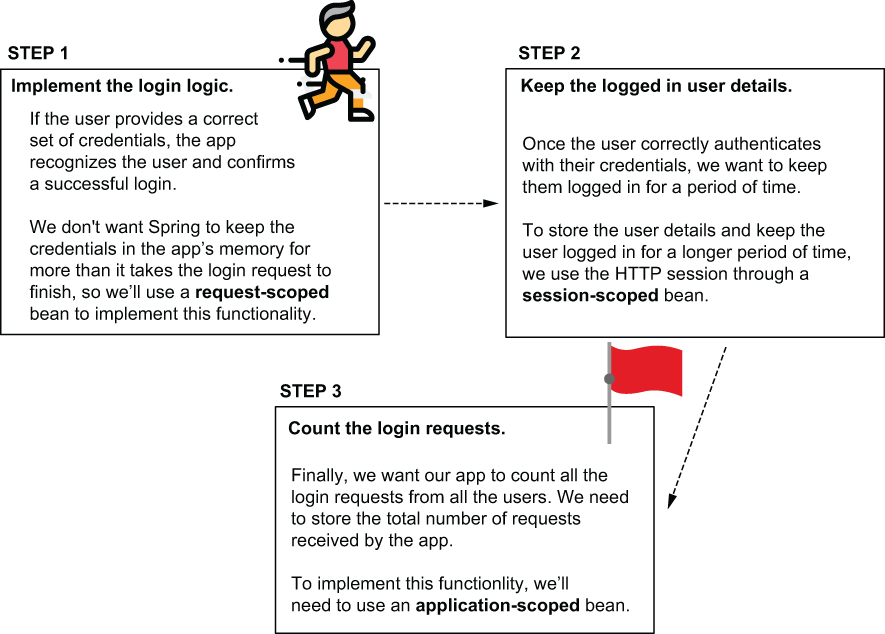


Figure 9.1 We’ll implement the login functionality in 3 steps. For each step we implement, we’ll need to use a different bean scope. In section 9.1, we’ll use a request-scoped bean to implement the login logic without risking storing the credentials for longer than the login request. We’ll then decide what details we need to store for the authenticated user in a session-scoped bean. Finally, we’ll implement a feature to count all the login requests, and we’ll use an application-scoped bean to keep the number.

**9.1 Using the request scope in a Spring web app**

A request-scoped bean is an object managed by Spring, for which the framework creates a new instance for every HTTP request. The app can use the instance only for the request that created it. Any new HTTP request (from the same or other clients) creates and uses a different instance of the same class (fig. 9.2).



Figure 9.2 For every HTTP request, Spring provides a new instance for the request-scoped bean. When using a request-scoped bean, you can be sure the data you add on the bean is available only on the HTTP request that created the bean. Spring manages the bean type (the plant) and uses it to get instances (coffee beans) for each new request.

Key aspects of request-scoped beans

|  |  |  |  |
| --- | --- | --- | --- |
| **Fact** | **Consequence** | **To consider** | **To avoid** |
| Spring creates a new instance for every HTTP request from any client. | Spring creates a lot of instances of this bean in the app’s memory during its execution. | The number of instances is usually not a big problem because these instances are short-lived. The app doesn’t need them for more than the time the HTTP request needs to complete. Once the HTTP request completes, the app releases the instances, and they are garbage-collected. | However, make sure you don’t implement a time-consuming logic Spring needs to execute to create the instance (like getting data from a database or implementing a network call). Avoid writing logic in the constructor or a *@PostConstruct* method for request-scoped beans. |
| Only one request can use an instance of a request-scoped bean. | Instances of request-scoped beans are not prone to multithread-related issues as only one thread (the one of the request) can access them. | You can use the instance’s attributes to store data used by request. | Don’t use synchronization techniques for the attributes of these beans. These techniques would be redundant, and they only affect the performance of your app. |

! A login example, is excellent for didactic purposes. However, in a production-ready app, it’s better to avoid implementing authentication and authorization mechanisms yourself. In a real-world Spring app, we use Spring Security to implement anything related to authentication and authorization. Using Spring Security (which is also part of the Spring ecosystem) simplifies your implementations and ensures you don’t (by mistake) introduce vulnerabilities when writing the application-level security logic.

To make things straightforward, we will consider a set of credentials that we bake into our application. In a real-world app, the app stores the users in a database. It also encrypts the passwords to protect them. For now, we focus only on the purpose of this chapter: discussing the Spring web bean scopes.

## **PROJECT 08 – Login – @RequestScope bean**

We’ll implement a web application’s login functionality, and we’ll use a request-scoped bean to manage the user’s credentials for the login logic.

Let’s create a Spring Boot project and add the needed dependencies. You can add the dependencies directly when creating the project (for example, using start.spring.io) or afterward in your pom.xml. For this example, we will use the web dependency and Thymeleaf as a templating engine:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

We’ll create a page that contains a login form asking for a user’s name and password. The app compares the username and the password with a set of credentials it knows (in my case, user “natalie” with password “password”). If we provide correct credentials (they match with the credentials the app knows), then the page displays a message “You are now logged in” under the login form. If the credentials we provide are not correct, then the app displays a message: “Login failed.”

We need to implement a page (representing our view) and a controller class. The controller sends the message it needs to display to the view according to the login’s result (figure 9.3).

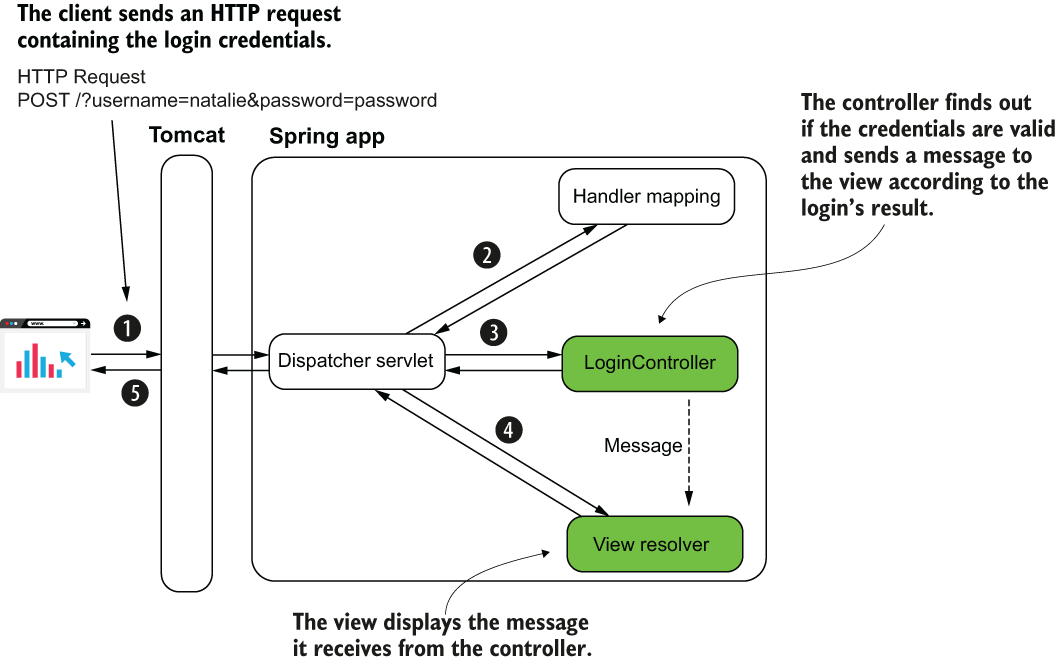


Figure 9.3 We need to implement the controller and the view. In the controller, we implement an action that finds out if the credentials sent in the login request are valid. The controller sends a message to the view, and the view displays this message.

The HTML login page that defines the view in our app you have to store in the resources/templates folder of your project. Let’s name the page “login.html.” To display the message with the logic’s result, we need to send a parameter from the controller to the view. I named this parameter “message,” and I used the syntax ${message} to display this in a paragraph under the login form.

Listing 9.1 The definition of the login page login.html

<!DOCTYPE html>

<html lang="en" xmlns:th="http://www.thymeleaf.org"> ❶

<head>

<meta charset="UTF-8">

<title>Login</title>

</head>

<body>

<form action="/" method="post"> ❷

Username: <input type="text" name="username" /><br /> ❸

Password: <input type="password" name="password" /><br /> ❸

<button type="submit">Log in</button> ❹

</form>

<p th:text="${message}"></p> ❺

</body>

</html>

❶ We define the “th” Thymeleaf prefix to use the templating engine’s capabilities.

❷ We define an HTML form to send the credentials to the server.

❸ The input fields are used to write the credentials, username, and password.

❹ When the user clicks the Submit button, the client makes an HTTP POST request with the credentials.

❺ We display a message with the result of the login request under the HTML form.

A controller action needs to get the HTTP request (from the dispatcher servlet), so let’s define the LoginController and the action that receives the HTTP request for the page we created. We map the controller’s action to the web app’s root path ("/").

Listing 9.2 The controller’s action mapped to the root path

@Controller ❶

public class LoginController {

@GetMapping("/") ❷

public String loginGet() {

return "login.html"; ❸

}}

❶ We use the @Controller stereotype annotation to define the class as a Spring MVC controller.

❷ We map the controller’s action to the root ("/ ") path of the application.

❸ We return the view name we want to be rendered by the app.

Now we want to implement the login logic. When a user clicks on the Submit button, we want the page to display a proper message under the login form. If the user submitted the correct set of credentials, the message is “You are now logged in”; otherwise, the displayed message will be “Login failed” (fig. 9.4)

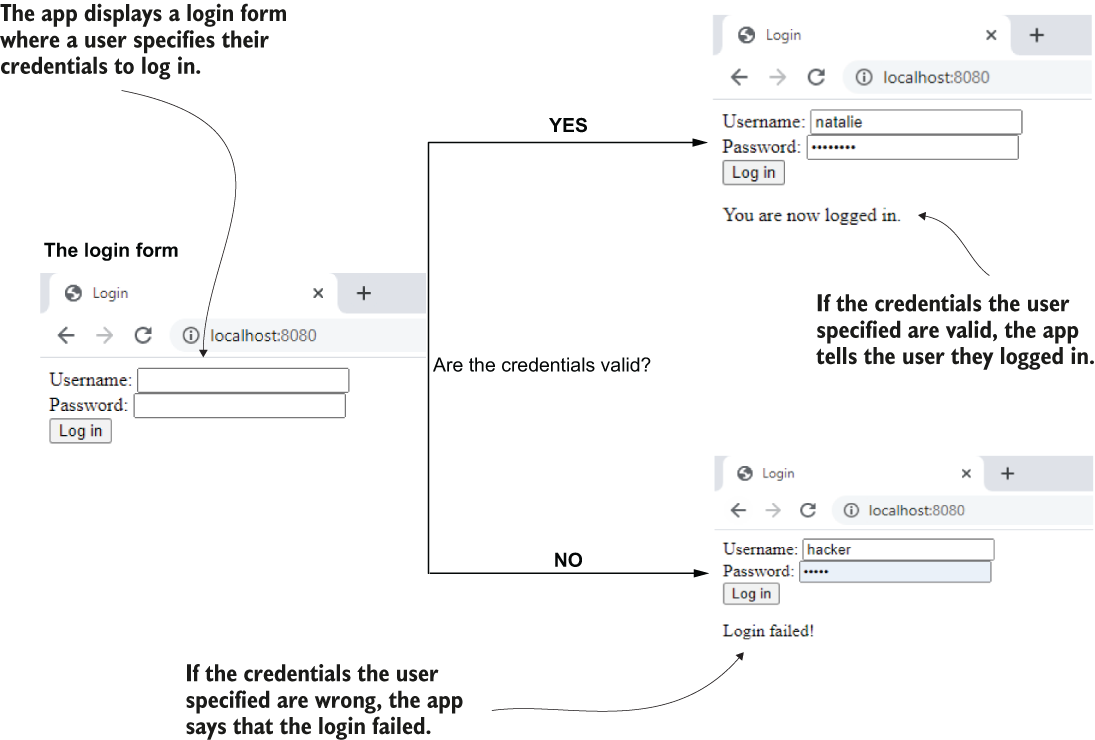


Figure 9.4 The functionality we implement in this section. The page displays a login form for the user. Then the user provides valid credentials, and the app displays a message that they successfully logged in. If the user provides incorrect credentials, the app tells the user that the login failed.

To process the HTTP POST request that the HTML form creates when the user clicks on the Submit button, we need to add 1 more action to our LoginController. This action takes the client’s request parameters (the username and the password) and sends a message to the view according to the login result.

Notice that we haven’t implemented the login logic. In the next listing, we take the request and send a message in response according to a variable representing the request’s result. But this variable is always “false.” In the next listings, we complete this action by adding a call to the login logic. This login logic will return the login result based on the credentials the client sent in the request.

Listing 9.3 The controller’s login action

@Controller

public class LoginController {

@GetMapping("/")

public String loginGet() {

return "login.html";

}

@PostMapping("/") ❶

public String loginPost(

@RequestParam String username, ❷

@RequestParam String password, ❷

Model model ❸

) {

boolean loggedIn = false; ❹

if (loggedIn) { ❺

model.addAttribute("message", "You are now logged in."); ❺

} else { ❺

model.addAttribute("message", "Login failed!"); ❺

} ❺

return "login.html"; ❻

}}

❶ We are mapping the controller’s action to the HTTP POST request of the login page.

❷ We get the credentials from the HTTP request parameters

❸ We declare a Model parameter to send the message value to the view.

❹ When we later implement the login logic, this variable will store the login request result.

❺ Depending on the result of the login, we send a specific message to the view.

❻ We return the view name, which is still login.html, so we remain on the same page.



Figure 9.5 The dispatcher servlet calls the controller’s action when someone submits the HTML login form. The controller’s action gets the credentials from the HTTP request parameters. According to the login result, the controller sends a message to the view, and the view displays this message under the HTML form.

The only class we wrote is the LoginController, and we left it a singleton, which is the default Spring scope. We don’t need to change the scope for LoginController as long as it doesn’t store any detail in its attributes. But, we need to implement the login logic. The login logic depends on the user’s credentials, and we have to take into consideration 2 things about these credentials:

1. The credentials are sensitive details, and you don’t want to store them in the app’s memory for longer than the login request.
2. More users with different credentials might attempt to log in simultaneously.

Considering these 2 points, we need to make sure that if we use a bean for implementing the login logic, each instance is unique for each HTTP request. We need to use a request-scoped bean. We’ll extend the app as presented in figure 9.5. We add a request-scoped bean LoginProcessor, which takes the credentials on the request and validates them (figure 9.6).

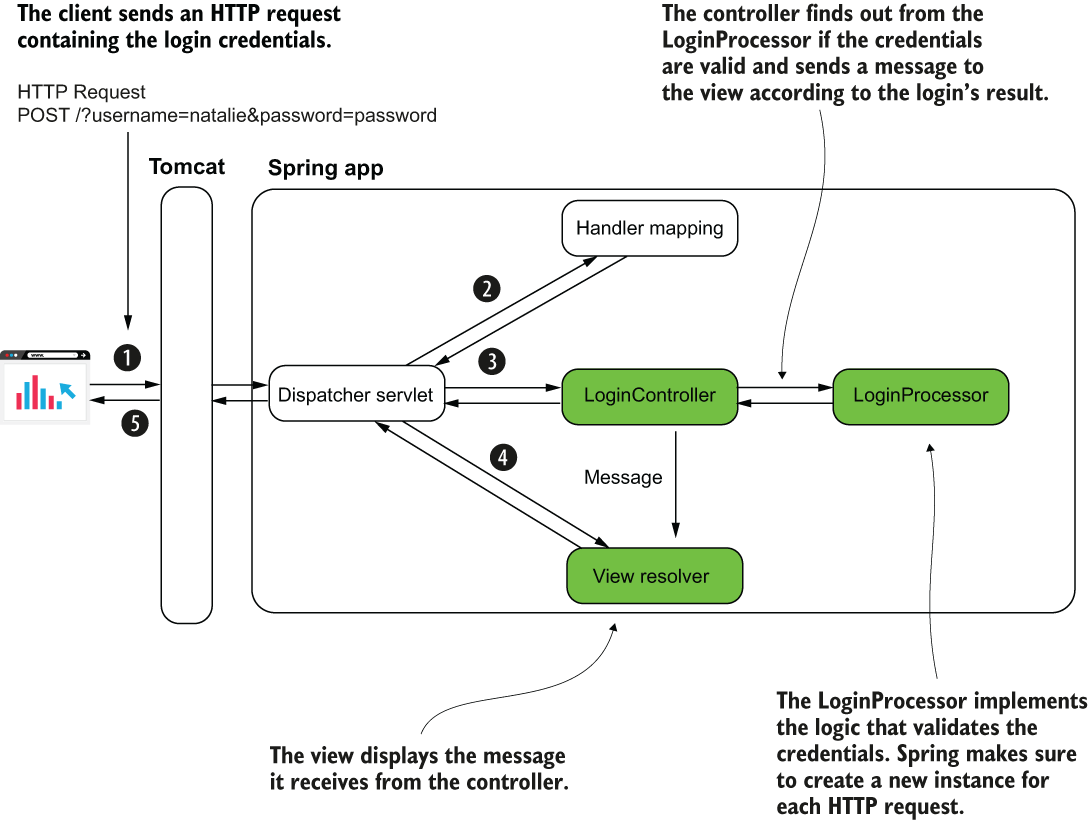


Figure 9.6 The LoginProcessor bean is request-scoped. Spring makes sure to create a new instance for each HTTP request. The bean implements the login logic. The controller calls a method it implements. The method returns true if the credentials are valid and false otherwise. Based on the value the LoginProcessor returns, the LoginController sends the right message to the view.

The implementation of the LoginProcessor class - to change the scope of the bean, we use the *@RequestScope* annotation. We need to make a bean of this class type in the Spring context by using the *@Bean* annotation in either a configuration class or a stereotype annotation. I chose to annotate the class with the *@Component* stereotype annotation.

Listing 9.4 Request-scoped LoginProcessor bean implementing the login logic

@Component ❶

@RequestScope ❷

public class LoginProcessor {

private String username; ❸

private String password; ❸

public boolean login() { ❹

String username = this.getUsername();

String password = this.getPassword();

if ("natalie".equals(username) && "password".equals(password)) {

return true;

} else {

return false;

}

}

// omitted getters and setters

}

❶ We annotate the class with a stereotype annotation to tell Spring this is a bean.

❷ We use the *@RequestScope* annotation to change the bean’s scope to request scope. This way, Spring creates a new instance of the class for every HTTP request.

❸ The bean stores the credentials as attributes.

❹ The bean defines a method for implementing the login logic.

You can run the application and access the login page using the localhost:8080 address in your browser’s address bar. Figure 9.7 shows you the app’s behavior after accessing the page and for using valid and incorrect credentials.

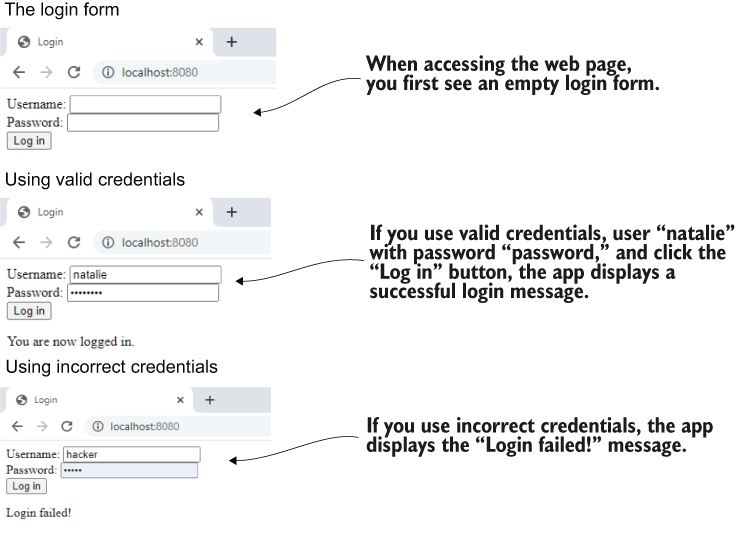


Figure 9.7 When accessing the page in a browser, the app shows a login form. You can use valid credentials, and the app displays a successful login message. If you use incorrect credentials, the app displays a “Login failed!” message.

## **PROJECT 09 – Login – @SessionScope bean**

**9.2 Using the session scope in a Spring web app**

When you enter a web app and log in, you expect to then surf through that app’s pages, and the app still remembers you’ve logged in. A session-scoped bean is an object managed by Spring, for which Spring creates an instance and links it to the HTTP session. Once a client sends a request to the server, the server reserves a place in the memory for this request, for the whole duration of their session. Spring creates an instance of a session-scoped bean when the HTTP session is created for a specific client. That instance can be reused for the same client while it still has the HTTP session active. The data you store in the session-scoped bean attribute is available for all the client’s requests throughout an HTTP session. This approach of storing the data allows you to store information about what users do while they’re surfing through the pages of your app.

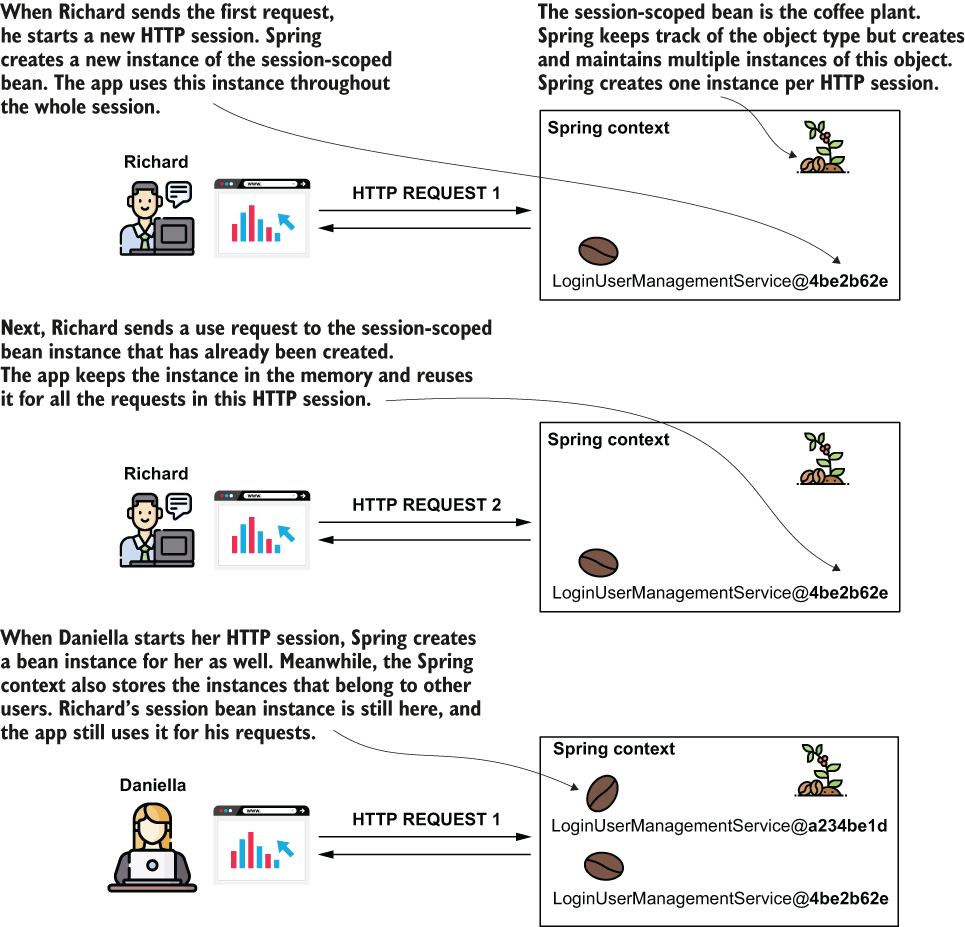


Figure 9.8 The session-scoped bean is used to keep a bean in the context throughout the client’s full HTTP session. Spring creates an instance of a session-scoped bean for each HTTP session a client opens. The client accesses the same instance for all the requests sent through the same HTTP session. Each user has their own session and accesses different instances of the session-scoped bean.

A session-scoped bean allows us to store data shared by multiple requests of the same client.

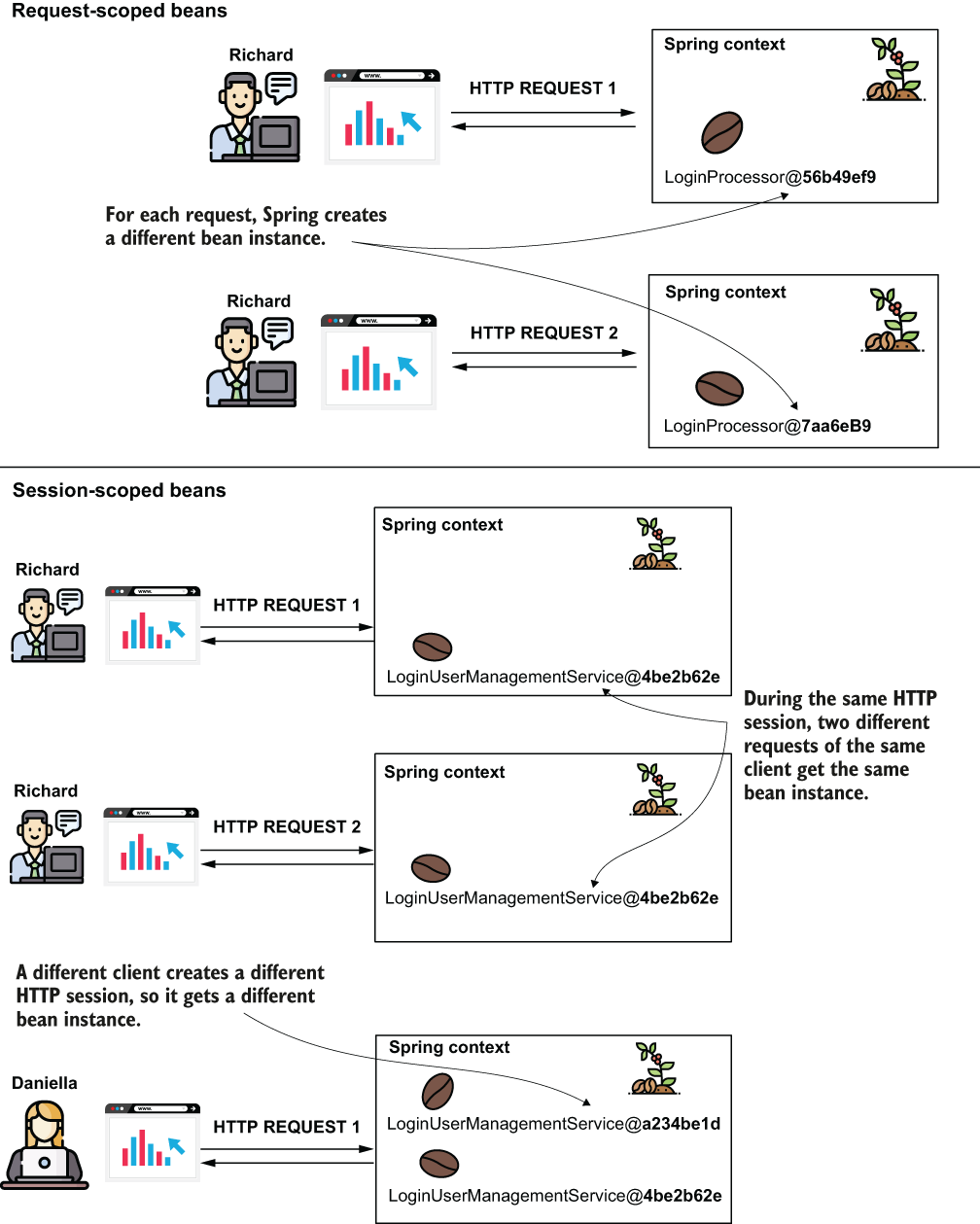


Figure 9.9 You use request-scoped beans when you want Spring to create a new instance for each request. You use a session-scoped bean when you want to keep the bean (together with any details it holds) throughout the client’s HTTP session.

A couple of features you can implement using session-scoped beans include the following examples:

* *A login* - Keeps details of the authenticated user while they visit different parts of your app and send multiple requests
* *An online shopping cart* - Users visit multiple places in your app, searching for products they add to the cart. The cart remembers all the products the client added.

Key aspects of session-scoped beans

|  |  |  |  |
| --- | --- | --- | --- |
| **Fact** | **Consequence** | **To consider** | **To avoid** |
| The session-scoped bean instances are kept for the entire HTTP session. | They have a longer life, and they are less frequently garbage-collected than the request-scoped beans. | The app keeps the data you store in the session-scoped beans for a more extended period. | Avoid keeping too much data on the session. It can potentially become a performance problem. Moreover, never store sensitive details (like passwords, private keys, or any other secret detail) in session-bean attributes. |
| Multiple requests can share the session-scoped bean instance. | If the same client issues multiple concurrent requests that change the data on the instance, you may encounter multithreading-related issues like race conditions. | When you know such a scenario is possible, you might need to use synchronization techniques to avoid concurrency. However, I generally recommend you see if this can be avoided and keep synchronization only as a last resort when it can’t be avoided. |  |
| The session-scoped beans are a way to share data among requests by keeping the data on the server side. | The logic you implement might imply requests become dependent one on the other. | When keeping details stateful in one app’s memory, you make clients dependent on that specific app instance. Before deciding to implement some feature with a session-scoped bean, consider alternatives, such as storing the data you want to share in a database instead of the session. This way, you can leave the HTTP requests independent one from another. |  |

We continue to use a session-scoped bean to make our app aware that a user logged in and recognize them as a logged-in user while they access different pages of the app.

Let’s change the application we implemented in section 9.1 to display a page that only logged-in users can access. Once a user logs in, the app redirects them to this page, which displays a welcome message containing the logged-in username and offers the user the option to log out by clicking a link.

These are the steps we need to take to implement this change (figure 9.10):

1. Create a session-scoped bean to keep the logged-in user’s details.
2. Create the page a user can only access after login.
3. Make sure a user cannot access the page created at point 1 without logging in first.
4. Redirect the user from login to the main page after successful authentication.

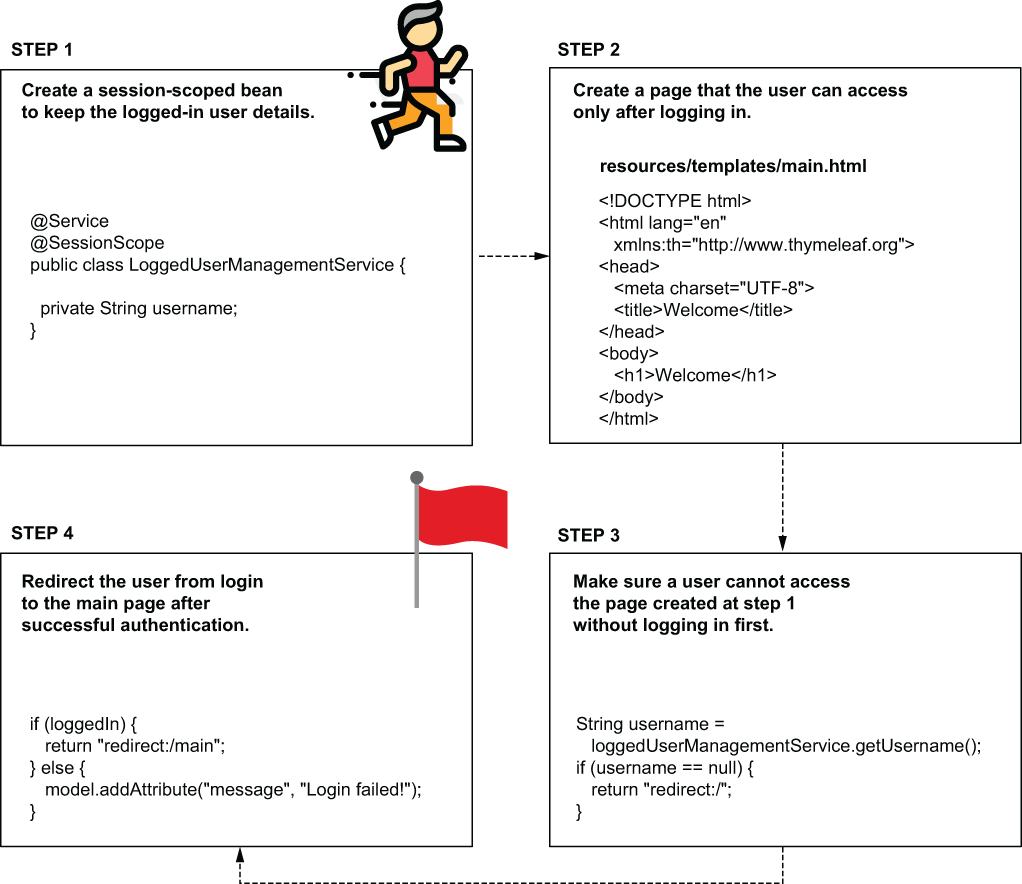


Figure 9.10 We use a session-bean to implement a section of the app that only a logged-in user can access. Once the user authenticates, the app redirects them to a page they can only access once authenticated. If the user tries to access this page before authentication, the app redirects them to the login form.

Creating a session-scoped bean in Spring is as simple as using the **@SessionScope** annotation with the bean class. Let’s create a new class, *LoggedUserManagementService,* and make it session-scoped.

Listing 9.5 Defining a session-scoped bean to keep the logged user details

@Service ❶

@SessionScope ❷

public class LoggedUserManagementService {

private String username;

// Omitted getters and setters

}

❶ We add the @Service stereotype annotation to instruct Spring to manage this class as a bean in its context.

❷ We use the @SessionScope annotation to change the scope of the bean to session.

Every time a user successfully logs in, we store its name in this bean’s username attribute. We auto-wire the LoggedUserManagementService bean in the LoginProcessor class, who takes care of the authentication logic:

Listing 9.6 Using the LoggedUserManagementService bean in the login logic

@Component

@RequestScope

public class LoginProcessor {

private final LoggedUserManagementService loggedUserManagementService;

private String username;

private String password;

public LoginProcessor( ❶

LoggedUserManagementService loggedUserManagementService) {

this.loggedUserManagementService = loggedUserManagementService;

}

public boolean login() {

String username = this.getUsername();

String password = this.getPassword();

boolean loginResult = false;

if ("natalie".equals(username) && "password".equals(password)) {

loginResult = true;

loggedUserManagementService.setUsername(username); ❷

}

return loginResult;

}

// Omitted getters and setters

}

❶ We auto-wire the LoggedUserManagementService bean.

❷ We store the username on the LoggedUserManagementService bean.

Observe that the LoginProcessor bean stays request-scoped. We still use Spring to create this instance for each login request. We only need the username and password attributes’ values during the request to execute the authentication logic.

Because the LoggedUserManagementService bean is session-scoped, the username value will now be accessible throughout the entire HTTP session. You can use this value to know if someone is logged in, and who. You don’t have to worry about the case where multiple users are logged in; the application framework makes sure to link each HTTP request to the correct session. Figure 9.11 visually describes the login flow.

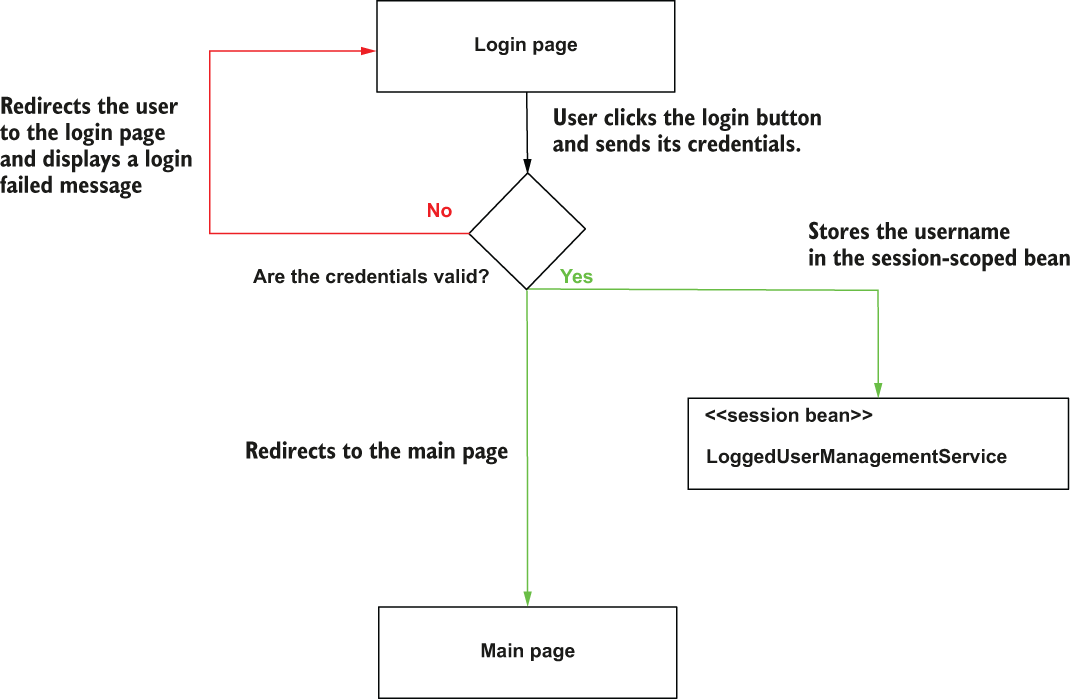


Figure 9.11 The login flow implemented in the example. When the user submits their credentials, the login process begins. If the user's credentials are correct, the username is stored in the session-scoped bean, and the app redirects the user to the main page. If the credentials are not valid, the app redirects the user back to the login page and displays a failed login message.

Now we create a new page and make sure a user can access it only if they have already logged in. We define a new controller (that we’ll call MainController) for the new page. We’ll define an action and map it to the /main path. To make sure a user can access this path only if they logged in, we check if the LoggedUserManagementService bean stores any username. If it doesn’t, we redirect the user to the login page. To redirect the user to another page, the controller action needs to return the string “redirect:” followed by the path to which the action wants to redirect the user. Figure 9.12 visually presents the logic behind the main page.

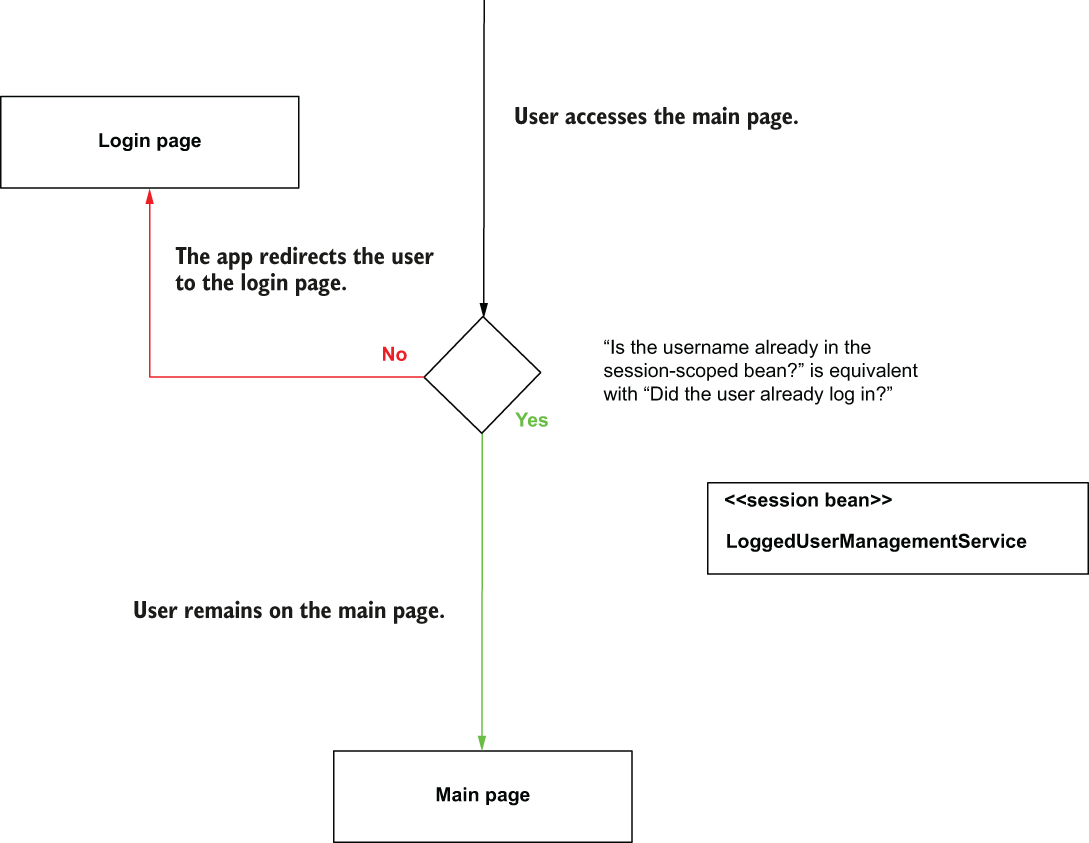


Figure 9.12 Someone can access the main page only after they are authenticated. When the app authenticates the user, it stores the username in the session-scoped bean. This way, the app knows the user had already logged in. When someone accesses the main page, and the username is not in the session-scoped bean (they did not authenticate), the app redirects them to the login page.

Listing 9.7 The MainController class

@Controller

public class MainController {

private final LoggedUserManagementService loggedUserManagementService;

public MainController( ❶

LoggedUserManagementService loggedUserManagementService) {

this.loggedUserManagementService = loggedUserManagementService;

}

@GetMapping("/main")

public String home() {

String username = ❷

loggedUserManagementService.getUsername();

if (username == null) { ❸

return "redirect:/";

}

return "main.html"; ❹

}

}

❶ We auto-wire the LoggedUserManagementService bean to find out if the user already logged in.

❷ We take the username value, which should be different than null if someone logged in.

❸ If the user is not logged in, we redirect the user to the login page.

❹ If the user is logged in, we return the view for the main page.

You need to add the main.html that defines the view in the “resources/templates” folder of the project.

Listing 9.8 The content of the **main.html** page

<!DOCTYPE html>

<html lang="en" xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="UTF-8">

<title>Welcome</title>

</head>

<body>

<h1>Welcome</h1>

</body>

</html>

To allow the user to log out is also easy. You just need to set the username in the LoggedUserManagementService session bean as null. Let’s create a logout link on the page and also add the logged-in username in the welcome message.

Listing 9.9 Adding a logout link to the **main.html** page

<!DOCTYPE html>

<html lang="en" xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="UTF-8">

<title>Login</title>

</head>

<body>

<h1>Welcome, <span th:text="${username}"></span></h1> ❶

<a href="/main?logout">Log out</a> ❷

</body>

</html>

❶ We get the username from the controller and display it on the page in the welcome message.

❷ We add a link on the page that sets an HTTP request parameter named “logout.” When the controller gets this parameter, it will erase the value of the username from the session.

These main.html page changes also assume some changes in the controller for the functionality to be complete. The next listing shows how to get the logout request parameter in the controller’s action and send the username to the view where it is displayed on the page.

Listing 9.10 Logging out the user based on the logout request parameter

@Controller

public class MainController {

// Omitted code

@GetMapping("/main")

public String home(

@RequestParam(required = false) String logout, ❶

Model model ❷

) {

if (logout != null) { ❸

loggedUserManagementService.setUsername(null);

}

String username = loggedUserManagementService.getUsername();

if (username == null) {

return "redirect:/";

}

model.addAttribute("username" , username); ❹

return "main.html";

}}

❶ We get the logout request parameter if present.

❷ We add a Model parameter to send the username to the view.

❸ If the logout parameter is present, we erase the username from the LoggedUserManagementService bean.

❹ We send the username to the view.

To complete the app, we’d like to change the LoginController to redirect users to the main page once they authenticate.

Listing 9.11 Redirecting the user to the main page after login

@Controller

public class LoginController {

// Omitted code

@PostMapping("/")

public String loginPost(

@RequestParam String username,

@RequestParam String password,

Model model

) {

loginProcessor.setUsername(username);

loginProcessor.setPassword(password);

boolean loggedIn = loginProcessor.login();

if (loggedIn) { ❶

return "redirect:/main";

}

model.addAttribute("message", "Login failed!");

return "login.html";

}

}

❶ When the user successfully authenticates, the app redirects them to the main page.

Now you can start the application and test the login. When you provide the correct credentials, the app redirects you to the main page (figure 9.13). Click the Logout link, and the app redirects you back to the login. If you try to access the main page without authenticating, the app redirects you to log in.

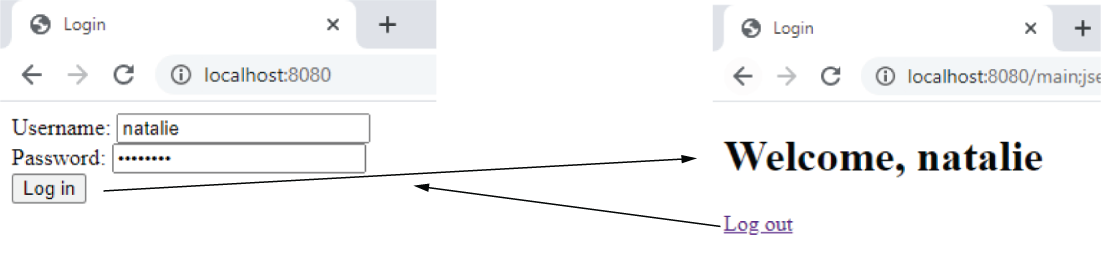


Figure 9.13 This flow between the two pages. When the user logs in, the app redirects them to the main page. The user can click on the logout link, and the app redirects them back to the login form.

## **PROJECT 10 – Login – @ApplicationScope bean**

**9.3 Using the application scope in a Spring web app**

In this section, we discuss the application scope. I want to mention its existence, make you aware of how it works, and emphasize that it’s better not to use it in a production app. All client requests share an application-scoped bean (figure 9.14).

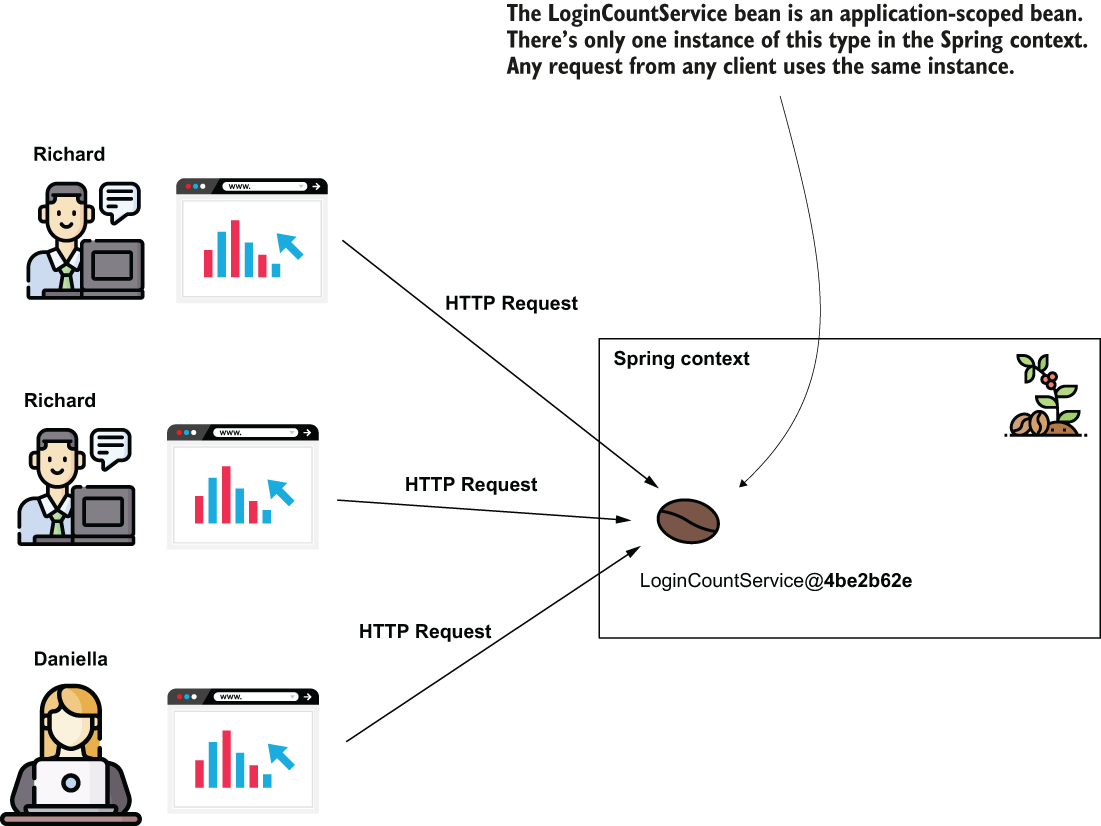


Figure 9.14 Understanding the application scope in a Spring web app. The instance of an application-scoped bean is shared by all the HTTP requests from all clients. The Spring context provides only 1 instance of the bean’s type, used by anyone who needs it.

The application scope is close to how a singleton works. The difference is that you can’t have more instances of the same type in the context and that we always use the HTTP requests as a reference point when discussing the life cycle of web scopes (including the application scope). We face the same concurrency problems we discussed in chapter 5 for the singleton beans for application-scoped beans: it’s better to have immutable attributes for the singleton beans. The same advice is applicable to an application-scoped bean. But if you make the attributes immutable, then you can directly use a singleton bean instead.

Generally, I recommend developers avoid using application-scoped beans. It’s better to directly use a persistence layer, such as a database.

Let’s change the application we worked on in this chapter and add a feature that counts the login attempts. Because we have to count the login attempts from all users, we’ll store the count in an application-scoped bean. Let’s create a *LoginCountService* application-scoped bean that stores the count in an attribute. The following listing shows the definition of this class.

Listing 9.12 The LoginCountService class counts the login attempts

@Service

@ApplicationScope ❶

public class LoginCountService {

private int count;

public void increment() {

count++;

}

public int getCount() {

return count;

}}

❶ The @ApplicationScope annotation changes the scope of this bean to the application scope.

The LoginProcessor can then auto-wire this bean and call the increment() method for any new login attempt, as presented in the following listing.

Listing 9.13 Incrementing the login count for every login request

@Component

@RequestScope

public class LoginProcessor {

private final LoggedUserManagementService loggedUserManagementService;

private final LoginCountService loginCountService;

private String username;

private String password;

public LoginProcessor( ❶

LoggedUserManagementService loggedUserManagementService,

LoginCountService loginCountService) {

this.loggedUserManagementService = loggedUserManagementService;

this.loginCountService = loginCountService;

}

public boolean login() {

loginCountService.increment(); ❷

String username = this.getUsername();

String password = this.getPassword();

boolean loginResult = false;

if ("natalie".equals(username) && "password".equals(password)) {

loginResult = true;

loggedUserManagementService.setUsername(username);

}

return loginResult;

}

// Omitted code

}

❶ We inject the LoginCountService bean through the constructor’s parameters.

❷ We increment the count for each login attempt.

The last thing you need to do is to display this value. You can use a Model parameter in the controller’s action to send the count value to the view. You can then use Thymeleaf to display the value in the view. The following listing shows you how to send the value from the controller to the view.

Listing 9.14 Sending the count value from controller to be displayed on the main page

@Controller

public class MainController {

// Omitted code

@GetMapping("/main")

public String home(

@RequestParam(required = false) String logout,

Model model

) {

if (logout != null) {

loggedUserManagementService.setUsername(null);

}

String username = loggedUserManagementService.getUsername();

int count = loginCountService.getCount(); ❶

if (username == null) {

return "redirect:/";

}

model.addAttribute("username" , username);

model.addAttribute("loginCount", count); ❷

return "main.html";

}

}

❶ Gets the count from the application-scoped bean

❷ Sends the count value to the view

Here how to display the count value on the page:

Listing 9.15 Displaying the count value on the main page

<!DOCTYPE html>

<html lang="en" xmlns:th="http://www.thymeleaf.org">

<head>

<meta charset="UTF-8">

<title>Login</title>

</head>

<body>

<h1>Welcome, <span th:text="${username}"></span>!</h1>

<h2>

Your login number is

<span th:text="${loginCount}"></span> ❶

</h2>

<a href="/main?logout">Log out</a>

</body>

</html>

❶ Displays the count on the page

When you run the app, you find the total number of login attempts on the main page, as presented in figure 9.15.

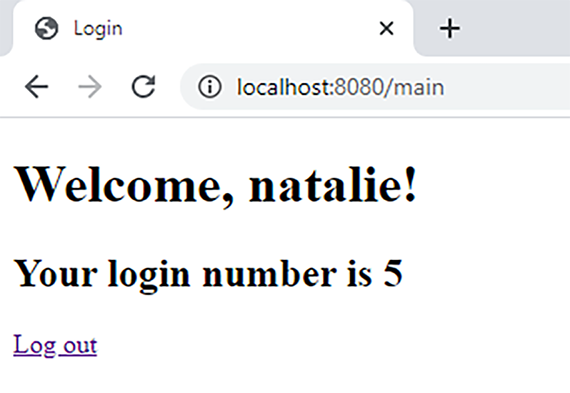


Figure 9.15 The result of the application is a web page that displays the total number of logins for all the users. This main page displays the total number of login attempts.

**Summary**

* Aside from the *singleton* and *prototype* bean scopes, you can benefit from using 3 more bean scopes in a Spring web app. These scopes only make sense in web apps, and that’s why we call them web scopes:
  + *Request scope*—Spring creates an instance of the bean for each HTTP request.
  + *Session scope*—Spring creates an instance of the bean per HTTP session of the client. Multiple requests from the same client can share the same instance.
  + *Application scope*—There’s only one instance for the whole application for that specific bean. Every request from any client can access this instance.
* Spring guarantees that a request-scoped bean instance is only accessible by one HTTP request. For this reason, you can use the instance’s attributes without worrying about concurrency-related problems. Also, you don’t need to worry that they might fill the app’s memory. Being they are short-lived, the instances can be garbage-collected once the HTTP request ends.
* Spring creates request-scoped bean instances for every HTTP request. This is quite often. You preferably shouldn’t make the instance’s creation difficult by implementing logic in the constructor or a @PostConstruct method.
* Spring links a session-scoped bean instance to the HTTP session of the client. This way, a session-scoped bean instance can be used to share data among multiple HTTP requests from the same client.
* Even if from the same client, the client can send HTTP requests concurrently. If these requests change data in the session-scoped instance, they might get into race-condition scenarios. You need to consider such situations and either avoid them or synchronize your code to support the concurrency.
* I recommend avoiding the use of application-scoped bean instances. With application-scoped bean instances being shared by all the web app requests, any write operation usually needs synchronization, creating bottlenecks and dramatically affecting the app’s performance. Moreover, these beans live in your app’s memory as long as the app itself, so they can’t be garbage-collected. A better approach is to directly store the data in a database.
* Both session- and application-scoped beans imply making requests less independent. We say the application manages the state the requests need (or that the app is stateful). A stateful app implies different architectural problems that are best to avoid. It it’s better to consider an alternative.

10 Implementing REST services

Representational state transfer (REST) services are one of the most often encountered ways to implement communication between 2 apps. REST offers access to functionality the server exposes through endpoints a client can call.

You use REST services to establish the communication between a client and a server in a web app. But you can also use REST services to develop the communication between a mobile app and a backend or even two backend services (figure 10.1).

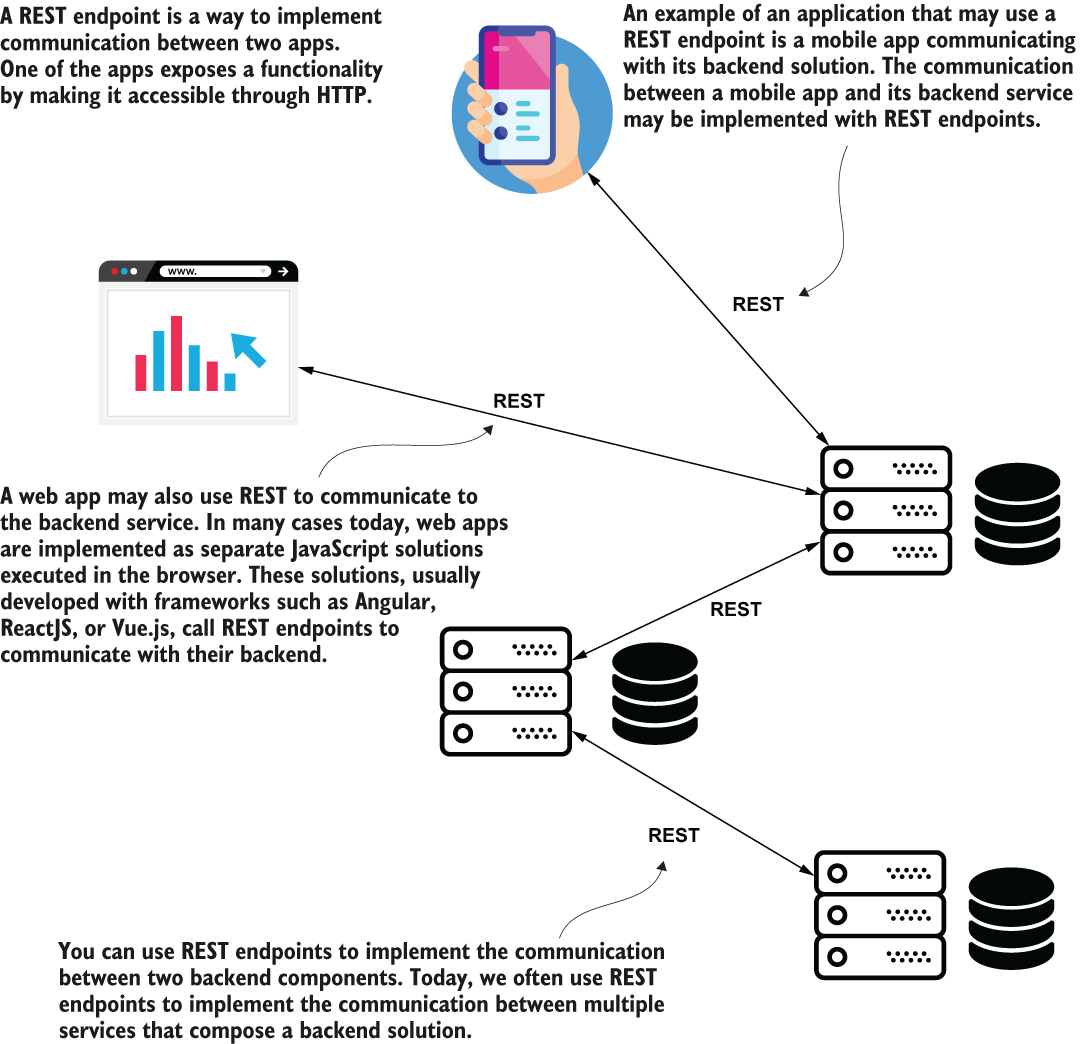


Figure 10.1 REST services are a communication method between 2 apps. A web client app or mobile app may call its backend solution through REST endpoints, but even backend services might communicate using REST web service calls.

**10.1 Using REST services to exchange data between apps**

Here we will discuss the way Spring supports implementing REST services through Spring MVC. REST endpoints are simply a way for implementing communication between 2 apps. REST endpoints are as simple as implementing a controller action mapped to an HTTP method and a path. An app calls this controller action through HTTP. Because it’s how an app exposes a service through a web protocol, we call this endpoint a web service.

Spring uses the same mechanism you learned for web apps for exposing REST endpoints. The only difference is that for REST services we’ll tell the Spring MVC dispatcher servlet not to look for a view. In the Spring MVC diagram you learned in chapter 7, the view resolver disappears. The server sends back, in the HTTP response to the client, directly what the controller’s action returns. Figure 10.2 presents the changes in the Spring MVC flow.

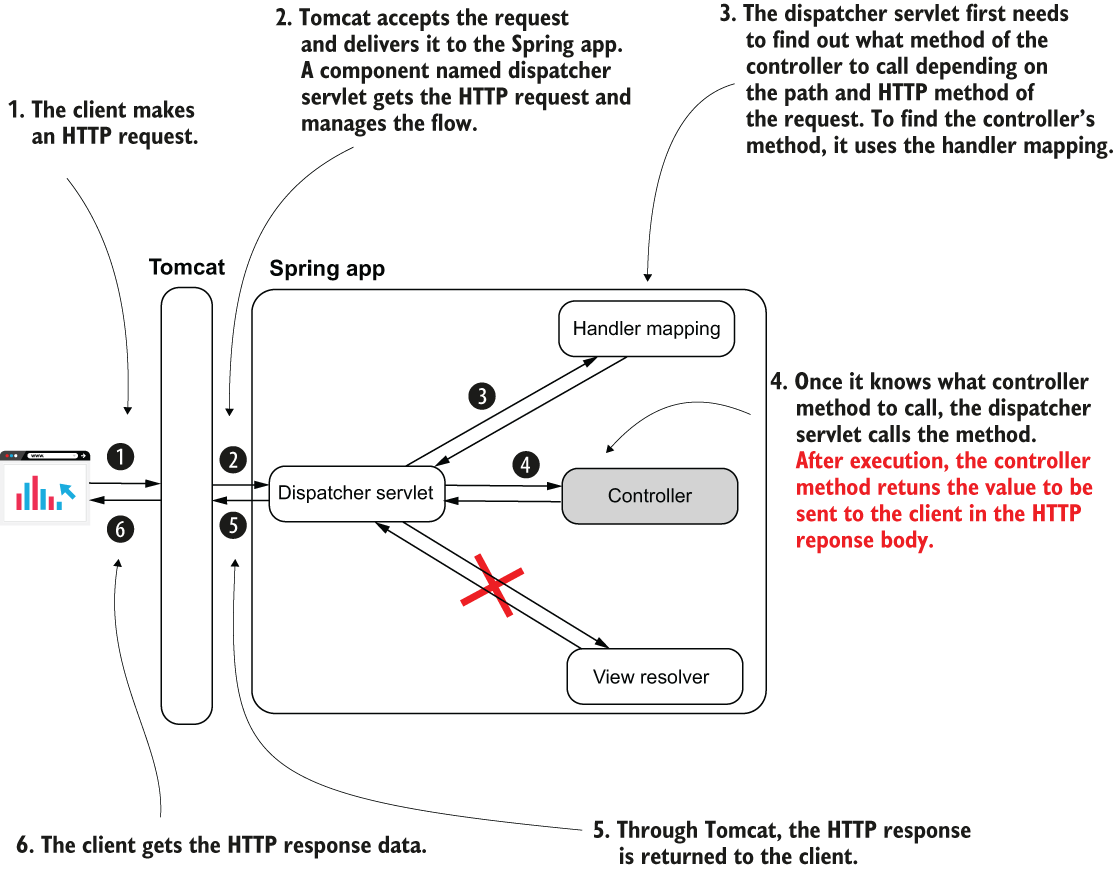


Figure 10.2 When implementing REST endpoints, the Spring MVC flow changes. The app no longer needs a view resolver because the client needs the data returned by the controller’s action directly. Once the controller’s action completes, the dispatcher servlet returns the HTTP response without rendering any view.

Some communication issues the REST endpoint might bring:

* If the controller’s action takes a long time to complete, the HTTP call to the endpoint might time out and break the communication.
* Sending a large quantity of data in one call (through the HTTP request) might cause the call to time out and break the communication. Sending more than a few megabytes through a REST call usually isn’t the right choice.
* Too many concurrent calls on an endpoint exposed by a backend component might put too much pressure on the app and cause it to fail.
* The network supports the HTTP calls, and the network is never 100% reliable. There’s always a chance a REST endpoint call might fail because of the network.

When you implement the communication between 2 apps using REST, you always need to consider what should happen if a call fails and how it might affect the app. Ask yourself if the data could be affected in any way. Could the way you designed your app lead to data inconsistencies if an endpoint call fails? In case the app needs to display an error to the user, how would you do that? These are complex issues and require architectural knowledge outside the scope of this book, but I recommend J. J. Geewax’s API Design Patterns (Manning, 2021), an excellent guide discussing the best practices of designing APIs.

**10.2 Implementing a REST endpoint**

Here you’ll learn how to transform a simple web controller into a REST controller to implement REST web services.

## **PROJECT 11 – @ResponseBody**

We annotate the controller class with the *@Controll*er stereotype annotation. This way, an instance of the class becomes a bean in the Spring context, and Spring MVC knows this is a controller that maps its methods to specific HTTP paths. Also, we used the *@GetMapping* annotation to specify the action path and HTTP method. The ***@ResponseBody*** annotation tells the dispatcher servlet that the controller’s action doesn’t return a view name but the data sent directly in the HTTP response.

Listing 10.1 Implementing a REST endpoint action in a **controller** class

@Controller ❶

public class HelloController {

@GetMapping("/hello") ❷

@ResponseBody ❸

public String hello() {

return "Hello!";

} }

❶ We use the @Controller annotation to mark the class as a Spring MVC controller

❷ We use the @GetMapping annotation to associate the GET HTTP method and a path with the controller’s action.

❸ We use the @ResponseBody annotation to inform the dispatcher servlet that this method doesn’t return a view name but the HTTP response directly.

But if we add more methods to the controller, repeating the @ResponseBody annotation on every method becomes annoying.

Listing 10.2 The @ResponseBody annotation becomes duplicated code

@Controller

public class HelloController {

@GetMapping("/hello")

@ResponseBody

public String hello() {

return "Hello!";

}

@GetMapping("/ciao")

@ResponseBody

public String ciao() {

return "Ciao!";

} }

## **PROJECT 12 – @RestController**

A best practice is avoiding code duplication. We want to somehow prevent repeating the @ResponseBody annotation for each method. To help us with this aspect, Spring offers the **@RestController** annotation, a combination of @Controller and @ResponseBody. You use @RestController to instruct Spring that all the controller’s actions are REST endpoints. This way, you avoid repeating the @ResponseBody annotation. Now you need to change the controller to use @RestController once for the class instead of @ResponseBody for each method.

Listing 10.3 Using the **@RestController** annotation to avoid code duplication

@RestController ❶

public class HelloController {

@GetMapping("/hello")

public String hello() {

return "Hello!";

}

@GetMapping("/ciao")

public String ciao() {

return "Ciao!";

} }

❶ Instead of repeating the @ResponseBody annotation for each method, we replace @Controller with @RestController.

A best practice is avoiding code duplication. We want to somehow prevent repeating the @ResponseBody annotation for each method. To help us with this aspect, Spring offers the **@RestController** annotation, a combination of @Controller and @ResponseBody. You use @RestController to instruct Spring that all the controller’s actions are REST endpoints. This way, you avoid repeating the @ResponseBody annotation. Now you need to change the controller to use @RestController once for the class instead of @ResponseBody for each method.

In this section, you’ll learn to call your endpoints using 2 tools you’ll often encounter in real-world scenarios:

* *Postman*—Offers a nice GUI and is comfortable to use
* *cURL*—A command-line tool useful in cases where you don’t have a GUI (e.g., when you connect to a virtual machine via SSH or when you write a batch script)

Both these tools are a must-learn for any developer. In chapter 15, you’ll learn a third approach for validating that an endpoint behaves as expected by writing an integration test.

Let’s change the application we worked on in this chapter and add a feature that counts the login attempts. Because we have to count the login attempts from all users, we’ll store the count in an application-scoped bean. Let’s create a *LoginCountService* application-scoped bean that stores the count in an attribute. The following listing shows the definition of this class.

## **POSTMAN**

Let’s discuss Postman first. You need to install the tool on your system as presented on their official website: <https://www.postman.com/> .

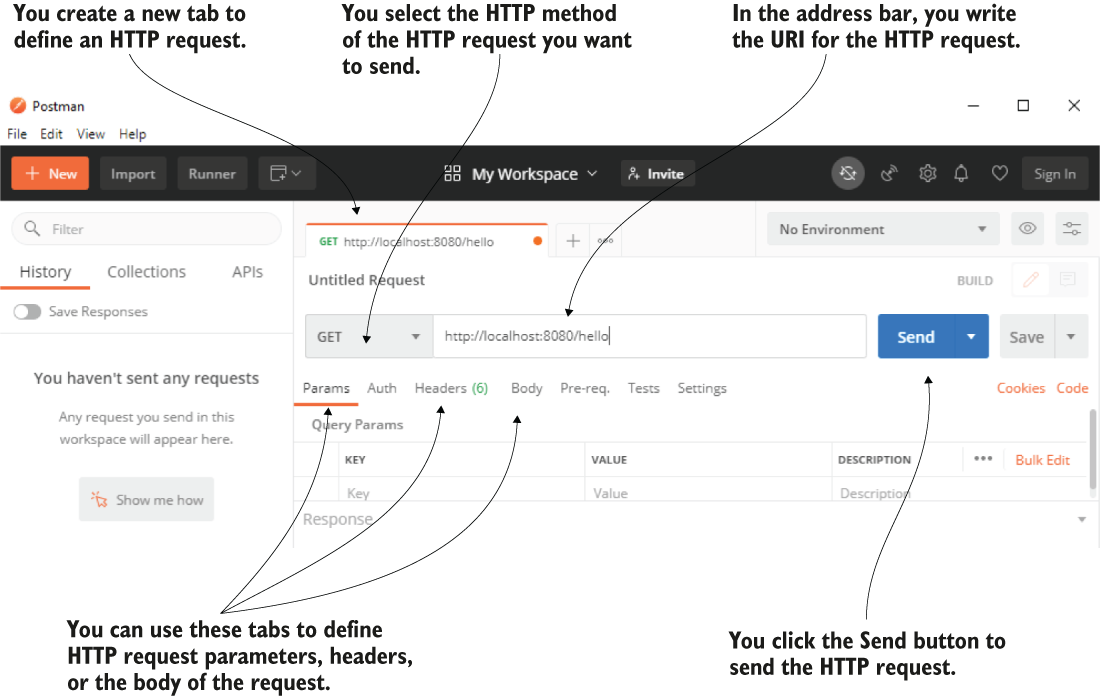


Figure 10.3 Postman offers a friendly interface to configure and send an HTTP request. You select the HTTP method, set the HTTP request URI, and then press the Send button to send an HTTP request. You can also define other configurations such as the request parameters, headers, or the request body if needed.

Once you press the Send button, Postman sends the HTTP request. When the request completes, Postman displays the HTTP response details:



Figure 10.4 Once the HTTP request completes, Postman displays the HTTP response details. You find the response status, the time it took the request to complete, the amount of transferred data in bytes, and the response body and the headers.

## **cURL**

In case you don’t have a GUI, you can use a command-line tool to call an endpoint. You’ll also find articles and books often use command-line tools for demonstrations rather than GUI tools because it’s a shorter way to represent the command.

If you choose to use cURL as a command-line tool like in the case of Postman, you need first to make sure you install it. You install cURL according to your operating system as described on the tool’s official web page: <https://curl.se/>

((((((((((((((((((((((( For Windows - <https://developer.zendesk.com/documentation/developer-tools/getting-started/installing-and-using-curl/> :

Replace line-ending backslashes

The cURL examples often have a backslash (\) at the end of lines to break up a long statement into easier-to-read lines. The backslash is a line continuation character in UNIX but not in Windows. In Windows, replace any backslash at the end of lines with the caret (^) character, which is an escape character in Windows. Don't leave any space after any ^ character or it won't work. The caret will escape the space instead of the new line.

Example:

curl https://mysubdomain.zendesk.com/api/v2/groups.json ^ -v -u myemail@example.com:mypassword

You can paste a multiline statement at the command prompt by clicking the icon in the upper-left corner and selecting **Edit** > **Paste**. If you prefer using the keyboard, press **Alt+spacebar** to open the menu, then press **E** and **P**.

Graphical user interface, website

Description automatically generated

Move JSON data to a file

The Windows command prompt doesn't support single quotes. It's a problem because cURL statements use single quotes to specify JSON data. Example:

curl https://{subdomain}.zendesk.com/api/v2/groups.json ^ -d '{"group": {"name": "My Group"}}' ^ -H "Content-Type: application/json" ^ -v -u {email\_address}:{password} -X POST

The statement specifies JSON data for creating a group (the -d flag stands for data). Because the JSON is enclosed in single quotes, the statement won't work on the command line.

To fix the problem, save the JSON in a separate file and import it into the cURL statement. To modify the example above, create a file named **json.txt** containing the following text:

{"group": {"name": "My Group"}}

Next, change the cURL statement to import the JSON data with the **@filename** syntax:

curl https://{subdomain}.zendesk.com/api/v2/groups.json ^ -d @json.txt ^ -H "Content-Type: application/json" ^ -v -u {email\_address}:{password} -X POST

Before running the statement, use the **cd** command (for change directory) to navigate to the folder that contains the file. Example:

**C:\> cd json\_files**

Then paste the cURL statement at the command prompt:

curl https://{subdomain}.zendesk.com/api/v2/groups.json ^ -d @json.txt ^ -H "Content-Type: application/json" ^ -v -u {email\_address}:{password} -X POST

An alternative to moving the JSON to a separate file is to use double quotes around the JSON data in the cURL statement and escape the inner ones with backslashes:

-d "{\"group\": {\"name\": \"My Group\"}}" \

It doesn't end there. The following special characters in strings must be escaped with the caret (^) character: &, \, <, >, ^, |. If the JSON includes HTML, such as when you try to create or update an article in Help Center, you need to find and escape all the angle brackets in the HTML.

This is tedious and error prone. Best to stick with importing the JSON from a file.

)))))))))))))))))))))))))))))

Once you have it installed and configured, you can use the curl command to send HTTP requests. The following snippet shows you the command you can use to send the HTTP request to test the /hello endpoint exposed by our app:

curl <http://localhost:8080/hello>

Upon completing the HTTP request, the console only displays the HTTP response body presented in the next snippet:

Hello!

If the HTTP method is HTTP GET, you don’t need to specify it explicitly. When the method is not HTTP GET, or if you want to specify it explicitly, you can use the -X flag:

curl -X GET <http://localhost:8080/hello>

If you want to get more details of the HTTP request, you can add the -v option to the command, as presented in the next snippet:

curl -v <http://localhost:8080/hello>

The result of this command is a bit more complicated. You also find details like the status, the amount of data transferred, and headers through the lengthy response:

Trying ::1:8080...

\* Connected to localhost (::1) port 8080 (#0)

> GET /hello HTTP/1.1

> Host: localhost:8080

> User-Agent: curl/7.73.0

> Accept: \*/\*

>

\* Mark bundle as not supporting multiuse

< HTTP/1.1 200 ❶

< Content-Type: text/plain;charset=UTF-8

< Content-Length: 6

< Date: Fri, 25 Dec 2020 23:11:02 GMT

<

{ [6 bytes data]

100 6 100 6 0 0 857 0 --:--:-- --:--:-- --:--:--

1000

Hello! ❷

\* Connection #0 to host localhost left intact

❶ The HTTP response status

❷ The HTTP response body

ex.

curl -X POST http:/localhost:9090/payment -H "content-type:application/json" -d "{\"amount\":1000}"}"

-X – verb

- H – header

-d - body

**10.3 Managing the HTTP response**

The HTTP response is how the backend app sends data back to the client due to a client’s request. The HTTP response holds data as the following:

* *Response headers* —Short pieces of data in the response (usually not more than a few words long)
* *The response body* —A larger amount of data the backend needs to send in the response
* *The response status* —A short representation of the request’s result

## **PROJECT 13 – DTO - sending objects as a response body - Country**

**10.3.1 Sending objects as a response body**

The only thing you need to do to send an object to the client in a response is make the controller’s action return that object. We define a model object named *Country* with the attributes *Name* (representing the country name) and *Population* (representing the number of millions of people located in that country). We implement a controller action to return an instance of type Country.

Listing 10.4 shows the class that defines the Country object. When we use an object (such as Country) to model the data transferred between 2 apps, we name this object a data transfer object (DTO). We can say that Country is our DTO, whose instances are returned by the REST endpoint we implement in the HTTP response body.

Listing 10.4 **Model** of the data the server returns in the HTTP response body

public class Country {

private String name;

private int population;

public static Country of( ❶

String name,

int population) {

Country country = new Country();

country.setName(name);

country.setPopulation(population);

return country;

}

// Omitted getters and setters

}

❶ To make a Country instance simpler, we define a static factory method that receives the name and the population. This method returns a Country instance with the provided values set.

The following listing shows the implementation of a controller’s action that returns an instance of type Country.

Listing 10.5 Returning an object instance from the **controller**’s action

@RestController ❶

public class CountryController {

@GetMapping("/france") ❷

public Country france() {

Country c = Country.of("France", 67);

return c; ❸

} }

❶ Marking the class as a REST controller to add a bean in the Spring context and also inform the dispatcher servlet not to look for a view when this method returns

❷ Mapping the controller’s action to the HTTP GET method and /france path

❸ Returning an instance of type Country

What happens when you call this endpoint? How would the object look in the HTTP response body? By default, Spring creates a string representation of the object and formats it as JSON. JavaScript Object Notation (JSON) is a simple way to format strings as attribute-value pairs.

When calling the /france endpoint, the response body looks as presented in the next snippet:

{

"name": "France",

"population": 67

}

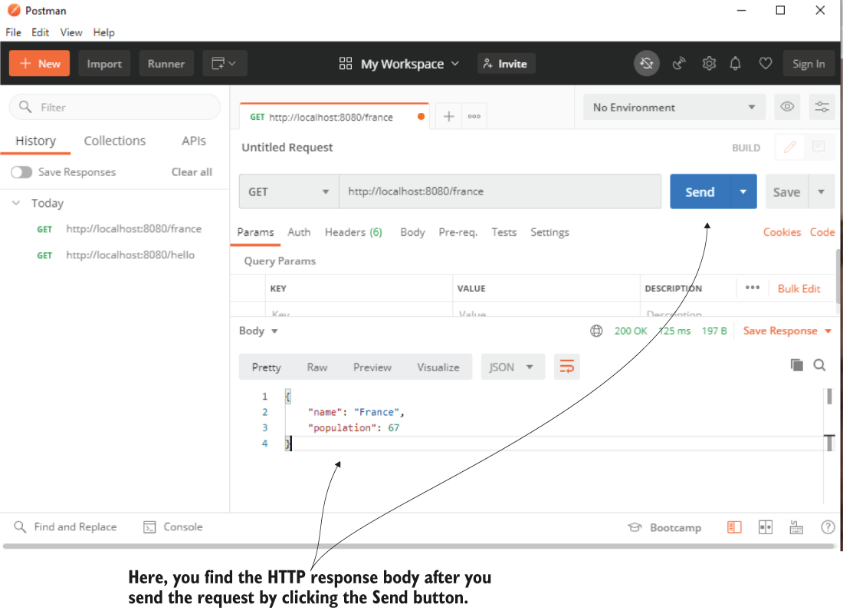


Figure 10.5 Once you press the Send button, Postman sends the request. When the request completes, Postman displays the response details, including the response body.

You could also send object collection instances in the response body. The next listing shows that we added a method that returns a List of Country objects.

Listing 10.6 Returning a collection in the response body

@RestController

public class CountryController {

// Omitted code

@GetMapping("/all")

public List<Country> countries() {

Country c1 = Country.of("France", 67);

Country c2 = Country.of("Spain", 47);

return List.of(c1,c2); ❶

} }

❶ Returns a collection in the HTTP response body

When you call this endpoint, the response body looks as presented in the next snippet:

[ ❶

{ ❷

"name": "France", ❷

"population": 67 ❷

}, ❷

{

"name": "Spain",

"population": 47

}

]

❶ In JSON, the list is defined with brackets.

❷ Each object is between curly braces, and the objects are separated with commas.

Using JSON is the most common way to represent objects when working with REST endpoints. Spring offers the possibility of using other ways to format the response body (like XML or YAML), by plugging in a custom converter for your objects. The chances you’ll need this in a real-world scenario are small.

**10.3.2 Setting the response status and headers**

In this section, we discuss setting the response status and response headers. Sometimes it’s more comfortable to send part of the data in the response headers. The response status is also an essential flag in the HTTP response you use to signal the request’s result. By default, Spring sets some common HTTP statuses:

* *200 OK* - if no exception was thrown on the server side while processing the request.
* *404 Not Found* - if the requested resource doesn’t exist.
* *400 Bad Request* - if a part of the request could not be matched with the way the server expected the data.
* *500 Error on server* - if an exception was thrown on the server side for any reason while processing the request. Usually, for this kind of exception, the client can’t do anything, and it’s expected someone should solve the problem on the backend.

## **PROJECT 14 - ResponseEntity – custom headers and response status**

However, in some cases, the requirements ask you to configure a custom status. The easiest and most common way to customize the HTTP response is using the **ResponseEntity** class. This class provided by Spring allows you to specify the response body, status, and headers on the HTTP response. In listing 10.7, a controller action returns a ResponseEntity instance instead of the object you want to set on the response body directly. The ResponseEntity class allows you to set the response body’s value and the response status and headers. We set 3 headers and change the response status to “202 Accepted.”

Listing 10.7 Adding **custom headers** and setting a response **status**

@RestController

public class CountryController {

@GetMapping("/france")

public ResponseEntity<Country> france() {

Country c = Country.of("France", 67);

return ResponseEntity

.status(HttpStatus.ACCEPTED) ❶

.header("continent", "Europe") ❷

.header("capital", "Paris") ❷

.header("favorite\_food", "cheese and wine") ❷

.body(c); ❸

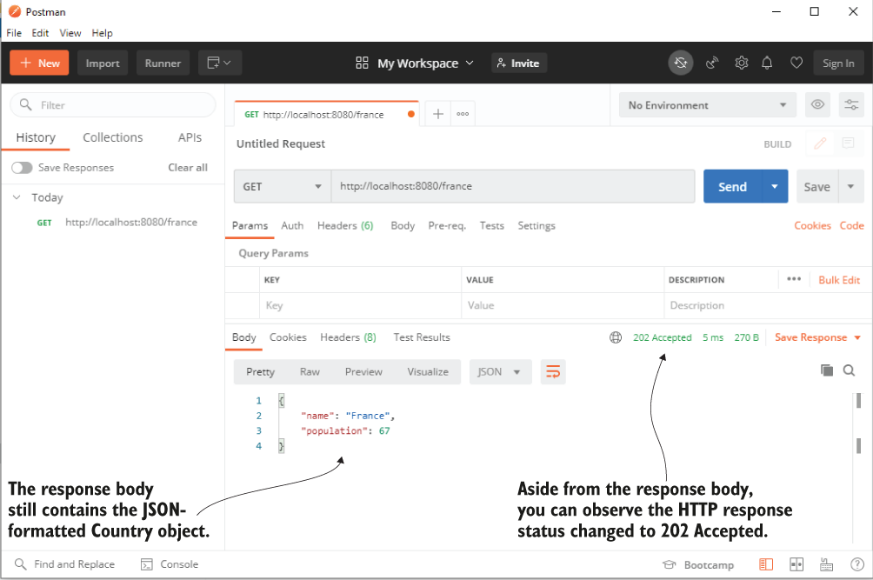
}}

❶ Changes the HTTP response status to 202 Accepted

❷ Adds three custom headers to the response

❸ Sets the response body

Once you send the request using Postman, you can verify the HTTP response status changed to “202 Accepted” (figure 10.6).



In the Headers tab of the HTTP response in Postman, you also find the 3 custom response headers you added (figure 10.7).

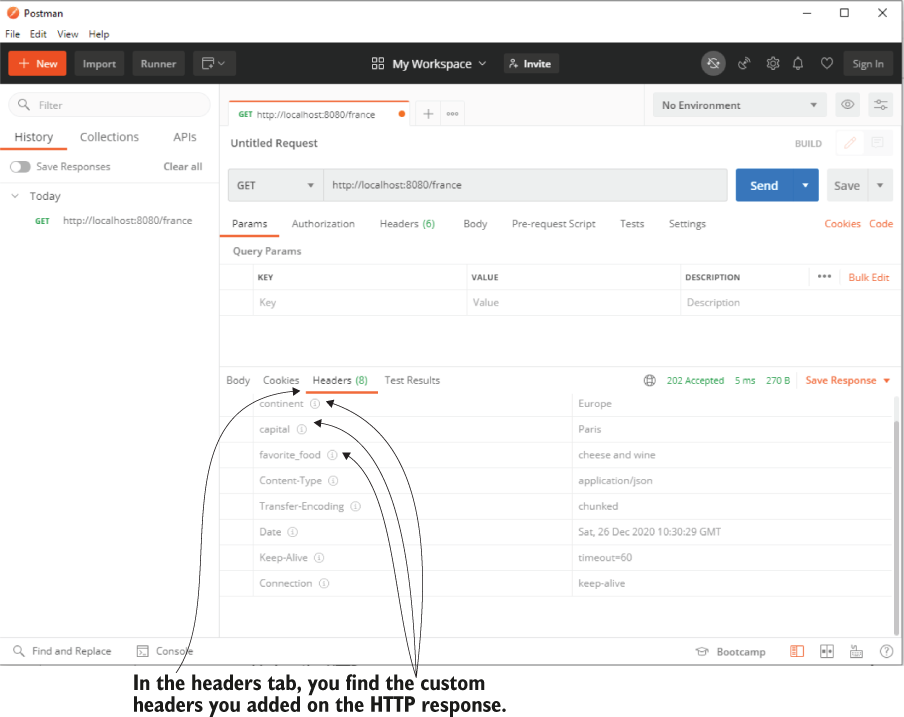


Figure 10.7 To see the customer headers in Postman, you have to navigate to the Headers tab of the HTTP response.

**10.3.3 Managing exceptions at the endpoint level**

It’s essential to consider what happens if the controller’s action throws an exception. In many cases, we use exceptions to signal specific situations, some of these related to the business logic. Suppose you create an endpoint the client calls to make a payment. If the user doesn’t have enough money in their account, the app might represent this situation by throwing an exception. In this case, you’ll probably want to set some details on the HTTP response to inform the client of the specific situation that occurred.

However, in some cases, the requirements ask you to configure a custom status. The easiest and most common way to customize the HTTP response is using the *ResponseEntity* class. This class provided by Spring allows you to specify the response body, status, and headers on the HTTP response. In listing 10.7, a controller action returns a ResponseEntity instance instead of the object you want to set on the response body directly. The ResponseEntity class allows you to set the response body’s value and the response status and headers. We set 3 headers and change the response status to “202 Accepted.”

## **PROJECT 15 - custom exception handling at the endpoint - ResponseEntity**

One of the ways you can manage exceptions is catching them in the controller’s action and using the **ResponseEntity** class, to send a different configuration of the response when the exception occurs.

We’ll start by demonstrating this approach with an example. I’ll then show you an alternative approach I prefer by using a REST controller advice class: an aspect that intercepts an endpoint call when it throws an exception, and you can specify a custom logic to be executed for that specific exception.

Let’s create a new project. For our scenario, we define an exception named NotEnoughMoneyException, and the app will throw this exception when it cannot fulfill the payment because the client doesn’t have enough money in their account:

public class NotEnoughMoneyException extends RuntimeException {

}

We also implement a service class that defines the use case. For our test, we directly throw this exception. In a real-world scenario, the service would implement the complex logic for making the payment.

@Service

public class PaymentService {

public PaymentDetails processPayment() {

throw new NotEnoughMoneyException();

} }

*PaymentDetails*, the returned type of the *processPayment()* method, is just a model class describing the response body we expect the controller’s action to return for a successful payment:

public class PaymentDetails {

private double amount;

// Omitted getters and setters

}

When the app encounters an exception, it uses another model class named *ErrorDetails* to inform the client of the situation. The ErrorDetails class is also simple and only defines the error message as an attribute:

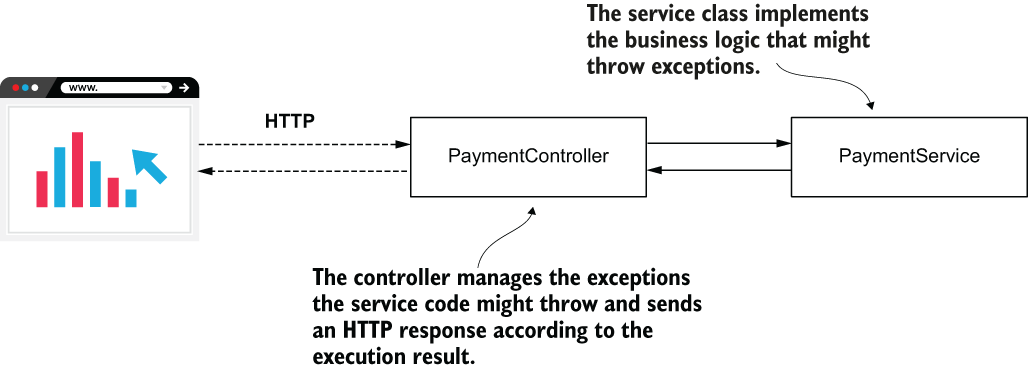
public class ErrorDetails {

private String message;

// Omitted getters and setters

}

How could the controller decide what object to send back depending on how the flow executed? When there’s no exception (the app successfully completes the payment), we want to return an HTTP response with the status “Accepted” of type PaymentDetails. Suppose the app encountered an exception during the execution flow. In that case, the controller’s action returns an HTTP response with the status “400 Bad Request” and an ErrorDetails instance containing a message that describes the issue. Figure 10.8 visually presents the relationship between the components and their responsibilities.



Listing 10.8 Managing the HTTP response for exceptions in the **controller**’s action

@RestController

public class PaymentController {

private final PaymentService paymentService;

public PaymentController(PaymentService paymentService) {

this.paymentService = paymentService;

}

@PostMapping("/payment")

public ResponseEntity<?> makePayment() {

try {

PaymentDetails paymentDetails = ❶

paymentService.processPayment();

return ResponseEntity ❷

.status(HttpStatus.ACCEPTED)

.body(paymentDetails);

} catch (NotEnoughMoneyException e) {

ErrorDetails errorDetails = new ErrorDetails();

errorDetails.setMessage("Not enough money to make the payment.");

return ResponseEntity ❸

.badRequest()

.body(errorDetails);

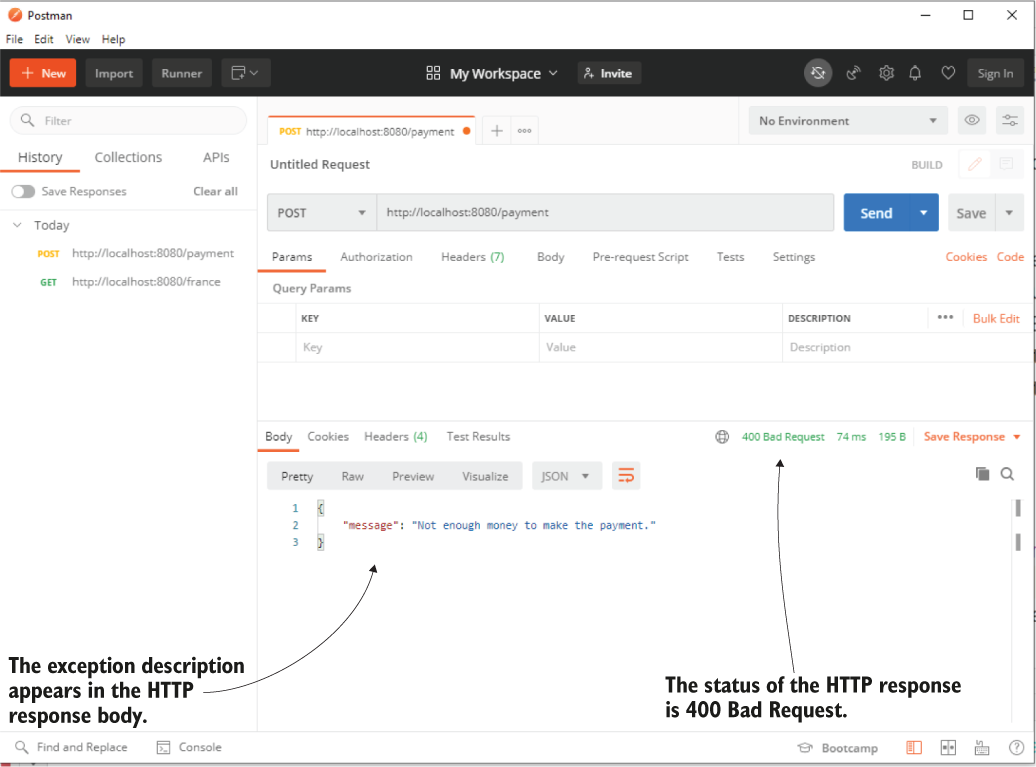
} }}

❶ We try calling the processPayment() method of the service.

❷ If calling the service method succeeds, we return an HTTP response with status Accepted and the PaymentDetails instance as a response body.

❸ If an exception of type NotEnoughMoneyException is thrown, we return an HTTP response with status Bad Request and an ErrorDetails instance as a body.

Start the application and call the endpoint using Postman or cURL. We know that we made the service method to always throw the NotEnoughMoneyException, so we expect to see the response status message is “400 Bad Request,” and the body contains the error message. Figure 10.9 presents the result of sending a request to the /payment endpoint in Postman.



This approach is good, and you’ll often find developers using it to manage the exception cases. However, in a more complex application, you would find it more comfortable to separate the responsibility of exception management.

## **PROJECT 16 - custom exception handling - aspect – rest controller advice**

First, sometimes the same exception has to be managed for multiple endpoints, and we don’t want to introduce duplicated code. Second, it’s more comfortable to know you find the exception logic all in one place when you need to understand how a specific case works. For these reasons, I prefer using a REST controller advice, an aspect that intercepts exceptions thrown by controllers’ actions and applies custom logic you define according to the intercepted exception.

Figure 10.10 presents the changes we want to make in our class design.

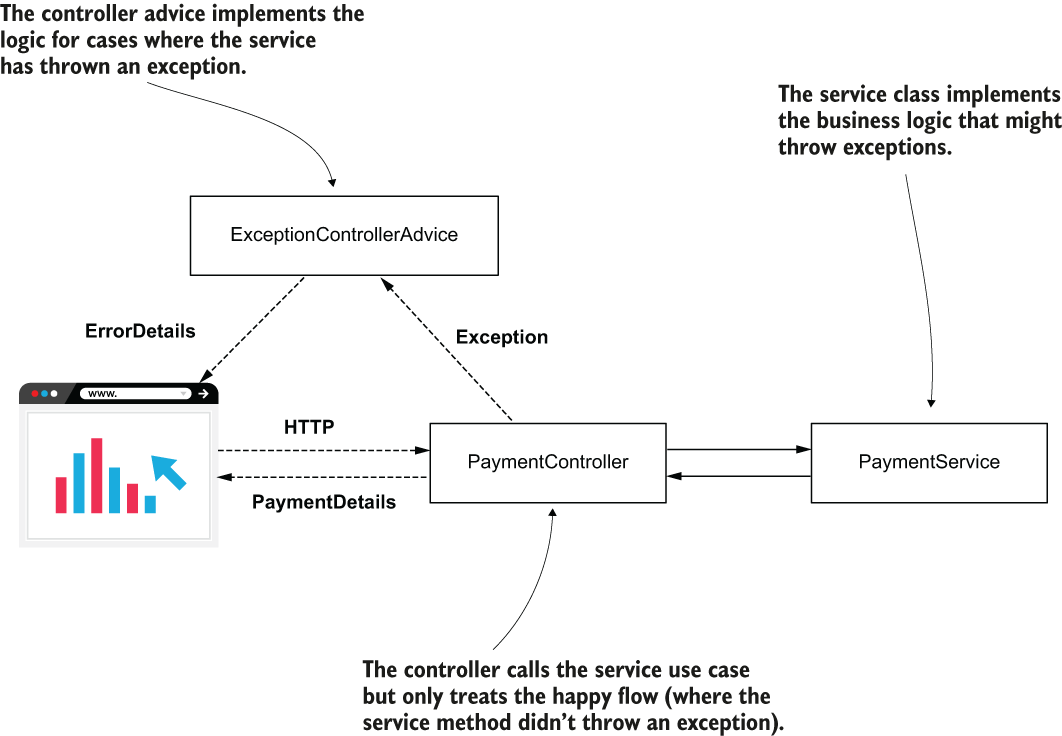


Figure 10.10 Instead of managing the exception cases, the controller now only takes care of the happy flow. We added a controller advice named ExceptionControllerAdvice to take care of the logic that will be implemented if the controller’s action throws an exception.

The controller action is much simplified because it no longer treats the exception case:

Listing 10.9 **Controller**’s action that no longer treats the exception case

@RestController

public class PaymentController {

private final PaymentService paymentService;

public PaymentController(PaymentService paymentService) {

this.paymentService = paymentService;

}

@PostMapping("/payment")

public ResponseEntity<PaymentDetails> makePayment() {

PaymentDetails paymentDetails = paymentService.processPayment();

return ResponseEntity

.status(HttpStatus.ACCEPTED)

.body(paymentDetails);

}}

Instead, we created a separate class named *ExceptionControllerAdvice* that implements what happens if the controller’s action throws a *NotEnoughMoneyException*. The *ExceptionControllerAdvice* class is a REST controller advice. To mark it as a REST controller advice, we use the **@RestControllerAdvice** annotation. The method the class defines is also called an exception handler. You specify what exceptions trigger a controller advice method using the *@ExceptionHandler* annotation over the method. The following listing shows the REST controller advice class’s definition and the exception handler method that implements the logic associated with the *NotEnoughMoneyException* exception.

Listing 10.10 Separating the exception logic with a **REST controller advice**

@RestControllerAdvice ❶

public class ExceptionControllerAdvice {

@ExceptionHandler(NotEnoughMoneyException.class) ❷

public ResponseEntity<ErrorDetails> exceptionNotEnoughMoneyHandler() {

ErrorDetails errorDetails = new ErrorDetails();

errorDetails.setMessage("Not enough money to make the payment.");

return ResponseEntity

.badRequest()

.body(errorDetails);

} }

❶ We use the @RestControllerAdvice annotation to mark the class as a REST controller advice.

❷ We use the @ExceptionHandler method to associate an exception with the logic the method implements.

NOTE In production apps, you sometimes need to send information about the exception that occurred, from the controller’s action to the advice. In this case, you can add a parameter to the advice’s exception handler method of the type of the handled exception. Spring is smart enough to pass the exception reference from the controller to the advice’s exception handler method. You can then use any details of the exception instance in the advice’s logic.

## **PROJECT 17 - @RequestBody – to get data from the client**

**10.4 Using a request body to get data from the client**

In this section, we discuss getting data from the client in the HTTP request body. You learned in chapter 8 that you can send data in the HTTP request using request parameters and path variables. Because REST endpoints rely on the same Spring MVC mechanism, nothing from the syntaxes you learned changes regarding sending data in request parameters and path variables. You can use the same annotations and implement the REST endpoints identically as for your web pages.

However, the HTTP request has a request body, and you can use it to send data from the client to the server. The HTTP request body is often used with REST endpoints. When you need to send a larger quantity of data (my recommendation is anything that takes more than 50 to 100 characters), you use the request body.

To use the request body, you just need to annotate a parameter of the controller’s action with **@RequestBody**. By default, Spring assumes you used JSON to represent the parameter you annotated and will try to decode the JSON string into an instance of your parameter type. In the case Spring cannot decode the JSON-formatted string into that type, the app sends back a response with status “400 Bad Request.” Here, we implement a simple example of using the request body. The controller defines an action mapped to the /payment path with HTTP POST and expects to get a request body of PaymentDetails type. The controller prints the amount of the PaymentDetails object in the server’s console and sends the same object in the response body back to the client.

Listing 10.11 Getting data from the client in the request body

@RestController

public class PaymentController {

private static Logger logger = Logger.getLogger(PaymentController.class.getName());

@PostMapping("/payment")

public ResponseEntity<PaymentDetails> makePayment(

@RequestBody PaymentDetails paymentDetails) { ❶

logger.info("Received payment " + paymentDetails.getAmount()); ❷

return ResponseEntity ❸

.status(HttpStatus.ACCEPTED)

.body(paymentDetails);

}}

❶ We get the payment details from the HTTP request body.

❷ We log the amount of the payment in the server’s console.

❸ We send back the payment details object in the HTTP response body, and we set the HTTP response status to 202 ACCEPTED.

Figure 10.11 shows how to use Postman to call the /payment endpoint with a request body.

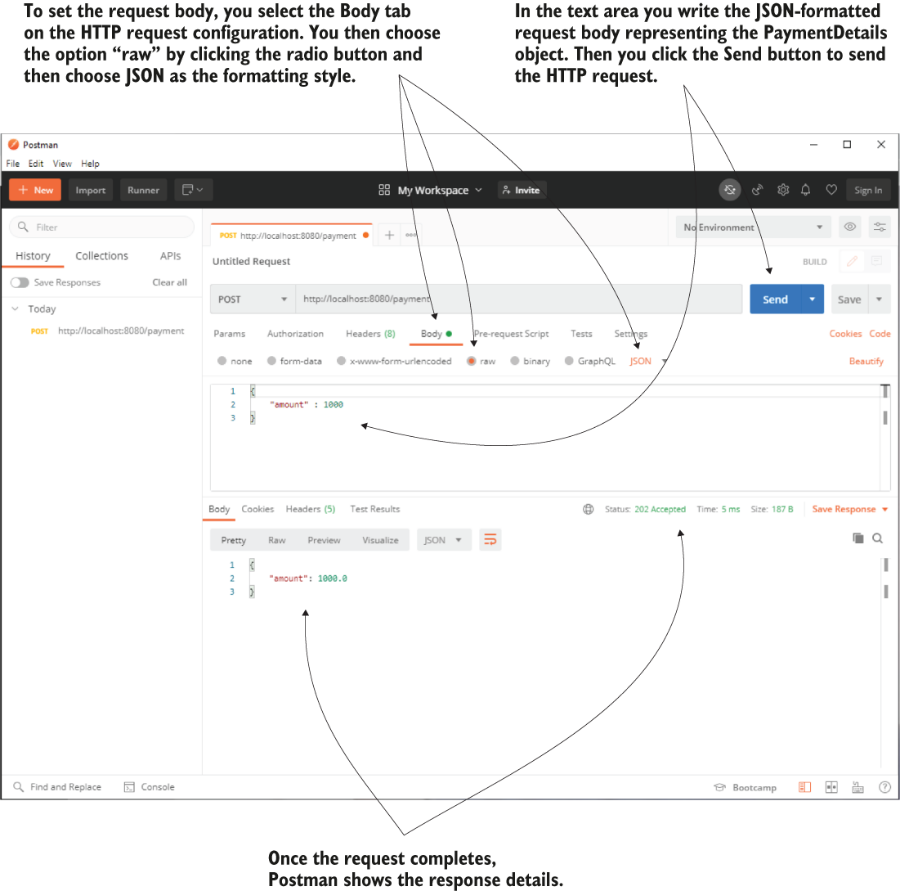


Figure 10.11 Using Postman to call the endpoint and specify the request body. You need to fill the JSON-formatted request body in the request body text area and select the data encoding as JSON. Once the request completes, Postman displays the response details.

If you prefer using cURL, you can use the command presented by the next snippet:

curl -v -X POST http://127.0.0.1:8080/payment -d '{"amount": 1000}' -H

➥ "Content-Type: application/json"

! Can an HTTP GET endpoint use a request body?

Before 2014, the HTTP protocol specification didn’t allow a request body for HTTP GET calls. No implementation for the client or server side allowed you to use a request body with an HTTP GET call.

The HTTP specification changed in 2014, and it now allows the use of the request body with an HTTP GET call. But sometimes students find old articles on the internet or read book editions that were not updated, and this seems to create confusion years later.

You can read more details about the HTTP GET method in section 4.3.1 of the HTTP specification, RFC 7231: <https://tools.ietf.org/html/rfc7231#page-24> .

**Summary**

* Representational state transfer (REST) web services are a simple way to establish communication between 2 applications.
* In a Spring app, the Spring MVC mechanism supports the implementation of REST endpoints. You either need to use the **@ResponseBody** annotation to specify that a method directly returns the response body or replace the **@Controller** annotation with **@RestController** to implement a REST endpoint. If you don’t use one of these, the dispatcher servlet will assume the controller’s method returns a view name and try to look for that view instead.
* You can make the controller’s action directly return the HTTP response body and rely on Spring default’s behavior for the HTTP status.
* You can manage the HTTP status and headers by making your controller’s action return a **ResponseEntity** instance.
* One way to manage exceptions is to treat them directly at the controller’s action level. This approach couples the logic used to treat the exception to that specific controller action. Sometimes using this approach can lead to code duplication, which is best to avoid.
* You can manage the exceptions in the controller’s action directly or separate the logic executed if the controller’s action throws an exception using a REST controller advice class.
* An endpoint can get data from the client through the HTTP request in request parameters, path variables, or the HTTP request body.

11 Implementing REST services

In chapter 10, we discussed implementing REST endpoints. REST services are a common way to implement the communication between 2 system components. The client of a web app can call the backend, and so can another backend component. In a backend solution composed of multiple services (appendix A), these components need to “speak” to exchange data, so when you implement such a service using Spring, you need to know how to call a REST endpoint exposed by another service (fig.11.1).

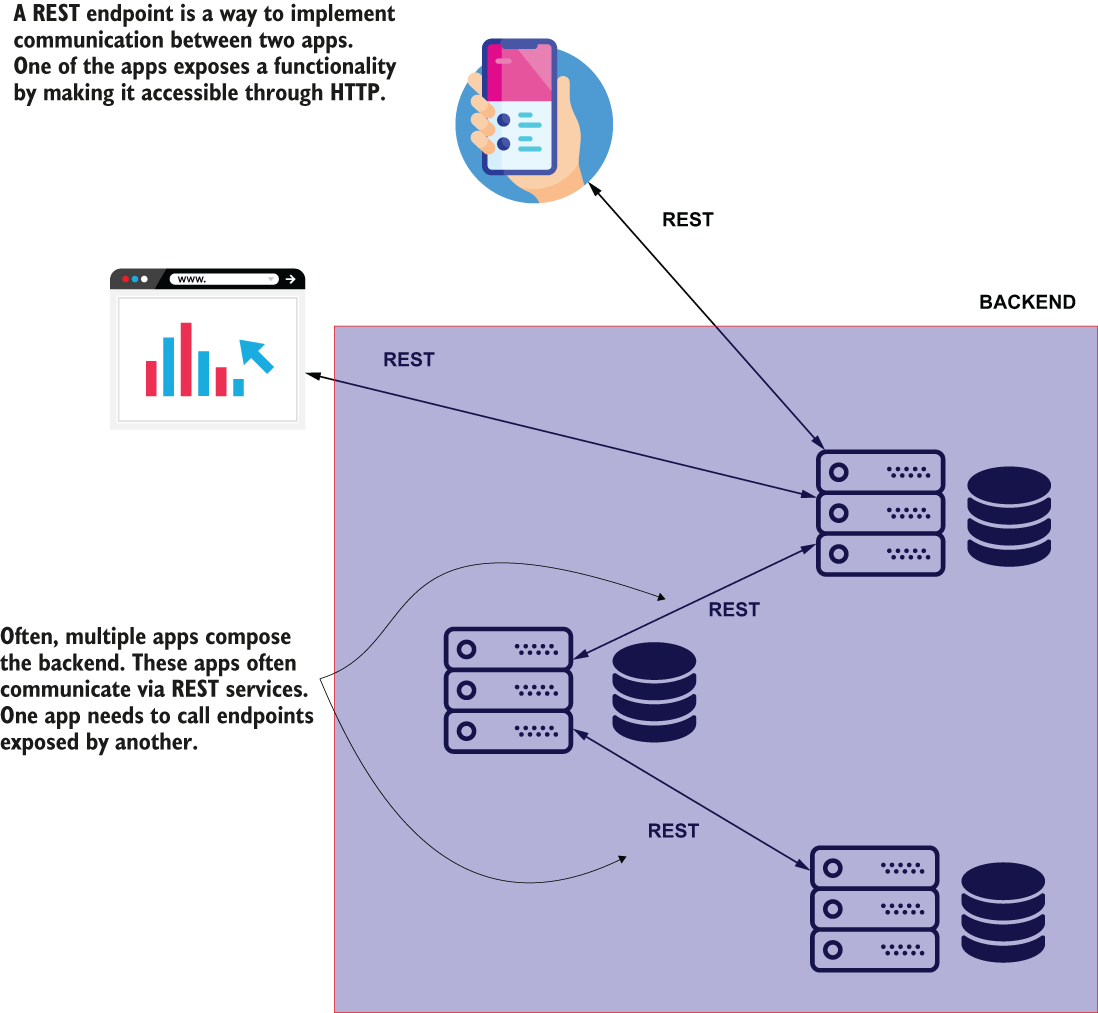


Figure 11.1 Often, a backend app needs to act as a client for another backend app, and calls exposed REST endpoints to work with specific data.

In this chapter, you’ll learn 3 ways to call REST endpoints from a Spring app:

1. **OpenFeign**—A tool offered by the Spring Cloud project. I recommend developers use this feature in new apps for consuming REST endpoints.
2. **RestTemplate**—A well-known tool developers have used since Spring 3 to call REST endpoints. RestTemplate is often used today in Spring apps. However, OpenFeign is a better alternative to RestTemplate, so if you work on a new app, you’ll probably avoid RestTemplate and use OpenFeign instead.
3. **WebClient**—A Spring feature presented as an alternative to RestTemplate. This feature uses a different programming approach named reactive programming, which we’ll discuss at the end of this chapter.

The first Spring capability we discuss, in section 11.1, is *OpenFeign*, which is part of the Spring Cloud family and a feature I recommend for all new implementations today. As you’ll learn, OpenFeign offers a simple syntax and makes calling a REST endpoint from a Spring app straightforward.

In section 11.2, we’ll use *RestTemplate*. But be careful! RestTemplate has been put in maintenance mode starting with Spring 5, and it will eventually be deprecated. Yet, most of today’s Spring projects use RestTemplate to call REST endpoints because they started when this was the only or the best solution for implementing such a capability. For some of these apps, RestTemplate’s capabilities are enough and work fine, so replacing them makes no sense. Sometimes the time needed to replace RestTemplate with a newer solution might be too costly, so learning it is still a must for a Spring developer.

Here’s an interesting fact that usually creates confusion for students. In the RestTemplate documentation (<http://mng.bz/7lWe> ), WebClient is given as a recommendation for replacing the use of RestTemplate. In section 11.3, I’ll explain why using WebClient is not always the best alternative to RestTemplate. We’ll discuss WebClient and clarify when it’s best to use this capability.

To teach you these 3 fundamental ways, we’ll write an example for each. We’ll first implement a project that exposes an endpoint. Our purpose is to call the endpoint in each approach we discuss in this chapter: OpenFeign, RestTemplate, and WebClient.

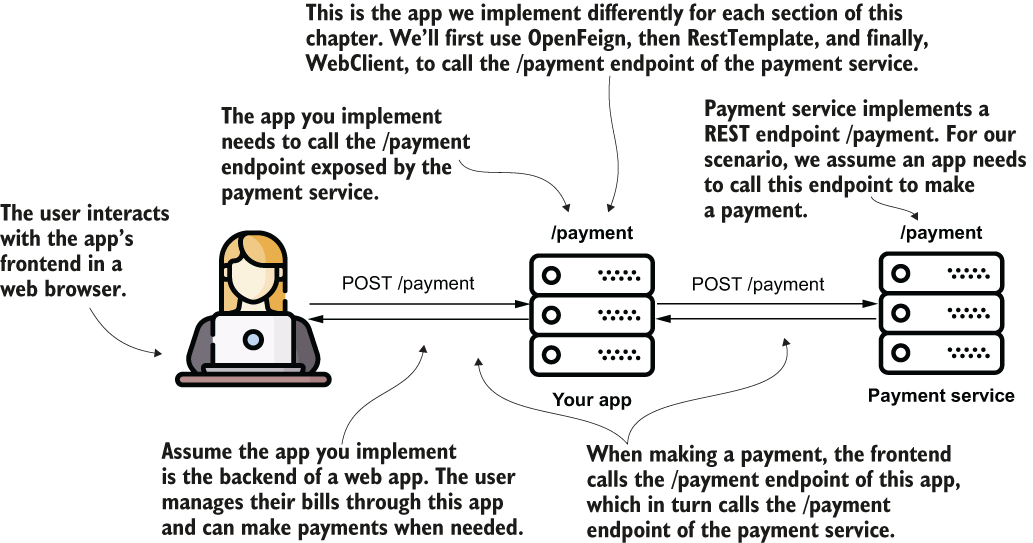


Figure 11.2 To properly teach you how to call REST endpoints, we’ll implement several examples. For each example, we implement 2 projects. One exposes a REST endpoint. The second demonstrates the implementation for calling that REST endpoint using OpenFeign, RestTemplate, and WebClient.

Suppose you implement an app that allows users to make payments. To make a payment, you need to call an endpoint of another system. Figure 11.2 visually presents this scenario. Figure 11.3 details the scenario showing the request and response details.

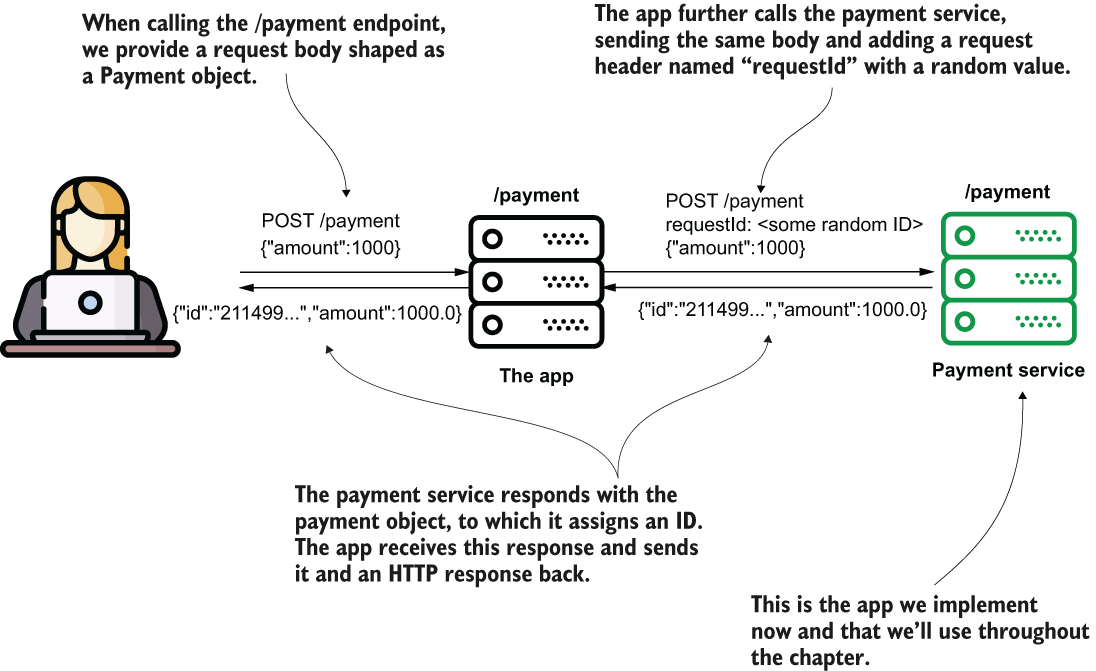


Figure 11.3 The payment service exposes an endpoint that requires an HTTP request body. The app uses OpenFeign, RestTemplate, or WebClient to send requests to the endpoint the payment service exposes.

## **PROJECT 18 – Payment service app**

With this first project, we implement the payment service app. We’ll use this app in all our next examples.

Let’s create the project “sq-ch11-payments,” which represents the payments service. It’s a web app, so, like all the projects we discussed in chapters 7 through 10, we need to add to the pom.xml file the web dependency, as presented in the next code snippet:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

We’ll model the payment with the Payment class:

public class Payment {

private String id;

private double amount;

// Omitted getters and setters

}

Listing 11.1 shows the endpoint’s implementation in the controller class. Technically, it doesn’t do much. The method receives a Payment instance and sets a random ID to the payment before returning it. The endpoint is simple but good enough for our demonstration. We use HTTP POST. We need to specify a request header and the request body. The endpoint returns a header in the HTTP response and the Payment object in the response body when called.

Listing 11.1 The /payment endpoint’s implementation in the controller class

@RestController

public class PaymentsController {

private static Logger logger = ❶

Logger.getLogger(PaymentsController.class.getName());

@PostMapping("/payment") ❷

public ResponseEntity<Payment> createPayment(

@RequestHeader String requestId, ❸

@RequestBody Payment payment ❸

) {

logger.info("Received request with ID " + requestId +

" ;Payment Amount: " + payment.getAmount());

payment.setId(UUID.randomUUID().toString()); ❹

return ResponseEntity ❺

.status(HttpStatus.OK)

.header("requestId", requestId)

.body(payment);

}}

❶ We use a logger to prove the right controller’s method gets the correct data when the endpoint is called.

❷ The app exposes the endpoint with HTTP POST at the path /payment.

❸ The endpoint needs to get a request header and the request body from the caller. The controller method gets these two details as parameters.

❹ The method sets a random value for the payment’s ID.

❺ The controller action returns the HTTP response. The response has a header and the response body that contains the payment with the random ID value set.

You can now run this app, and it will start Tomcat on port 8080, which is the Spring Boot default. The endpoint is accessible, and you could call it with cURL or Postman. But the purpose of this chapter is to learn how to implement an app that calls the endpoint, so this is what we will do in sections 11.1, 11.2, and 11.3.

## **PROJECT 19 – @OpenFeign – to call Payment service app endpoint**

**11.1 Calling REST endpoints using Spring Cloud OpenFeign**

In this section, we discuss a modern approach for calling REST endpoints from a Spring app. In most apps, developers have used RestTemplate. RestTemplate is in maintenance mode starting with Spring 5. Moreover, RestTemplate will be deprecated soon, so I prefer to begin this chapter by discussing the alternative to RestTemplate I recommend you use: OpenFeign.

With OpenFeign, you only need to write an interface, and the tool provides you with the implementation.

To teach you how OpenFeign works, we’ll create the project “sq-ch11-ex1” and implement an app that uses OpenFeign to call the endpoint app “sq-ch11-payments” exposes (figure 11.4).



Figure 11.4 We now implement the app that consumes the /payment endpoint the payment service exposes. We use OpenFeign to implement the functionality that consumes the REST endpoint.

We’ll define an interface where we declare the methods that consume REST endpoints. The only thing we need to do is annotate these methods to define the path, the HTTP method, and eventually parameters, headers, and the body of the request. The interesting thing is that we don’t need to implement the methods ourselves. You define with the interface methods based on the annotations, and Spring knows to implement them. We rely again on the excellent magic of Spring.

Figure 11.5 shows the class design for the application we’ll build that consumes a REST endpoint.

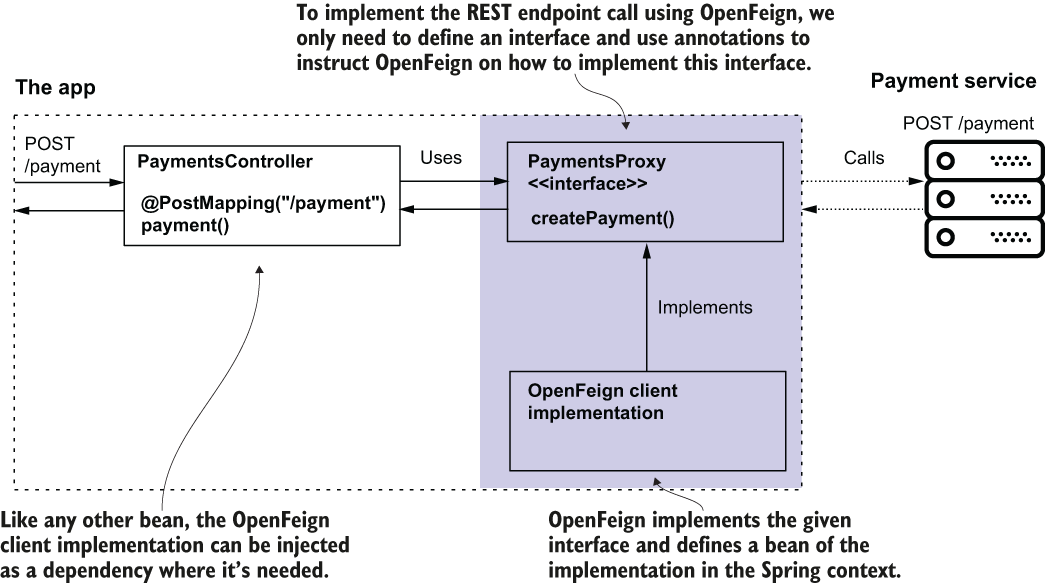


Figure 11.5 With OpenFeign, you only need to define an interface (a contract) and tell OpenFeign where to find this contract to implement it. OpenFeign implements the interface and provides the implementation as a bean in the Spring context based on configurations you define with annotations. You can inject the bean from the Spring context anywhere you need it in your app.

Your pom.xml file needs to define the dependency:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

Now, you can create the proxy interface. In OpenFeign terminology, we also name this interface the OpenFeign client. OpenFeign implements this interface, so you don’t have to bother writing the code that calls the endpoint. You only need to use a few annotations to tell OpenFeign how to send the request. The following listing shows you how simple the definition of the request is with OpenFeign.

Listing 11.2 Declaring an OpenFeign client interface

@FeignClient(name = "payments", ❶

url = "${name.service.url}")

public interface PaymentsProxy {

@PostMapping("/payment") ❷

Payment createPayment(

@RequestHeader String requestId, ❸

@RequestBody Payment payment); ❸

}

❶ We use the @FeignClient annotation to configure the REST client. A minimal configuration defines a name and the endpoint base URI.

❷ We specify the endpoint’s path and HTTP method.

❸ We define the request headers and body.

The first thing to do is annotate the interface with the **@FeignClient** annotation to tell OpenFeign it has to provide an implementation for this contract. We have to assign a name to the proxy using the name attribute of the @FeignClient annotation, which OpenFeign internally uses. The name uniquely identifies the client in your app. The @FeignClient annotation is also where we specify the base URI of the request. You can define the base URI as a string using the URL attribute of @FeignClient.

NOTE Ensure you always store URIs and other details that might differ from one environment to another in the properties files and never hardcode them in the app.

You can define a **property** in the project’s “application.properties” file and refer it from the source code using the following syntax: ${property\_name}. Using this practice, you don’t need to recompile the code when you want to run the app in different environments.

Each method you declare in the interface represents a REST endpoint call. You use the same annotations you learned in chapter 10 for the controller’s actions to expose REST endpoints:

* To specify the path and HTTP method: *@GetMapping*, *@PostMapping*, *@PutMapping*, etc.
* To specify a request header: *@RequestHeader*
* To specify the request body: *@RequestBody*

I find this aspect of reusing the annotation beneficial. Here, by “reusing the annotation,” I mean OpenFeign uses the same annotations we use when we define the endpoints. You don’t have to learn something specific to OpenFeign. Just use the same annotations as for exposing the REST endpoints in the Spring MVC controller classes.

OpenFeign needs to know where to find the interfaces defining the client contracts. We use the **@EnableFeignClients** annotation on a configuration class to enable the OpenFeign functionality and tell OpenFeign where to search for the client contracts.

Listing 11.3 Enabling the OpenFeign clients in the configuration class

@Configuration

@EnableFeignClients( ❶

basePackages = "com.example.proxy")

public class ProjectConfig {

}

❶ We enable the OpenFeign clients and tell the OpenFeign dependency where to search for the proxy contracts.

You can now inject the OpenFeign client through the interface you defined in listing 11.2. Once you enable OpenFeign, it knows to implement the interfaces annotated with @FeignClient. In chapter 5, we discussed that Spring is smart enough to provide you with a bean instance from its context when you use an abstraction, and this is exactly what you do here. The following listing shows you the controller class that injects the FeignClient.

Listing 11.4 Injecting and using the OpenFeign client

@RestController

public class PaymentsController {

private final PaymentsProxy paymentsProxy;

public PaymentsController(PaymentsProxy paymentsProxy) {

this.paymentsProxy = paymentsProxy;

}

@PostMapping("/payment")

public Payment createPayment(@RequestBody Payment payment) {

String requestId = UUID.randomUUID().toString();

return paymentsProxy.createPayment(requestId, payment);

}}

Now start both projects (the payments service and this section’s app) and call the app’s /payment endpoint using cURL or Postman. Using cURL, the request command looks the following snippet:

curl -X POST -H 'content-type:application/json' -d '{"amount":1000}'

➥ <http://localhost:9090/payment>

curl -X POST http:/localhost:9090/payment -H "content-type:application/json" -d "{\"amount\":1000}"}"

In the console where you executed the cURL command, you’ll find a response, as presented in the next snippet:

{"id":"1c518ead-2477-410f-82f3-54533b4058ff","amount":1000.0}

In the payment service’s console, you find the log proving that the app correctly sent the request to the payment service:

Received request with ID 1c518ead-2477-410f-82f3-54533b4058ff ;Payment

➥ Amount: 1000.0

## **PROJECT 20 – RestTemplate – to call Payment service app endpoint**

**11.2 Calling REST endpoints using RestTemplate**

Here again implement the app that calls the /payment endpoint of the Payment service, but this time we use a different approach: RestTemplate.

Don’t conclude that RestTemplate has any problems. It is being put to sleep not because it’s not working properly or because it’s not a good tool. But as apps evolved, we started to need more capabilities. Developers wanted to be able to benefit from different things that aren’t easy to implement with RestTemplate, such as the following:

* Calling the endpoints both synchronously and asynchronously
* Writing less code and treating fewer exceptions (eliminate boilerplate code)
* Retrying call executions and implementing fallback operations (logic performed when the app can’t execute a specific REST call for any reason)

In other words, developers prefer to get more things out of the box rather than implement them wherever possible. Remember that reusing code and avoiding boilerplate code is one of the primary purposes of frameworks. If you compare the examples in sections 11.1 and 11.2, you will observe that using OpenFeign is much easier than using RestTemplate.

**NOTE** Here is a good lesson I learned in my experience: When something is called “deprecated” or “legacy,” it doesn’t necessarily mean you shouldn’t learn it. Sometimes, deprecated technologies are still used in projects many years after being declared deprecated, including RestTemplate and the Spring Security OAuth project.

The steps for defining the call are as follows (figure 11.6):

1. Define the HTTP headers by creating and configuring an ***HttpHeaders*** instance.
2. Create an ***HttpEntity*** instance that represents the request data (headers and body).
3. Send the HTTP call using the ***exchange()*** method and get the HTTP response.

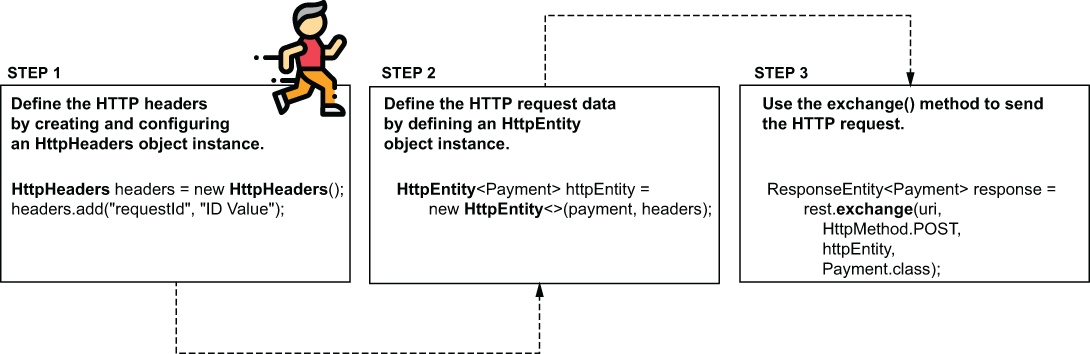


Figure 11.6 To define a more complex HTTP request, you have to use the HttpHeaders class to define the headers, then the HttpEntity class to represent the full request data. Once you defined the data on the request, you call the exchange() method to send it.

In listing 11.5, you find the definition of the proxy class. Observe how the createPayment() method defines the header by creating an HttpHeaders instance and adding the needed header “requestId” to this instance using the add() method. It then creates a HttpEntity instance based on the headers and the body (received by the method as a parameter). The method then sends the HTTP request using RestTemplate’s exchange() method. The exchange() method’s parameters are the URI and the HTTP method, followed by the HttpEntity instance (that holds the request data) and the type expected for the response body.

Listing 11.5 The app’s PaymentsProxy calling the /payment endpoint

@Component

public class PaymentsProxy {

private final RestTemplate rest;

@Value("${name.service.url}")

private String paymentsServiceUrl; ❶

public PaymentsProxy(RestTemplate rest) { ❷

this.rest = rest;

}

public Payment createPayment(Payment payment) {

String uri = paymentsServiceUrl + "/payment";

HttpHeaders headers = new HttpHeaders(); ❸

headers.add("requestId", ❸

UUID.randomUUID().toString()); ❸

HttpEntity<Payment> httpEntity = ❹

new HttpEntity<>(payment, headers);

ResponseEntity<Payment> response = ❺

rest.exchange(uri, ❺

HttpMethod.POST, ❺

httpEntity, ❺

Payment.class); ❺

return response.getBody(); ❻

} }

❶ We take the URL to the payment service from the properties file.

❷ We inject the RestTemplate from the Spring context using constructor DI.

❸ We build the HttpHeaders object to define the HTTP request headers.

❹ We build the HttpEntity object to define the request data.

❺ We send the HTTP request and retrieve the data on the HTTP response.

❻ We return the HTTP response body.

We define a simple endpoint to call this implementation.

Listing 11.6 Defining a controller class to test the implementation

@RestController

public class PaymentsController {

private final PaymentsProxy paymentsProxy;

public PaymentsController(PaymentsProxy paymentsProxy) {

this.paymentsProxy = paymentsProxy;

}

@PostMapping("/payment") ❶

public Payment createPayment(

@RequestBody Payment payment ❷

) {

return paymentsProxy.createPayment(payment); ❸

}}

❶ We define a controller action and map it to the /payment path.

❷ We get the payment data as a request body.

❸ We call the proxy method, which in turn calls the endpoint of the payments service. We get the response body and return the body to the client.

We run both apps on different ports to validate our implementation works as expected. For this example, I kept the same configuration from section 11.1.1: port 8080 for the payment service and port 9090 for this section’s app.

Using cURL, you can call the app’s endpoint, as presented in the next snippet:

curl -X POST -H 'content-type:application/json' -d '{"amount":1000}'

➥ <http://localhost:9090/payment>

curl -X POST http:/localhost:9090/payment -H "content-type:application/json" -d "{\"amount\":1000}"}"

In the console where you executed the cURL command, you’ll find a response, as presented in the next snippet:

{

"id":"21149959-d93d-41a4-a0a3-426c6fd8f9e9",

"amount":1000.0

}

In the payment service’s console, you find the log proving that the app correctly sent the payment service request:

Received request with ID e02b5c7a-c683-4a77-bd0e-38fe76c145cf ;Payment

➥ Amount: 1000.0

## **PROJECT 21 – WebClient – reactive - to call Payment service app endpoint**

**11.3 Calling REST endpoints using WebClient**

In this section, we discuss using **WebClient** to call REST endpoints. WebClient is a tool used in different apps and is built on a methodology we call a reactive approach. The reactive methodology is an advanced approach, and I recommend studying it once you know the basics well. A good starting point is reading chapters 12 and 13 of Spring in Action, 6th ed., by Craig Walls (Manning, 2021).

Spring’s documentation recommends using WebClient, but that’s only a valid recommendation for reactive apps. If you aren’t writing a reactive app, use OpenFeign instead. Like anything else in software, it fits well for some cases, but might complicate things for others. Choosing WebClient to implement the REST endpoint calls is strongly coupled to making your app reactive.

**NOTE** If you decide not to implement a reactive app, use OpenFeign to implement the REST client capabilities. If you implement a reactive app, you should use a proper reactive tool: WebClient.

Even though reactive apps are a bit beyond the basics, I’d like to make sure you know what using WebClient looks like and how this tool differs from others we have discussed so that you can compare the approaches. Let me tell about reactive apps and then use WebClient to call the /payment endpoint.

In a nonreactive app, a thread executes a business flow. Multiple tasks compose a business flow, but these tasks are not independent. The same thread executes all the tasks composing a flow. Let’s take an example to observe where this approach might face issues and how we can enhance it.

Suppose you implement a banking application where a bank’s client has one or more credit accounts. The system component you implement calculates the total debt of a bank’s client. To use this functionality, other system components make a REST call to send a unique ID to the user. To calculate this value, the flow you implement includes the following steps (figure 11.7):

1. The app receives the user ID.
2. It calls a different service of the system to find out if the user has credits with other institutions.
3. It calls a different service of the system to get the debt for internal credits.
4. If the user has external debts, it calls an external service to find out the external debt.
5. The app sums the debts and returns the value in an HTTP response.

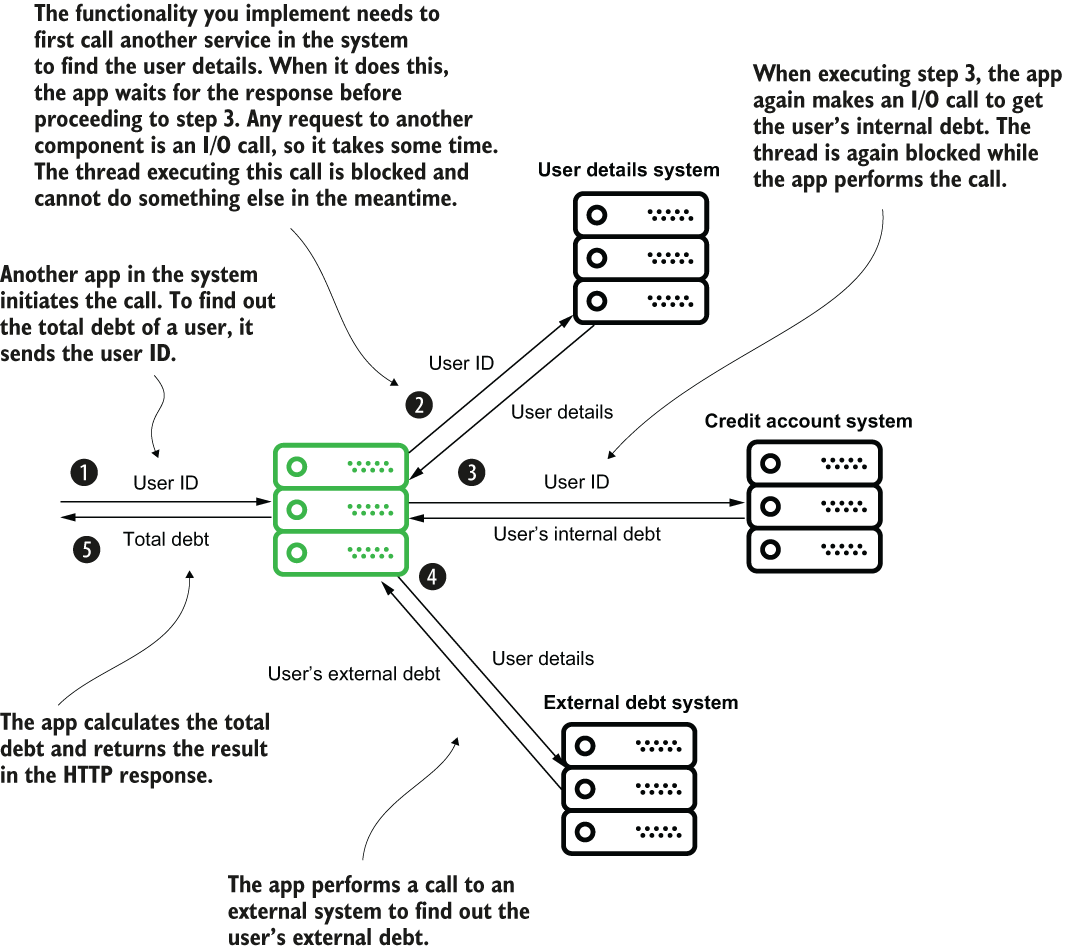


Figure 11.7 A functionality scenario for demonstrating the usefulness of a reactive approach. A banking app needs to call several other apps to calculate the total debt of a user. Due to these calls, the thread executing the request is blocked several times while waiting for I/O operations to finish.

These are just fictive steps of functionality, but I designed them to prove where using a reactive app could be helpful. Let’s analyze these steps deeper. Figure 11.8 presents the execution of the scenario from the thread’s point of view. The app creates a new thread for each request, and this thread executes the steps one by one. The thread has to wait for a step to finish before proceeding to the next one and is blocked every time it waits for the app to perform an I/O call.

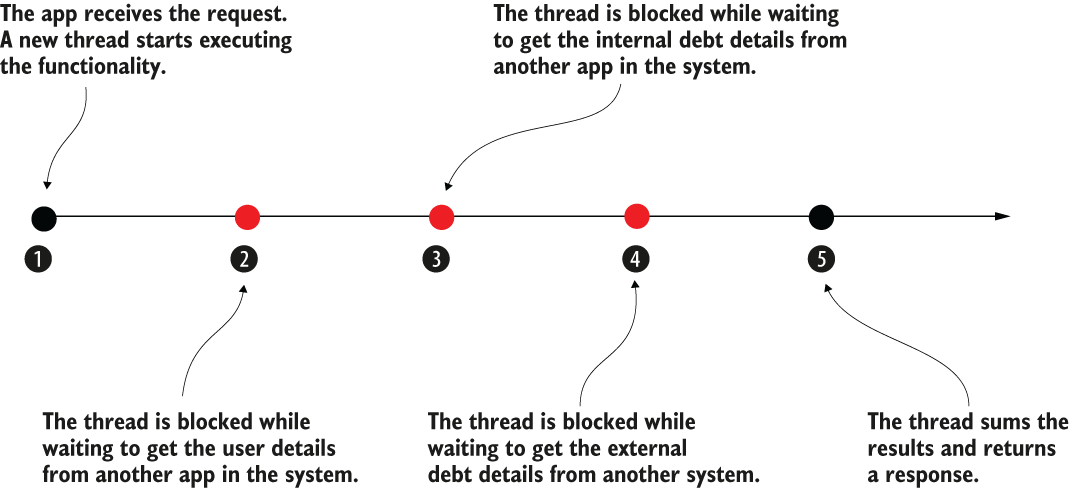


Figure 11.8 The execution of the scenario functionality from the thread point of view. The arrow represents the timeline of the thread. Some of the steps cause details to block the thread, which needs to wait for the task to finish before proceeding.

We observe 2 significant issues here:

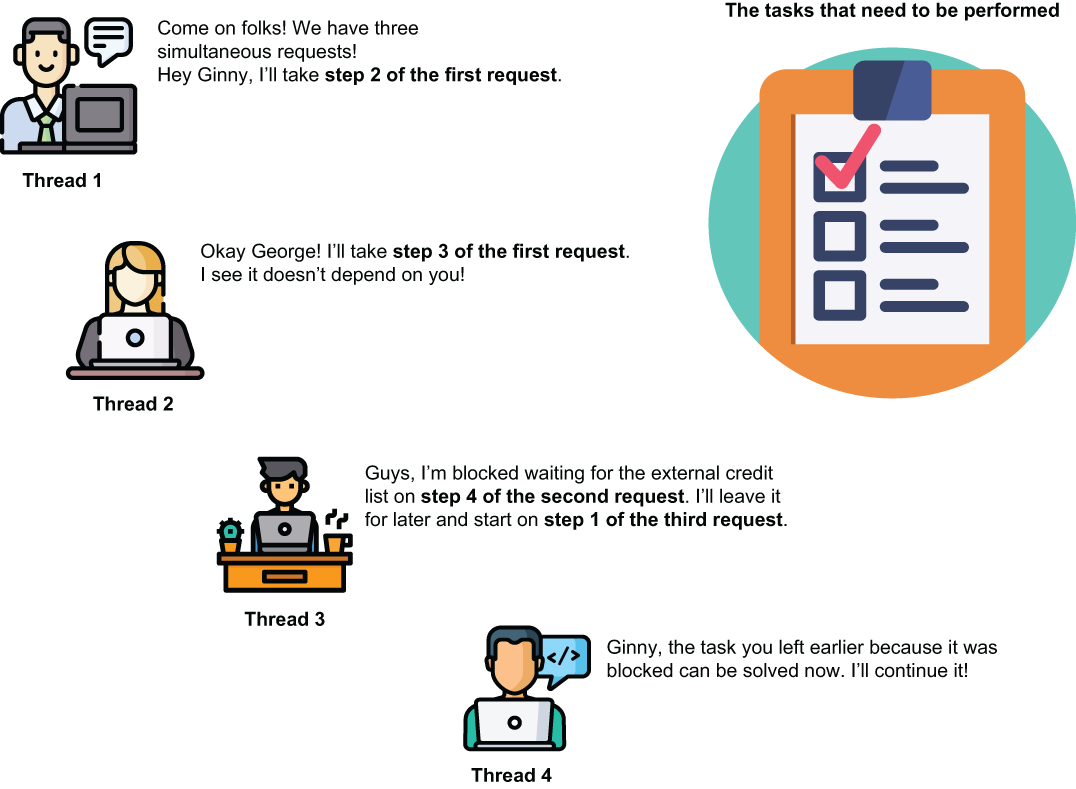
1. The thread is idle while an I/O call blocks it. Instead of using the thread, we allow it to stay and occupy the app’s memory. We consume resources without gaining any benefit. With such an approach, you could have cases where the app gets 10 requests simultaneously, but all the threads are idle simultaneously while waiting for details from other systems.
2. Some of the tasks don’t depend on one another. For example, the app could execute step 2 and step 3 at the same time. There’s no reason for the app to wait for step 2 to end before executing step 3. The app just needs, in the end, the result of both to calculate the total debt.

Reactive apps change the idea of having one atomic flow in which one thread executes all its tasks from the beginning to the end. With reactive apps, we think of tasks as independent, and multiple threads can collaborate to complete a flow composed of multiple tasks.

Instead of imagining this functionality as steps on a timeline, imagine it as a backlog of tasks and a team of developers solving them. With this analogy, I’ll help you imagine how a reactive app works: the developers are threads, and the tasks in the backlog are the steps of a functionality.

Two developers can implement two different tasks simultaneously if they don’t depend on one another. If a developer gets stuck on a task because of an external dependency, they can leave it temporarily and work on something else. The same developer can get back to the task once it’s not blocked anymore, or another developer can finish solving it (figure 11.9).

Figure 11.9 An analogy of the way a reactive app works. A thread doesn’t take a request’s tasks in order and wait when it’s blocked. Instead, all tasks from all requests are on a backlog. Any available thread can work on tasks from any request. This way, independent tasks can be solved in parallel, and the threads don’t stay idle.



Using this approach, you don’t need one thread per each request. You can solve multiple requests with fewer threads because the threads don’t have to stay idle. When blocked on a certain task, the thread leaves it and works on some other task that isn’t blocked.

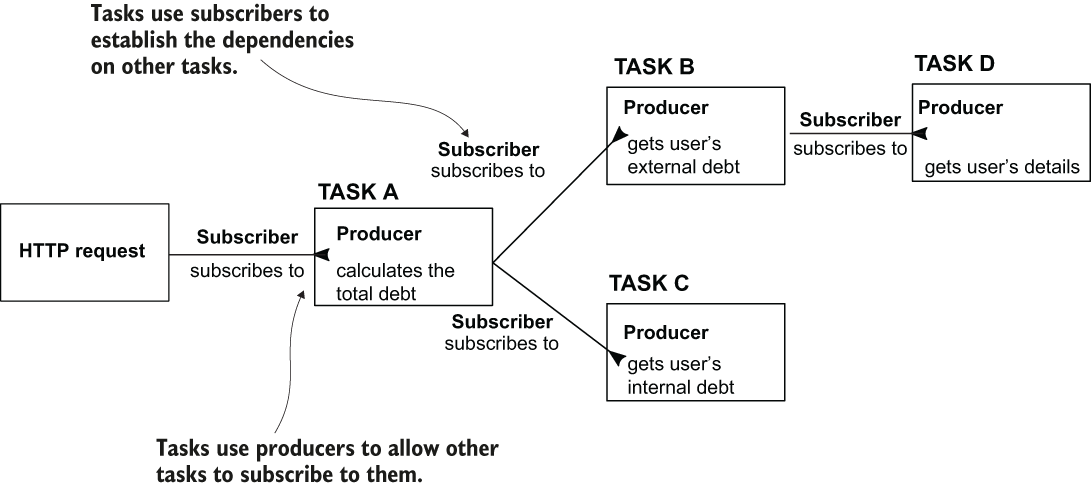
Technically, in a reactive app, we implement a flow by defining the tasks and the dependencies between them. The reactive app specification offers us 2 components: the producer and the subscriber to implement the dependencies between tasks.

A task returns a producer to allow other tasks to subscribe to it, marking the dependency they have on the task. A task uses a subscriber to attach to a producer of another task and consume that task’s result once it ends.

Figure 11.10 shows the discussed scenario implemented in a reactive approach. Compare this visual with figure 11.8. Instead of being steps on a timeline, the tasks are independent of any thread and declare their dependencies. Multiple threads can execute these tasks, and no thread has to wait for a task when an I/O communication blocks it. The thread can begin executing another task.

Moreover, tasks that are not dependent on one another can be simultaneously executed. In figure 11.10, tasks C and D, which were initially steps 2 and 3 in the nonreactive design, can now be executed simultaneously, which helps make the app more performant.

Figure 11.10 In a reactive app, the steps become tasks. Each task marks its dependencies on other tasks and allows other tasks to depend on them. Threads are free to execute any task.



For this section’s app, we use WebClient to send requests to the endpoint the payment service exposes.

Because WebClient imposes a reactive approach, we need to add a dependency named WebFlux instead of the standard web dependency. The next code snippet shows the WebFlux dependency, which you can add to your **pom.xml** file or choose where you build the project using start.spring.io:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-webflux</artifactId>

</dependency>

To call the REST endpoint, you need to use a WebClient instance. The best way to create easy access is to put it in the Spring context using the **@Bean** annotation with a **configuration** class method.

Listing 11.7 Adding a WebClient bean to the Spring context in the configuration class

@Configuration

public class ProjectConfig {

@Bean

public WebClient webClient() {

return WebClient

.builder() ❶

.build();

}}

❶ Creates a WebClient bean and adds it in the Spring context

Listing 11.8 shows the **proxy** class’s implementation, which uses WebClient to call the endpoint the app exposes. The logic is similar to RestTemplate. You take the base URL from the properties file; specify the HTTP method, headers, and body; and execute the call. WebClient’s methods’ names differ, but it’s quite easy to understand what they’re doing after reading their names.

Listing 11.8 Implementing a proxy class with WebClient

@Component

public class PaymentsProxy {

private final WebClient webClient;

@Value("${name.service.url}") ❶

private String url;

public PaymentsProxy(WebClient webClient) {

this.webClient = webClient;

}

public Mono<Payment> createPayment( String requestId, Payment payment) {

return webClient.post() ❷

.uri(url + "/payment") ❸

.header("requestId", requestId) ❹

.body(Mono.just(payment), Payment.class) ❺

.retrieve() ❻

.bodyToMono(Payment.class); ❼

}}

❶ We take the base URL from the properties file.

❷ We specify the HTTP method we use when making the call.

❸ We specify the URI for the call.

❹ We add the HTTP header value to the request. You can call the header() method multiple times if you want to add more headers.

❺ We provide the HTTP request body.

❻ We send the HTTP request and obtain the HTTP response.

❼ We get the HTTP response body.

We use a class named **Mono**. This class defines a producer. In listing 11.8, you find this case, where the method performing the call doesn’t get the input directly. Instead, we send a Mono. This way, we can create an independent task that provides the request body value. The WebClient subscribed to this task becomes dependent on it.

The method also doesn’t return a value directly. Instead, it returns a Mono, allowing another functionality to subscribe to it. This way, the app builds the flow, not by chaining them on a thread, but by linking the dependencies between tasks through producers and consumers (figure 11.11).

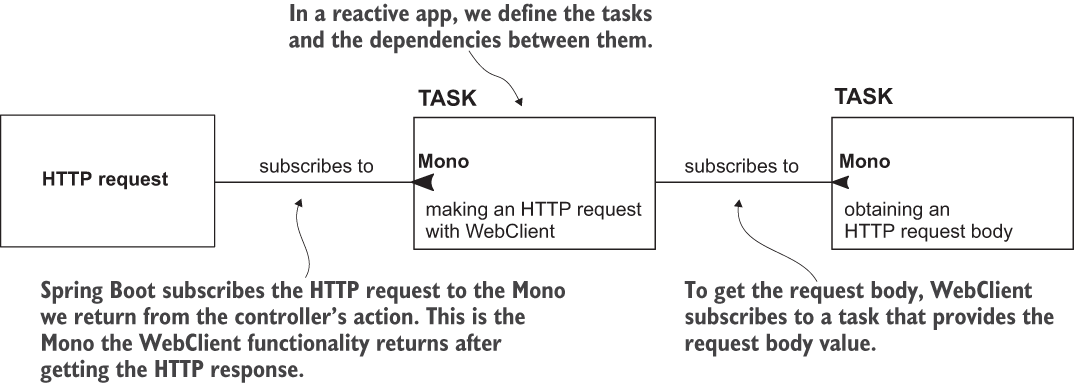


Figure 11.11 The tasks chain in a reactive app. When building a reactive web app, we define the tasks and the dependencies between them. The WebFlux functionality initiating the HTTP request subscribes to the task we create through the producer the controller’s action returns. In our case, this producer is the one we get by sending the HTTP request with WebClient. For WebClient to make the request, it subscribes to another task that provides the request body.

Listing 11.8 also shows the proxy method that consumes a Mono producing the HTTP request body and returns it to what the WebFlux functionality subscribes.

To prove the call works correctly, as we did in this chapter’s previous examples, we implement a controller class that uses the proxy to expose an endpoint we’ll call to test our implementation’s behavior.

Listing 11.9 A controller class exposing an endpoint and calling the proxy

@RestController

public class PaymentsController {

private final PaymentsProxy paymentsProxy;

public PaymentsController(PaymentsProxy paymentsProxy) {

this.paymentsProxy = paymentsProxy;

}

@PostMapping("/payment")

public Mono<Payment> createPayment( @RequestBody Payment payment) {

String requestId = UUID.randomUUID().toString();

return paymentsProxy.createPayment(requestId, payment);

}}

You can test the functionality of both apps by calling the /payment endpoint with cURL or Postman. Using cURL, the request command looks like the following snippet:

curl -X POST -H 'content-type:application/json' -d '{"amount":1000}'

➥ <http://localhost:9090/payment>

curl -X POST http:/localhost:9090/payment -H "content-type:application/json" -d "{\"amount\":1000}"}"

In the console where you executed the cURL command, you’ll find a response like the next snippet:

{

"id":"e1e63bc1-ce9c-448e-b7b6-268940ea0fcc",

"amount":1000.0

}

In the payment service console, you find the log proving that this section’s app correctly sends the request to the payment service:

Received request with ID e1e63bc1-ce9c-448e-b7b6-268940ea0fcc ;Payment

➥ Amount: 1000.0

**Summary**

* In a real-world backend solution, you often find cases when a backend app needs to call endpoints exposed by another backend app.
* Spring offers multiple solutions for implementing the client side of a REST service. 3 of the most relevant solutions are as follows:
  + **OpenFeign** - A solution offered by the Spring Cloud project that successfully simplifies the code you need to write to call a REST endpoint and adds several features relevant to how we implement services today
  + **RestTemplate** - A simple tool used to call REST endpoints in Spring apps
  + **WebClient** - A reactive solution for calling REST endpoints in a Spring app
* You shouldn’t use RestTemplate in new implementations. You can choose between OpenFeign and WebClient to call REST endpoints.
* For an app following a standard (nonreactive) approach, the best choice is using OpenFeign.
* WebClient is an excellent tool for an app designed on a reactive approach. But before using it, you should deeply understand the reactive approach and how to implement a reactive app with Spring.