Appendix C. A quick introduction to HTTP

In this appendix, we discuss the essential aspects of HTTP any developer needs to know. Fortunately, you don't have to be an expert in HTTP and know its reference by heart to implement excellent web apps. On your journey as a software developer, you'll also learn other HTTP aspects, but I want to make sure you have all the needed information to understand the examples we work on in this book, starting with chapter 7.

Why learn about HTTP in a book about Spring? Because today most of the apps we implement with an application framework (such as Spring) are web apps—and web apps use HTTP.

We'll begin with what HTTP is, and we'll analyze its definition in a visual. We'll then discuss the details you need to know about HTTP requests a client makes and how the server responds.

C.1 What is HTTP?

In this section, we discuss what HTTP is. I prefer simple definitions, so I describe it as how a client communicates with the server in a web app. Applications prefer to have rigid ways to "speak," and the protocols offer the rules they need to exchange information. Let's analyze the HTTP definition with a visual (figure C.1).

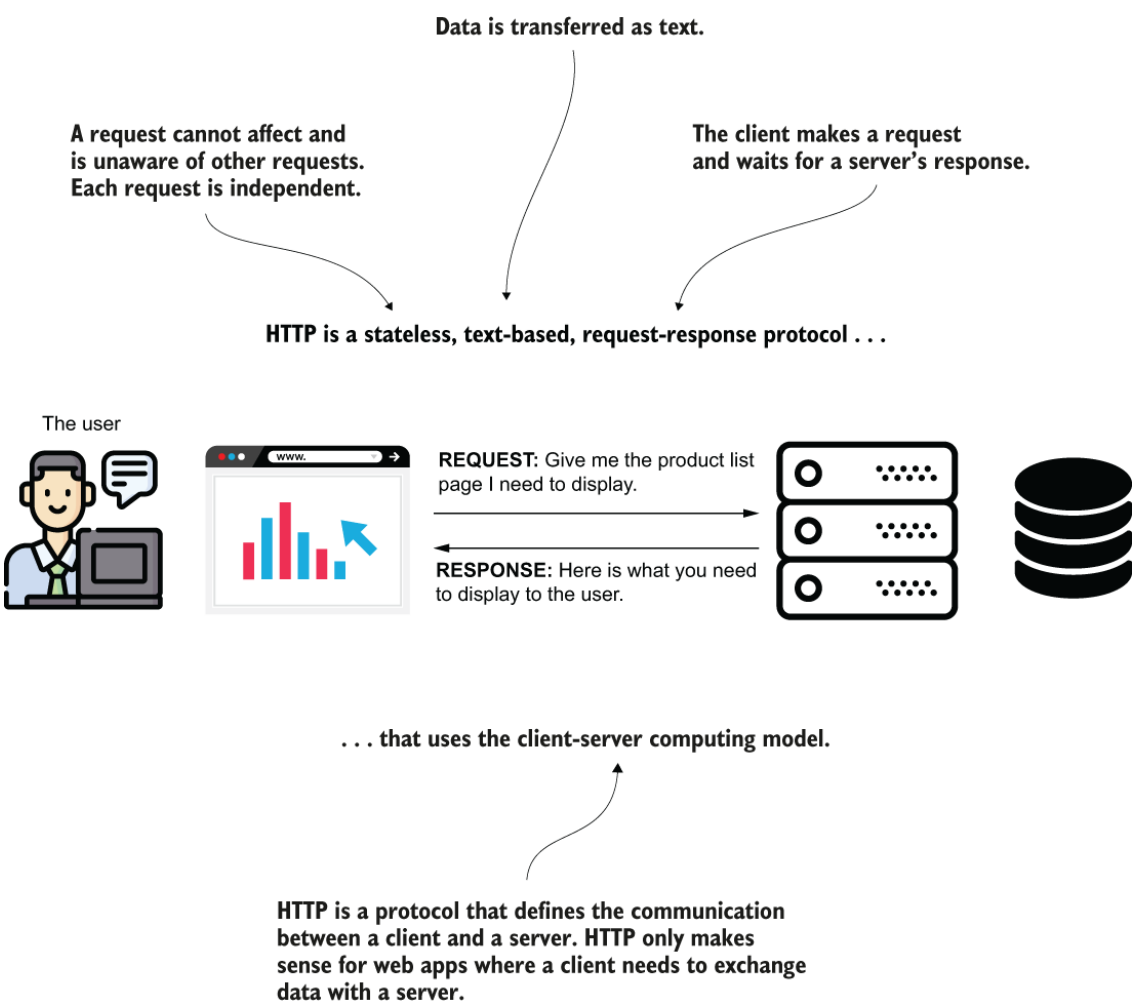

Figure C.1 HTTP is a protocol that describes how a client and a server talk. HTTP assumes a client makes a request, and the server responds. The protocol describes what the client's request and the server's response look like. HTTP is stateless, meaning the requests are independent of one another, and text-based, which means the information is exchanged as plain text.

HTTP Stateless, text-based, request-response protocol that uses the client-server computing model.

C.2 The HTTP request as a language between client and server

In this section, we discuss the HTTP request. In the apps you implement with Spring, you'll need to use the HTTP request to send data from client to server. If you implement the client, you'll need to add data on the HTTP request. If you implement the server, you'll need to get data from the request. Either way, you need to understand the HTTP request.

The HTTP request has a simple format. The things you have to take into consideration are the following:

1. *The request URI*—The client uses the path to tell the server what resource it requests. The request URI looks like this one: **http://www.manning.com/books/spring-start-here**
2. *The request method*—A verb that the client uses to indicate what action it will do with the requested resource. For example, when you write an address in a web browser's address bar, the browser always uses an HTTP method named GET. In other circumstances, you'll find in the next paragraphs, the client can issue an HTTP request with a different method such as POST, PUT, or DELETE.
3. *The request parameters (optional)*—Data in small quantity the client sends to the server with the request. When I say "small quantity," I refer to something that can be expressed in maybe 10 to 50 characters. Parameters on the request aren't mandatory. The request parameters (also referred to as query parameters) are sent in the URI by appending a query expression.
4. *The request headers (optional)*—Data in small quantity sent in the request header. Unlike request parameters, these values are not visible in the URI.
5. *The request body (optional)*—A larger quantity of data the client sends to the server in the request. When the client needs to send data composed of some hundreds of characters, it can use the HTTP body. A body on the request is not mandatory.

The following snippet these details in an HTTP request:

```
POST /servlet/default.jsp HTTP/1.1                    ❶

Accept: text/plain; text/html                         ❷
Accept-Language: en-gb                                ❷
Connection: Keep-Alive                                ❷
Host: localhost                                       ❷
Referer: http://localhost/ch8/SendDetails.html        ❷
User-Agent: Mozilla/4.0 (MSIE 4.01;Windows 98)        ❷
Content-Length: 33                                    ❷
Content-Type: application/x-www-form-urlencoded       ❷
Accept-Encoding: gzip, deflate                        ❷

lastName=Einstein&firstName=Albert                    ❸
```

❶ The request specifies the method and the path.

❷ Different headers with values can be added as request data.

❸ Request parameters can also be used to transfer request data.

The request URI identifies a resource on the server side the client wants to work with. The URI is the part of the HTTP request most people know about because we have to write a URI in our browser's address bar each time we access a web site. The URI has a format like that in the next snippet. In the snippet, <server_location> is the network address of the system where the server app runs, <application_port> is the port number identifying the server app instance running, and <resource_path> is a path the developer associated with a specific resource. The client needs to request a specific path to work with a particular resource:

```
http://<server_location>:<application_port>/<resource_path>
```

Figure C.2 analyzes the format of an HTTP request URI.

Figure C.2 The HTTP request URI identifies the resource the client requests to work with. The first part of the URI identifies the protocol and the server that runs the server app. The path identifies a resource exposed by the server.

NOTE A uniform resource identifier (URI) includes a uniform resource locator (URL) and a path. We can say the formula is URI = URL + path. But in many cases, you'll find people confusing the URI with the URL or considering them to be the same thing. You need to remember that the URL identifies the server and the application. When you add a path to a specific resource of that application, it becomes a URI.

Once the client identifies the resource in the request, it uses a verb named *HTTP request method* to specify what it will do with the resource. The way the client specifies the method depends on how it sends the call to the server. For example, if the call is made directly by the browser, when you write an address in the address bar, the browser will send a GET request. In most cases, when you click a submit button on a form on a web page, the browser uses POST. The developer of the web page decides what method the browser should use when sending a request that

originated as a result of submitting a form. You'll learn more about this aspect in chapter 8. An HTTP request can also be sent by a script written in a client-side language such as JavaScript. In this case, the developer of the script decides what HTTP method the request will use.

The HTTP methods you'll find most often in web apps are as follows:

- *GET*—Expresses the client's intention to obtain some data from the server
- *POST*—Expresses the client's intention to add data on the server
- *PUT*—Expresses the client's intention to change data on the server
- *DELETE*—Expresses the client's intention to remove some data from the server

NOTE Always remember that the verbs are not a constraint to what you implement. The HTTP protocol cannot force you not to implement an HTTP GET functionality that changes data on the backend side. However, you should never misuse the HTTP methods! Always consider the meaning of the HTTP method used to ensure your app's reliability, security, and maintainability.

Less often encountered, but often enough to be relevant are the following HTTP methods:

- *OPTIONS*—Tells the server to return a list of parameters it supports for request. For example, the client can ask which HTTP methods the server supports. The most encountered functionality that uses the OPTIONS method is cross-origin resource sharing (CORS) related to security implementations. You can find an excellent discussion about CORS in chapter 10 of another book I wrote, *Spring Security in Action* (Manning, 2020; **https://livebook.manning.com/book/spring-security-in-action/chapter-10/**).
- *PATCH*—May be used if only part of the data representing a specific resource on the backed is changed. HTTP PUT is used only when the client's action completely replaces a specific resource or even adds it where the data to be updated doesn't exist. In my experience, developers still tend to use HTTP PUT in most cases—even where the action represents only a PATCH.

The URI and the HTTP method are mandatory. The client needs to mention the resource it works with (through URI) and what it does with that resource (the method) when making an HTTP request.

For example, the request represented in the next snippet could be a way to instruct the server to return all the products it manages. We consider here that the product is a resource the server manages:

```
GET http://example.com/products
```

The request represented in the next snippet could mean that the client wants to remove all the products from the server:

```
DELETE http://example.com/products
```

But sometimes the client also needs to send data with the request. The server needs this data to complete the request. Say the client doesn't want to delete all the products, just a specific one. Then the client needs to tell the server what product to delete and send this detail in the request. The HTTP request could look like the one presented in the next snippet, where the client uses a parameter to tell the server they want to delete the product "Beer":

```
DELETE http://example.com/products?product=Beer
```

A client uses either the *request parameters*, *request headers*, or the *request body* to send data to the server. The request parameters and the request body are optional for the HTTP request. A client needs to add them only if they want to send specific data to the server.

The request parameters are key-value pairs the client can attach to the HTTP request to send specific information to the server. We use the request parameters to send small, individual quantities of data. If more data needs to be exchanged, the best way to send the data is through the HTTP request body. In chapters 7 through 10, we use both these approaches to send data from client to server in the HTTP request.

2.3 The HTTP response: The way the server responds

In this section, we discuss the HTTP response. HTTP is the protocol that allows the client to communicate with the server in a web app. Once you take care of the client's request in an app, it's time to implement the server's response. In response to a client's request, the server sends the following:

- *The response status*—An integer between 100 and 599 that defines a short representation of the request's result.
- *Response headers (optional)*—Similar to request parameters, they represent key-value pair data. They are designed for sending a small amount of data (10 to 50 characters) from server to client in response to a client's request.
- *The response body (optional)*—A way for the server to send a larger quantity (e.g., the server needs to send some hundreds of characters or entire files) of data back to the client.

The next snippet helps you visualize the HTTP response.

```
HTTP/1.1 200 OK                                    ❶

Server: Microsoft-IIS/4.0                          ❷
Date: Mon, 14 May 2012 13:13:33 GMT                ❷
Content-Type: text/html                            ❷
Last-Modified: Mon, 14 May 2012 13:03:42 GMT       ❷
Content-Length: 112                                ❷

<html>                                             ❸
<head><title>HTTP Response</title></head>          ❸
<body>Hello Albert!</body>                         ❸
</html>                                            ❸
```

❶ The HTTP response specifies the HTTP version and the response code and message.

❷ The HTTP response can send data through the response headers.

❸ The HTTP response can send data in the response body.

The response status is the only mandatory detail a server delivers in response to a client's request. The status tells the client if the server understood the request and everything worked fine, or if the server encountered issues while processing the client's request. For example, the server returns a status value starting with 2 to tell the client that everything was fine. The HTTP status is a short representation of the result of the full request (including if the server was able to manage the business logic of the request). You don't need to learn all the statuses in detail. I'll enumerate and describe the ones you'll find more often in real-world implementations:

- Starting with 2, meaning the server correctly processed the request. The request processing is okay, and the server executed what the client asked.
- Starting with 4, where the server tells the client something is wrong with its request (it's a problem on the client side). For example, the client requested a resource that doesn't exist, or the client sent some request parameters that the server didn't expect.
- Starting with 5, where the server communicates that something went wrong on its side. For example, the server needed to connect to a database but the it was not accessible. In this case, the server sends back a status telling the client that it couldn't complete the request but not because of something the client didn't do well.

NOTE I will skip the values starting with 1 and 3, which you'll encounter less often in apps, so that you can focus on the other three essential categories.

Different values starting with 2 are variations of messages saying that the server correctly processed the client's request. A few examples are as follows:

- *200—OK* is the most known and most straightforward of the response statuses. It just tells the client the server didn't encounter any issue when processing its request.
- *201—Created* might be used, for example, in response to a POST request to tell the client that the server managed to add the requested resource. It's not always mandatory to add such detail to the response status, and that's why 200—OK is, in general, the most used response status to identify that everything's okay.

- *204—No Content* could tell the client it shouldn't expect a response body for this response.

When an HTTP response status value starts with 4, the server tells the client something was wrong with the request. The client did something wrong when requesting a specific resource. It could be that the resource doesn't exist (the well-known 404—Not Found), or maybe some validation of the data didn't go well. Some of the most often encountered client error response statuses are as follows:

- *400—Bad Request*—A generic status often used to represent any kind of problem with the HTTP request (e.g., validation of the data or problem with reading a specific value in the request body or a request parameter).
- *401—Unauthorized*—A status value generally used to communicate to the client that the request needs authentication.
- *403—Forbidden*—A status value generally sent by the server to tell the client it's not authorized to execute its request.
- *404—Not Found*—A status value sent by the server to inform the client the requested resource doesn't exist.

When the response status starts with 5, it means something went wrong on the server side, but it's the server's issue. The client sent a valid request, but the server could not complete it for some reason. The most often used status from this category is *500—Internal Server Error*. This response status is a generic error value the server sends to inform the client that an issue occurred while the backend was processing its request.

If you want to go more in-depth and learn about more status codes, this page is nice to read: **https://datatracker.ietf.org/doc/html/rfc7231**

Optionally, the server sends back data to the client in response through either the response headers or the response body.

2.4 The HTTP session

In this section, we discuss the HTTP session, a mechanism that allows a server to store data between multiple request-response interactions with the same client. Remember that for HTTP every request is independent of another. In other words, a request doesn't know anything about other

previous, next, or simultaneous requests. A request cannot share data with order requests or access the details the backend responds for them.

However, you'll find scenarios where the server needs to correlate some requests. A good example is the cart functionality of an online shop. A user adds multiple items to their cart. To add an item to the cart, the client makes a request. To add a second item, the client makes another request. The server needs a way to know that the same client previously added an item to the same cart (figure C.3).

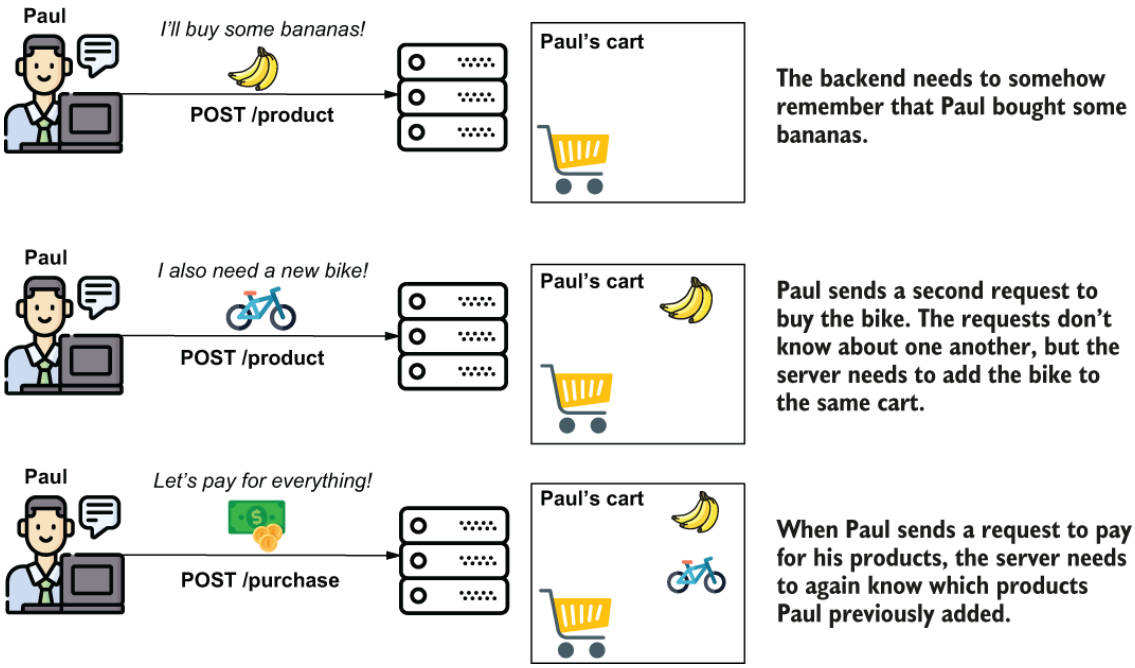

Figure C.3 For an online shop, the backend needs to identify the clients and remember the products they added to their carts. The HTTP requests are independent one from another, so the backend needs to find another way to remember the products added by each client.

One way to implement such behavior is using the HTTP session. The backend assigns a unique identifier named "session ID" to a client and then associates it with a place in the app's memory. Each request the client sends after being assigned the session ID needs to contain the session ID in a request header. This way, the backend app knows to associate the specific session requests (figure C.4).

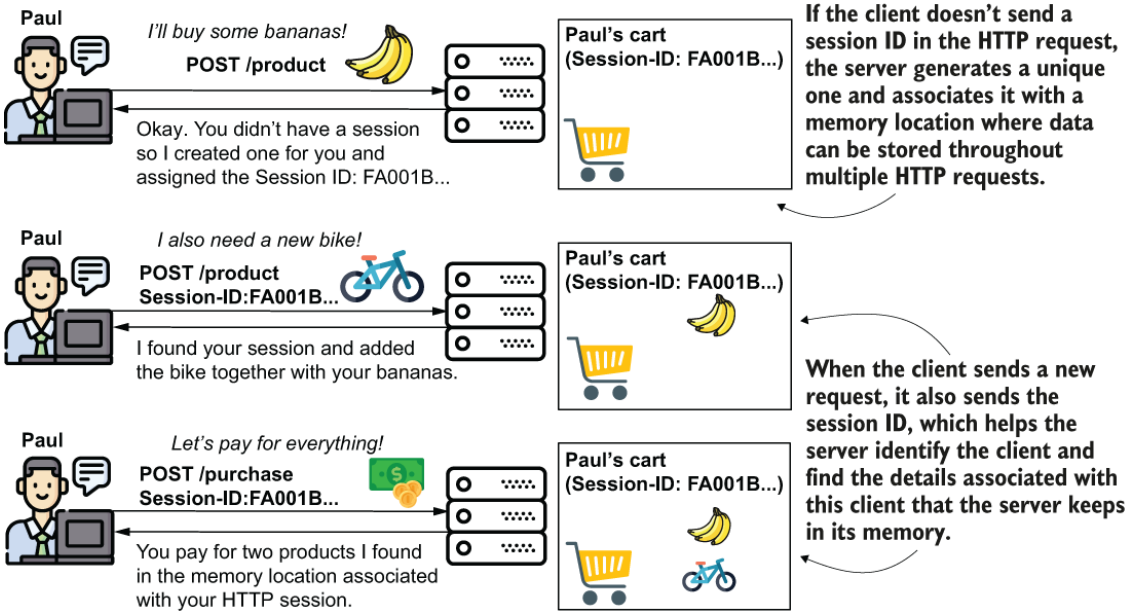

Figure C.4 The HTTP session mechanism. The server identifies the client with a unique session ID it generates. The client sends the session ID in the next requests, so the backend app knows which memory location it reserved earlier for the client.

The HTTP session usually ends after a time if the client doesn't send more requests. You can configure this time, usually both in the servlet container and the app. It shouldn't be more than maybe a few hours. If the session lives too long, the server will spend a lot of memory. For most apps, a session ends after less than one hour if the client doesn't send more requests.

If the client sends another request after the session ended, the server will start a new session for that client.