Reactive Programming

Overview

In this lab you'll explore the role of reactive programming in HTTP/2. HTTP/2 is a major evolution of the HTTP network protocol used on the World Wide Web, to support efficient communication in contemporary web applications.

> If you're new to HTTP/2, here's a quick summary of how it differs from HTTP 1…
>
> HTTP 1 is based on the classic request-response message exchange pattern. The client makes a request, and effectively waits for the whole response to be received. The client can make the request on a separate thread if it likes (to avoid blocking the client application), but that thread still has to wait for the whole response to be returned before it can process the response.
>
> HTTP/2 supports reactive streams to handle request and response bodies in a more piecemeal manner. The client can issue a request and provide a *subscriber* to receive chunks of response data a bit at a time. The client uses a *subscription* to throttle the amount of data pushed at it by the server.
>
> HTTP/2 is well suited to server-side events (SSE) and streaming data. Most modern browsers support HTTP/2 nowadays.

Support for HTTP/2 was added in Java version 11. This lab will introduce you to the Java API for HTTP/2, and show how to make web requests in 3 different ways:

- Via a blocking (synchronous) call, which eventually returns a response when the whole response has been completely received from the server.
- Via an asynchronous call, which returns a **`CompletableFuture`** immediately whilst the response is being accumulated. The **`CompletableFuture`** is fulfilled when the whole response has been completely received.
- Via a reactive call, which returns a **`Subscription`**. The server pushes as much data at the **`Subscription`** as the client asks it to, so that the client can receive data in piecemeal chunks rather than a monolithic complete result.

Note: The instructions in this lab doc are quite detailed, to make sure you understand the HTTP/2 API and get a good grasp on how its reactive model differs from synchronous / asynchronous communication. This will stand you in good stead for our discussions about reactive web apps (i.e. Spring WebFlux) and reactive data sources (i.e. R2DBC) in subsequent chapters.

IntelliJ projects

Starter project: **`student\student-reactive`**

Solution project: **`solutions\solution-reactive`**

Roadmap

There are 6 exercises in this lab, of which the last exercise is "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Reviewing the starter code
2. Creating an **`HttpClient`** object
3. Making a blocking (synchronous) HTTP/2 call
4. Making an asynchronous HTTP/2 call
5. Making a reactive HTTP/2 call
6. (If time permits) Additional suggestions

Exercise 1: Reviewing the starter code

In IntelliJ, locate the project named **`student-reactive`**. It's a Spring Boot application, and we've also included the *Lombok* dependency for logging and code generation purposes.

The purpose of the application will be to request a web page at a particular URL and search the response to see if it contains a particular search term. You'll implement 3 different ways to make the HTTP request, using the **`HttpClient`** class introduced in Java 11 (more on this shortly):

- Synchronously
- Asynchronously
- Reactively

In each case, the search operation will need to return a result containing the following info:

- The URL that was pinged
- The term that was searched for (a string)
- A Boolean flag indicating whether the term was found in the HTTP response body

With this in mind, we've defined a class called **`Result`** to hold this information. The class has two static methods to create a **`Result`** object in two different ways:

- "Normally", i.e., where the search ran to completion (either finding the term or not). The method writes an informational log message to indicate whether the term was found, and creates/returns a **`Result`** object with the details of the search.
- "Exceptionally", i.e., where the search operation threw an exception for some reason. The method writes an error log message to indicate the exception info, and creates/returns a **`Result`** object with the details of the search.

We've also defined an interface named **`PageSearcher`**, with a single method that takes a string URL and a string search term, and returns a **`Result`** object.

In Exercises 3, 4, and 5, you'll define three separate component classes that implement this interface in three different ways:

- In Exercise 3, you'll implement a **`PageSearcherSync`** class
- In Exercise 4, you'll implement a **`PageSearcherAsync`** class
- In Exercise 5, you'll implement a **`PageSearcherReactive`** class

Exercise 2: Creating an `HttpClient` object

The key Java ingredient in HTTP/2 is the **`java.net.http.HttpClient`** class, which is new in Java 11. This class provides far more flexibility in how you make HTTP requests, compared to the synchronous request/response pattern in HTTP 1.

Take a look at the **`HttpClient`** class on JavaDoc. Note that the constructor is *protected*, so you can't create an **`HttpClient`** object using **`new`** – you must use a *static* factory method instead (the use of static factory methods has become a popular technique in modern Java):

- **`newHttpClient()`** – creates an **`HttpClient`** object with default settings
- **`newBuilder()`** – returns a builder object, to build a custom **`HttpClient`** object

Once you've created an **`HttpClient`** object, it's immutable. Immutability has become a hot topic in functional programming recently – when an object is immutable, you know for sure it's safe to pass around and to use it freely on any thread.

Each of the three "page searcher" classes in this application will need an **`HttpClient`** object, in order to make the synchronous/asynchronous/reactive call. Given the fact that **`HttpClient`** is immutable, you may as well just create a single **`HttpClient`** instance (e.g., as a singleton Spring bean) so it can be injected into each of the three "page searcher" classes. To do this:

- In your project, locate the **`Application`** class.
- In Spring Boot, the application class is also a *configuration* class. This means you can define *bean methods* in the class, to create bean instances. Like so:

  ```
  @Bean
  public HttpClient httpClient() {
     return HttpClient.newHttpClient();
  }
  ```

  This method creates a Spring Boot bean of type **`HttpClient`**. The bean is a singleton by default – a single bean instance will be created at application startup, and it can be autowired into any component that needs it. Very cool indeed 😎.

Exercise 3: Making a blocking (synchronous) HTTP/2 call

Now that you've got the foundations of the application in place, it's time to define the first "page searcher" class. The simplest scenario is a blocking (synchronous) call, so this is what you'll do in this exercise. This will also give you some important insights into how the **`HttpClient`** class works, which will be handy for more advanced scenarios (i.e., asynchronous/reactive calls).

Define a class named **`PageSearcherSync`** as follows:

- Define it as a Spring Boot component class, i.e., a class annotated with **`@Component`**
- Implement the **`PageSearcher`** interface
- Autowire an **`HttpClient`** object

Implement the **`searchPageFor()`** method as follows:

- Create a **`java.net.http.HttpRequest`** object, to represent the HTTP request. The **`HttpRequest`** class is new in Java 11 – take a look on JavaDoc. Note that it too uses the builder pattern, so you can build up the HTTP request a step at a time. You need to create an **`HttpRequest`** object that represents a GET request to the specified URL.
- Now you're ready to make a blocking (synchronous) call to the server. To do this, call **`send()`** on the **`HttpClient`** object as follows.
    - The first parameter is the **`HttpRequest`** object you created just now.
    - The second parameter is an **`HttpResponse.BodyHandler<T>`**, which specifies a strategy for handling the HTTP response body when it is returned.

      The easiest way to create an **`HttpResponse.BodyHandler<T>`** is via the **`HttpResponse.BodyHandlers`** class – take a look on JavaDoc. It has various static methods that create an **`HttpResponse.BodyHandler<T>`** object to handle the body according to different strategies.

      In this example, call **`HttpResponse.BodyHandlers.ofString()`**. This method creates an **`HttpResponse.BodyHandler`** that consumes the entire HTTP response as a single large string.
- The **`send()`** method is synchronous. It blocks until the entire HTTP response has been returned, and you get back an **`HttpResponse<String>`** object. Call **`body()`** to get the HTTP response body as a large string, and then see if the string contains the search term you're looking for. When you've done all this, create a **`Result`** object containing the URL that was searched, the term that was searched for, and the boolean indicating if the term was found. Return the **`Result`** object from the method.
- Your method should also have a try-catch block surrounding all the code, in case anything goes wrong in the HTTP request. If so, create and return a **`Result`** object embodying the exception info.

Back in the main application class, add code to get a **`PageSearcherSync`** bean and call its **`searchPageFor()`** method with a URL for a very large web page and a suitable search term.

The method will return a **`Result`** object, so display this info on the console. Run the application and verify it works correctly.

Exercise 4: Making an asynchronous HTTP/2 call

In this exercise you'll use the **`HttpClient`** class to make an asynchronous call to the server, via the **`sendAsync()`** method. This method makes the call on a separate thread and doesn't block – it returns a **`CompletableFuture`** immediately, so that the main thread can do some other work in the meantime. When the HTTP response is ready, the **`CompletableFuture`** completes and you can provide some call-back code to be invoked on the separate thread.

To do all this, define a class named **`PageSearcherAsync`**. The class will be broadly similar to the **`PageSearcherSync`** class, except for the way you make the HTTP call and handle the response. Here are some hints:

- On the **`HttpClient`** object, call **`sendAsync()`** with the same parameters as before. **`sendAsync()`** returns a **`CompletableFuture<HttpResponse<String>>`** object. In other words, it returns a pending **`HttpResponse<String>`** result.
- Call **`thenApply()`** on the **`CompletableFuture<HttpResponse<String>>`** object, to perform the following actions on the response when it has arrived (note, if you need any help with this bit, feel free to take a sneak look at our solution code):
    - Extract the HTTP response body as a string.
    - See if the string contains the search string.
    - Create a **`Result`** object containing the overall result of the search (i.e. URL, search term, and boolean outcome).
- Also call **`exceptionally()`** to deal with any exception that might have occurred. Create a **`Result`** object embodying the exception info.

All of the above code will run *on a separate thread when the response has finally been returned*. The main thread of the application doesn't hang around waiting – to prove the point, do some other useful work on the main thread, and then when you've had enough of that, call **`get()`** on the **`CompletableFuture<HttpResponse<String>>`** object to wait for the response.

Back in the main application class, add code to get a **`PageSearcherAsync`** bean and test it in the same way as you did for the **`PageSearcherSync`** bean in the previous exercise.

Exercise 5: Making a reactive HTTP/2 call

In this exercise you'll use the **`HttpClient`** class to make a reactive call to the server. You'll still call **`sendAsync()`** to make the call asynchronously; what will be different is the type of **`HttpResponse.BodyHandler`** you'll provide to handle the response body…

- Previously you've called **`BodyHandlers.ofString()`**, which creates an **`HttpResponse.BodyHandler<String>`**. This type of body-handler waits for the entire HTTP response to be returned, then loads it all into one giant string. Not very reactive, is it? What if the response is enormous, or actually unbounded…?
- In this exercise you'll call **`BodyHandlers.fromLineSubscriber()`**. You'll pass a **`Flow.Subscriber`** object as a parameter, and that object will subscribe to the response reactively.

So, the first step is to define a subscriber class. The subscriber class will request one line at a time from the server, and will check that line to see if it contains the search term. If it does, the subscriber will stop subscribing and indicate a "found" outcome. Here are some hints on how to implement all this behavior:

- Define a class named **`StringMatchSubscriber`** (for example), which implements the **`Flow.Subscriber<String>`** interface.
- Ultimately the subscriber class should yield a **`Result`** object, encapsulating all the information about the search. Therefore, your class needs to remember the URL that was pinged and the term that was sought. The client code will want to interrogate the subscriber for the **`Result`** object, but it's clearly not ready initially – therefore your subscriber should have a **`CompletableFuture<Result>`** instance variable, where it can stick the result when it's finally available.
- The class must implement all the normal callback methods for a subscriber:
    - **`onSubscribe()`** – This method should stash the **`Flow.Subscription`** object in an instance variable, it'll be needed later. The method should also request 1 line of text from the server (i.e. publisher).
    - **`onNext()`** – This method receives the next line of text from the server. The method should see if the line contains the search term.

      If the line contains the search term, then we're done. Create a **`Result`** object with a "found" outcome and stick it into the **`CompletableFuture<Result>`** bucket (so clients can see it).

      If the line doesn't contain the search term, request another line from the server.
    - **`onComplete()`** – If this method is invoked, it means the server stopped sending us data. In other words, it got all the way to the end of the web page without us finding the search term. In this case, create a **`Result`** object with a "not found" outcome and stick it into the **`CompletableFuture<Result>`** bucket.
    - **`onError()`** – If this method is invoked, it means an exception occurred during the subscription. Create a **`Result`** object to embody the exception info.

Now you're ready to define a reactive "page searcher" class. Give the class a name such as **`PageSearcherReactive`**. The class will be quite similar to **`PageSearcherAsync`**, but note the following differences:

- You'll need to create a **`StringMatchSubscriber`** object, to subscribe to the reactive flow of data from the server (publisher).
- When you call the **`HttpClient`**'s **`sendAsync()`**, for the second parameter, use the **`BodyHandlers.fromLineSubscriber()`** method. Pass in your subscriber, so that it will receive the data published by the server.
- All of the text-searching logic now lives in your subscriber class, so there's no need for any of that code in the **`PageSearcherReactive`** class. So maybe you can do some other useful work for a while, and then get the **`Result`** from the subscription when it's available.
- Note that all this code should be in a try/catch block to handle any exceptions. In this case, create a Result **`object`** to embody the exception info.

Back in the main application class, add code to get a **`PageSearcherReactive`** bean and test it in the same way as you did in the last two exercises.

Exercise 6 (If time permits): Additional suggestions

Take a look through the chapter notes and see if there are any techniques you can sprinkle into your application to explore how they work.