Iurii Mednikov

Friendly Webflux: A Practical Guide to Reactive Applications Development with Spring Webflux

1st edition

Published by Iurii Mednikov (Yuri Mednikov) at Leanpub

Copyright 2021 Iurii Mednikov (Yuri Mednikov)

1st edition

Contacts

For all questions please contact the author directly with mednikov.iurii@gmail.com

Source code listings

This book is supplied with source code listings, that readers can download from this github repository.

Table of contents

Foreword

This book originated from my personal notes and case studies, where I documented my own solutions for problems, that I faced during my experience with Spring Webflux. I could consider myself as an evangelist of reactive programming paradigm, and I worked with Eclipse Vertx for many years (for instance, my another book is devoted to that framework). Certainly, when Spring Webflux came to the market, I was very curious to "have my hands dirty" and to try it in various applications. For a long time, I had a limited exposure to the Spring ecosystem, mainly due to its "heavy weight" and sticked to more unopinionated technologies. I was surprised to find out, that Spring Webflux did not only bring asynchronous programming model to the Spring universe, but it is also a fast and lightweight solution to serve as a foundation of modern scalable web systems, such as microservices.

I believe that a good piece of technical writing should unify both theoretical concepts and practical experience. With this book, I made an attempt to combine notes on reactive programming with Project Reactor (chapter 1), application architecture and design (chapters 3, 4, 13) with an overview of most crucial and used components of Spring Webflux. The book also includes instructions on software testing, authentication and authorization, working with a data layer. Finally, as a full stack software engineer myself, I feel, that it is important to include notes about client side development and how these apps can interact with systems, built with Spring Webflux. Frontend examples are created with Vue.js, however they are not tied to the particular framework and can be translated to React.js or Svelte.js.

The book is targeted for intermediate Java programmers, yet a previous experience with Spring framework is not required.

Acknowledgments

A good tradition of authors is to acknowledge people whose in a certain way influenced a process of book writing. And I would not be an exception here. First of all, I would like to thank my parents as without their support and patience I would not become a software engineer. I would like to thank all readers of my books and my writings in general for their valuable feedback. Being a member of Java community is a big honor.

Chapter 1. An introduction to reactive programming with Project Reactor

The goal of this chapter is to provide a quick intro to readers on most important subjects, related to the reactive software development using Spring Boot, Spring Webflux and Project Reactor. These topics include a usage of reactive streams and reactive streams provided by the Project Reactor framework (Mono and Webflux).

Reactive Streams

The idea of streams gains its popularity with rise of parallel computations. It is based on principle of lazy evaluation. From a technical point of view, means, that actual evaluation is suspended until the result is needed. The laziness of operations states, that the actual computation on the source data is performed after the final operation will be invoked, and source elements are consumed only as needed.

As it was already mentioned, Spring Webflux is based on the Project Reactor - an open source framework from Pivotal, which has an objective to unifies reactive API in Java. Project Reactor is built around the idea of reactive streams. There are several characteristics of reactive streams:

- A reactive stream should be non-blocking,
- It should be a stream of data,
- It should work asynchronously,
- And it should be able to handle back pressure

So, let return to the example with database access. When we use JDBC-based solutions, requests block the thread until response. That is why this model is called blocked. Compare to it, non-blocking execution relies on async computations. Such model is built on the usage of an observer design pattern, which specifies communication between objects: observable and observers:

- Observable is a component, that has a sequence of observers and notifies them on each change of its state
- Observer is a component, that subscrbies on updates of an observable

These concepts can be named in a different manner, which is accepted for concrete reactive implementation. In Project Reactor we operate publishers and consumers. In this way, publisher is an observable, that sends data to consumers (observers).

Project Reactor types

In this section we will observe two core reactive types, that are brought by the Project Reactor and are used as main building blocks by Spring Webflux - `Mono` and `Flux`. Both of these classes implement the *publisher* pattern, that is familiar by RxJava and other reactive libraries. Under this idea, event handlers are subscribed to the publisher component and listen for its updates in the async way.

This way, Project Reactor is not the only approach to implement reactive architecture for the JVM. You can also know RxJava and Vertx. The last one has its own way (you can get more about the Vertx framework in my book "Principles of Vertx"). In this chapter we will cover following aspects:

- Creating Mono and Flux instances from Java objects and from async providers (such as Runnable or CompletableFuture classes)
- Use cases of Mono and Flux classes
- Mapping and the difference between `map()` and `flatMap()` methods
- How to subscribe on Mono and Flux events
- Build pipelines with Mono and Flux
- How to merge several Mono/Flux instances to a single result

- Blocking vs non-blocking
- Error handling

Create Mono instances

The `Mono` class implements a reactive publisher that can emit zero or one item. There are several characteristics of Mono, that we need to keep in mind:

- Mono can emit zero or one item at most using `onNext` signal
- Mono terminates with the `onComplete` signal
- Mono can emit at most one error signal with the `onError` signal

That is how it described in theory. But what does this mean for developers? Consider the Mono class as an abstraction that allows us to subscribe to *some event* (we don't know an output yet, because it is computed asynchronously). Let say, that we have a repository (if you are not yet familiar with Spring Data, don't worry - we will cover them later on). In the "classical" Spring, repository exposed an access to *data entities*, like following:

(Code snippet 1-1)

```
Optional<Invoice> getInvoiceById (UUID invoiceId);

List<Invoice> getAllInvoices (UUID userId);

// etc.
```

Nothing wrong with this, yet as we utilize blocking data access clients, like JDBC, the following code blocks (simply speaking, makes you wait) the app thread until getting results from database. To overcome that, the reactive version of Spring Data exposes *reactive publishers* instead of entities:

(Code snippet 1-2)

```
Mono<Invoice> getInvoiceById (UUID invoiceId);

Flux<Invoice> getAllInvoices (UUID userId);

// etc
```

Again, if you are not familiar well with Spring repositories, don't worry - we will cover them later on. For now, I just want to show you, that from the developer perspective, both Mono and Flux components are simply abstractions and do not contain data directly. Consider them as containers, that permits to subscribe on some event, that *possible could produce result in the future*. Or could fail. The last situation is called *a failed Mono*.

So, that was a small theoretical introduction of what is Mono. Let know go to practive and see how to create Mono instances. From the technical point of view, we can go for two main approaches:

- Create Mono instance directly from plain Java objects, like data entities, errors or create empty Mono
- Create Mono from reactive events (like CompletableFuture or Runnable). In this case we can wrap these Java components in a compatible with Project Reactor containers.

Create Mono from plain Java objects

The simplest way to obtain the new Mono instance is to use `just()` method and to create the Mono container with the entity inside. Technicaly speaking, the emited object is captured in the instantiation time. We can get then the item using the `block()` method. NB, this is just for educational purposes, in a real life don't call the `block()` and we will see later why.

Take a look on the following example:

(Code snippet 1-3)

```
Mono<Student> mono = Mono.just(new Student("Aneta", 3.9));

Student result = mono.block();

assertThat(result.getName()).isEqualTo("Aneta");
```

As I have said, this is the most straightforward approach. Of course, we can add some protection of potentially absent value. The Mono does not check that the item is null, so we can call `just(null)`, like it is shown below:

(Code snippet 1-4)

```
Mono<Student> mono = Mono.just(null);

Student result = mono.block();
```

In such way, we can utilize the `justOrEmpty()` method, that could accept a nullable entity as an argument. In this way, the `Mono.block()` method call will return a *null* value instead of raising the `NullPointerException`. So, this code will perfectly complie without any error:

(Code snippet 1-5)

```
Mono<Student> mono = Mono.justOrEmpty(null);
```

```
Student result = mono.block();
```

Another path is to use `blockOptional()` method, which will return an `Optional` object instead of the direct entity. This is an acceptable to write null-safe code in Java:

(Code snippet 1-6)

```
Mono<Student> mono = Mono.justOrEmpty(null);

Optional<Student> optional = mono.blockOptional();

assertThat(optional).isEmpty();
```

The last thing I want to highlight in this subsection is how to create a Mono container that explicitly does not hold any value. That task can be achieved using the `empty()` static method. This type of Mono completes without emitting any item.

(Code snippet 1-7)

```
Mono<Student> mono = Mono.empty();
```

I want to emphasize here, that while we did this introduction, it means to be for educational purposes only and to bring you a better understanding what is the Mono class. That means, I advice you against using `block()` methods. I also discourage you to use reactive publishers as a containers of the data, received synchronously. The smart approach is to use Mono to wrap up Java asynchronous objects to use them with Project Reactor and Webflux as we will see later.

Create a failed Mono

We have already mentioned, that the failed Mono stands for the Mono that terminates with the specified error immediately after being subscribed to. That means, that when we create an explicitly failed Mono instance and try to call `block()` method it will terminate with an error, not with a null result. Take a look on the following example below:

(Code snippet 1-8)

```
Mono<String> mono = Mono.error(new NoSuchFieldException());

String name = mono.block();

System.out.println(name);
```

The exection of this code will terminates with an error. We can explicitly specify a *default* (or fallback) return for the Mono, likewise we do that using the `Optional.orElse()`. That can be done with the `onErrorReturn()` function.

(Code snippet 1-9)

```
Mono<String> mono = Mono.error(new NoSuchFieldException());

String name = mono.onErrorReturn("Barbora").block();

assertThat(name).isEqualTo("Barbora");
```

So, you can observe, that the failed Mono **does not emit a value**, rather it **terminates with an error**. We have seen it, when tried to subscribe with the `block()` for the failed Mono. I found this important to emphasize, because I have seen some "experts", that tried to teach that the failed Mono returns *an error type*. No, it does not.

Use Mono as a wrapper for Java asynchronous types

We did some introduction on how to create Mono instances using plain Java objects, how to instanciate empty and failed Mono objects. This is very useful for the purpose of learning, but I would like to discourage you to use these methods unless you really have just a single plain Java object. I emphasize this, because I have seen a lot of developers (and, I would not lie, I was also like them) who do following:

- Compute the result in the blocking way (for example, calling the API endpoint with the HTTP client)
- Then wrap up the result using the `Mono` or `Flux` and return that to the Webflux

Please, **do not do that!** It may seem naive, but the Project Reactor supplies us with a range of methods, that allow to wrap up "native" Java concurrent (async) events as Mono and Flux. For the purposes of this book I call them "from-methods", because their signatures start with "from..." word. Basically, you can create Mono instance from following Java async types:

- From `java.util.concurrent.Callable` with the static factory method `Mono.fromCallable()`
- From `java.util.concurrent.Future` with the static factory method `Mono.fromFuture()`
- From the runnable implementation with the static factory method `Mono.fromRunnable()`

Let me illustrate this point using the example with HTTP call. For that you need at least JDK 11, because I would like to use the `java.net.http.HttpClient`, that was introduced in the aforesaid release. This class permits to perform HTTP calls both in a synchronous way and in an asynchronous way (`sendAsync` method). Let implement a simple use case of performing the GET request asynchronously and then to wrap it as a Mono instance. Due to the fact, the `sendAsync()` method returns a `CompletableFuture` object we can then use the static method `Mono.fromFuture()`.

(Code snippet 1-10)

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()

         .GET()

         .uri(URI.create
("https://jsonplaceholder.typicode.com/todos/1"))

                .build();

CompletableFuture<String> response = client

                .sendAsync(request,
HttpResponse.BodyHandlers.ofString())

                .thenApplyAsync(result -> result.body());

Mono<String> mono = Mono.fromFuture(response);
```

```
String body = mono.block();

System.out.println(body);
```

The execution of this code will print the todo with ID 1 from the JSONPlaceholder API. Here we perform following steps:

- Create a new HttpClient instance
- Define a new GET request
- Create a new async response as a `CompletableFuture`. Note, that by default the `sendAsync` returns `CompletableFuture<HttpResponse>`, where the `HttpResponse` object contains all data from the response, including status code, errors etc. As we need only body we map the result to get only body of the response
- Create a new Mono instance from the future object
- Subscribe to the Mono's computation result
- Print an output

You can note here, that this approach is better than getting the HTTP response in a blocking way and then to wrap the String object to Mono with the `just()` method.

Create Flux instances

An another reactive publisher is `Flux`. This component can emit from zero to many elements. Please note that "many". If you intend to obtain 0 or 1 element you should stick with Mono. From the other side, if you know, that you will emit more than 1 element, use Flux. To make things simplier, consider Flux as a reactive publisher that emit sequence of elements.

In this subsection we will observe approaches to create Flux instances. As Mono, the Flux class permits to create an instance with the `just()` static method. You can explicitly create Flux instances from `Iterable` implementations (collections) and Streams. From the other side, you can not create Flux instances directly from Java async types. For that you need to:

- Create Mono instance from the Java async type (Runnable, Future, Callable) as we have observed in the previous subsection
- Convert Mono to Flux with `Mono.flux()`

Create Flux from iterables

The simplest way is to create new Flux instance from the Java `Iterable` implementation (e.g. from collection). However, please remember that the Flux object does not emit the `Iterable` itself (yet you can map to the `Iterable` return). The following statement applies:

- `Flux<T>` allows to subscribe for a sequence of `T` elements where each element is provided as it is ready
- `Mono<Iterable<T>>` emits the `Iterable` of `T` elements that is provided as whole

Let create a new Flux from the `java.util.List`, that contains a list of student records:

(Code snippet 1-11)

```
List<Student> students = List.of(

                new Student("Aneta", 3.8),

                new Student("Barbora", 3.5),

                new Student("Carolina", 3.4),

                new Student("Denisa", 2.9),
                new Student("Eva", 2.9)

        );

Flux<Student> flux = Flux.fromIterable(students);
```

And here comes the mentioned difference! Please be careful, the Flux abstracts not an iterable itself, but the *sequence* of elements. That means we can't go and directly use `block()` and get the `List` result. Rather, we could (have to) subscribe to elements of a *sequence* as following:

- `blockFirst` to subscribe for the 1st element of a sequence
- `blockLast` to subscribe for the last element of a sequence

Take a look on the following code example:

(Code snippet 1-12)

```
List<Student> students = List.of(

                new Student("Aneta", 3.8),

                new Student("Barbora", 3.5),

                new Student("Carolina", 3.4),

                new Student("Denisa", 2.9),

                new Student("Eva", 2.9)

        );
```

```
Flux<Student> flux = Flux.fromIterable(students);

Student first = flux.blockFirst();

Student last = flux.blockLast();

assertThat(first.getName()).isEqualTo("Aneta");

assertThat(last.getName()).isEqualTo("Eva");
```

Separately, I would like to review the method `elementAt(n)`. This function permits to subscribe for the specific element of a sequence, using following conditions:

- The element index `n` includes 0 and excludes the length of sequence (like with arrays).
- The method throws an exception if the element's index is out of bounds
- The method returns not an element directly but `Mono` to subscribe for the element

Here is an example:

(Code snippet 1-13)

```
List<Student> students = List.of(

        new Student("Aneta", 3.8),

        new Student("Barbora", 3.5),

        new Student("Carolina", 3.4),
        new Student("Denisa", 2.9),

        new Student("Eva", 2.9)

);

Flux<Student> flux = Flux.fromIterable(students);

Mono<Student> mono = flux.elementAt(2);

Student student = mono.block();

assertThat(student.getName()).isEqualTo("Carolina");
```

Finally, I would like to mention, that you can transform [lazily] the sequence into the iterable with `toIterable()` method. However, this method is blocking and I advice you to avoid it.

(Code snippet X-14)

```
List<Student> students = List.of(

        new Student("Aneta", 3.8),

        new Student("Barbora", 3.5),

        new Student("Carolina", 3.4),

        new Student("Denisa", 2.9),
        new Student("Eva", 2.9)

);

Flux<Student> flux = Flux.fromIterable(students);

Iterable<Student> iterable = flux.toIterable();

assertThat(iterable).hasSameElementsAs(students);
```

Due to the fact, that Flux emits to many elements and abstacts a sequence, it may seem logical to instanciate a Flux from the `Iterable`.

Create Flux from Streams

Besides iterables, you could use Java streams as a source for Flux instances. In Java, *stream* defines a sequence of elements supporting sequential and parallel aggregate operations. The idea of streams gains its popularity with rise of parallel computations. It is based on principle of lazy evaluation. From a technical point of view, this means, that actual evaluation is suspended until the result is needed. The laziness of operations states, that the actual computation on the source data is performed after the final operation will be invoked, and source elements are consumed only as needed.

In the following example we will use a stream, that contains even numbers as a source for the Flux instance:

(Code snippet 1-15)

```
List<Integer> numbers =
List.of(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16);
```

```
Stream<Integer> stream = numbers.stream().filter(n -> n %2 == 0);

Flux<Integer> flux = Flux.fromStream(stream);

int first = flux.blockFirst();

assertThat(first).isEqualTo(2);
```

However, here you need to consider the important aspect of streams - they are meant to be consumed once and after consumption, the stream is closed. That means, that the execution of the following code snippet would lead to an error:

(Code snippet 1-16)

```
List<Integer> numbers =
List.of(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16);

Stream<Integer> stream = numbers.stream().filter(n -> n %2 == 0);

Flux<Integer> flux = Flux.fromStream(stream);

int first = flux.blockFirst();

int last = flux.blockLast();

//...assertions
```

You can find out, that this exposes an `IllegalStateException`, because the stream was already cosumed and closed. We can overcome this issue by converting Flux to the `Iterable`:

(Code snippet 1-17)

```
List<Integer> numbers =
List.of(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16);

Stream<Integer> stream = numbers.stream().filter(n -> n %2 == 0);

Flux<Integer> flux = Flux.fromStream(stream);

Iterable<Integer> iterable = flux.toIterable();
```

That is how you can create Flux instances using Java objects, such as iterables and streams. However, I discourage you from this practice. While we did this for educational purposes, I strongly suggest you to use reactive types `Mono` and `Flux` to wrap up Java **async** types and make them compatible with Project Reactor. You should avoid the practice of using these types as wrappers for results of synchronous computations.

Mapping patterns

In my practice, I found that mapping operations are by far most frequent, when you work with Project Reactor and Webflux. By mapping we understand a conversion of the Mono/Flux item to a different type. That is similar of how we do with Java streams. Let first take a look on how it is done with streams. Imagine, that we have a list of student entities, which contain data about student's name and a GPA score. We want to convert each student entity to `Double` value, that represents a GPA score for further calculations (find average per group, for instance).

(Code snippet 1-18)

```
List<Student> students = List.of(

            new Student("Aneta", 3.8),

            new Student("Barbora", 3.5),

            new Student("Carolina", 3.4),

            new Student("Denisa", 2.9),

            new Student("Eva", 2.9)

        );

Double average = students.stream().map(s ->
s.getGpa()).average().get();
```

So, here we converted an initial POJO entity (`Student`) into the `Double` value and then performed computations. With Project Reactor types we utilize same principles. Let me return to the preceding example with a HTTP request. You remember, that Java uses an `HttpResponse` object as a return of HTTP call. That objects contains different data about the actual response result, such as:

- Body
- Status code
- Headers
- Request data
- SSL session data
- URI

- Protocol version

We don't need all of that information. We want to get only a String, which contains body of the response. You remember that we used `thenApply` method to map the result as string. We can rewrite that example and use `Mono.map()` instead.

(Code snippet 1-19)

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()

.GET().uri(URI.create("https://jsonplaceholder.typicode.com/todos/1"))

.build();

CompletableFuture<HttpResponse<String>> response = client

                .sendAsync(request,
HttpResponse.BodyHandlers.ofString());

Mono<HttpResponse<String>> mono = Mono.fromFuture(response);

Mono<String> mapped = mono.map(r -> r.body());

String body = mapped.block();

System.out.println(body);
```

The result of an execution is same, but in this case we use Project Reactor mapping to get a Mono, that emits a body value. Hope, you got an idea.

Now, let move to another question - Project Reactor exposes two mapping methods `map()` and `flatMap()`. I have seen many developers that misunderstand the difference, therefore I think it is crucial to mark it here.

- `map()` - transforms the item emitted by the Mono by applying a **synchronous** function to it
- `flatMap()` - transforms the item emitted by the Mono **asynchronously**.

From the technical point of view, reactive types (Mono, Flux) serve as abstractions or containers, so mapping functions transform the underlaying item (likewise we did with streams). The difference is that the `map()` transforms the item directly by an application of a sync function, while `flatMap()` transforms the item, wrapped as an another `Mono` and therefore does it asynchronously. Let me show this with the following example:

(Code snippet 1-20)

```
Mono<Student> studentMono = Mono.just(new Student("Marketa", 3.5));

Mono<String> nameMap = studentMono.map(student -> student.getName());

Mono<String> nameFlatMap = studentMono.flatMap(student ->
Mono.just(student.getName()));
```

You could note, that in the first case (`map()`) the mapper function returns an entity object directly. In the second situation (`flatMap()`) the mapper returns another Mono. Because, the `flatMap()` meant to return the result, computed in an async way, it is better to use it to map the result of another async operation.

Subscribe for events

Both reactive types that we review in this section implement a publisher pattern. This design pattern also sometime is called an observer pattern. Let me remind you its definition: it is a behavioral architectural pattern which allows to create a subscription mechanism to notify multiple objects (subscrbiers or observers) about any changes that happen to the publsher they are observing.

You remember, that I have already asked you to avoid the `block()` method and here is a reason why. This function is meant to be a blocking subscription, in other words we block the main thread until we get a result (item or error). But reactive programming is built on an idea of async computations. So, we *subscribe* for future results. In this way, reactive types serve as abstractions that provide an access to that future results. To do that we register callbacks.

Types of events depend on concrete publisher type (Mono or Flux), but can be groupped into these groups:

- Events triggered on successful result: `Mono.doOnSuccess`, `Flux.doOnComplete`. The successful result can emit either empty (null) item or a single (in case of Mono) or many items (in case of Flux)
- Events triggered on error result: `Mono.doOnError`, `Flux.doOnError`.
- Events triggered on any completing, despite the result is successful or failed: `Mono.doOnTerminate`, `Flux.doOnTerminate`
- Events triggered when a publisher receives *any* request - `Mono.doOnRequest`, `Flux.doOnRequest`
- Events triggered on cancellation of the publisher: `Mono.doOnCancel`, `Flux.doOnCancel`
- Events triggered anyway, no matter if the publisher was completed, failed or cancelled.

These callbacks accept functions as arguments, which define behavior. We will see their power once we will move to practical cases of Spring Webflux. For example, you can register different behavior on publisher that emit a result of a business service operation and therefore provide an output for success, error etc.

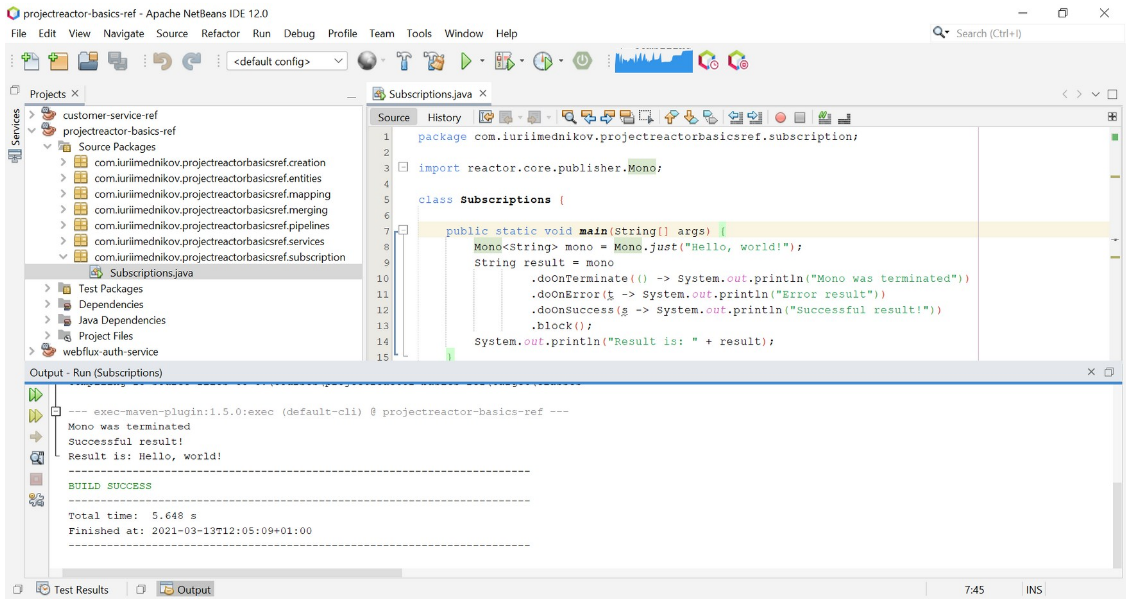
But let see how we can register callbacks for publisher events. Take a look on the code snippet below:

(Code snippet 1-21)

```
Mono<String> mono = Mono.just("Hello, world!");

String result = mono

        .doOnTerminate(() -> System.out.println("Mono was 
terminated!"))

        .doOnError(t -> System.out.println("Error result!"))

        .doOnSuccess(s -> System.out.println("Successful result!"))

        .block();

System.out.println("Message is: " + result);
```

This seems to be pretty artificial! However, the purpose is to illustrate the principle. Because this code is meant to be completed successfully, the result will look like this:

(Image 1-1)



```
package com.iuriimednikov.projectreactorbasicsref.subscription;

import reactor.core.publisher.Mono;

class Subscriptions {

    public static void main(String[] args) {
        Mono<String> mono = Mono.just("Hello, world!");
        String result = mono
                .doOnTerminate(() -> System.out.println("Mono was terminated"))
                .doOnError(t -> System.out.println("Error result"))
                .doOnSuccess(s -> System.out.println("Successful result!"))
                .block();
        System.out.println("Result is: " + result);
```

```
--- exec-maven-plugin:1.5.0:exec (default-cli) @ projectreactor-basics-ref ---
Mono was terminated
Successful result!
Result is: Hello, world!
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  5.648 s
Finished at: 2021-03-13T12:05:09+01:00
------------------------------------------------------------------------
```

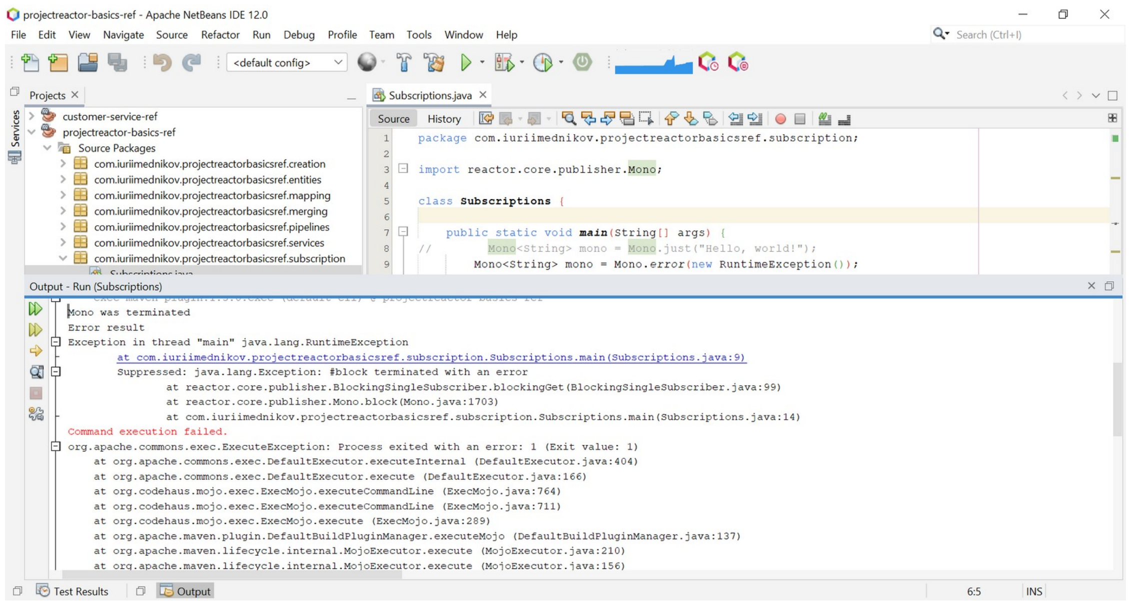
You can observe, that both `doOnSuccess` and `doOnTerminate` callbacks were triggered. Let modify the code to make the failed Mono:

(Code snippet 1-22)

```
Mono<String> mono = Mono.error(new RuntimeException());

String result = mono

        .doOnTerminate(() -> System.out.println("Mono was 
terminated!"))

        .doOnError(t -> System.out.println("Error result!"))

        .doOnSuccess(s -> System.out.println("Successful result!"))

        .block();

System.out.println("Message is: " + result);
```

Once executed, the code will end with an exception.

(Image 1-2)



```java
package com.iuriimednikov.projectreactorbasicsref.subscription;

import reactor.core.publisher.Mono;

class Subscriptions {

    public static void main(String[] args) {
//          Mono<String> mono = Mono.just("Hello, world!");
        Mono<String> mono = Mono.error(new RuntimeException());
```

```
Mono was terminated
Error result
Exception in thread "main" java.lang.RuntimeException
        at com.iuriimednikov.projectreactorbasicsref.subscription.Subscriptions.main(Subscriptions.java:9)
        Suppressed: java.lang.Exception: #block terminated with an error
                at reactor.core.publisher.BlockingSingleSubscriber.blockingGet(BlockingSingleSubscriber.java:99)
                at reactor.core.publisher.Mono.block(Mono.java:1703)
                at com.iuriimednikov.projectreactorbasicsref.subscription.Subscriptions.main(Subscriptions.java:14)
Command execution failed.
org.apache.commons.exec.ExecuteException: Process exited with an error: 1 (Exit value: 1)
    at org.apache.commons.exec.DefaultExecutor.executeInternal (DefaultExecutor.java:404)
    at org.apache.commons.exec.DefaultExecutor.execute (DefaultExecutor.java:166)
    at org.codehaus.mojo.exec.ExecMojo.executeCommandLine (ExecMojo.java:764)
    at org.codehaus.mojo.exec.ExecMojo.executeCommandLine (ExecMojo.java:711)
    at org.codehaus.mojo.exec.ExecMojo.execute (ExecMojo.java:289)
    at org.apache.maven.plugin.DefaultBuildPluginManager.executeMojo (DefaultBuildPluginManager.java:137)
    at org.apache.maven.lifecycle.internal.MojoExecutor.execute (MojoExecutor.java:210)
    at org.apache.maven.lifecycle.internal.MojoExecutor.execute (MojoExecutor.java:156)
```

Yet, in this situation are called `doOnError` and `doOnError` events. In next sections we will see practical examples of event subscriptions in Spring Webflux.

Error handling

Before we will go to next things, I want to stop here and talk a bit about such crucial topic like an error handling. We will need to understand it before we will move to practical stuff and I think it requires a separate section on it. Remember we talked about error callbacks, failed and empty Mono/Flux instances. Let me show you a common misunderstanding about that.

Imagine we have a service, that encapsulates a business logic to deal with students. Let name it as the `StudentService` class:

(Code snippet 1-23)

```
public class StudentService {

    public Mono<Student> findStudentByName(String name) {

    //..

    }

    public Flux<Student> findAllStudents(){

    //..

    }

}
```

Let write some code, that uses the service. For example, we look for a particular student. We register an error callback to deal with an error outcome and we provide a default values with `switchIfEmpty` and `onErrorReturn`:

(Code snippet 1-24)

```
StudentService service = new StudentService();

Mono<Double> result = service.findStudentByName("Aneta")

        .flatMap(s -> Mono.just(s.getGpa()))

        .doOnError(err -> System.out.println("Error occured!"))
        .onErrorReturn(5.0)

        .switchIfEmpty(Mono.just(4.0));

Double value = result.block();

System.out.println(value);
```

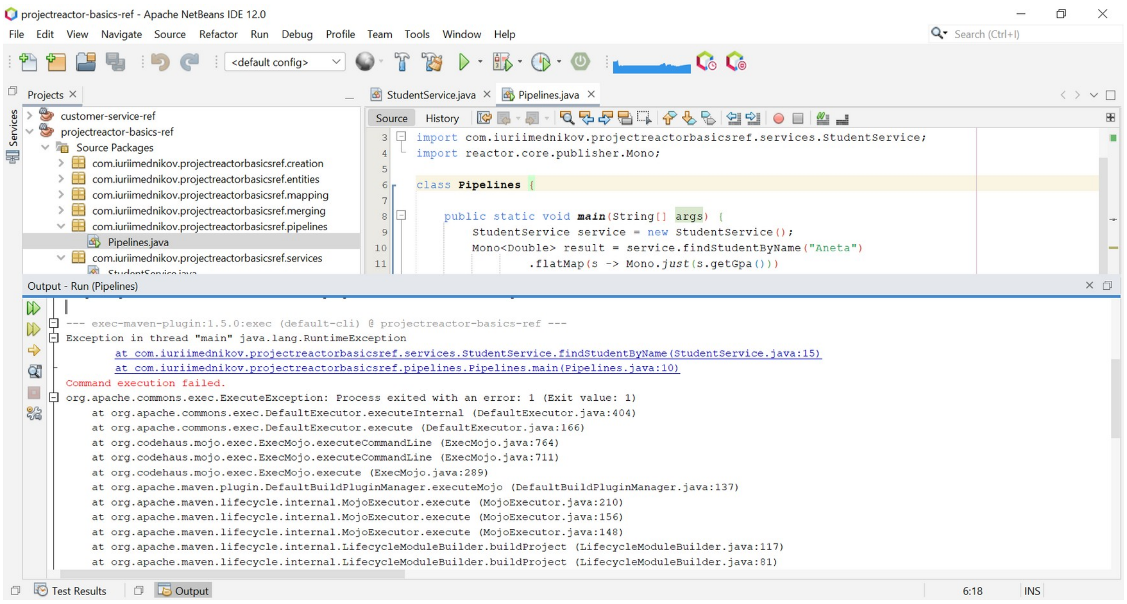
What happens if we will raise an exception *directly*, like that:

(Code snippet 1-25)

```
public Mono<Student> findStudentByName(String name){

       throw new RuntimeException();

}
```

An attempt of an execution of this code will result into an error - however, subscriptions will not be triggered:

(Image 1-3)



```java
import com.iuriimednikov.projectreactorbasicsref.services.StudentService;
import reactor.core.publisher.Mono;

class Pipelines {

    public static void main(String[] args) {
        StudentService service = new StudentService();
        Mono<Double> result = service.findStudentByName("Aneta")
                .flatMap(s -> Mono.just(s.getGpa()))
```

```
--- exec-maven-plugin:1.5.0:exec (default-cli) @ projectreactor-basics-ref ---
Exception in thread "main" java.lang.RuntimeException
        at com.iuriimednikov.projectreactorbasicsref.services.StudentService.findStudentByName(StudentService.java:15)
        at com.iuriimednikov.projectreactorbasicsref.pipelines.Pipelines.main(Pipelines.java:10)
Command execution failed.
org.apache.commons.exec.ExecuteException: Process exited with an error: 1 (Exit value: 1)
    at org.apache.commons.exec.DefaultExecutor.executeInternal (DefaultExecutor.java:404)
    at org.apache.commons.exec.DefaultExecutor.execute (DefaultExecutor.java:166)
    at org.codehaus.mojo.exec.ExecMojo.executeCommandLine (ExecMojo.java:764)
    at org.codehaus.mojo.exec.ExecMojo.executeCommandLine (ExecMojo.java:711)
    at org.codehaus.mojo.exec.ExecMojo.execute (ExecMojo.java:289)
    at org.apache.maven.plugin.DefaultBuildPluginManager.executeMojo (DefaultBuildPluginManager.java:137)
    at org.apache.maven.lifecycle.internal.MojoExecutor.execute (MojoExecutor.java:210)
    at org.apache.maven.lifecycle.internal.MojoExecutor.execute (MojoExecutor.java:156)
    at org.apache.maven.lifecycle.internal.MojoExecutor.execute (MojoExecutor.java:148)
    at org.apache.maven.lifecycle.internal.LifecycleModuleBuilder.buildProject (LifecycleModuleBuilder.java:117)
    at org.apache.maven.lifecycle.internal.LifecycleModuleBuilder.buildProject (LifecycleModuleBuilder.java:81)
```

"But why?" - you may ask. We have a raised exception, but neither `doOnError` callback was triggered neither the default value `onErrorReturn` was provided. This is a frequent mistake. Let distinguish things.

From a technical point of view, the `findStudentByName` method raises an ordinary Java exception. It breakes a reactive flow and propogates outside of the Mono execution. We can prove that by wrapping it with a normal `try...catch` block:

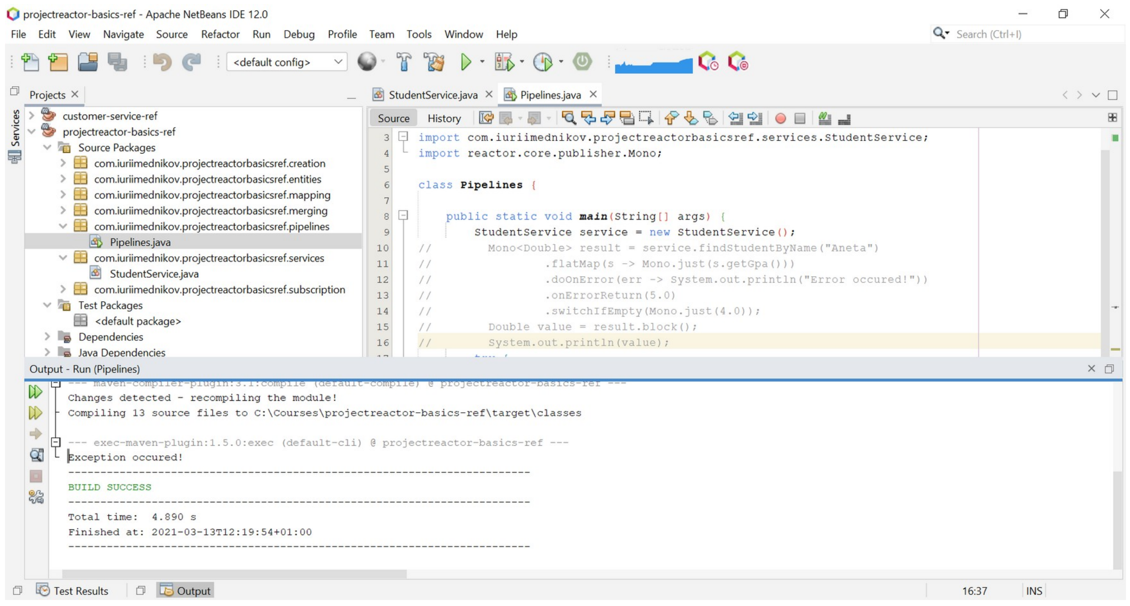
(Code snippet 1-26)

```java
try {

Mono<Double> result = service.findStudentByName("Aneta")

            .flatMap(s -> Mono.just(s.getGpa()))

            .doOnError(err -> System.out.println("Error occured!"))

            .onErrorReturn(5.0)
```

```java
                .switchIfEmpty(Mono.just(4.0));

        Double value = result.block();

        System.out.println(value);

    } catch (RuntimeException ex){

        System.out.println("Exception occured!");

    }
```

Now, the code builds and runs successfuly and the `catch` block logic is executed.

(Image 1-4)



```java
import com.iuriimednikov.projectreactorbasicsref.services.StudentService;
import reactor.core.publisher.Mono;

class Pipelines {

    public static void main(String[] args) {
        StudentService service = new StudentService();
//        Mono<Double> result = service.findStudentByName("Aneta")
//                .flatMap(s -> Mono.just(s.getGpa()))
//                .doOnError(err -> System.out.println("Error occured!"))
//                .onErrorReturn(5.0)
//                .switchIfEmpty(Mono.just(4.0));
//        Double value = result.block();
//        System.out.println(value);
```

```
Changes detected - recompiling the module!
Compiling 13 source files to C:\Courses\projectreactor-basics-ref\target\classes

--- exec-maven-plugin:1.5.0:exec (default-cli) @ projectreactor-basics-ref ---
Exception occured!
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  4.890 s
Finished at: 2021-03-13T12:19:54+01:00
------------------------------------------------------------------------
```

And here comes the truth - the Project Reactor error handling mechanism **is involved on failed Mono and Flux, but does not deal with normal Java**

exceptions/null values. In other words you **have to** wrap these results either with `empty()` or `error` factory methods. Let return to the service code and refactor it:

(Code snippet 1-27)

```
public Mono<Student> findStudentByName(String name){

    return Mono.error(new RuntimeException());

}
```

Now, the code works as supposed to be. The error callback is triggered and the default value is returned. Same applies for the empty return: the return of null value will raise an exception. You have to return `Mono.empty()` in that case.

Merging

The last topic we will review in this section is how to merge several Mono or Flux instances into a single one. The first thing is to combine several Mono instances into a single Mono object:

- `zipWith` = takes an another Mono and combines result into a new Mono that emits a *tuple*
- `zip` = this is a **static** method, that takes up to 8 Mono instances to combine result into a new Mono that emits a *tuple*

Let have a look on the following example below:

(Code snippet 1-28)

```
Mono<String> nameMono = Mono.just("Veronika");
```

```java
Mono<Double> gpaMono = Mono.just(3.5);

Mono<Tuple2<String, Double>> zippedMono = nameMono.zipWith(gpaMono);

Mono<Student> studentMono = zippedMono.map(t -> new Student(t.getT1(),
t.getT2()));

Student result = studentMono.block();

assertThat(result.getName()).isEqualTo("Veronika");

assertThat(result.getGpa()).isEqualTo(3.5);
```

Let me draw an important line here. In this example we have 2 Mono publishers, one emits the name of the student and the second emits the GPA score. We use the `zipWith` method to combine them into a new tuple. Remember, we can combine up to 8 Mono instances into a tuple. These instances can be of **different type**, like here we have `String` and `Double`. This is very useful, for instance, when you write a code, that needs to get an input of entities of several types and then use some logic to prepare a result of the another type. Think about reporting: you create an accounting app, you have a publisher that gives you invoices, a publisher that gives you bills and then you `zip` them and create a new cash flow report.

An another aspect of this topic is to merge two Mono instances of the **same type** and to provide a new unified container that hold both. For that there are two ways:

- `mergeWith`
- `concatWith`

Both of these methods accept a `Publisher` that emits the same type as Mono/Flux and produce a new Flux, that holds emissions of both instances. That is what they have same. "But, wait, why do we have two different methods? What is a difference?". Let see how they perform under the hood: xoncatination combines emissions **without** interleave, while for merging the interleave is *possible*.

That is how it sounds from the technical perspective. The difference in practice is better to overview using examples, that takes some delays. To illustrate these principles, I will utilize static methods `Flux.merge()` and `Flux.concat()`.

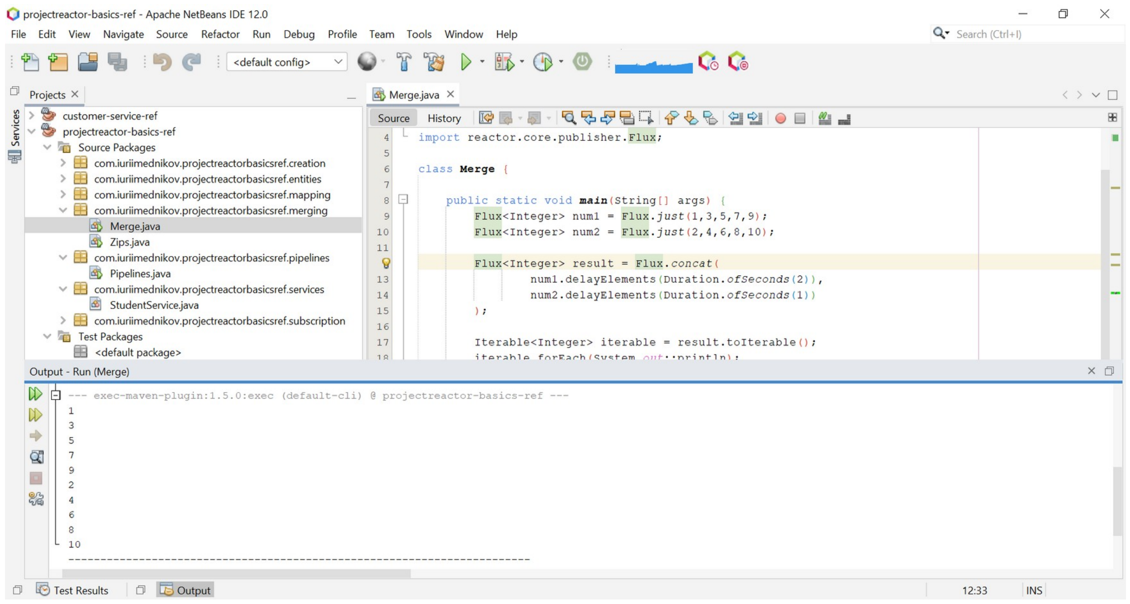
Take a look on the following example below:

(Code snippet 1-29)

```
Flux<Integer> num1 = Flux.just(1,3,5,7,9);

Flux<Integer> num2 = Flux.just(2,4,6,8,10);

Flux<Integer> result = Flux.concat(

                num1.delayElements(Duration.ofSeconds(2)),

                num2.delayElements(Duration.ofSeconds(1))

        );

Iterable<Integer> iterable = result.toIterable();

iterable.forEach(System.out::println);
```

When you run this code snippet, you will check, that numbers appear **sequentially** - first, values from `num1`, then from `num2`.

(Image 1-5)


projectreactor-basics-ref - Apache NetBeans IDE 12.0
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
<default config>
Projects
customer-service-ref
projectreactor-basics-ref
Source Packages
com.iuriimednikov.projectreactorbasicsref.creation
com.iuriimednikov.projectreactorbasicsref.entities
com.iuriimednikov.projectreactorbasicsref.mapping
com.iuriimednikov.projectreactorbasicsref.merging
Merge.java
Zips.java
com.iuriimednikov.projectreactorbasicsref.pipelines
Pipelines.java
com.iuriimednikov.projectreactorbasicsref.services
StudentService.java
com.iuriimednikov.projectreactorbasicsref.subscription
Test Packages
<default package>
Merge.java
Source
History
Output - Run (Merge)
Test Results
Output
12:33
INS

```java
import reactor.core.publisher.Flux;

class Merge {

    public static void main(String[] args) {
        Flux<Integer> num1 = Flux.just(1,3,5,7,9);
        Flux<Integer> num2 = Flux.just(2,4,6,8,10);

        Flux<Integer> result = Flux.concat(
                num1.delayElements(Duration.ofSeconds(2)),
                num2.delayElements(Duration.ofSeconds(1))
        );

        Iterable<Integer> iterable = result.toIterable();
        iterable.forEach(System.out::println);
```

```
--- exec-maven-plugin:1.5.0:exec (default-cli) @ projectreactor-basics-ref ---
1
3
5
7
9
2
4
6
8
10
------------------------------------------------------------------------
```

Then, take an example of using the `merge` operation:

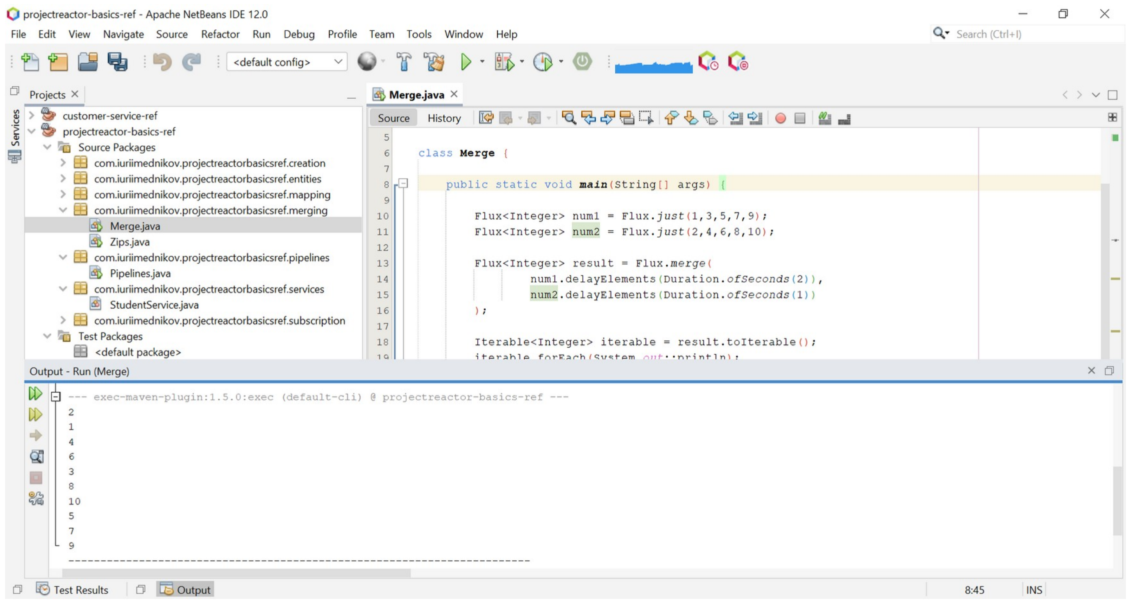
(Code snippet 1-30)

```java
Flux<Integer> num1 = Flux.just(1,3,5,7,9);

Flux<Integer> num2 = Flux.just(2,4,6,8,10);

Flux<Integer> result = Flux.merge(

                num1.delayElements(Duration.ofSeconds(2)),

                num2.delayElements(Duration.ofSeconds(1))

        );
```

```
Iterable<Integer> iterable = result.toIterable();

iterable.forEach(System.out::println);
```

In this case, sources are subscribed eagerly:

(Image 1-6)



```
class Merge {

    public static void main(String[] args) {

        Flux<Integer> num1 = Flux.just(1,3,5,7,9);
        Flux<Integer> num2 = Flux.just(2,4,6,8,10);

        Flux<Integer> result = Flux.merge(
                num1.delayElements(Duration.ofSeconds(2)),
                num2.delayElements(Duration.ofSeconds(1))
        );

        Iterable<Integer> iterable = result.toIterable();
        iterable.forEach(System.out::println);
```

```
--- exec-maven-plugin:1.5.0:exec (default-cli) @ projectreactor-basics-ref ---
2
1
4
6
3
8
10
5
7
9
------------------------------------------------------------------------
```

With the added delay, you can clearly spot the difference of these methods.

Summary

In this section we covered reactive publisher types from Project Reactor: `Mono` and `Flux`. We compared them and reviewed most important aspects of their usage, including:

- Creating Mono and Flux instances from Java objects and from async providers (such as `Runnable` or `CompletableFuture`)
- Use cases of Mono and Flux classes
- Mapping and the difference between `map()` and `flatMap()` methods
- How to subscribe on Mono and Flux events

- Build pipelines with Mono and Flux components
- How to merge several Mono/Flux instances to a single result
- Blocking vs non-blocking
- Error handling

Chapter 2. Your first Spring Webflux REST API

The best teacher is practice, and for that, in this section we will create your first REST API using the Spring Webflux framework. In order to make things simple, we will concentrate on most essential features, while skipping others for now. Especially, in this section we will develop a small REST API service to handle customers management for a hypothetical CRM platform. This is a basic CRUD (Create-Retrieve-Update-Delete) functionality:

- Create a new customer
- Retrieve all customers
- Retrieve a single customer using its ID
- Update a customer
- Delete a customer

As a data source we will utilize a NoSQL MongoDB which is a perfect choice for microservices and APIs. Because a project is very simple and is limited to a single domain, I will stick to a **package by layer** architecture (please see the related section for details about structural patterns in Spring).

I recommend you to use an IDE with installed Spring Boot support plugin, although you can follow this section if you don't use an IDE and rely on Maven CLI and a text editor of your choice (like Atom, VS Code or Vim). When it is necessary I will mark explicitly how to deal with things using Maven CLI.

1. Create a new Spring Boot project

While *it is not technically required for Webflux to use Spring Boot*, we will use it in this section (and in the book in general), because it is a great way to bootstrap your application with such essential features like dependency injection, data source management, deployment etc.

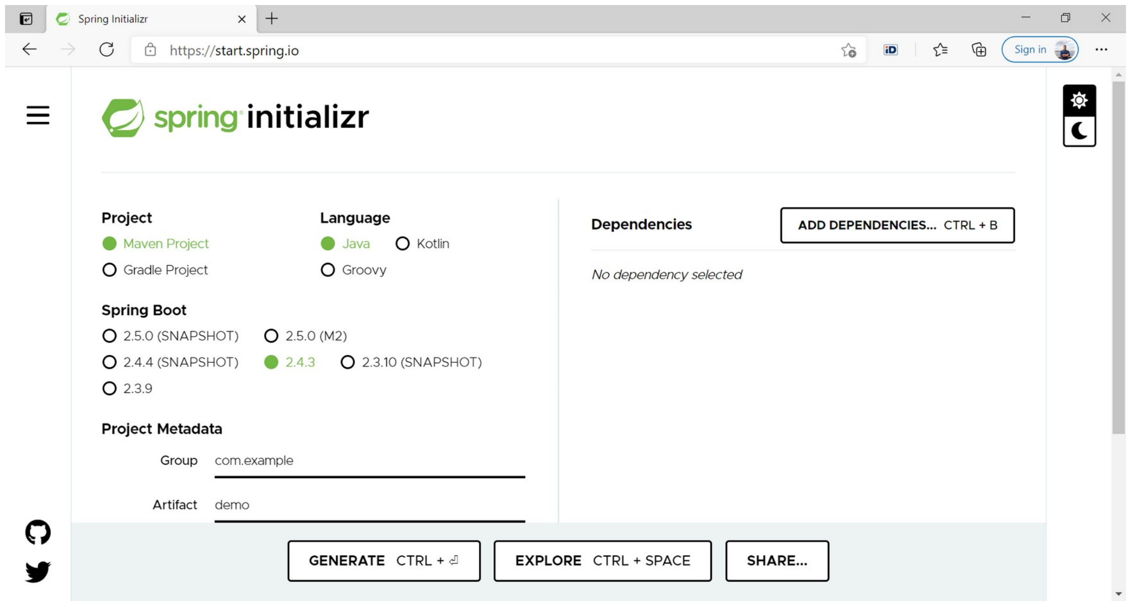
Basically, we can create a new Spring Boot app in three ways:

- Use your IDE's Spring Boot integration and create a new app using a wizard (the most recommended for any developer)
- Use Spring Initialzr website and then import a generated project into your IDE
- Use Spring Initialzr and then work with a project with Maven CLI tool

I personally recommend you to use the Option 1, yet we will not cover it here. This is due to the fact, that each IDE may have a different proccess, and it is better to consult your IDE documentation here. But, basically, all these integrations just wrap up a Spring Initialzr into your IDE's interface.

You can go to the Spring Initialzr website and there use the tool to generate a new Spring Boot project. Most important here is to select correct dependencies and Java version. In this book we use Java 11 as it is a LTS edition for the time of writing.

(Image 2-1)



Project: [✓ Maven Project] Gradle Project

Language: [✓ Java] Kotlin Groovy

Spring Boot: 2.5.0 (SNAPSHOT) 2.5.0 (M2) 2.4.4 (SNAPSHOT) [✓ 2.4.3] 2.3.10 (SNAPSHOT) 2.3.9

Project Metadata

Group: com.example

Artifact: demo

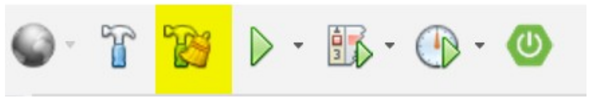
Dependencies

No dependency selected

Once you download a project, you can import to your IDE or use it with Maven directly.

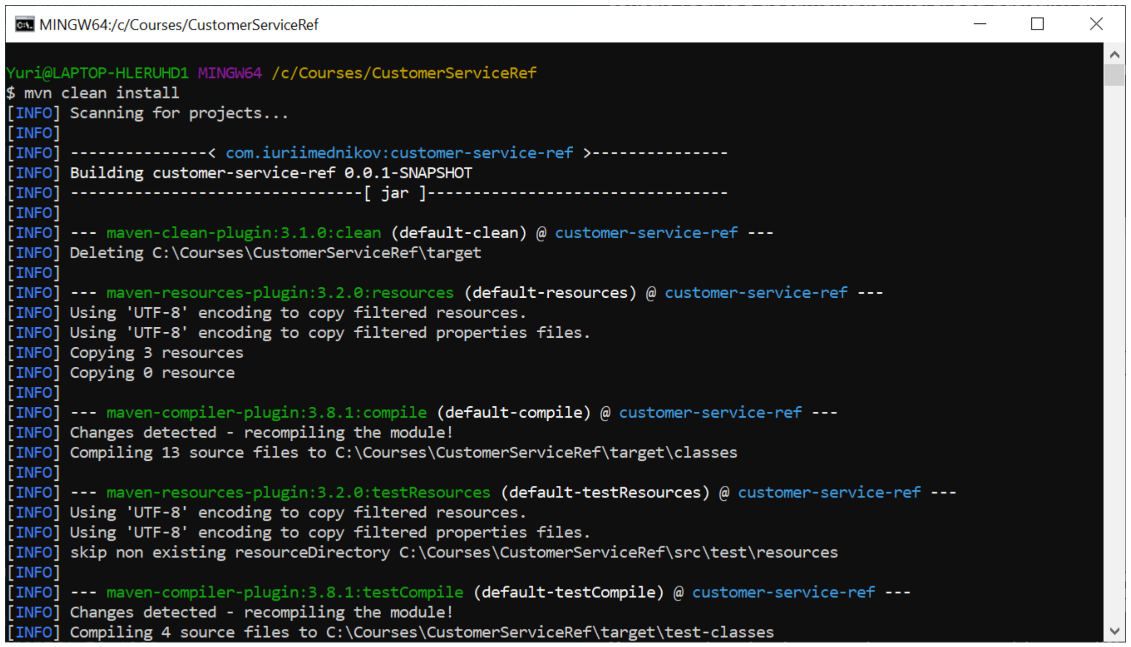
For Apache Netbeans, use the `File -> Open Project` menu item and select a root folder of the project (the one where resides the `pom.xml`). Then, you need to build to download dependencies and check that there are no erros. Use the menu item `Run -> Clean and build project` or press the icon from the toolbar:

(Image 2-2)



If you are using Maven CLI directly, then navigate to the root folder in your terminal and run `mvn clean install` to build a project:

(Image 2-3)



```
MINGW64:/c/Courses/CustomerServiceRef

Yuri@LAPTOP-HLERUHD1 MINGW64 /c/Courses/CustomerServiceRef
$ mvn clean install
[INFO] Scanning for projects...
[INFO]
[INFO] --------------< com.iuriimednikov:customer-service-ref >--------------
[INFO] Building customer-service-ref 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-clean-plugin:3.1.0:clean (default-clean) @ customer-service-ref ---
[INFO] Deleting C:\Courses\CustomerServiceRef\target
[INFO]
[INFO] --- maven-resources-plugin:3.2.0:resources (default-resources) @ customer-service-ref ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Using 'UTF-8' encoding to copy filtered properties files.
[INFO] Copying 3 resources
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ customer-service-ref ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 13 source files to C:\Courses\CustomerServiceRef\target\classes
[INFO]
[INFO] --- maven-resources-plugin:3.2.0:testResources (default-testResources) @ customer-service-ref ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Using 'UTF-8' encoding to copy filtered properties files.
[INFO] skip non existing resourceDirectory C:\Courses\CustomerServiceRef\src\test\resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:testCompile (default-testCompile) @ customer-service-ref ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 4 source files to C:\Courses\CustomerServiceRef\target\test-classes
```

2. Set up profiles

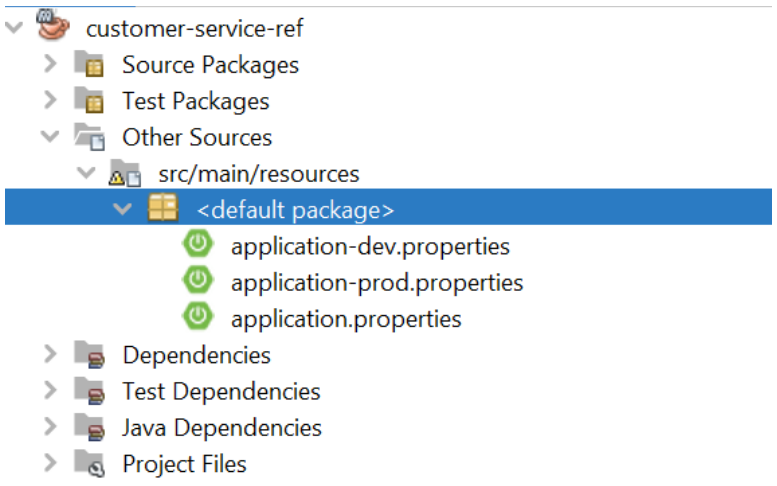
A good software design requires developers to separate a configuration from a code, because an application should be able in a same way in different environemnts. For example, you can have a development environment on your machine, a production and a staging environments or a separate configuration for CI/CD pipelines.

In Spring there is a concept of *profiles* for this. From a techical point of view, profiles is a way to separate parts of the application configuration and make it only available in certain environments. You can use profiles to control:

- Application properties
- Which beans (components) to load

Because our app is simple we will concentrate in this section only on the first thing and in next sections also will review how to control beans using profiles. So, application properties live in the `src/main/resources` folder:

(Image 2-4)



In the `application.properties` we define an application-wide configuration. For specific environments we can create dedicated properties files, namely in this project we will use:

- `application-dev.properties` for the local development
- `application-prod.properties` for the production environment

Basically, we need to specify two things: a port number, that app will listen and MongoDB URI. For the development profile we can hardcode values in the file like that:

(Code snippet 2-1)

```
spring.data.mongodb.uri=mongodb://localhost:27017/authdb

server.port=4568
```

For the production configuration things are a bit different. First, we don't know yet exact credentials. Second, as we would put our code under source control, we don't want to make visible paid credentials. Hopefully, Spring allows to listen for environment variables to inject properties. The syntax is show below:

(Code snippet 2-2)

```
spring.data.mongodb.uri=${MONGODB_URI}

server.port=${PORT}
```

Finally, we need to tell to the Spring Boot which profile is *active*. You can set up in the `application.properties` file and override using command line arguments on runtime.

(Code snippet 2-3)

```
spring.profiles.active=dev,prod
```

3. Models and data layer

In this subsection we will cover two elements: model layer and data layer. Due to the fact, that our example application is fairly simple, *the model* will stand both for a database entity and a business object. In more complex applications, you may need to use *data-transfer objects* instead.

We have two models here:

- Customer
- Address

First, let define a new `AddressModel` class:

(Code snippet 2-4)

```
@Value

public class AddressModel {

    String addressLine;

    String postalCode;

    String city;
    @JsonCreator
    public AddressModel(
            @JsonProperty("addressLine") String addressLine,
            @JsonProperty("postalCode") String postalCode,
            @JsonProperty("city") String city) {
        this.addressLine = addressLine;
        this.postalCode = postalCode;
        this.city = city;
    }
}
```

Let quickly stop here to showcase important aspects, presented in this code snippet. The entity is *immutable*, e.g. an object whose state cannot be modified after it is created. That means that it has:

- All private fields
- All final fields
- No setters
- Fields are intialized in creation time, so it has an all-args constructor

We can do it manually, of course. However, it is more convinient to use the `@Value` annotation from Lombok, which does that job for us.

Another important aspect is the constructor. In my implementation I explicitly provide an all-args constructor, combined with Jackson annotations. From a technical point of view, Jackson depends on no-args constructor and setters to deserialize a JSON payload into a Java object. But, we don't have such things, because we use immutable objects. For that, we create a constructor and tell to Jackson to use it for deserialization (`@JsonCreator`). Jackson does not have a way to use a reflection in runtime to get field names, so we specify them for constructor's arguments with the `@JsonProperty`.

In a same way, we define the customer model. Take a look to my implementation below:

(Code snippet 2-5)

```
@Value

@Document(collection = "customers")

public class CustomerModel {

    @Id String customerId;

    String companyName;

    String companyEmail;

    String taxId;

    AddressModel billingAddress;

    AddressModel shippingAddress;
    @JsonCreator

    public CustomerModel(

            @JsonProperty("customerId") String customerId,

            @JsonProperty("companyName") String companyName,

            @JsonProperty("companyEmail") String companyEmail,

            @JsonProperty("taxId") String taxId,

            @JsonProperty("billingAddress") AddressModel
billingAddress,

            @JsonProperty("shippingAddress") AddressModel
shippingAddress) {

        this.customerId = customerId;

        this.companyName = companyName;

        this.companyEmail = companyEmail;

        this.taxId = taxId;

        this.billingAddress = billingAddress;

        this.shippingAddress = shippingAddress;

    }
```

```
}
```

This object is used as a database entity (or a *document* in terms of MongoDB), so it has two additional annotations:

- `@Document` identifies a domain object to be persisted to MongoDB
- `@Id` specifies an identifier

In this section I would like to cover a data layer. In our example we don't have any custom functionality, and will just use a basic CRUD functionality from Spring Data repositories.

(Code snippet 2-6)

```
public interface CustomerRepository extends
ReactiveMongoRepository<CustomerModel, String> {}
```

4. Business layer

In Spring, service is a component that contains a business logic. This is a core of your application. In our example application a service will provide a basic CRUD functionality. First step is to define an interface (contract), which will have required methods. Take a look on the following code snippet below:

(Code snippet 2-7)

```
public interface CustomerService {

    Mono<CustomerModel> createCustomer (CustomerModel customer);
    Mono<CustomerModel> updateCustomer (CustomerModel customer);

    Mono<Void> removeCustomer (String customerId);

    Mono<CustomerModel> findCustomerById (String customerId);

    Flux<CustomerModel> findAllCustomers ();

}
```

If you are familiar with Spring MVC, you can note, that this service looks generally same as in Spring MVC, however it returns reactive types. This will allow to callers to subscribe for results of asynchronous computations instead of doing blocking calls.

Now, let create an implementation of this interface:

(Code snippet 2-8)

```
@Component("CustomerService")

@Value

public class CustomerServiceImpl implements CustomerService {

    CustomerRepository customerRepository;

    @Override

    public Mono<CustomerModel> createCustomer(CustomerModel customer)
{
        return customerRepository.save(customer);

    }

    @Override

    public Mono<CustomerModel> updateCustomer(CustomerModel customer)
{

        return customerRepository.save(customer);

    }

    @Override

    public Mono<Void> removeCustomer(String customerId) {

        return customerRepository.deleteById(customerId);

    }

    @Override

    public Mono<CustomerModel> findCustomerById(String customerId) {

        return customerRepository.findById(customerId);

    }

    @Override

    public Flux<CustomerModel> findAllCustomers() {
        return customerRepository.findAll();

    }

}
```

I found it important to clarify some aspects. From a technical perspective, the service "wraps up" around the data layer, however in more complex solutions you many need to provide a more serious functionality. I want to stop on two aspects:

- We use the `@Value` annotation to mark field dependencies final and private and to create an all-args constructor (and to use constructor-based dependency injection)
- We annotate the implementation with the `@Component` annotation to mark for Spring Container to use this bean as `CustomerService` instance for its callers

5. Web layer

There are two approaches to build a web layer in Spring Webflux - with annotated reactive controllers and router functions (router handlers). We talk about it more in a detailed way in the corresponding section. To start, I would like to use here the first approach. If you are familiar with Spring MVC you will find it familiar. If you are not yet familiar with Spring MVC, you will find this pattern more easier to understand. Also, it is important to mention here, that this architecture allows to utilize Spring web components, like `@RequestBody`, `ResponseEntity`, validations etc. With router functions, developers have to implement this functionality on their own.

Take a look on my implementation of the REST controller for our application:

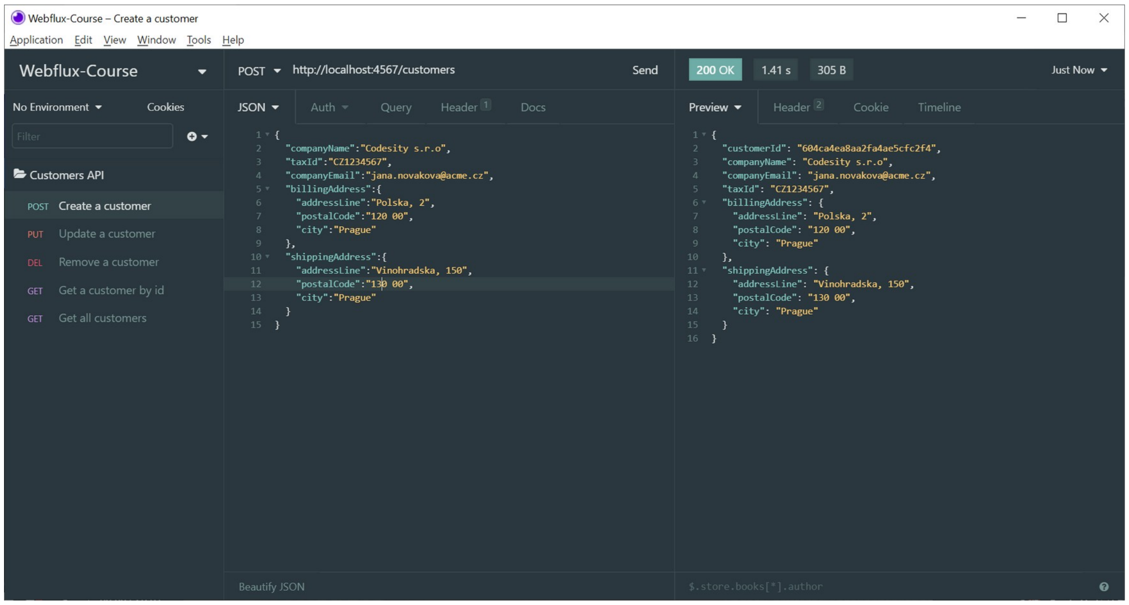
(Code snippet 2-9)

```
@Value

@RestController

@RequestMapping(

        produces = MediaType.APPLICATION_JSON_VALUE,

        consumes = MediaType.APPLICATION_JSON_VALUE)

public class CustomerRestController {

    CustomerService customerService;

    @GetMapping("/customer/{id}")

    Mono<ResponseEntity<CustomerModel>> getCustomerById
(@PathVariable("id") String id){

        return customerService.findCustomerById(id)

                .map(customer -> ResponseEntity.ok(customer));

    }

    @PostMapping("/customers")
    Mono<ResponseEntity<CustomerModel>> postCustomer (@RequestBody
CustomerModel customer) {

        return customerService.createCustomer(customer).map(result ->
ResponseEntity.ok(result));

    }

    @PutMapping("/customer/{id}")

    Mono<ResponseEntity<CustomerModel>> putCustomer (@RequestBody
CustomerModel customer) {

        return customerService.createCustomer(customer).map(result ->
ResponseEntity.ok(result));

    }

    @GetMapping("/customers")

    Flux<CustomerModel> getAllCustomers(){

        return customerService.findAllCustomers();

    }

    @DeleteMapping("/customer/{id}")

    Mono<Void> deleteCustomer(@PathVariable("id") String id){

        return customerService.removeCustomer(id);
    }

}
```

For those readers who have an experience of working with Spring MVC, most aspects are familiar. What is important, is that controller's methods return reactive types, instead of direct return of `ResponseEntity` objects. Also, we have to provide mapping, because services return business objects. In the opposite, `ResponseEntity` is a *web* object (belongs to a web layer). We can run our application and verify that everything works as expected.

You can test the API with your favorite HTTP client (I prefer Insomnia). In the beginning, let try to create a new customer object:

(Image 2-5)



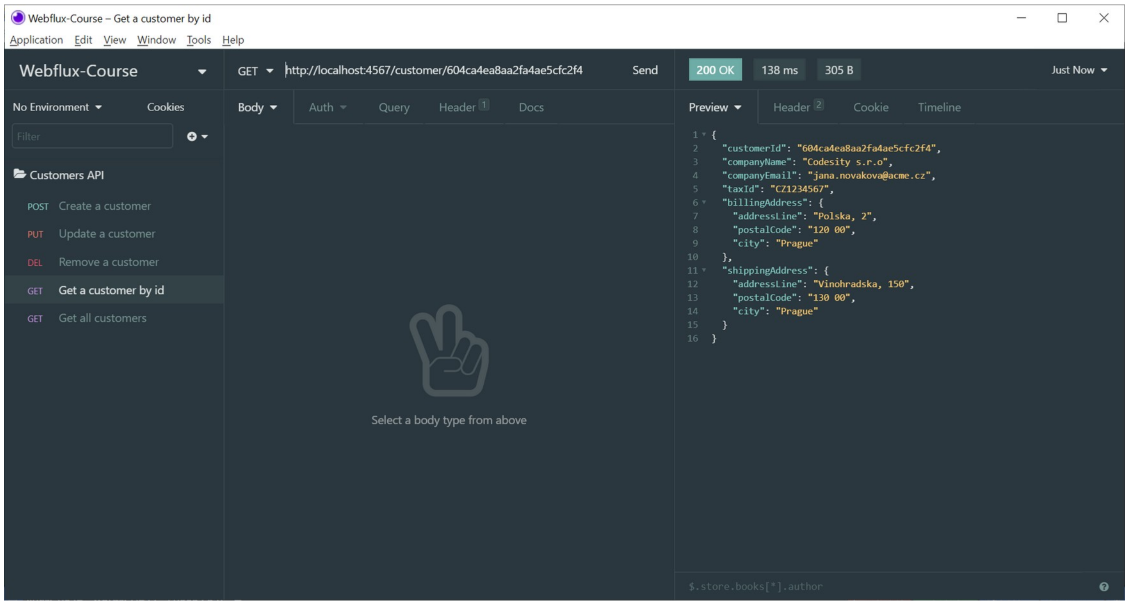
POST http://localhost:4567/customers

```
{
  "companyName":"Codesity s.r.o",
  "taxId":"CZ1234567",
  "companyEmail":"jana.novakova@acme.cz",
  "billingAddress":{
    "addressLine":"Polska, 2",
    "postalCode":"120 00",
    "city":"Prague"
  },
  "shippingAddress":{
    "addressLine":"Vinohradska, 150",
    "postalCode":"130 00",
    "city":"Prague"
  }
}
```

```
{
  "customerId": "604ca4ea8aa2fa4ae5cfc2f4",
  "companyName": "Codesity s.r.o",
  "companyEmail": "jana.novakova@acme.cz",
  "taxId": "CZ1234567",
  "billingAddress": {
    "addressLine": "Polska, 2",
    "postalCode": "120 00",
    "city": "Prague"
  },
  "shippingAddress": {
    "addressLine": "Vinohradska, 150",
    "postalCode": "130 00",
    "city": "Prague"
  }
}
```

We should be able to retrieve the entity using the identifier with the `GET` request:

(Image 2-6)

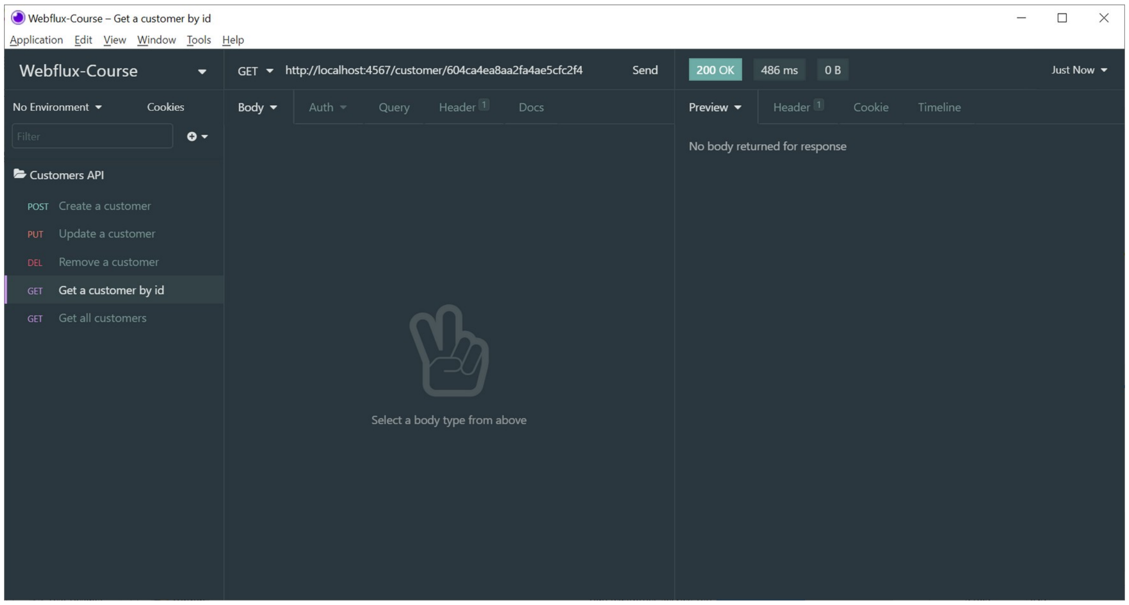


```
http://localhost:4567/customer/604ca4ea8aa2fa4ae5cfc2f4
```

```
{
  "customerId": "604ca4ea8aa2fa4ae5cfc2f4",
  "companyName": "Codesity s.r.o",
  "companyEmail": "jana.novakova@acme.cz",
  "taxId": "CZ1234567",
  "billingAddress": {
    "addressLine": "Polska, 2",
    "postalCode": "120 00",
    "city": "Prague"
  },
  "shippingAddress": {
    "addressLine": "Vinohradska, 150",
    "postalCode": "130 00",
    "city": "Prague"
  }
}
```

When we remove a single entity, it should be deleted from a database, and therefore we do not expect it to exist anymore:

(Image 2-7)



```
http://localhost:4567/customer/604ca4ea8aa2fa4ae5cfc2f4
```

```
No body returned for response
```

But wait, what happened? We don't get a result body but still have a `200 OK` status. This violates REST API principles, because we expect to get a `404 Not found` status code in this situation. Let change that.

6. Fixing an empty response output

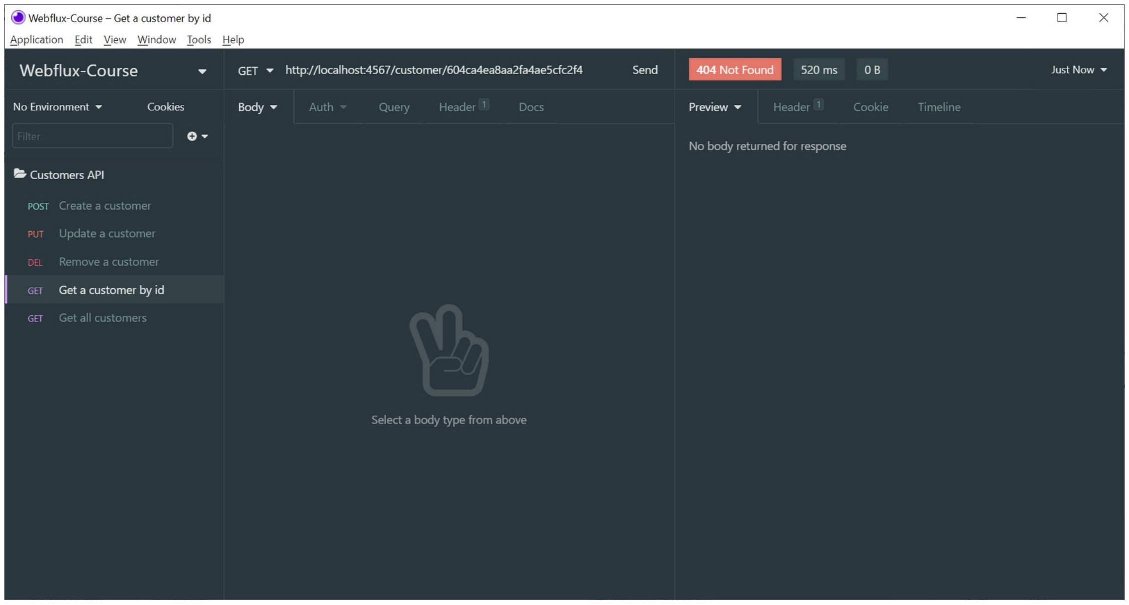
From a technical point of view, an empty response is a special case of an error handling. We overview this topic in the dedicated section of this book. For now, as we have an annoted controller, we can use the `ResponseEntity` to display the empty response. To achieve that, utilize the `switchIfEmpty` function to make a Mono to emit a *default* entity, when the main entity is empty. Take a look on my implementation of the controller method:

(Code snippet 2-10)

```
@GetMapping("/customer/{id}")

Mono<ResponseEntity<CustomerModel>> getCustomerById
(@PathVariable("id") String id){

    return customerService.findCustomerById(id)

            .map(customer -> ResponseEntity.ok(customer))

            .switchIfEmpty(Mono.just(ResponseEntity.notFound().build()
));

}
```

Now, the problem should be fixed:

(Image 2-8)



http://localhost:4567/customer/604ca4ea8aa2fa4ae5cfc2f4

No body returned for response

Summary

In this section we created a simple REST API with Spring Webflux. Of course, there is a lot of things you can add here; this is just a barebone skeleton. However, it demonstrated most important aspects, including architecture, implementation of layers, using profiles etc. There are other topics to be covered outside the scope of this section, like error handling, package structure etc. You will get more information in dedicated sections of the book.

Chapter 3. Annotated controllers vs router functions

The Spring Webflux introduces a new way to write a web layer with reactive router functions (router handlers). Prior to that, the Spring MVC used an idea of *controllers* to handle this. The new framework allows to use both techniques, which means developers can select either a more familiar approach either a new pattern, designed as a reactive-first way. In this section we will talk about both tecniques and discuss their use cases.

Annotated controllers with Spring Webflux

In the section, devoted to building a simple REST API, we used an annotated controller in order to implement a web layer of our project. Let review how we used it. Take a look on the code snippet below:

(Code snippet 3-1)

```
@Value

@RestController

@RequestMapping(

       produces = MediaType.APPLICATION_JSON_VALUE,

       consumes = MediaType.APPLICATION_JSON_VALUE)

public class CustomerRestController {

    CustomerService customerService;
    @GetMapping("/customer/{id}")

    Publisher<ResponseEntity<CustomerModel>> getCustomerById
(@PathVariable("id") String id){

        return customerService.findCustomerById(id)

                .map(customer -> ResponseEntity.ok(customer))

                .switchIfEmpty(Mono.just(ResponseEntity.notFound().bui
ld()));

    }

    @PostMapping("/customers")

    Publisher<ResponseEntity<CustomerModel>> postCustomer
(@RequestBody CustomerModel customer) {

        return customerService.createCustomer(customer).map(result ->
ResponseEntity.ok(result));

    }

    @PutMapping("/customer/{id}")

    Publisher<ResponseEntity<CustomerModel>> putCustomer (@RequestBody
CustomerModel customer) {

        return customerService.createCustomer(customer).map(result ->
ResponseEntity.ok(result));
```

```
    }

    @GetMapping("/customers")

    Publisher<CustomerModel> getAllCustomers(){

        return customerService.findAllCustomers();

    }

    @DeleteMapping("/customer/{id}")

    Publisher<Void> deleteCustomer(@PathVariable("id") String id){

        return customerService.removeCustomer(id);

    }

}
```

For those developers, who are familiar with Spring MVC, the usage of annotated REST controllers may be a good choice to transit to the Spring Webflux universe. Basically, it depends on how your application uses Spring MVC feautures. For solutions, that depend on template rendering (server side rendering), Spring MVC web components, error handling or validation, it is better to stick with annotated controllers, because the amount of work to move them to reactive railways is minimal. Another advantage of this approach is that it is much easier to use and implement for novice developers, as you have only a single component to represent a web layer, compare to router functions.

Let summarize these points. Advantages of an usage of annotated REST controllers are following:

- Simplicity of an implementation and an usage, especially for developers, familair with Spring MVC, as well for newcomers
- Minimal amount of change for existing Spring MVC apps
- A possibility to use Spring MVC features with Spring Webflux

However, nothing comes without a price. This approach has following drawbacks:

- Support of reactive programming model is limited
- Developers have to use Spring MVC under the hood, which makes an application heavier, and this may not be a solution for microservices and serverless

Now, let review router handlers.

Using router functions (handlers)

Being a reactive framework, Spring Webflux introduced a new paradigm to write a web layer. It is known as router functions (or router handlers; routers). The key difference with the older technique is that the logic is separated into **two** components:

- Router handler
- Router function configuration

The first component (handler) provides a logic, that deals with requests and responses. It contains a number of methods, that accept the `ServerRequest` object and return a reactive publisher (of type of Mono), that emits the `ServerResponse` entity. This allows to do server calls in a non-blocking way. Let take a look to the simple implementation:

(Code snippet 3-2)

```
@Value

@Component

class CustomerHandler {

    CustomerService customerService;

    Mono<ServerResponse> postCustomer(ServerRequest request) {

        Mono<CustomerModel> body =
request.bodyToMono(CustomerModel.class);

        Mono<CustomerModel> result = body.flatMap(b ->
customerService.createCustomer(b));

        return
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(resul
t, CustomerModel.class);

    }

    Mono<ServerResponse> putCustomer(ServerRequest request) {

        return request.bodyToMono(CustomerModel.class)

                .flatMap(customerService::updateCustomer)

                .flatMap(result -> ServerResponse

                        .ok()
                        .contentType(MediaType.APPLICATION_JSON)

                        .bodyValue(result));

    }

    Mono<ServerResponse> deleteCustomer(ServerRequest request) {


        String id = request.pathVariable("id");

        Mono<Void> result = customerService.removeCustomer(id);

        return
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(resul
t, Void.class);

    }

    Mono<ServerResponse> getCustomerById(ServerRequest request) {


        String id = request.pathVariable("id");

        return customerService.findCustomerById(id)

                .flatMap(result ->
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).bodyValue(
result))

                .switchIfEmpty(ServerResponse.notFound().build());
    }

    Mono<ServerResponse> getAllCustomers(ServerRequest request) {


        Flux<CustomerModel> result = 
customerService.findAllCustomers();

        return 
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(resul
t, CustomerModel.class);

    }

}
```

This component also serves as a glue between business components layer (services) and a web layer. However, we do much inside a handler, to name a few:

- Error handling
- Validation
- Type conversions

Compare to annotated controllers, handlers could not utilize a power of Spring MVC web components. In other words, developers have to do these operations manually. From the other view, you can use this technique to build an application **without Spring Boot**, which can be a choice for lightweight solutions, like serverless or microservices.

The second part is the router configuration. It is a class, annotated with a `@Configuration`, which exposes a `RouterFunction` bean, which basically connects handler methods with corresponding URLs and HTTP methods (similar to what we did with request mappings in annotated controllers).

(Code snippet 3-3)

```
@Configuration

class CustomerRouter {

    @Bean

    RouterFunction<ServerResponse> customerEndpoint(CustomerHandler
handler){

        return RouterFunctions

                .route(POST
("/customers").and(accept(MediaType.APPLICATION_JSON)),

                        handler::postCustomer)

                .andRoute(GET
("/customers").and(accept(MediaType.APPLICATION_JSON)),

                        handler::getAllCustomers)

                .andRoute(GET
("/customer/{id}").and(accept(MediaType.APPLICATION_JSON)),

                        handler::getCustomerById)

                .andRoute(DELETE
("/customer/{id}").and(accept(MediaType.APPLICATION_JSON)),

                        handler::deleteCustomer)

                .andRoute(PUT
("/customer/{id}").and(accept(MediaType.APPLICATION_JSON)),

                        handler::putCustomer);

    }

}
```

To sum up, let review advantages of the new architecture:

- A possibility to build lightweight applications that do not depend on Spring Boot and Spring MVC
- Better implementation of reactive programming models

There are drawbacks too:

- No built-in functionality for error handling, validation etc. This should be done manually
- Server side rendering is available, but is limited compare to Spring MVC

Summary

In this section we provided an overview of two approaches to build a web layer for Spring Webflux applications. Developers can use either annotated REST controllers like with Spring MVC with minimal changes (mainly to handle reactive types) or use new router function paradigm. Differences come depending on the concrete solution and how it depends on Spring

MVC itself. For greenfield projects and for lightweight apps, like serverless or microservices I could advice to use new patterns. If the application already uses a lot of Spring MVC features, including rendering, validation, error handling, it is better to stick with annotated controllers.

Chapter 4. How to structure Spring Webflux applications

An important aspect of a good software design is a selection of project structure. This is not just about namespaces and folders. There are several software architecture principles that are targeted by this:

- Single responsibility principle
- Separation of concerns principle
- Dont-repeat-yourself (DRY) principle

Therefore, you need to choose a project structure that is adequate for your goals and application. Talking about Spring apps, the industry created three main approaches:

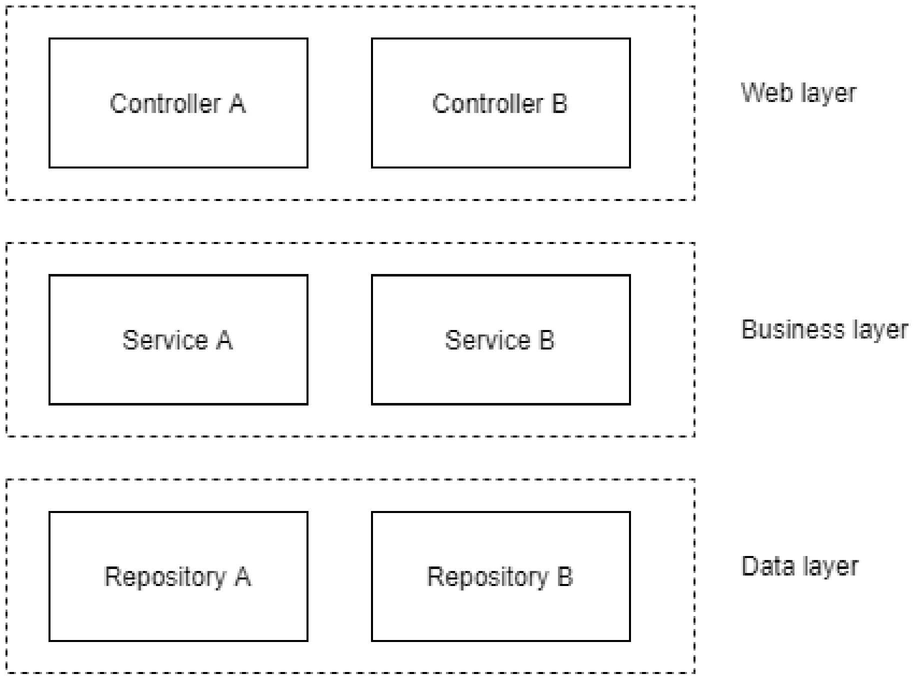
- Packaging by technical layer
- Packaging by feature (business domain)

In this section we will observe both of them and review their advantages and drawbacks.

Packaging by technical layer

The idea of packaging by technical layer came to Spring from the "Four tier architecture" of Java EE applications. The general graphical representaion of this pattern is shown below:

(Image 4-1)



Unfortunately, many software enginners wrongly think that this principle is only tied with MVC solutions (Model-View-Controller), that includes a *presentaion layer* (web layer). This is partly true, that modern REST API does not include server-side rendering to expose views and delegate this responsibility to client applications. However, it is also true that the definition of the web layer goes beyond of just rendering views. This layer contains a logic that deals with HTTP request/response handling, including JSON, validation, error handling etc.

The second main component is a *data access layer*. In Spring community it is also known as a *repository layer*, because Spring uses *repositories* to abstract database operations. So, in most cases, your "intervention" in a data layer is minimal and often is limited to providing custom query logic.

The third block stands for a *business logic layer*, or as it called in Spring, a *service layer*. This is a place where you normally should do most of your work. Some developers think about this level as just a connection between a web layer and a database layer. However, this is not true. In this layer you incorporate all code that corresponds to a business logic of your application, including:

- Communication with external APIs
- Report generations
- Business domain communications (for example, tax calculations, money operations, statistical operations etc.)

The idea of a "package by layer" architecture has a number of advantages:

- Each layer can be tested in isolation, while external dependencies are mocked
- As layers are abstracted for their dependencies, you can easily replace concrete implementations. For example, if you think about database operations as a "data layer", you can change concrete mechanisms without affecting your business logic and web layer.
- It makes simple to separate work between developers, where each developer can work on aa particular layer

These reasons made this pattern a common practice for enterprise companies. However, drawbacks also exist:

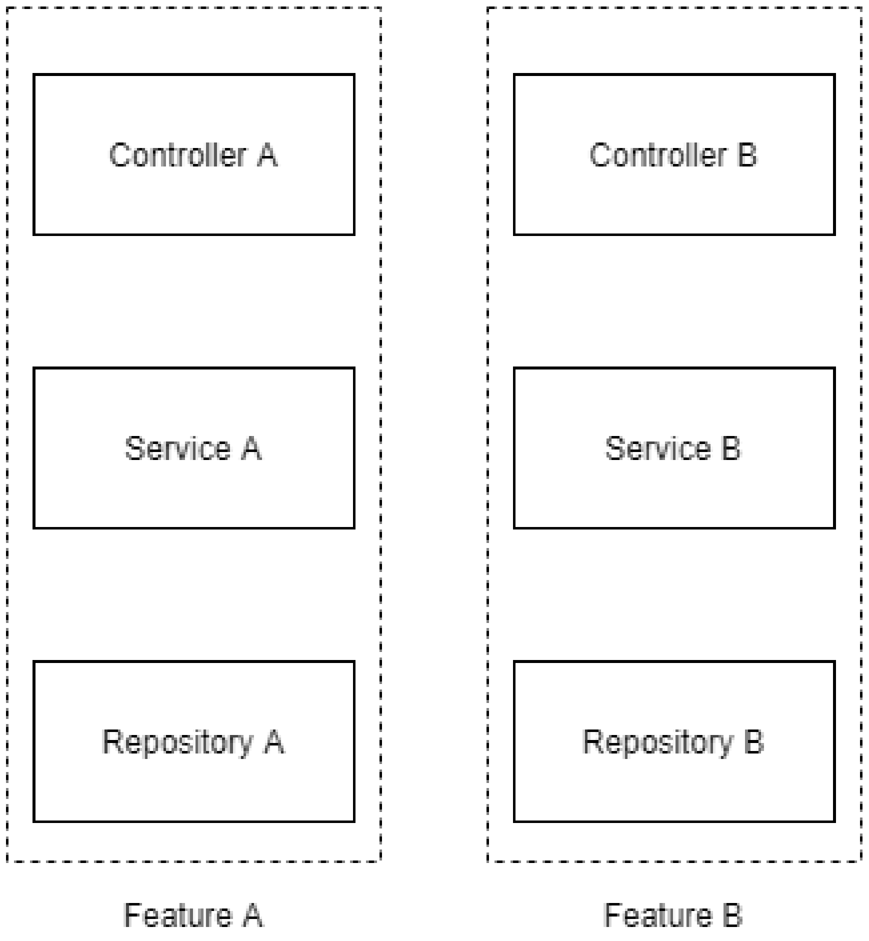
- The codebase become monolith, it is very hard to decompose it to microservices, because the logic is placed in several layers
- Security issues, mainly because most of components are `public` in order to be accessed by other packages

Usually, this architecture is used with monolith applications, which is not something bad. Another pattern is called "package by feature".

Packaging by feature (business domain)

In order to solve issues of the layered architecture, was created an alternative pattern, which is called "package by feature", or by business domain. Take a look on the following graph below:

(Image 4-4)



This idea is based on what is called *domain-driven design*. Technically speaking, you need to decompose your application features into its business domains. For example, when you work with a CRM application, it may include following domains:

- Customers

- Projects
- Tasks
- Invoices
- Payments

Each of this package incorporates **all** technical tiers that are connected with a feature, including web handling, data access, business logic. This architecture has a number of advantages:

- Each feature can be shipped separately as a decoupled micro service
- Domain code is hidden from outside callers that increases a level of security
- It easier to add new features, especially when they are not unified. For example, some of your features may use database as a data source, other can get data from outside APIs

As it was mentioned, this is a perfect candidate for microservices. Some people say this architecture does not have disadvantages, however not everyting is so. There are drawbacks too:

- If one domain depends on another, you still have to provide an access, which violates hiding and makes the feature exposed for other callers
- It makes your feature code really messy, so for big projects you need to use a hybrid approach - package your application by feature and by a layer inside a domain
- There is a danger of having a lot of repeatable code

Summary

In this section we did an overview of two common architectural patterns that are used with Spring applications:

- Package by layer
- Package by feature

Both of them have advantages and disadvantages and there is no such thing as "one approach fits all". In my practice I found out that both of them should be seen as guidelines, however with real projects you may need to adapt things to concrete needs. Basically, the "package by layer" structure is often used with monolith apps, while the "package by feature" is common for microservices and serverless.

Chapter 5. Calling external APIs with the WebClient

From a technical point of view, we distinguish between blocking and non-blocking IO. Prior to the Spring WebFlux, Spring ecosystem relied on Apache HttpClient library as an underlying implementation to work with HTTP. Blocking means, that a system call *halts* the execution flow until it receives an reply. Opposite to it stands non-blocking IO. In that case, execution *continues* and it deals with the result later. That principle lies in the idea of reactive programming. Spring 5 introduces among other async tools new `WebClient` component, that is a higher-level abstraction to HTTP client implemetations.

Create a new WebClient instance

First, we need to obtain an instance of `WebClient`. Basically, we can do it using one of three approaches:

- `WebClient.create()` is a static method without arguments, that returns a client with default settings, which you can customize later
- `WebClient.create(baseUrl)` is another static method, but it accepts a base url as an argument. This means that your requests are executed against this base url as a root
- Using the builder pattern `WebClient.builder()` allows to customize a client instance in a flexible way

Take a look on the following code snippet below, that demonstrates usages of aforesaid methods:

(Code snippet 5-1)

```
WebClient basicClient = WebClient.create();
WebClient rootClient = WebClient.create("https://myserver.com");

WebClient builderClient = WebClient.builder()

    .baseUrl("https://myserver.com")

    .defaultCookie("key", "value")

    .defaultHeader(HttpHeaders.CONTENT_TYPE,
MediaType.APPLICATION_JSON_VALUE)

    .build();
```

Once created, we can reuse the HttpClient instance in code.

Retrieve data

In this subsection we will review how to perform a simple `GET` request and to access response body as Java type. Like in every case with Java, you first need to prepare a data model, that will be used for a deserialization.

The Webflux HttpClient operates two main categories - a request and a response. In order to prepare a request there are several approaches:

- Use methods `get()`, `head()`, `delete()` to obtain the `RequestHeadersUriSpec` instance, which allows to specify request headers and URI
- Use a function `method()` which takes a desired http method and returns the `RequestBodyUriSpec` instance, that goes for specifying request headers, body and URI for a request
- Use methods `put()`, `post()` that creates an instance of `RequestBodyUriSpec` to define headers, body and URI for a request.

On the other side, the response is defined by the only `ResponseSpec` interface, which encapsulates the response data. That instance can be mapped to a body or the `ResponseEntity` object, which contains all information about the response, including response body, status code, headers, method etc. Both operations return a reactive type, that allows to subscribe for a result.

To start, let have a look on how to perform a get request, that emits a single entity in a body and to map this entity to a Java type.

(Code snippet 5-2)

```
@Test

void getOnePostRequestTest(){

    Mono<PostModel> result = client.get()

            .uri(URI.create
("https://jsonplaceholder.typicode.com/posts/1"))

            .accept(MediaType.APPLICATION_JSON)

            .retrieve()

            .bodyToMono(PostModel.class);

    PostModel post = result.block();

    Assertions.assertThat(post)

            .hasFieldOrPropertyWithValue("userId", 1);
}
```

The example code call a remote endpoint and then retrieves a body payload as a Java class. This does not allow to get other data, than payload; so in the case, you need to access a response, use the `toEntity()` method.

(Code snippet 5-3)

```
@Test

void getRequestDataTest(){

    Mono<ResponseEntity<PostModel>> result = client.get()

            .uri(URI.create
("https://jsonplaceholder.typicode.com/posts/1"))

            .accept(MediaType.APPLICATION_JSON)

            .retrieve()

            .toEntity(PostModel.class);

    ResponseEntity<PostModel> response = result.block();

    Assertions.assertThat(response.getStatusCode())

        .isEqualTo(HttpStatus.OK);

    Assertions.assertThat(response.hasBody()).isTrue();
}
```

In both cases we subscribed for a Mono publisher. In a situation, when you expect a sequence of entities, you can subscribe for a Flux with the `bodyToFlux()` function. Please note, that in this case, you will access a Flux, which emits entities and not the response data.

(Code snippet 5-4)

```
@Test

void getPostsTest(){

    Flux<PostModel> result = client.get()

            .uri(URI.create
("https://jsonplaceholder.typicode.com/posts"))

            .accept(MediaType.APPLICATION_JSON)

            .retrieve()

            .bodyToFlux(PostModel.class);

    Iterable<PostModel> posts = result.toIterable();

    Assertions.assertThat(posts).hasSize(100);

}
```

That is about retrieving data using `GET` requests. Another important aspect is how to actually send data to the remote server using `POST` and `PUT` HTTP requests.

Send data

We have mentioned in the previous subsection, that for `PUT` and `POST` requests we can utilize corresponding methods in order to define headers, body payload and URI. Take a look on the following code sample, where we perform a `POST` operation to send a new post entity to the remote API.

(Code snippet 5-5)

```
@Test

void postRequestTest(){

    PostModel payload = new PostModel("body", "title", 101, 101);

    Mono<PostModel> result = client.post()

            .uri(URI.create
("https://jsonplaceholder.typicode.com/posts"))

            .accept(MediaType.APPLICATION_JSON)

            .bodyValue(payload)

            .retrieve()

            .bodyToMono(PostModel.class);

    PostModel post = result.block();
    Assertions.assertThat(post).isEqualTo(payload);

}
```

The difference with previous examples is that we provide a body payload using the `bodyValue()` function. Namely, we can use two main approaches to define a payload, depending on its "availability":

- `bodyValue()` accepts an entity directly
- `body()` accepts a *publisher* that allows to subscribe to Mono that will emit an actual data entity

In other words, I can recommend you to use the `bodyValue()` method if you already have an actual payload (it passed as an argument for your client component), otherwise provide a publisher that will permit to subscribe for a result of an async computation.

Exchange() vs retrieve()

The last point, which I would like to address in this section is a difference between two methods - `exchange()` and `retrieve()`. From a technical point of view, they both are used in order to execute a request, but they return different type of response:

- `Mono<ClientResponse>   exchange()` returns a Mono that allows to subscribe for the `ClientResponse` item. It provides access to the response status and headers, and also methods to consume the response body
- `WebClient.ResponseSpec retrieve()` allows to use the `ResponseSpec` object. This allows to declare how to extract the response

Please note, that this subsection is provided here for general knowledge (or historical purposes). **Starting the Spring version 5.3**, the `exchange()` method is marked as deprecated due to the possibility to leak memory and/or connections. Thefore it is advised to use the `retrieve()` function instead.

Summary

In this section we did an overview of the Webflux WebClient. It is intended to replace blocking `RestTemplate` component, that was used to perform HTTP calls before and relied on Apache HttpClient library. The new component allows to execute requests in non-blocking way and to subscribe to responses with reactive publishers - Mono and Flux.

Chapter 6. Server-side validation with Yavi

In the Spring MVC framework a server-side validation was implemented using annotations. We placed a `@Valid` annotation before the body payload in the controller method and then Spring did all the work for us. On the other side, the Spring Webflux implements a functional programming model, and its web layer is based on router function handlers. This is a non-blocking API and things are different from what we know from older approaches. When we need to validate an input, we actually need to do it manually, as developers can't use Spring MVC web components with router handlers. The validation in Spring Webflux reactive handlers is implemented in a declarative way.

An anatomy of a validator pattern

The main entry point for our API is a validator class. This component performes a bean assertion; in other words, it validates that the supplied entity satisfies some rules. Comparing to an annotation-based approach, these rules are defined not in a bean (POJO), rather, in a validator itself. That brings a better encapsulation and makes entities reusable, because they are free from annotation dependencies.

To start, we create an interface `BaseValidator<T>`, which has a single abstract method - `validate()`. This function returns a Mono publisher which can emit either an instance of the supplied object or an error in case of an invalid input. Take a look on the following code snippet:

(Code snippet 6-1)

```
intreface BaseValidator <T> {

    Mono<T> validate (T data);

}
```

Once the `validate()` method asserts an invalid entity, the pipeline will break up with the `ValidatonException` error. In this section, let take a look on an implementation of this interface with the Yavi library. This is a lambda based type safe validation for Java. It allows to build a declarative validation, instead of an annotation-based validation approach.

Use constraint validations

Yavi supplies a number of built-in *validators*. Each validator contains a number of *constraints* (conditions to check). Following validators are supplied out of the box:

- notNull()
- isNull()
- pattern()

- email()
- url()
- greaterThanOrEqual() / greaterThan()
- lessThanOrEqual() / lessThan()

In this subsection we will implement a sample validator for customer endpoints. The model is valid with following conditions:

- `companyName` field is not empty
- `companyEmail` field is not empty and is a valid email address
- `taxId` field is not empty and is valid VAT format

For the last point, we can use the `pattern()` method and supply the regular expression. First step is to obtain a validator instance, that will be used to perform validations behind the scenes.

(Code snippet 6-2)

```
public class CustomerValidator implements BaseValidator<CustomerModel>
{

    private final Validator<CustomerModel> validator;

    public CustomerValidator(){

        validator = ValidatorBuilder.of(CustomerModel.class)

                .constraint(CustomerModel::getCompanyEmail,
"companyEmail", c-> c.notNull().email())

                .constraint(CustomerModel::getCompanyName,
"companyName", c -> c.notNull())
```

```
                .constraint(CustomerModel::getTaxId, "taxId", c ->
c.pattern(""))

                .build();

    }

    //...

}
```

Call the validator

Once we obtained the validator, we can use it to perform validations on the entity. The Yavi library has the `isValid()` method, which helps us to determine if the input is valid or not. In the first case we just pass the data forward into a reactive stream to the next subscriber. Otherwise, we raise an error, which can be used later for an error handling.

(Code snippet 6-3)

```
@Override

public Mono<CustomerModel> validate(CustomerModel model) {

    ConstraintViolations violations = validator.validate(model);

    if (violations.isValid()) {

        return Mono.just(model);

    } else {
        return Mono.error(new
ValidationException(violations.violations()));

    }

}
```

Use validators in router functions

We already discussed, that Spring Webflux reactive router functions do not allow to use Spring MVC web components, including validation annotations. This means, that developers have to implement this task manually. The validation component is incorporated into a general handler pipeline. Consider the following router function:

(Code snippet 6-4)

```
Mono<ServerResponse> putCustomer(ServerRequest request) {

    return request.bodyToMono(CustomerModel.class)

            .flatMap(customerService::updateCustomer)

            .flatMap(result -> ServerResponse

                    .ok()

                    .contentType(MediaType.APPLICATION_JSON)

                    .bodyValue(result));
}
```

In order to use the validator here, we need to accomplish several steps:

- Create a new `CustomerValidator` instance
- Call the `CustomerValidator::validate` on the body as a first action
- Use the `onErrorResume` method to intercept and to handle possible validation errors

The modified implementation is presented in the code snippet below:

(Code snippet 6-5)

```
Mono<ServerResponse> putCustomer(ServerRequest request) {

    CustomerValidator validator = new CustomerValidator();

    return request.bodyToMono(CustomerModel.class)

            .flatMap(validator::validate)

            .flatMap(customerService::updateCustomer)

            .flatMap(result -> ServerResponse

                    .ok()

                    .contentType(MediaType.APPLICATION_JSON)

                    .bodyValue(result))

            .onErrorResume(err ->
ServerResponse.badRequest().build());

}
```

Summary

Spring Webflux provides two approaches to build a web layer: with annotated controllers and router functions. The first architecture allows to use Spring MVC components, including annotation-based validations. The later approach does not have this support and developers have to implement this functionality on their own. The Yavi library is a solution, that helps to build validations pipelines in a declarative way and to avoid using heavy validation-based techniques.

Chapter 7. CORS handling

If you develop web applications (especially apps, which interact with remote APIs), you saw errors in the browser console about CORS. That is because your web browser by default does not permit you to load resources, coming outside the same origin. You need to tell it, that it is safe - and this is implemented using CORS (Cross-Origin-Resource-Sharing) protocol.

Technically, the server application has to write certain response headers, telling to the browser, that the origin is trusted and is allowed to get resources from the server. CORS is a powerful security mechanism, which can include or exclude certain methods, headers or even MIME types from the access by the certain origin and you can use it as a big part of your security policy. Spring Webflux makes it is really easy to configure your CORS policy. In this post we will examine how to do it, when you use **reactive handler functions** at your web layer (which is a bit different from how it is handled in Spring MVC).

What does CORS mean?

Imagine, that you are working on an web application, that consists of two separate components: REST API and front end app. This client application communicates with the remote server by sending HTTP requests (for example, using fetch). From a technical point of view, these parts reside on different *domains* (e.g. *domain A* for client, *domain B* for server), and the web browser does not permit you to access resources from the *different* origin. So, in other words, you need to tell the browser, that the server's origin is safe.

This task is implemented using CORS (stands for Cross Origin Resource Sharing) mechanism. In a nutshell it relies on specific headers, which are attached by the server, and specify *who* can access that resources.

Moreover, CORS protocol allows to define not only the recevier itself (origin), but also can oermit or reject certain HTTP methods, MIME types or headers. Developers also can use CORS requests to inform clients, what types of credentials are required (e.g. Basic authentication or cookies).

Types of CORS requests

The standard defines two types of CORS requests: simple requests and so-called preflighted requests. Basically, we can say that all requests, that do not meet criteria for simple requests are preflighted requests. That got their name due to the fact, that they require a client to send *preflight* `OPTIONS` request before executing requests in order to check the server's security policy.

In this subection we will briefly overview both types of CORS requests. As we are interested more in how it is done in Spring, I do a brief explanation of concepts. I really encourage you to check an actual standard to get a solid understanding of CORS protocol.

Simple requests

From a technical point of view, not all requests ask frontend apps to send a checking request. If the HTTP request meets certain requirements, it is considered as simple and can be used directly. Although, simple requests also put some limitations, that we will also observe later.

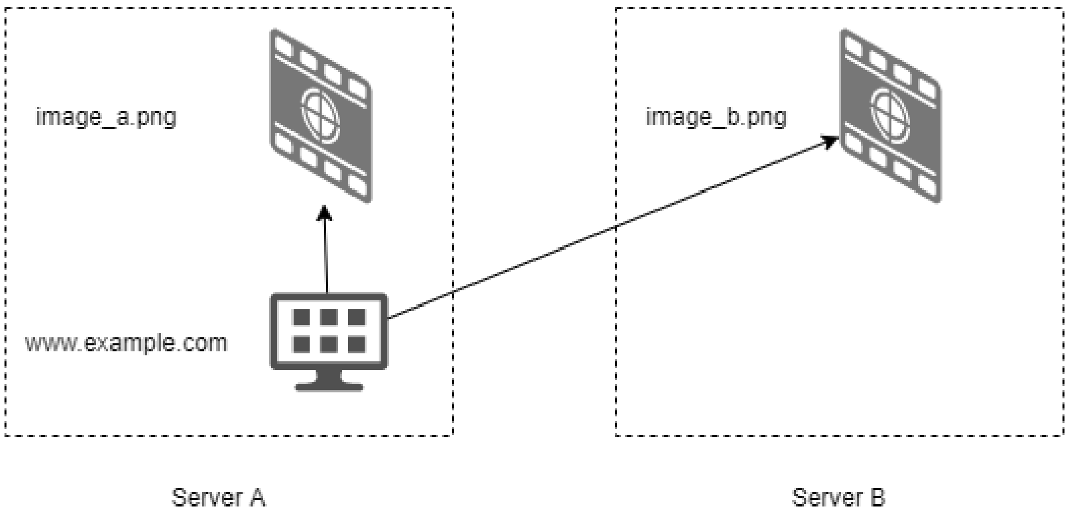
So, what kind of criteria will make a simple request? They can be summarized into three categories: method, content-type and headers:

- Methods: only `GET`, `POST` or `HEAD`
- For the content type you can utilize only certain values: `application/x-www-form-urlencoded`, `multipart/form-data` and `text/plain`
- Apart of headers, that are set automatically (for example, `Accept-Charset`, `Content-Length` or other headers, that are known in a standard

as a forbidden header name, you can use only those headers, which are listed as safelisted headers. This includes `Accept`, `Accept-Language`, `Content-Language` and some others.

Basically all cross-origin mechanism for simple requests requires you to just put an `Origin` header (which is placed automatically). The server validates the origin and added to the response an another header - `Access-Control-Allow-Origin`. Its value can be either same as `Origin` or `*` (that is not allowed if the request requires cookies). Otherwise, the response is considered as an error. Take a look on the graph below, which represents a process of simple requests:

(Image 7-1)



Besides these listed requirements, simple requests place a number of limitations. First of all, this does not allow you to work with REST API, as you can not send/receive JSON data or to use JWT tokens as an authentication measure. Moreover, if your application uses custom headers, they are also prohibited.

The second issue is that the server also limits types of headers it can write to the response. For example, if you send a file at the response you need to remember, that the `Content-Length` header is not placed by the server. So, if you want to measure a download progress on the client side, you need to explicitly allow this header. It can be done by writing `Access-Control-Expose-Headers` header's value by the server.

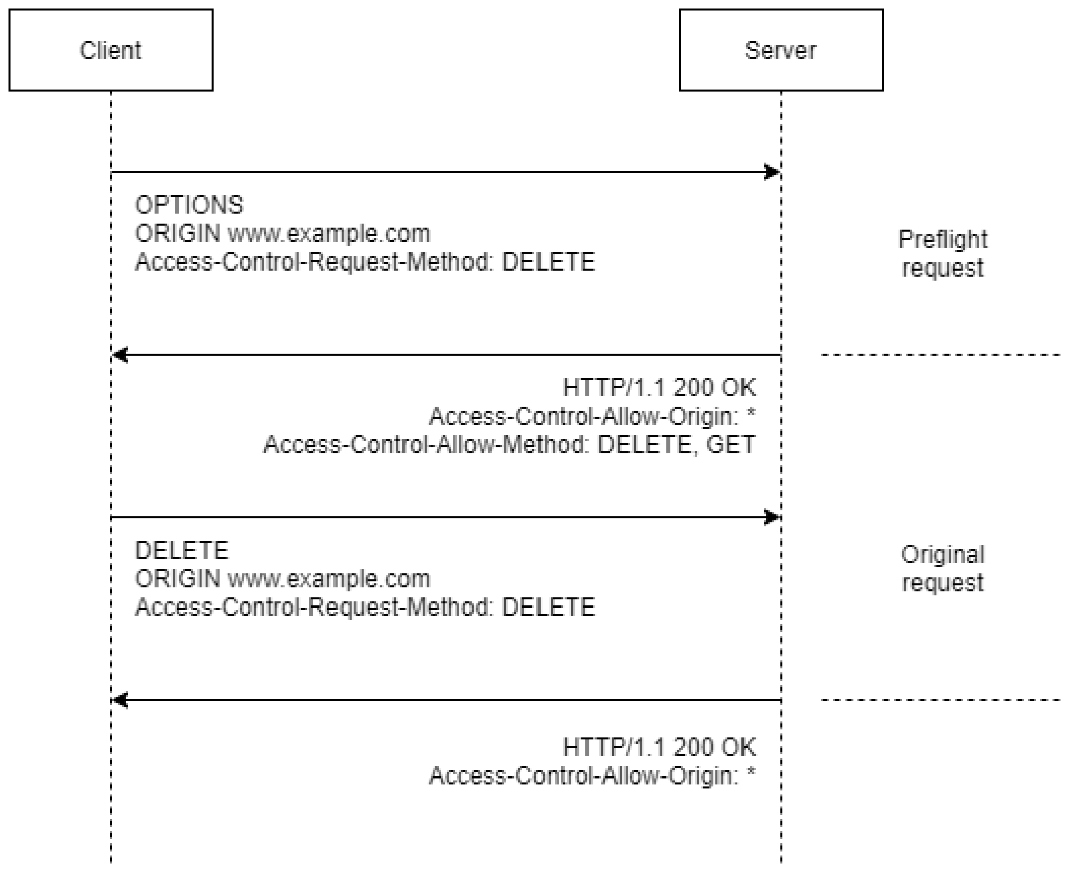
In other words, if you want to access other functionality (and for REST API you basically do not have a choice) you need to consider *preflighted* CORS requests. We will observe them in the next subsection.

Preflight requests

As we already discussed, this kind of requests received its name, because it requires a client to execute a *preflight* request in order to check the security policy. This can be done using the `OPTIONS` method in order to assert that the actual request is safe to execute. The server in its turn will validate this "pre-request" and either allows of disallows the actual one.

Take a look on this diagram below, which shows a process of the preflighted requests:

(Image 7-2)


Client
Server
OPTIONS
ORIGIN www.example.com
Access-Control-Request-Method: DELETE
Preflight request
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Access-Control-Allow-Method: DELETE, GET
DELETE
ORIGIN www.example.com
Access-Control-Request-Method: DELETE
Original request
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *

From a technical point of view, the preliminary `OPTIONS` request "asks" the server for allowed for the actual request methods (in the `Accept-Control-Request-Method` header) and headers (`Access-Control-Request-Headers`). In its turn, the server writes the list of allowed methods (inside the `Access-Control-Allow-Methods` header), allowed headers (in the `Access-Control-Allow-Header`). Finally, the server has to specify an allowed origin (`Access-Control-Allow-Origin`) - as in the case of simple requests it can be either * (any origin - you should not use it unless you write a public API) or a fully qualified name of the client.

That is a helicopter view on the CORS protocol. Due to the fact we concentrate on how it is implemented on the server side (particularly with Spring Webflux), I have omitted the browser side code. In any way, I recommend you to check the original standard as it is written in a quite understandle language and it will help you to get a deeper understanding. So, that was a theoretical introduction, and now we move to the practical side.

How to configure CORS in Spring Webflux

From a technical point of view, the actual way of a CORS configuration depends on which type of web layer architecture do you use. The difference is that for reactive handlers is used the CorsWebFilter. If you stick with reactive REST controllers you can either use the `@CrossOrigin` annotation (is not covered in this book) or the WebFluxConfigurer component.

Approach 1. Using WebFluxConfigurer

This approach is generally used when you create your web layer with controllers. Reactive controllers allow to use same techniques with Webflux, like with Spring MVC, to name them:

- `@CrossOrigin` annotation-based configuration
- `WebFluxConfigurer` component which basically a Webflux-variant of a global CORS configuration

We will not cover the first approach in this book, as it is similar with how it is done in Spring MVC. Rather, I would like to demonstrate you the usage of the `WebFluxConfigurer` class.

In order to use it, we create a new `@Configuration` component, that implements the `WebFluxConfigurer` interface. This contract defines callback methods to customize the configuration for WebFlux applications. The component should be also annotated with the `@EnableWebFlux` annotation. Take a look on the code snippet below:

(Code snippet 7-1)

```
@Configuration

@EnableWebFlux

class CorsConfiguration implements WebFluxConfigurer {

    @Override

    public void addCorsMappings(CorsRegistry corsRegistry) {

        corsRegistry.addMapping("/**")

          .allowedOrigins("http://localhost:8080")

          .allowedMethods("POST", "PUT", "DELETE", "GET")

          .allowedHeaders("X-USER-ID");

    }

}
```

Let break down this flow. The interface specify a method, that deals with the `CorsRegistry` reference. This reference allows to add `CorsRegistration` for defined URL pattern. Please note, that origins, methods and headers are specified as varargs.

Approach 2. Using CorsWebFilter

The recommended way to handle CORS configuration with functional reactive handlers is to use the `CorsWebFilter` component. From a technical point of view, we need to register a corresponding bean. To create it we accomplish following steps:

- Create a new `CorsConfiguration` instance
- Specify allowed origins as list of strings
- Specify allowed methods as list of strings
- Specify allowed headers as list of strings
- Create a new `UrlBasedCorsConfigurationSource` instance
- Register the configuration for the url pattern
- Create a new `CorsWebFilter` with the given source instance

Take a look on this sample implementation:

(Code snippet 7-2)

```
@Bean

CorsWebFilter corsWebFilter(){

    CorsConfiguration configuration = new CorsConfiguration();

    configuration.setAllowedOrigins(List.of("http://localhost:8080"));

    configuration.setAllowedMethods(List.of("POST", "GET", "PUT",
"DELETE"));

    configuration.setAllowedHeaders(List.of("X-USER-ID"));

    UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();

    source.registerCorsConfiguration("/**", configuration);
    return new CorsWebFilter(source);

}
```

Please note, that out-of-the-box the `CorsConfiguration` class is restrictive, which means that we need to specify explicitly *what is allowed*, rather than *what is not allowed*. However, it possible to create a new configuration with default values:

(Code snippet 7-3)

```
CorsConfiguration configuration = new CorsConfiguration();

configuration.applyPermitDefaultValues()

//... use this config
```

This approach can be used with controllers too, but cannot be combined with `@CrossOrigin` annotations.

Summary

The CORS mechanism is a header-based protocol, that is a powerful part of your application’s security policy. In order to implement it, your server have to expose a certain CORS configuration for clients. If you are using Spring Webflux framework you can follow an approach that depends on your architectural preferences - controller-based or router-based.

Chapter 8. Error handling

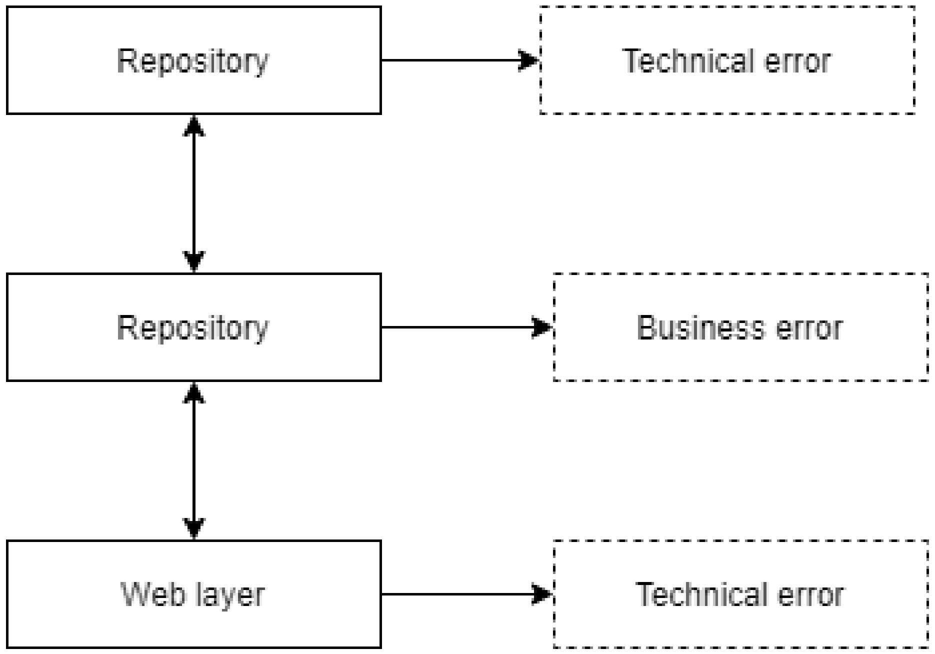
The topic of error handling in web applications is very important. From a client perspective it is essential to know on how was the request proceeded and in case of any error is crucial to provide to the client a valid reason, especially if the error was due to the client's actions. There are different situations, when notifying callers about concrete reasons is important - think about server-side validations, business logic errors that come due to bad requests or simple not found situations.

The mechanism of error handling in Webflux is different, from what we know from Spring MVC. Core building blocks of reactive apps - `Mono` and `Flux` bring another way to deal with error situations, and while old exception-based error handling still may work for some cases, it violates Spring Webflux nature. In this chapter will will do an overview of how to process errors in Webflux when it comes to business errors and absent data. We will not cover technical errors, as they are handled by Spring framework.

When do we need to handle errors in Webflux

Before we will move to the subject of error handling, let define what we want to achieve. Assume, that we build application using a conventional architecture with a vertical separation by layers and a horizontal separation by domains. That means, that our app consists of three main layers: *repositories* (one that handle data access), *services* (one that do custom business logic) and *handlers* (to work with HTTP requests/responses; understand them as controllers in Spring MVC). Take a look on the graph below and you can note that **potentially** errors may occur on any layer:

(Image 8-1)



Although, I need to clarify here, that from a technical perspective errors are not *same*. On the repository level (and let abstract here also *clients*, that deal with external APIs and all data-access components) usually occur what is called *technical errors*. For instance, something can be wrong with a database, so the repository component will throw an error. This error is a subclass of `RuntimeException` and if we use Spring is handled by framework, so we don't need to do something here. It will result to 500 error code.

An another case is what we called *business errors*. This is a violation of custom business rules of your app. While it may be considered as not as a good idea to have such errors, they are unavoidable, because, as it was mentioned before, we **have to** provide a meaningful response to clients in case of such violations. If you will return to the graph, you will note, that such errors usually occur on the service level, therefore we have to deal with them.

Now, let see how to handle errors and provide error responses in Spring Webflux APIs.

Start from business logic

Let take an example of the authentication API, which is organized from repositories, services and handlers. As it was already mentioned, *business errors* take place inside a service. Prior to reactive Webflux, we often used *exception-based* error handling. It means that you provide a custom runtime exception (a subclass of `ResponseStatusException`) which is mapped with specific http status.

However, Webflux approach is different. Main building blocks are `Mono` and `Flux` components, that are chained throughout an app's flow (note, from here I refer to both `Mono` and `Flux` as `Mono`). Throwing an exception on any level will break an async nature. Also, Spring reactive repositories, such as `ReactiveMongoRepository` don't use exceptions to indicate an error situation. `Mono` container provides a functionality to propagate error condition and empty condition:

- `Mono.empty()` = this static method creates a `Mono` container that completes without emitting any item
- `Mono.error()` = this static method creates a `Mono` container that terminates with an error immediately after being subscribed to

With this knowledge, we can now design a hypothetical login or signup flow to be able to handle situations, when 1) an entity is absent and 2) error occured. If an error occurs on the repository level, Spring handles it by returning `Mono` with error state. When the requested data is not found - empty `Mono`. We also can add some validation for business rules inside the *service*. Take a look on the refactored code of signup flow from this post:

(Code snippet 8-1)

```
@Override
```

```
public Mono<SignupResponse> signup(SignupRequest request) {

    String email = request.getEmail().trim().toLowerCase();

    String password = request.getPassword();

    String salt = BCrypt.gensalt();

    String hash = BCrypt.hashpw(password, salt);

    User user = new User(null, email, hash, salt);

    return repository.findByEmail(email)

            .defaultIfEmpty(user)

            .flatMap(result -> {

                if (result.getUserId() == null) {

                    return repository.save(result).flatMap(result2 ->
{

                        String userId = result2.getUserId();

                        String token =
tokenManager.issueToken(userId);

                        SignupResponse signupResponse = new
SignupResponse(userId, token, secret);

                        return Mono.just(signupResponse);
                    });

                } else {

                    return Mono.error(new AlreadyExistsException());

                }

            });

}
```

Now, we have a business logic. The next phase is to map results as HTTP responses on the *handler* level.

Display http responses in handlers

This level can correspond to the old good *controllers* in Spring MVC. The purpose of *handlers* is to work with HTTP requests and responses and by this to connect a business logic with the outer world. In the most simple implementation, the handler for signup process looks like this:

(Code snippet 8-2)

```
// signup handler

Mono<ServerResponse> signup (ServerRequest request){

    Mono<SignupRequest> body =
request.bodyToMono(SignupRequest.class);
```

```
    Mono<SignupResponse> result = body.flatMap(service::signup);

    return ServerResponse.ok().contentType(json).body(result,
SignupResponse.class);

}
```

This code does essential things:

- Accepts a body payload from the HTTP request
- Call a business logic component
- Returns a result as HTTP response

However, it does not takes an advantage of custom error handling, that we talked about in the previous section. We need to handle a situation, when user already exists. For that, let refactor this code block:

(Code snippet 8-3)

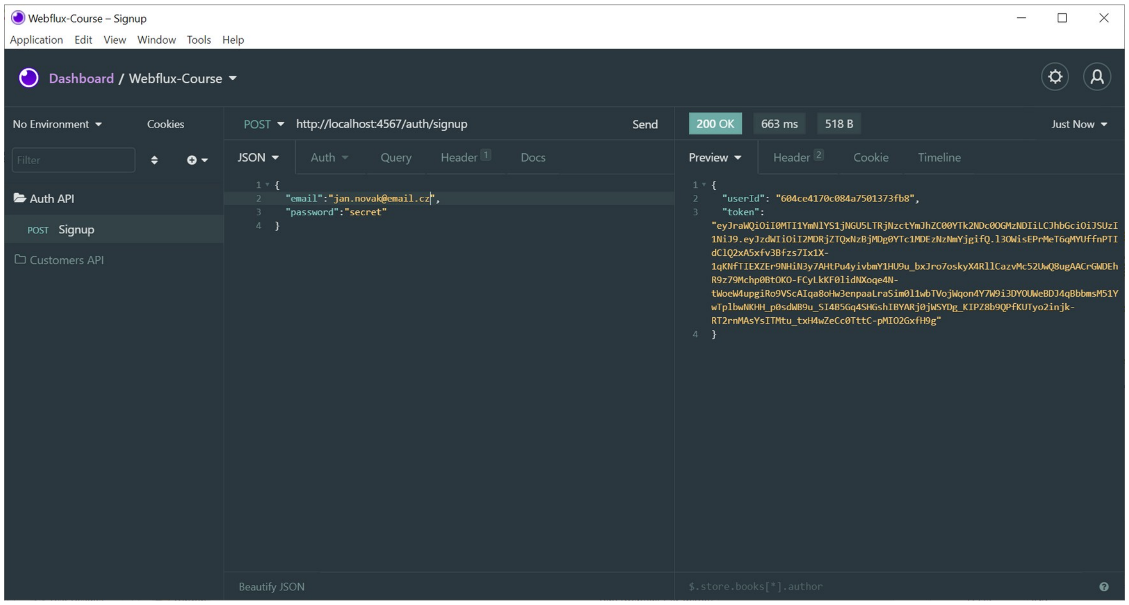
```
// signup handler refactored

Mono<ServerResponse> signup (ServerRequest request){

    Mono<SignupRequest> body =
request.bodyToMono(SignupRequest.class);

    Mono<SignupResponse> result = body.flatMap(service::signup);

    return result.flatMap(data ->
ServerResponse.ok().contentType(json).bodyValue(data))

            .onErrorResume(error ->
ServerResponse.badRequest().build());

}
```

Note, that compare to the previous post, I use here `bodyValue()` instead of `body()`. This is because `body` method actually accepts producers, while `data` object here is entity (`SignupResponse`). For that I use `bodyValue()` with passed value data.

In this code we can specify to the client, what was a reason of the error. If user does exist already in database, we will provide `409` error code to the caller, so she/he have to use an another email address for signup procedure. That is what is about *business errors*. For *technical errors* our API will output the `500` error code.

We can validate, that when we create a new user, everything works ok and the expected result is `200` success code:

(Image 8-2)



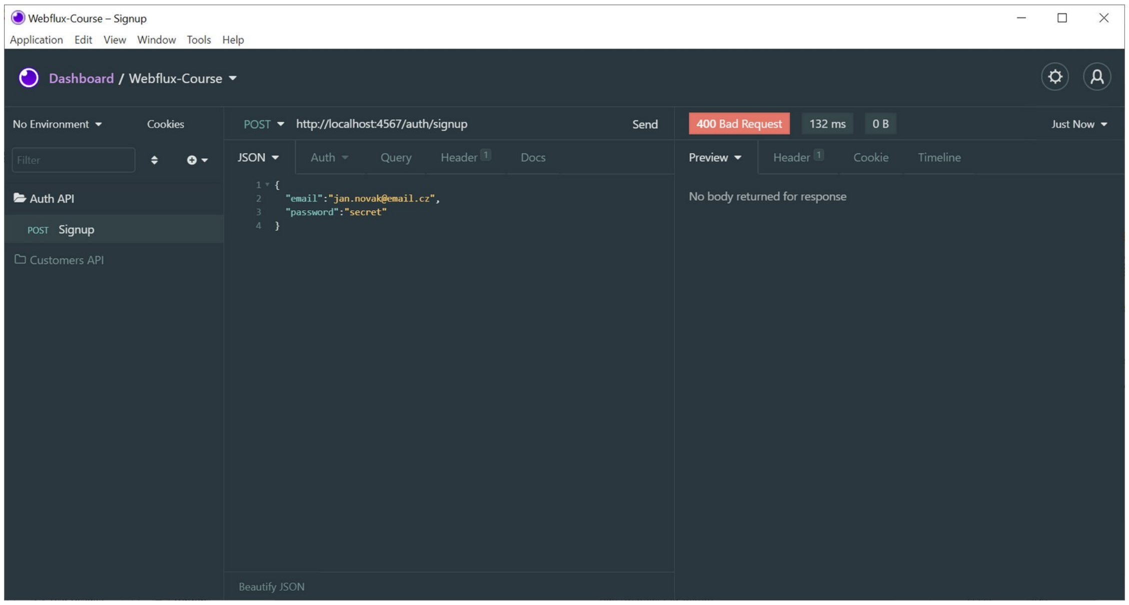
POST http://localhost:4567/auth/signup

```
{
  "email":"jan.novak@email.cz",
  "password":"secret"
}
```

```
{
  "userId": "604ce4170c084a7501373fb8",
  "token":
"eyJraWQiOiI0MTI1YmNlYS1jNGU5LTRjNzctYmJhZC00YTk2NDc0OGMzNDIiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiI2MDRjZTQxNzBjMDg0YTc1MDEzNzNmYjgifQ.l3OWisEPrMeT6qMYUffnPTIdClQ2xA5xfv3Bfzs7Ix1X-1qKNfTIEXZEr9NHiN3y7AHtPu4yivbmY1HU9u_bxJro7oskyX4RllCazvMc52UwQ8ugAACrGWDEhR9z79Mchp0BtOKO-FCyLkKF0lidNXoqe4N-tWoeW4upgiRo9VScAIqa8oHw3enpaaLraSim0l1wbTVojWqon4Y7W9i3DYOUWeBDJ4qBbbmsM51YwTplbwNKHH_p0sdWB9u_SI4B5Gq4SHGshIBYARj0jWSYDg_KIPZ8b9QPfKUTyo2injk-RT2rnMAsYsITMtu_txH4wZeCc0TttC-pMIO2GxfH9g"
}
```

On the other hand, if you will try to signup with the **same** email address, API should response with `409` error code, like it is shown on the screenshot below:

(Image 8-3)



POST http://localhost:4567/auth/signup

```
{
  "email":"jan.novak@email.cz",
  "password":"secret"
}
```

400 Bad Request 132 ms 0 B

No body returned for response

Moreover, we don't need `success` field for `SignupResponse` entity anymore, as unsucessful signup is handled with error codes. There is an another situation, I want to mention is this post - the problem of empty responses. This is what we would observe on a login example.

Special case: response on empty result

Why this is a special case? Well, technically, empty response is not an error, neither business error or technical error. There are different opinions among developers how to handle it properly. I think, that even it is not an exception in a normal sence, we still need to expose `404` error code to the client, to demonstrate an absence of the requested information.

Let have a look on the login flow. For login flow is common a situation, opposite to the signup flow. For signup flow we have to ensure, that *user* *does not exist yet,* however for login we have to know that *user does already exist*. In the case of the absence we need to return an error response.

Take a look on the `login` handler initial implementation:

(Code snippet 8-4)

```
Mono<ServerResponse> login (ServerRequest request){

    Mono<LoginRequest> body = request.bodyToMono(LoginRequest.class);

    Mono<LoginResponse> result = body.flatMap(service::login);

    return ServerResponse.ok().contentType(json).body(result,
LoginResponse.class);

}
```

From the service component's perspective there could be three scenarios:

- User exists and login is successful = return `LoginResponse`
- User exists but login was denied = return an error
- User does not exist = return an *empty* user

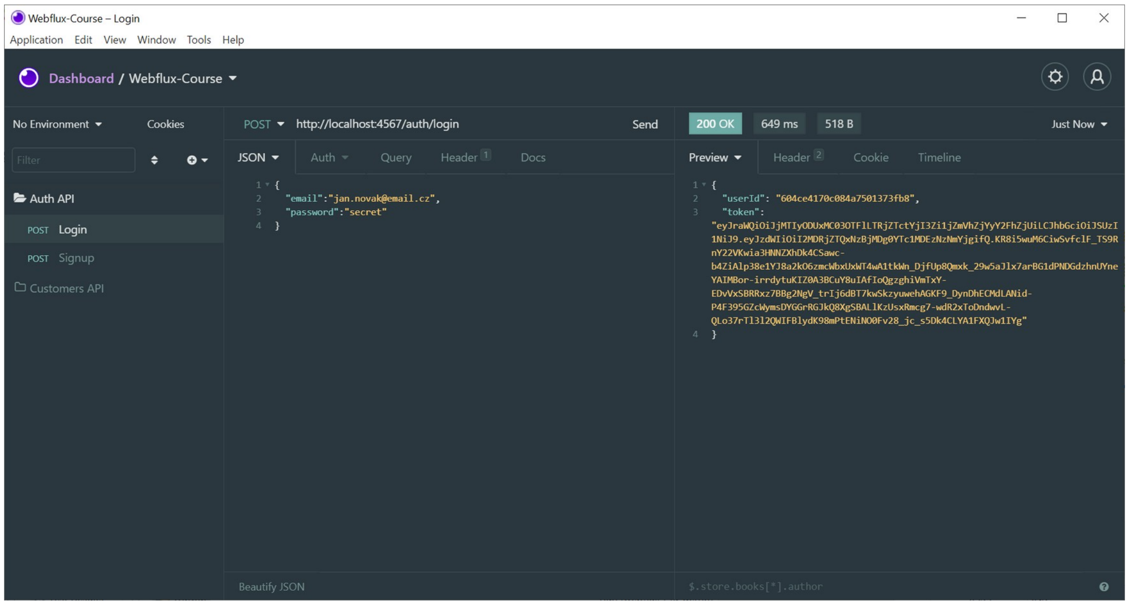
We have already seen how to work with errors in handlers. The empty response situation is managed using `switchIfEmpty` method. Take a look on the refactored implementation:

(Code snippet 8-5)

```
Mono<ServerResponse> login (ServerRequest request){

    Mono<LoginRequest> body = request.bodyToMono(LoginRequest.class);
    Mono<LoginResponse> result = body.flatMap(service::login);

    return result.flatMap(data ->
ServerResponse.ok().contentType(json).bodyValue(data))

            .switchIfEmpty(ServerResponse.notFound().build())

            .onErrorResume(error -> {

                if (error instanceof LoginDeniedException){

                    return ServerResponse.badRequest().build();

                }

                return ServerResponse.status(500).build();

            });

}
```

Note, that unlike `onErrorResume` method, `switchIfEmpty` accepts as an argument an alternative `Mono`, rather than function. Now, let check that everything works as expected. The login for existed user entity and valid credentials should return a valid response:

(Image 8-4)

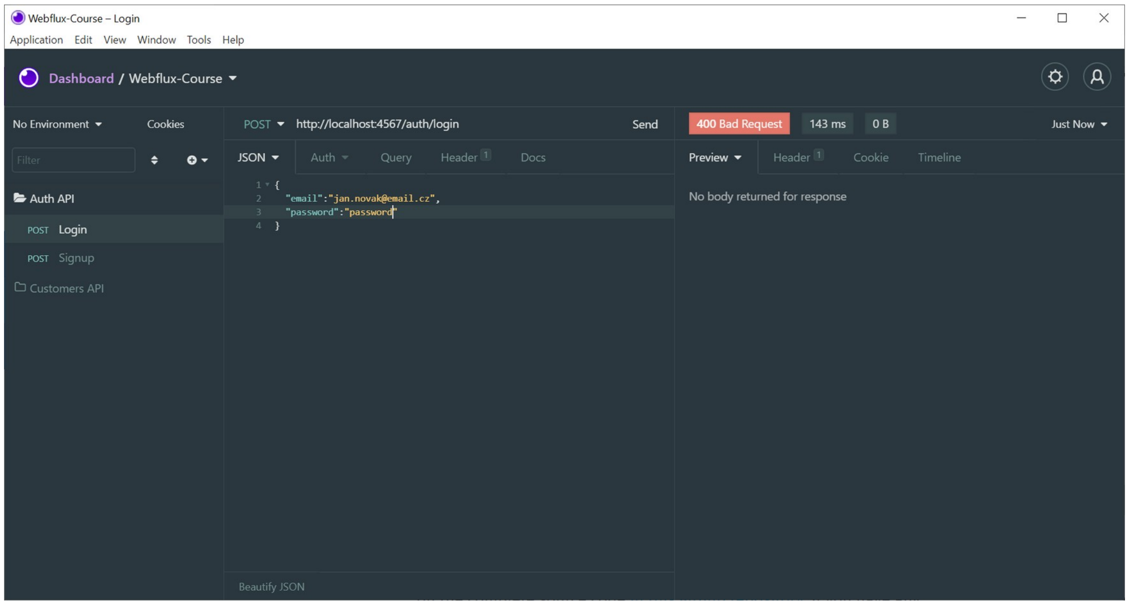


```
POST http://localhost:4567/auth/login
{
  "email":"jan.novak@email.cz",
  "password":"secret"
}
```

```
200 OK  649 ms  518 B
{
  "userId": "604ce4170c084a7501373fb8",
  "token":
"eyJraWQiOiJjMTIyODUxMC03OTFlLTRjZTctYjI3Zi1jZmVhZjYyY2FhZjUiLCJhbGciOiJSUzI
1NiJ9.eyJzdWIiOiI2MDRjZTQxNzBjMDg0YTc1MDEzNzNmYjgifQ.KR8i5wuM6CiwSvfclF_TS9R
nY22VKwia3HNNZXhDk4CSawc-
b4ZiAlp38e1YJ8a2kO6zmcWbxUxWT4wA1tkWn_DjfUp8Qmxk_29w5aJlx7arBG1dPNDGdzhnUYne
YAIMBor-irrdytuKIZ0A3BCuY8uIAfIoQgzghiVmTxY-
EDvVxSBRRxz7BBg2NgV_trIj6dBT7kwSkzyuwehAGKF9_DynDhECMdLANid-
P4F395GZcWymsDYGGrRGJkQ8XgSBALlKzUsxRmcg7-wdR2xToDndwvL-
QLo37rTl3l2QWIFBlydK98mPtENiNO0Fv28_jc_s5Dk4CLYA1FXQJw1IYg"
}
```

If credentials are wrong, but user does exist (case no.2), we will obtain a `Bad request` error code:

(Image 8-5)

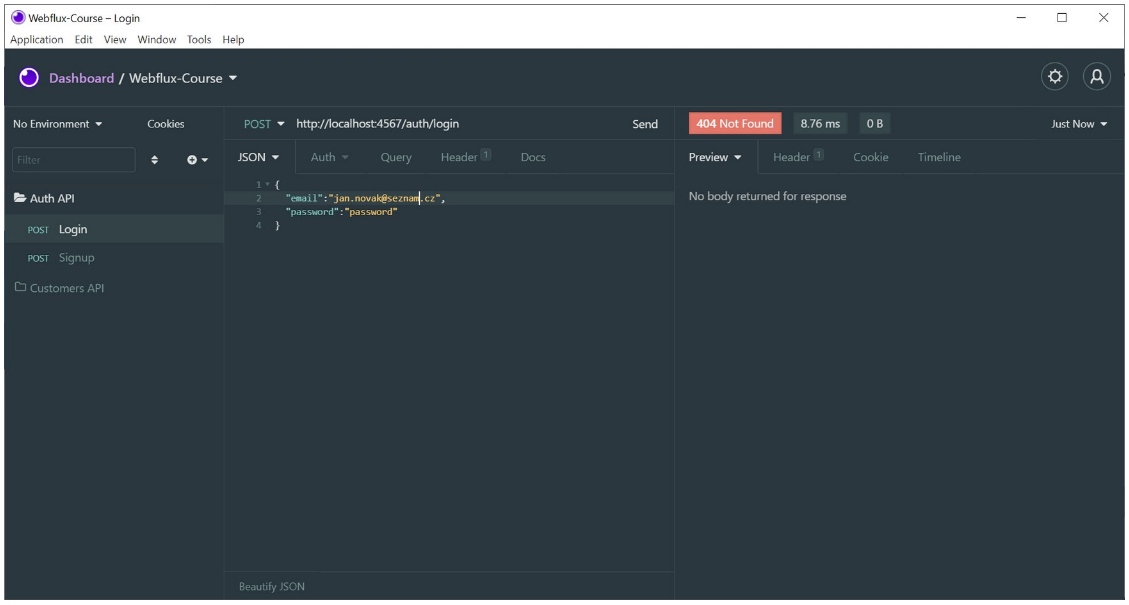


```
POST http://localhost:4567/auth/login
{
  "email":"jan.novak@email.cz",
  "password":"password"
}
```

```
400 Bad Request  143 ms  0 B
No body returned for response
```

Finally, if repository is unable to find a user entity, handler will answer with `Not found`:

(Image 8-6)



POST http://localhost:4567/auth/login

```
{
  "email":"jan.novak@seznam.cz",
  "password":"password"
}
```

404 Not Found 8.76 ms 0 B

No body returned for response

Summary

In this chapter we did an overview of an error handling on the level of router functions. This means, that if you prefer to use controller as a foundation of your web layer, you are supplied with all tools, that are familiar for you from Spring MVC. However, doing error handling with router functions will require to perform a bit more manual work to create it.

The way of error handling in Webflux is different, from what we know from Spring MVC. Core building blocks of reactive apps - `Mono` and `Flux` have a special way to deal with error situations, and while old exception-based error handling still may work for some cases, it violates Spring Webflux nature.

Chapter 9. Authentication patterns in Spring Webflux

In this chapter we will cover important aspects of dealing with authentication and authorization in Spring Webflux-based applications. While, developers can rely on Spring Security project, it may seem a bit overcomplicated for most use cases. Here we will observe security techniques using only Spring Webflux.

Traditionally, Spring MVC operated a concept of Webfilters. They can be still used in annotation-based controllers, written with Webflux. However, if you work with functional-style reactive routes, you need to use the `HandlerFilterFunction` component. From a technical point of view, the `HandlerFilterFunction` class represents a function that filters a handler function. We place it to the given router function in order to apply it before the request will go to the handler.

We will see how to use these handler filter functions in order to implement a token-based authentication (JWT) and a two-factor authentication (TOTP).

Creating a handler filter function

First logical step is to implement a new filter function, because it is an interface. A typical implementation should define `filter()` method, which, according to docs, applies this filter to the given handler function. Take a look on the code snippet below:

(Code snippet 9-1)

```
@Override

public Mono<ServerResponse> filter(ServerRequest request,
                                    HandlerFunction<ServerResponse>
handlerFunction) {

    List<String> headers = request.headers().header("Authorization");

    if (headers.isEmpty()) return ServerResponse.status(403).build();

    String token = headers.get(0);

    return authService.parseToken(token)

            .flatMap(res -> handlerFunction.handle(request))

            .onErrorResume(err -> ServerResponse.status(403).build());

}
```

Here we will do following things:

- Access headers. Note, that Webflux actually returns a list of headers on key
- Validate that `Authorization` header is not empty
- Get the first item in the list
- Call `AuthService.parseToken` method, which returns a `Mono` object with userId (on successful result) or an error (in case of bad or invalid token)

Another thing to note, here is how `filter()` method moves a request next by the pipeline. The return type is `Mono<ServerResponse>`. In a failed outcome, it terminates an execution of a handler, by returning finished server response. In case of success, it applies handler function, which represents the next entity in the chain.

Applying handler filter functions

Once we defined an auth function, we need to apply it to given routes. I created a simple `ProtectedHandler` component, that actually does not contain any valuable logc, but we need to have some protected route to demonstrate. Take a look on my implementation below.

(Code snippet 9-2)

```
@Bean

public RouterFunction<ServerResponse> protectedEndpoint
(ProtectedHandler handler){

    return RouterFunctions

            .route(GET("/secured/hello").and(accept(json)),
handler::sayHello);

}
```

Let apply our function to it. For that, you need to:

- Inject auth function as an argument to the bean method
- Call `RouterFunctions.filter` method at the end and pass here auth function

Check the another code snippet:

(Code snippet 9-3)

```
@Bean
```

```
public RouterFunction<ServerResponse> protectedEndpoint
(ProtectedHandler handler,

                                                               AuthFunction
authFunction){

    return RouterFunctions

            .route(GET("/secured/hello").and(accept(json)),
handler::sayHello)

            .filter(authFunction::filter);

}
```

Now, being prepared with foundational tools, we can see, how to implement token-based authentication. From a technical point of view, these patterns utilize handler filter functions to intercept incoming requests, to extract a token data and to proccess it (in other words, to validate it and to check caller's authorization rights).

JWT authentication in Spring Webflux

In this subsection we will implement a basic token authentication flow with just "core" Spring Webflux - without using Spring Security. We need to have two additional dependencies - the one is to work with tokens (generating, asserting) and the another to hash passwords.

Important note: I need to advise you against using MD5 or SHA functions to hash password data or to generate secure salts. The

acceptable practice is to utilize BCrypt; in case of Java you need to use the jBcrypt library or a similar project.

In this book, we will use the Nimbus JOSE-JWT library in order to work with tokens. However, with a proper software design, you should abstract implementations from contracts and you can change the concrete implementation to your favorite JWT library.

Before we will dive to an actual code, let understand concepts behind it. From a technical point of view, the JWT model takes two main duties to log in a user:

- Generate a token
- Validate a token

We can break this process into isolated steps:

- The client sends the `POST` request, that contains an username (email) and a password in a body payload to the server
- The server performs the logic to check the received data against a data in a database
- In the success case, the server sends to the client a new JWT token. Otherwise, the server returns an error code
- To call protected endpoints, the client needs to attach the token as an `Authorization` header
- The server validates the token using a handler filter function and allows or denies an access to a secured resource

This logic can be encapsulated into two main components: `AuthService` and `TokenManager`. Let describe a functionality of each part:

- `AuthService` - is a component, that encapsulates all business logic, related to authentication/authorization, including signup, login and token validation

- `TokenManager` - is a component, that abstracts a code to generate and validate JWT tokens. This makes main business logic independent from concrete JWT implementations.

The next step is to implement these contracts using Spring Webflux concepts. In this chapter we will create both signup and login flows. Let start with the `AuthServiceImpl` class.

Signup a new user

In this section we will complete a signup process, that involves following steps:

- Get signup request from client
- Check that user does not exists already
- Hash a password with a salt value
- Store user in a database
- Issue a token
- Return a response with user ID and token

The main component `AuthServiceImpl::signup` accepts `SignupRequest` and returns `SignupResponse`. Behind the scenes, it is responsible for whole signup logic. First, take a look on its implementation below.

(Code snippet 9-4)

```
@Override

public Mono<SignupResponse> signup(SignupRequest request) {

    // generating a new user entity params

    // step 1
    String email = request.getEmail().trim().toLowerCase();

    String password = request.getPassword();

    String salt = BCrypt.gensalt();

    String hash = BCrypt.hashpw(password, salt);

    User user = new User(null, email, hash, salt);

    // preparing a Mono

    Mono<SignupResponse> response = repository.findByEmail(email)

            .defaultIfEmpty(user) // step 2

            .flatMap(result -> {

                // assert, that user does not exist

                // step 3

                if (result.getUserId() == null) {

                    // step 4

                    return repository.save(result).flatMap(result2 ->
{

                        // prepare token
                        // step 5

                        String userId = result2.getUserId();

                        SignupResponse signupResponse = new
SignupResponse(userId);

                        return Mono.just(signupResponse);

                    });

                } else {

                    // step 6

                    // scenario - user already exists

                    return Mono.error(new UserExistsException());

                }

            });

    return response;

}
```

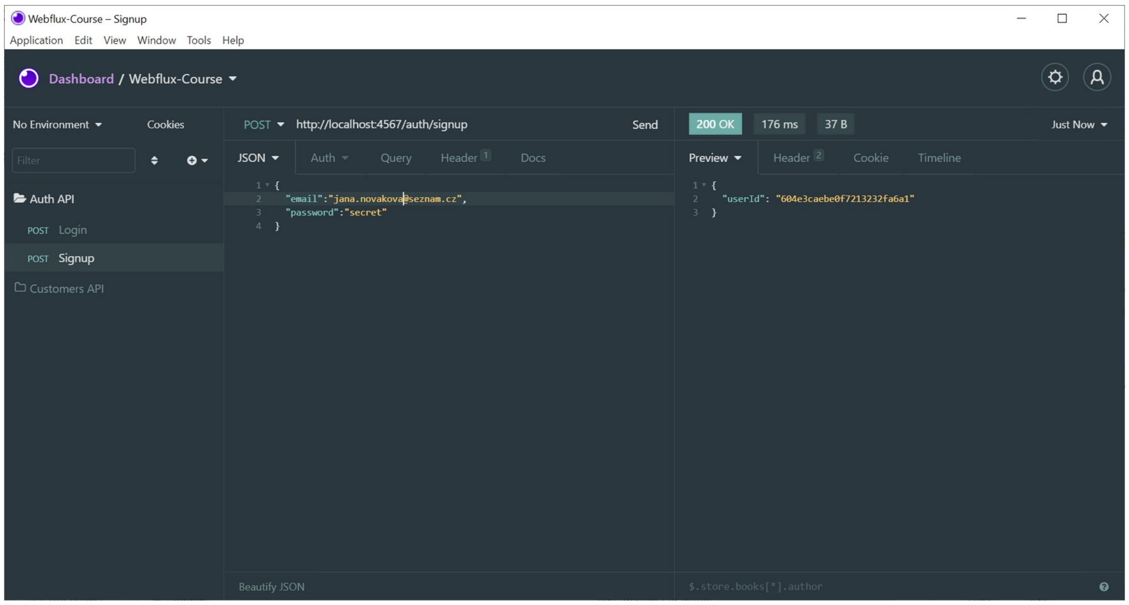
Now, let go step by step through my implementation. Basically, we can two possible scenarions with signup process: user is new and we sign it up or it is already presented in database, so we have to reject it.

Consider these steps:

- We create a new user entity from the request data
- Provide the new entity as *default*, if user does not exist
- Assert the repository's call result
- Save a user in the database and obtain a `userId`
- Return a rejecting response, if user does already exist

After the signup logic was implemented, we can validate that everything works as expected. First, let call `signup` endpoint to create a new user. Result object contains an ID value and a token:

(Image 9-1)

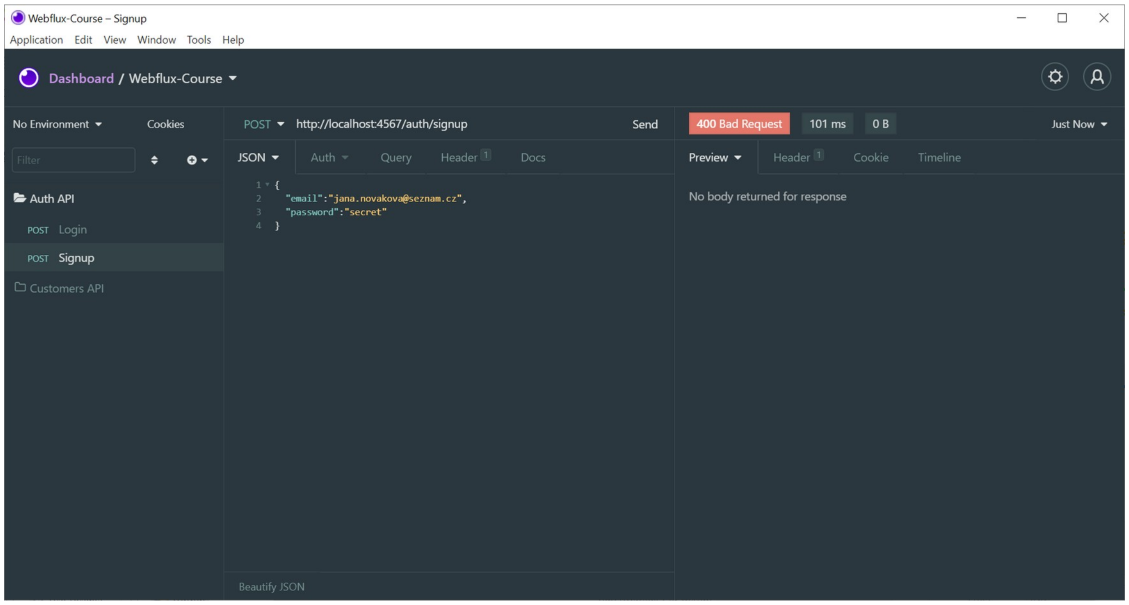


```
POST http://localhost:4567/auth/signup
{
  "email":"jana.novakova@seznam.cz",
  "password":"secret"
}
```

```
200 OK   176 ms   37 B
{
  "userId": "604e3caebe0f7213232fa6a1"
}
```

However, we should not be allowed to signup twice with the **same** email. Let try this case to assert, that app actually looks for the existed email before creating a new user:

(Image 9-2)



POST http://localhost:4567/auth/signup

```
{
  "email":"jana.novakova@seznam.cz",
  "password":"secret"
}
```

400 Bad Request 101 ms 0 B

No body returned for response

The next section will guide you to a login flow implementation

Logging in

As in the previous sub section, I start with presenting required steps. For log in it would be:

- Get a login request from a client
- Find a user in a databse
- Asserting the existing password with supplied at a request
- Return a login response with a token

Main business logic is implemeted in `AuthServiceImpl::login`. It does the majority of work. First we need to find a user by an email address in a database, otherwise we provide a *default* value that has null fields. The condition `user.getUserId() == null` means that user does not exist and a log in flow should be aborted.

Next, we need to assert that passwords do match. As we stored password hash in a database, we need first to hash a password from the request payload with a stored salt and then asserts both values:

The successful result of validation is followed by a generation of a token and creation of a `LoginResponse` instance. The final source code for this part is presented to you below:

(Code snippet 9-5)

```
@Override

public Mono<LoginResponse> login(LoginRequest request) {

    String email = request.getEmail().trim().toLowerCase();

    String password = request.getPassword();

    String code = request.getCode();

    Mono<LoginResponse> response = repository.findByEmail(email)

    // step 1

            .defaultIfEmpty(new User())

            .flatMap(user -> {

                // step 2

                if (user.getUserId() == null) {

                    // no user

                    return Mono.error(mew UserNotFoundException());
                } else {

                    // step 3

                    // user exists

                    String salt = user.getSalt();

                    String secret = user.getSecretKey();

                    boolean passwordMatch = BCrypt.hashpw(password,
salt).equalsIgnoreCase(user.getHash());

                    // step 4

                    // password matched

                    if (passwordMatch) {

                        // step 5

                        String token =
tokenManager.issueToken(user.getUserId());

                        LoginResponse loginResponse = new
LoginResponse(userId, token);

                    } else {

                        return Mono.error(mew
AccessDeniedException());
```

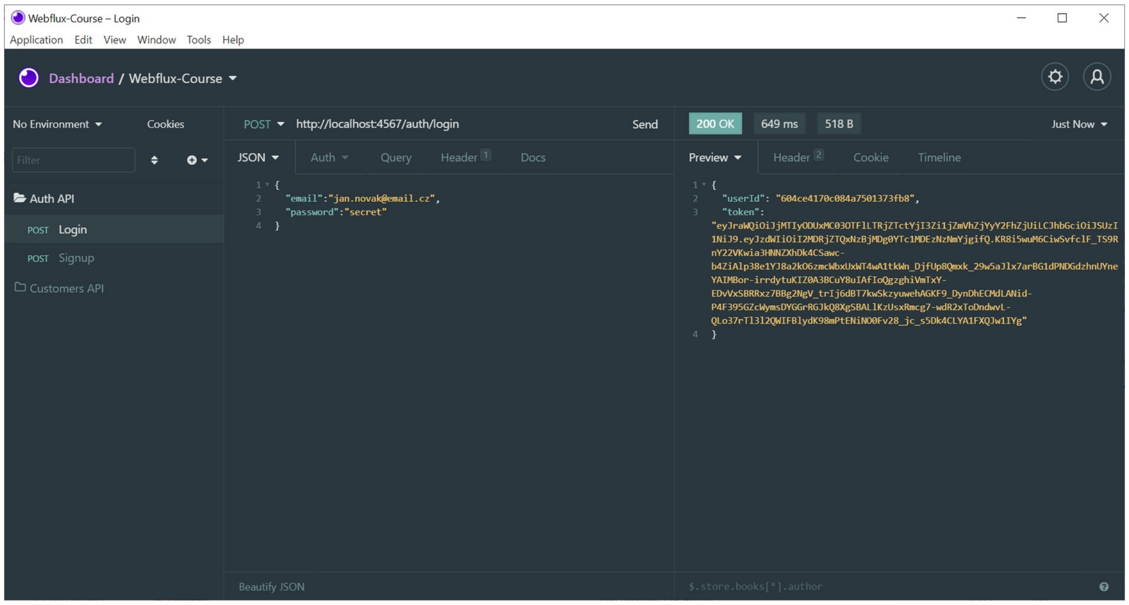
```
                        }
                    }
                });
        return response;
}
```

So, let observe steps:

- Provide a default user entity with null fields
- Check that the user **does** exist
- Produce a hash string of a password from request and a salt value, that stored in a database
- Assert that passwords do match
- Issue a token

Finally, we can do a testing to verify that the login process works as we planned. Let call `login` endpoint with login request payload.

(Image 9-3)



POST http://localhost:4567/auth/login

```
{
  "email":"jan.novak@email.cz",
  "password":"secret"
}
```

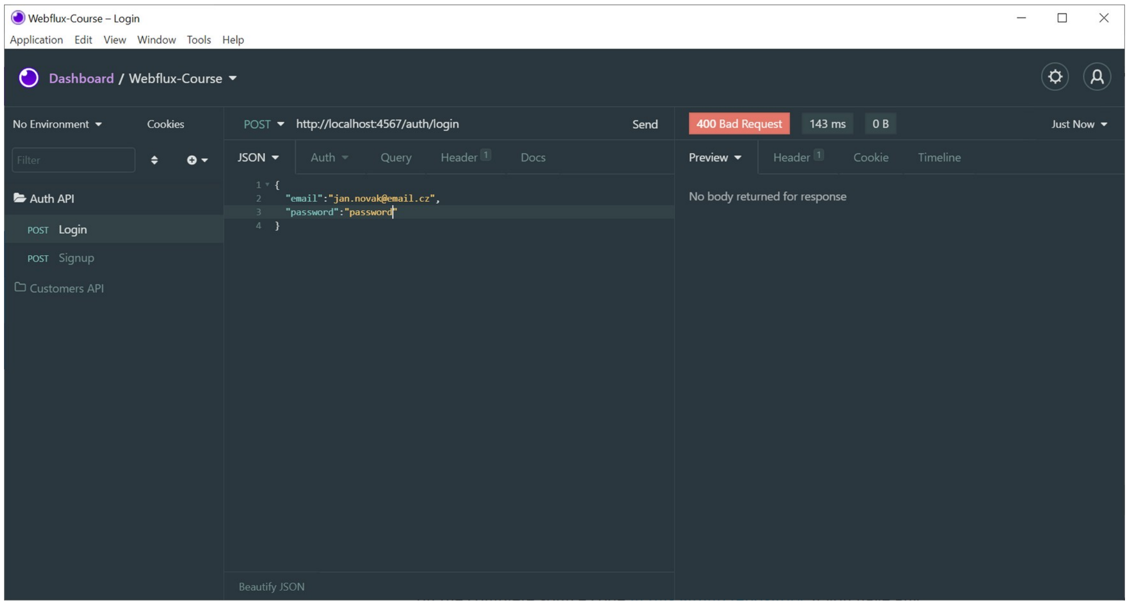
200 OK 649 ms 518 B Just Now

```
{
  "userId": "604ce4170c084a7501373fb8",
  "token":
  "eyJraWQiOiJjMTIyODUxMC03OTFlLTRjZTctYjI3Zi1jZmVhZjYyY2FhZjUiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiI2MDRjZTQxNzBjMDg0YTc1MDEzNzNmYjgifQ.KR8i5wuM6CiwSvfclF_TS9RnY22VKwia3HNNZXhDk4CSawc-b4ZiAlp38e1YJ8a2kO6zmcWbxUxWT4wA1tkWn_DjfUp8Qmxk_29w5aJlx7arBG1dPNDGdzhnUYneYAIMBor-irrdytuKIZ0A3BCuY8uIAfIoQgzghiVmTxY-EDvVxSBRRxz7BBg2NgV_trIj6dBT7kwSkzyuwehAGKF9_DynDhECMdLANid-P4F395GZcWymsDYGGrRGJkQ8XgSBALlKzUsxRmcg7-wdR2xToDndwvL-QLo37rTl3l2QWIFBlydK98mPtENiNO0Fv28_jc_s5Dk4CLYA1FXQJw1IYg"
}
```

$.store.books[*].author

An another case to check is the wrong password.

(Image 9-4)



POST http://localhost:4567/auth/login

```
{
  "email":"jan.novak@email.cz",
  "password":"password"
}
```

400 Bad Request 143 ms 0 B Just Now

No body returned for response

Another topic to address in this chapter is a two-factor authentication (also called a multi factor authentication or MFA). A multi factor authentication became a common practice for many applications, especially for enterprise ones, or those that deal with sensetive data (for example, finance apps). Moreover, MFA is enforced by law for a number of industries, and if you are working on the application, that by some requirement, has to enable two-factor auth in some way, you may need to implement such technique.

Multi factor authentication in Spring Webflux

The two-factor authentication (or the multi-factor authentication) can be described as **a security process, where users have to provide two or more factors to verify themselves**. That means, that usually user supplies alongside a password also an another factor. It can be an one-time password, hardware tokens, biometric factors (like fingerprints) or other factors.

From a technical point of view, such practice requires to accomplish several steps:

- User enters email (username) and password
- Alongside credentials, user submits one-time code, generated by an authenticator app
- App authenticates email (username), password and verifies one-time code, using user's secret key, issued during signup process

That means, that usage of authetnicator apps (like Google Authenticator, Microsoft Authenticator, FreeOTP etc) brings a number of advantages, compare to an usage of SMS for code delivery. They are not sensetive for SIM attacks and can work without a cell or an internet connection.

We can easily base a multi factor authentication implementation upon the core JWT solution from the previous subsection. The only difference is to create a component that is used to generate user's secret and to assert supplied short codes. For that, I utilize the java-totp library. In the sample implementation this contract is called `TotpManager`.

Signup a new user

The signup flow is based on what we have observed already. However, we also use a `TotpManager` component in order to generate a secret key and return it to a user. It should be used in an authenticator application.

(Code snippet 9-6)

```java
@Override

public Mono<MFASignupResponse> signupMFA(SignupRequest request) {

    // generating a new user entity params

    String email = request.getEmail().trim().toLowerCase();

    String password = request.getPassword();

    String salt = BCrypt.gensalt();

    String hash = BCrypt.hashpw(password, salt);

    String secret = totpManager.generateSecret();

    MFAUser user = new MFAUser(null, email, hash, salt, secret);

    // preparing a Mono

    Mono<MFASignupResponse> response = repository.findByEmail(email)

            .defaultIfEmpty(user) // step 2
            .flatMap(result -> {

                // assert, that user does not exist

                if (result.getUserId() == null) {

                    return repository.save(result).flatMap(result2 ->
{

                        // prepare token

                        String userId = result2.getUserId();

                        MFASignupResponse signupResponse = new
MFASignupResponse(userId, secret);

                        return Mono.just(signupResponse);

                    });

                } else {

                    // scenario - user already exists

                    return Mono.error(new UserExistsException());

                }

            });

        return response;
```

```
}
```

Next, we need to implement a function to generate new secret key. It is abstracted inside `TotpManager.generateSecret()`. Take a look on the code below:

(Code snippet 9-7)

```
@Override

public String generateSecret() {

    SecretGenerator generator = new DefaultSecretGenerator();

    return generator.generate();

}
```

Logging in

In the log in flow we need to add some changes. That way, if passwords do match, we need to verify a one-time code using the secret key, which we saved in a database.

(Code snippet 9-8)

```
@Override

public Mono<LoginResponse> loginMFA(MFALoginRequest request) {

    String email = request.getEmail().trim().toLowerCase();
    String password = request.getPassword();

    String code = request.getCode();

    Mono<MFALoginResponse> response = repository.findByEmail(email)

            .defaultIfEmpty(new MFAUser())

            .flatMap(user -> {

                if (user.getUserId() == null) {

                    // no user

                    return Mono.error(new UserNotFoundException());

                } else {

                    // user exists

                    String salt = user.getSalt();

                    String secret = user.getSecretKey();

                    boolean passwordMatch = BCrypt.hashpw(password,
salt).equalsIgnoreCase(user.getHash());

                    if (passwordMatch) {

                        // step 4
                        // password matched

                        boolean codeMatched =
totpManager.validateCode(code, secret);

                        if (codeMatched) {

                            // step 5

                            String token =
tokenManager.issueToken(user.getUserId());

                            LoginResponse loginResponse = new
LoginResponse(usreId, token);

                            return Mono.just(loginResponse);

                        } else {

                            return Mono.error(new
AccessDeniedException());

                        }

                    } else {

                        return Mono.error(new
UserNotFoundException());

                    }

                }
```

```
            });

    return response;

}
```

To validate one time codes, generated by apps, we have to provide for TOTP library both code and secret, that we saved as a part of user entity. Take a look on the implementation below:

(Code snippet 9-9)

```
@Override

public boolean validateCode(String code, String secret) {

    TimeProvider timeProvider = new SystemTimeProvider();

    CodeGenerator codeGenerator = new DefaultCodeGenerator();

    CodeVerifier verifier = new DefaultCodeVerifier(codeGenerator,
timeProvider);

    return verifier.isValidCode(secret, code);

}
```

Therefore, it is possible to verify now the log in proccess using an one time code, generated by an authenticator application from the secret key. Without the one time or with a wrong value, the log in flow would be aborted with an error. By using these blueprints, readers can successfully implement own solutions for other authentication factors.

Client side security with Vue.js

Being a full stack software engineer, I strongly believe, that it is necessary to include in the book, devoted to back end development, also chapters on front end development. Because, your APIs do not exist separately, they are always called by some clients of various types (web, mobile or other web services). In next chapters we will observe how to work with Spring Webflux REST APIs from Vue js applications. Vue js is a good choice, as this framework allows to bootstrap applications very quickly, similar to Spring Boot, and in a same time, it gives a strong level of control for developers.

In this sub section we will observe how to implement a login flow and a signup flow for Spring Webflux API in an application, written with Vue.js.

Create an AuthService

This component encapsulates an actual logic, that calls HTTP endpoints. Because in this book we are focused on Spring Webflux, we will keep our JS code as simple as possible. Here, we utilize the Axios library to perform HTTP calls. We have two endpoints: the one for login and the one for signup.

(Code snippet 9-10)

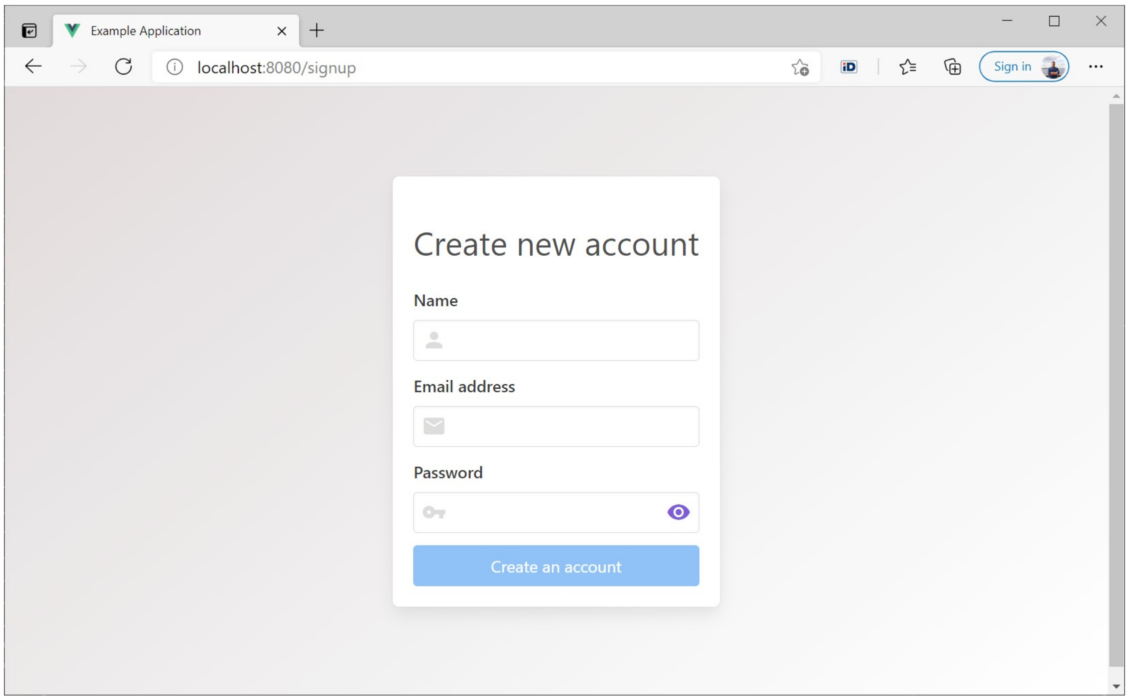
```
import axios from 'axios'

export default {
    login(data){
        return axios.post('login', data)
    },
    signup(data){
        return axios.post('signup', data)
    }
}
```

Please note, that these methods takes only endpoint names, without a fully qualified URL. We can set a base url for Axios using the `axios.defaults.baseUrl` property.

Implement a signup flow

In this subsection we will focus on a signup process. We will omit an actual layout creation, as we are focused on a business logic. Consider an example signup flow, that is presented on the screenshot below:

(Image 9-5)



On a front end side, in order to signup, we need to accomplish following steps:

- Take a data from a user through a signup form. Note, that we need to validate a correctness of data (e.g. all fields are filled etc) and we can do it either manually (for small forms) or with Vue validation libraries, such as Vuelidate. In this code example we use Vuelidate library

- Send a data to a server using the `AuthService.signup()` method
- Assert results. In a success case we redirect a user to a login page or in a case of an error we show an error message

Take a look on the implementation of this sequence of steps.

(Code snippet 9-11)

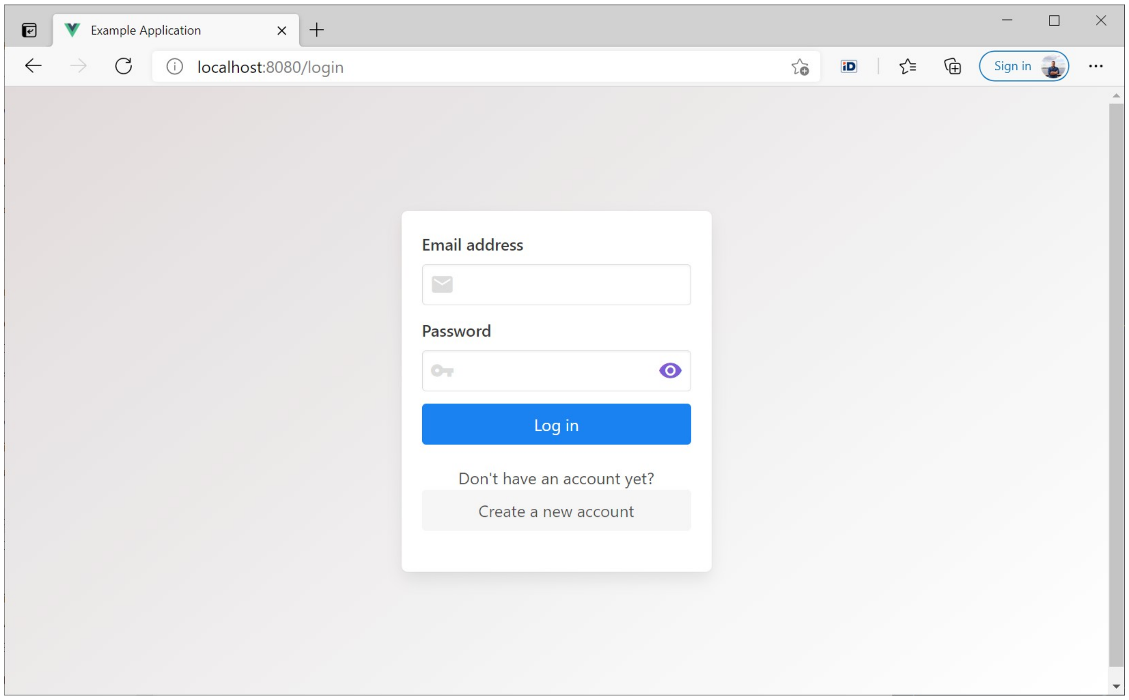
```
import AuthService from '@/services/AuthService'
import {required, email, minLength} from 'vuelidate/lib/validators'

export default {
  data(){
    return {
      signupRequest: {
        email: '',
        password: '',
        name: ''
      },
      isError: false,
      isLoading: false
    }
  },
  validations: {
    signupRequest: {
      email: {required, email},
      password: {required, minLength: minLength(8)},
      name: {required}
    }
  },
  methods:{
    handleSignup(){
      this.isLoading = true
      AuthService.signup(this.signupRequest).then(result => {
        this.isLoading = false
        this.isError = false
        this.$router.push('/login')
      }).catch(e => {
        console.log(e)
        this.isLoading = false
        this.isError = true
      })
    }
  }
}
```

Once the new entity was created on the backend side, we need to redirect a user to a login page.

Implement a login flow

The login process allows a user to use her credentials in order to get a token, that then would be attached to calls to protected endpoints. Likewise, with the signup flow, we will concentrate on a code part here, and will skip an actual form's design. You can observe an example login form below:

(Image 9-6)



Let review which steps do we need to perform in order to get a token from a server:

- Accept user credentials using a login form. We need to validate a correctness of a form using either manually written logic or third-party libraries. We assume that data is valid
- Send a data to a server with the `AuthService.login()` method
- Assert a response

In a success result we need to store a token and a user ID to re-use it later in calls to protected endpoints. Generally, we can do in several ways:

- Use local storage (either directly or from LocalForage)
- Use Vuex state management
- Specify a default authorization header for Axios

In our example we will keep things simple and will use a local storage. Check a following implementation of a login flow below:

(Code snippet 9-12)

```
import AuthService from '@/services/AuthService'

export default {
  data(){
    return {
      loginRequest: {
        email: '',
        password: ''
      },
      isError: false,
      isLoading: false
    }
  },
  methods: {
    handleLogin(){
      this.isLoading = true
      AuthService.login(this.loginRequest).then(result => {
        this.isLoading = false
        this.isError = false
        let credentials = result.data
        localStorage.setItem('userId', credentials.userId)
        localStorage.setItem('token', credentials.token)
        location.reload()
      }).catch(e => {
        console.log(e)
        this.isLoading = false
        this.isError = true
      })
    }
  }
}
```

Please note, that after a successful authorization, we reload a page using the `location.reload()` method.

Using tokens with protected endpoints

The final aspect we would observe in this subsection is how to use an obtained token during calls for protected endpoints. From a technical point of view, this depends on how do you store tokens on a client side. We have already mentioned that there are numerous ways. In the previous subsection we use a local storage to save a token and a userId. In this case we can use it inside service calls like following:

- Get a token from a local storage
- Create a configuration object with an `Authorization` header
- Use a configuration object with an Axios call

Consider that we have a service, that calls a protected endpoint to remove a project. In this case we need to call a `DELETE` endpoint with a project ID and a security token.

(Code snippet 9-13)

```
deleteProject(id){
    let token = localStorage.getItem('token')
    let config = {
        headers: {
            'Authorization': 'JWT ' + token
        }
    }
    return axios.delete(`project/${id}`, config)
}
```

When we also have a payload to send to a server (like in `POST` and `PUT`), we specify a config object as a **third argument**. In other words, if we create a new project we will do it like following:

(Code snippet 9-14)

```
saveProject(data){
    let token = localStorage.getItem('token')
    let config = {
        headers: {
            'Authorization': 'JWT ' + token
        }
    }
    if (data.projectId !== null) {
        let id = data.projectId
        return axios.put(`project/${id}`, data, config)
    } else {
        let userId = localStorage.getItem('userId')
        data.userId = userId
        return axios.post(`projects/${userId}`, data, config)
    }
}
```

Please note, that as we have mentioned before, an actual way to use tokens depends on how did you store credentials in a client application.

Summary

The Spring Security project is a part of the Spring ecosystem, that is responsible for dealing with authentication and authorization tasks. However, this approach has some drawbacks. It is not a hard task to implement same logic with just "core" Spring Webflux, which will be more flexible. To do that, developers need to understand how to use handler filter functions. In this big chapter we have observed two most common security patterns for reactive micro services: the token based security with JWT and the multi factor security with one time codes.

Chapter 10. Unit testing in Spring Webflux

Software testing is an important aspect of the development in general. In this chapter, I would like to highlight ways of how to perform unit testing of components, as services and router functions in reactive manner. To do that, you will need to utilize Project Reactor test library, which allows to test reactive publishers.

Unit tests serve to verify individual components, mostly on business logic level, and to ensure that components perform as expected. In Spring Webflux apps this level is a level of services. Here we want to check that the defined flow is executed properly. To do this we usually use a testing framework, like JUnit and abstract (isolate) component's dependencies with mocks to control the service under test.

What is unit testing?

Unit testing is a level of software testing, when we test individual software's components in isolation. For example, we have `UserService`, that may have various connected dependencies: `UserRepository` component to connect with a datasource, `EmailClient` to communicate with remote API that will send confirmation emails. For unit testing, we isolate `UserService` and then *mock* external dependencies.

The unit testing offers us a number of benefits, to name few:

- It increases our confidence, when we change code. If unit tests are good written and if they are run every time any code is changed, we can notice any failures, when we introduce new features
- It serves as documentation. Certanly, documenting your code includes several instruments, and unit testing is one of them - it describes an *expected* behaviour of your code to other developers.

- It makes your code more *reusable*, as for good unit tests, code components should be modular.

These advantages are just few of numerous, that are provided us by unit testing. Now, when we defined what is unit testing and why do we use it, we can move to how it works with Spring Webflux.

Introduction to Project Reactor Test

In previous sections we did a broad overview of an usage of reactive types `Mono` and `Flux`, however, we demonstrated them in quite naive way. The Project Reactor framework has a special extension to write unit tests that assert these reactive components.

The core element here is the `StepVerifier` class. It provides a declarative way of creating a verifiable script for an async `Publisher` sequence, by expressing expectations about the events that will happen upon subscription. Basically, this definition can be translated into the following pipeline:

- Create a new `StepVerifier` for the given `Publisher` (can be Mono or Flux type) using the static method `create()`
- Provide *values assertions* that will expect the publisher to emit a certain value
- It possible to subscribe for certain events, such as success, error or cancellation using respected callback functions on the `StepVerifier`
- Finally, trigger the testing scenario with the `verify()` method (optionally, can be used an overriden version, that takes a duration value) or the `verifyComplete()` method

Let take a look into the sample code snippet, where I demonstrate a principle of work of the `StepVerifier` component:

(Code snippet 10-1)

```java
class UserService {

    Mono<User> findUserByEmail (String email);

    //.. other methods

}

class UserServiceTest {

    private UserService service;

    @Test

    void findUserByEmailTest(){

        final String email = "jana.novakova@seznam.cz";

        StepVertifier.create(service.findUserByEmail(email))

            .assertNext(value ->

                Assertions.assertThat(result).isNotNull())

            .verifyComplete();

    }

}
```

Surely, the provided example is quite naive, yet it gives you an idea of how reactive tests work. Now, let move into more practical things and see how to write unit tests for Spring Webflux services.

Writing a service testing scenario

In the Spring world, a service is a component that encapsulates a business logic. In certain way, we can say that this is what distinguishes your application from others. The practice is to isolate "technical" dependencies (which usually stands for the data layer) out of services. When we write test cases, we basically mock such external dependencies using one of mocking libraries (in case of Spring, it is Mockito).

If you are familiar with JUnit 5 and Mockito libraries, you will like how the Spring framework makes it easier to set up everything to write unit tests, compare to "bare metal" JUnit5/Mockito. From a technical perspective, to set everything up we need:

- Create a mock external dependency (repository)
- Initialize a service under test with an injected dependency

Classicaly, we do this inside the `setup()` method, but with Spring we just need to utilize two annotations:

(Code snippet 10-2)

```
@ExtendWith(MockitoExtension.class)

class CustomerServiceImplTest {

    @Mock

    private CustomerRepository repository;
    @InjectMocks

    private CustomerServiceImpl service;

//...

}
```

For the goal 1 we use a `@Mock` annotation and for the goal 2 we use an `@InjectMocks` annotation. Also, we need to extend our test case with the Mockito extension to make things work. *NB*, that some old guides will point you to the `@RunWith` annotation, but it is outdated for JUnit 5.

Now we are ready to write our first test case. To start, let assert the "create" logic (from CRUD). We will verify, that the service under test does correctly create a new customer entity.

The logic flow includes following steps:

- Get a customer entity from the external caller (as an argument)
- Save it within the data layer
- Return back to the caller a new entity (with a generated ID)

So, in our test case we need to:

- Create a "dummy" entity
- Specify a behaviour for the mock repository
- Assert results

Let have a look on how it can be implemented in practice:

(Code snippet 10-3)

```
@Test
```

```
void createCustomerTest(){

    final CustomerModel customer = new CustomerModel("customerId",
"Acme s.r.o", "jana.dvorakova@acme.cz", "CZ1234567", null, null);


Mockito.when(repository.save(customer)).thenReturn(Mono.just(customer)
);

StepVerifier.create(service.createCustomer(customer))

    .assertNext(result ->
Assertions.assertThat(result).isEqualTo(customer))

    .verifyComplete();

}
```

In the next step we will do a testing of a "retrieve" logic (as of CRUD), however we need to say that this is divided into two separate blocks:

- Find an individual entity
- Find all entities

Let start from the end and verify the flow of retrieving all entites. Here we need to just to propagate result from the data layer, that can emit 0 to many entities (Flux).

- Create a "list" of entities
- Specify a behavior of mock dependency
- Assert results

Take a look on my implementation below:

(Code snippet 10-4)

```
@Test

void findAllCustomersTest(){

        List<CustomerModel> customers = List.of(

                new CustomerModel("customerId1", "Acme s.r.o",
"jana.dvorakova@acme.cz", "CZ1234567", null, null),

                new CustomerModel("customerId2", "Acme s.r.o",
"jana.dvorakova@acme.cz", "CZ1234567", null, null),

                new CustomerModel("customerId3", "Acme s.r.o",
"jana.dvorakova@acme.cz", "CZ1234567", null, null),

                new CustomerModel("customerId4", "Acme s.r.o",
"jana.dvorakova@acme.cz", "CZ1234567", null, null),

                new CustomerModel("customerId5", "Acme s.r.o",
"jana.dvorakova@acme.cz", "CZ1234567", null, null)

        );


Mockito.when(repository.findAll()).thenReturn(Flux.fromIterable(custom
ers));
StepVerifier.create(service.findAllCustomers()).expectNextCount(5).ver
ifyComplete();

}
```

Opposite, when we retrieve a *single* entity we may face two outcomes:

- An entity exists and the result is a Mono which emits that entity
- An entity does not exist and the result is an empty Mono

Let verify the first case. The code snippet below demonstrates that:

(Code snippet 10-5)

```
@Test

void findCustomerByIdSuccessTest(){

    final CustomerModel customer = new CustomerModel("customerId",
"Acme s.r.o", "jana.dvorakova@acme.cz", "CZ1234567", null, null);

    final String customerId = customer.getCustomerId();


Mockito.when(repository.findById(customerId)).thenReturn(Mono.just(cus
tomer));

    StepVerifier.create(service.findCustomerById(customerId))

            .assertNext(result ->
Assertions.assertThat(result).isEqualTo(customer))

            .verifyComplete();
}
```

Opposite, when we do not expect an entity to exist, we need to specify a mock repository to return an empty Mono. Because, there is no entity to be emitted, we do not use the `assertNext()` assertion. Take a look on an example implementation:

(Code snippet 10-6)

```
@Test

void findCustomerByIdNotFoundTest(){

    final String customerId = "customerId";


Mockito.when(repository.findById(customerId)).thenReturn(Mono.empty())
;

    StepVerifier

    .create(service.findCustomerById(customerId))

    .verifyComplete();

}
```

Testing Spring Webflux router functions

Router functions play a similar role in Spring Webflux as annotated controllers in Spring MVC. Both are responsible for handling HTTP requests/responses. However, compare to controllers, new approach is not represented by a single component, but consists of two parts: a router function and a handler, which serves as glue between a service layer (business logic) and an HTTP layer.

This means, that developers need to test these components as a group. This section overviews techniques of testing of Spring Webflux router handlers with the `WebTestClient` class.

Prior writing testing scenaros, let take a moment to configure a test class itself. To set up tests for router functions testing in Soring Webflux we need to complete following steps:

- Extend your test case with the Spring extension
- Specifiy a configuration (e.g. components to be injected)
- Provide external mock dependencies (like a service or a repository)
- Create a `WebTestClient` instance and bind to a context

Take a look on the code snippet below:

(Code snippet 10-7)

```
@ExtendWith(SpringExtension.class)

@ContextConfiguration(classes = {CustomerHandler.class,
CustomerRouter.class})

@WebFluxTest

class CustomerRouterTest {
    @Autowired private ApplicationContext context;

    @MockBean private CustomerService service;

    private WebTestClient client;

    @BeforeEach

    void setup(){

        client =
WebTestClient.bindToApplicationContext(context).build();

    }

//...

}
```

I would like to encapsulate a process of a creation of a mock entity into a separate function (so it could be reused by all test cases):

(Code snippet 10-8)

```
private CustomerModel createMockCustomer(String id){

    CustomerModel customer = new CustomerModel(id, "Acme s.r.o",
"jana.dvorakova@acme.cz", "CZ1234567", null, null);

    return customer;
}
```

The first test scenario is to create a new customer item. It corresponds with a `POST` endpoint. From a technical point of view, a test flow is following:

- Prepare a new customer entity
- Configure a mock's (service) behaviour
- Prepare the POST request
- Execute the request
- Assert results

Take a look on the sample implementation below:

(Code snippet 10-9)

```
@Test

void postCustomerTest(){

    final CustomerModel customer = createMockCustomer("customerId");


Mockito.when(service.createCustomer(customer)).thenReturn(Mono.just(cu
stomer));

    client.post()

            .uri(URI.create("/customers"))

            .accept(MediaType.APPLICATION_JSON)

            .body(Mono.just(customer), CustomerModel.class)
```

```
                .exchange()

                .expectStatus().isOk()

                .expectBody(CustomerModel.class)

                .value(result ->
Assertions.assertThat(result).isEqualTo(customer));

}
```

The next scenario is to retrieve a single customer by the identifier. This is basically divided into two situations:

- Successful result = the customer does exist
- Failed result = the customer does not exist

Let move to the code.

As it was defined already, the first test case means, that we should get a `CustomerModel` entity back from the router function. Here we proceed through foloowing steps:

- Define a mock data entity and ID
- Configure the service behaviour to return data
- Create an URI path: consists of two parts - base part and ID
- Execute the request
- Assert results

Take a look on the implementation, presented below:

(Code snippet 10-10)

```
@Test

void getCustomerByIdSuccessTest(){

    final String customerId = "customerId";

    final CustomerModel customer = createMockCustomer(customerId);


Mockito.when(service.findCustomerById(customerId)).thenReturn(Mono.jus
t(customer));

    client.get()

            .uri(URI.create("/customer/customerId"))

            .accept(MediaType.APPLICATION_JSON)

            .exchange()

            .expectStatus().isOk()

            .expectBody(CustomerModel.class)

            .value(result ->
Assertions.assertThat(result).isEqualTo(customer));

}
```

Observe, that we use the `value()` function in order to get an entity from a Mono publisher.

In the previous test scenario we checked a scenario with an existing customer. In case of an absence, the handler respondes with `404 Not found` status code. We talked about an error handling in Spring Webflux in the book.

What do we need to accomplish here:

- Create (or generate) an identifier that *should not exist*
- Configure a mock service to return an *empty* response
- Execute the request
- Assert that response has not found code

Below you can find the test case code:

(Code snippet 10-11)

```
@Test

void getCustomerByIdNotFoundTest(){

    final String customerId = "customerId";


Mockito.when(service.findCustomerById(customerId)).thenReturn(Mono.emp
ty());

    client.get()

            .uri(URI.create("/customer/customerId"))

            .accept(MediaType.APPLICATION_JSON)

            .exchange()
            .expectStatus().isNotFound();

}
```

Due to the fact, that this response does not contain a body payload, we just call the `expectStatus` method in order to verify a status code.

Summary

The unit testing is a foundational technique that every software developer should now. The process of doing unit tests in Spring Webflux is a bit different from "non reactive" Spring, especially when it comes to deal with reactive types, as Mono and Flux. From the other side, the Spring framework brings a number of helpers to deal with such tasks as assertions or mocking.

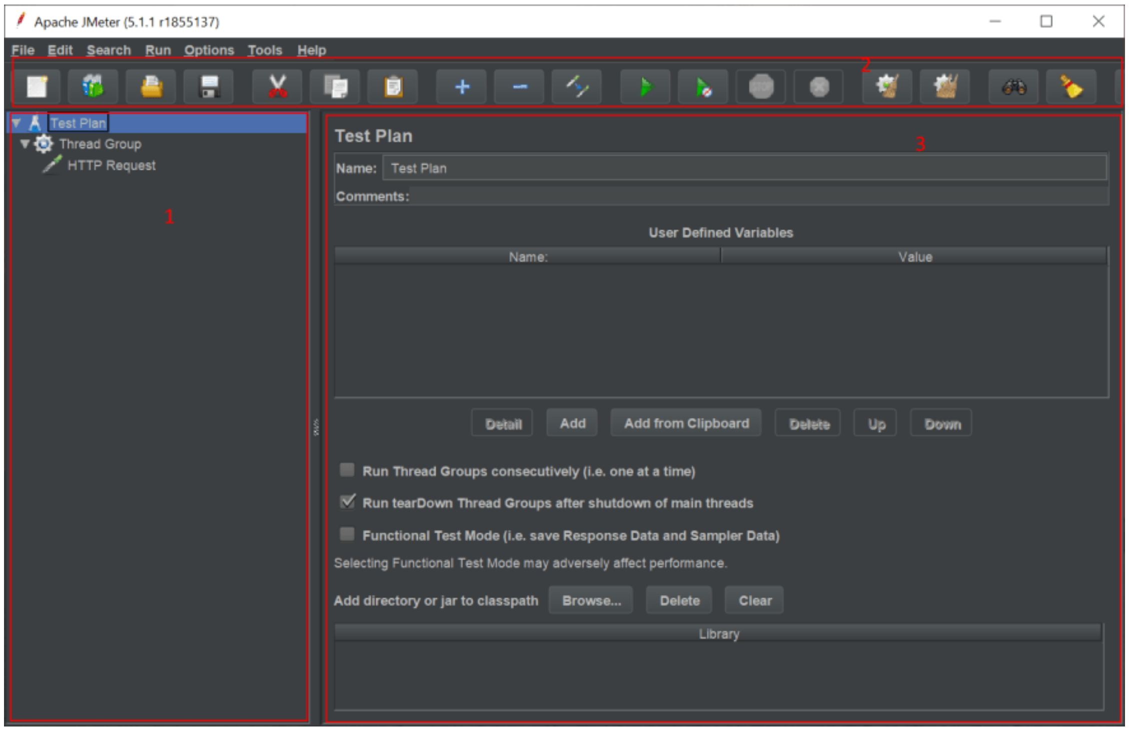
Chapter 11. Load testing with Apache JMeter

A load testing is a technique of non-functional testing: it refers to aspects of the software that may not be related to a specific function or user action. We use a load testing in order to study app's behaviour under a specific load. We need to distinguish a load testing and a performance testing. We use a performance testing to analyze overall app's performance under a heavy load.

Introduction to Apache JMeter interface

In this sub section, we will observe JMeter's interface. When you run it, you would see a similar window as on the screenshot below:

(Image 11-1)



Name: Test Plan

Comments:

[ ] Run Thread Groups consecutively (i.e. one at a time)

[✓] Run tearDown Thread Groups after shutdown of main threads

[ ] Functional Test Mode (i.e. save Response Data and Sampler Data)

Selecting Functional Test Mode may adversely affect performance.

The JMeter window can be divided into 3 functional areas:

- Part 1 = **Test plan area** = this is a tree-based structure, where you can manage your test's structure, add/remove items (see next about them)
- Part 2 = **Toolbar** = this bar contains most important actions, applied to the test - save/open a saved test, run, etc.
- Part 3 = **Element area** = in this part you can configure an individual item (thread group, elements etc.)

Work with test plans

JMeter operates its own terminology, that you should be familiar with. One of key concepts here is a *test plan*. Test plan is a root structure of load tests and represents what to test and how to go about it. It can include *thread groups*, that control the number of threads JMeter will use during the test. Thread groups include different *elements*, for example *samplers* (used to send requests to server; for example HTTP or FTP requests), *logic controllers* (to define logical conditions; if/for etc.), *listeners* (used to visualize tests output in the form of tables, graphs, trees, or simple text in some log files). These elements are most used, but not only ones. There also others, like *assertions*, *configuration elements*, *timers* etc., that you can use in your tests.

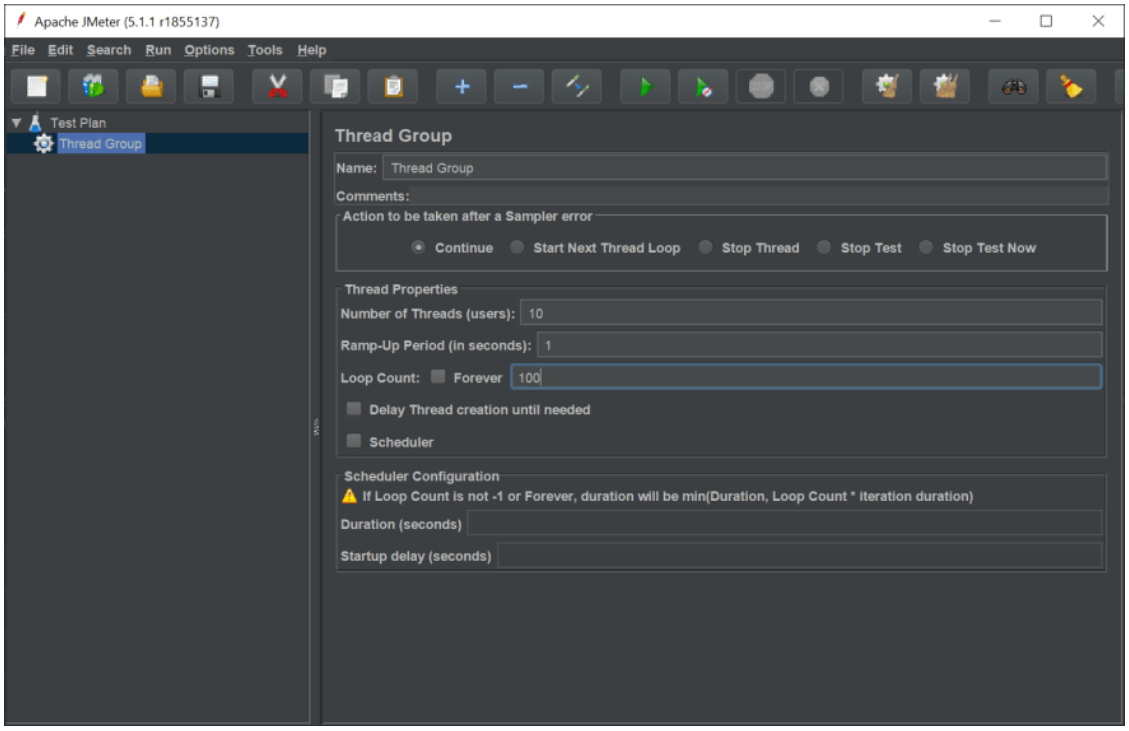
It is crucial to remember an execution order of test plan elements:

- Configuration elements
- Pre-Processors
- Timers
- Samplers
- Post-Processors (unless SampleResult is null)
- Assertions (unless SampleResult is null)
- Listeners (unless SampleResult is null)

As the next step, we will create a simple test plan with Apache JMeter to test a REST API.

Creating new Thread Group

(Image 11-2)


Apache JMeter (5.1.1 r1855137)
File Edit Search Run Options Tools Help
Test Plan
Thread Group

Thread Group

Name: Thread Group

Comments:

Action to be taken after a Sampler error

[✓ Continue] Start Next Thread Loop Stop Thread Stop Test Stop Test Now

Thread Properties

Number of Threads (users): 10

Ramp-Up Period (in seconds): 1

Loop Count: Forever 100

Delay Thread creation until needed

Scheduler

Scheduler Configuration

⚠ If Loop Count is not -1 or Forever, duration will be min(Duration, Loop Count * iteration duration)

Duration (seconds)

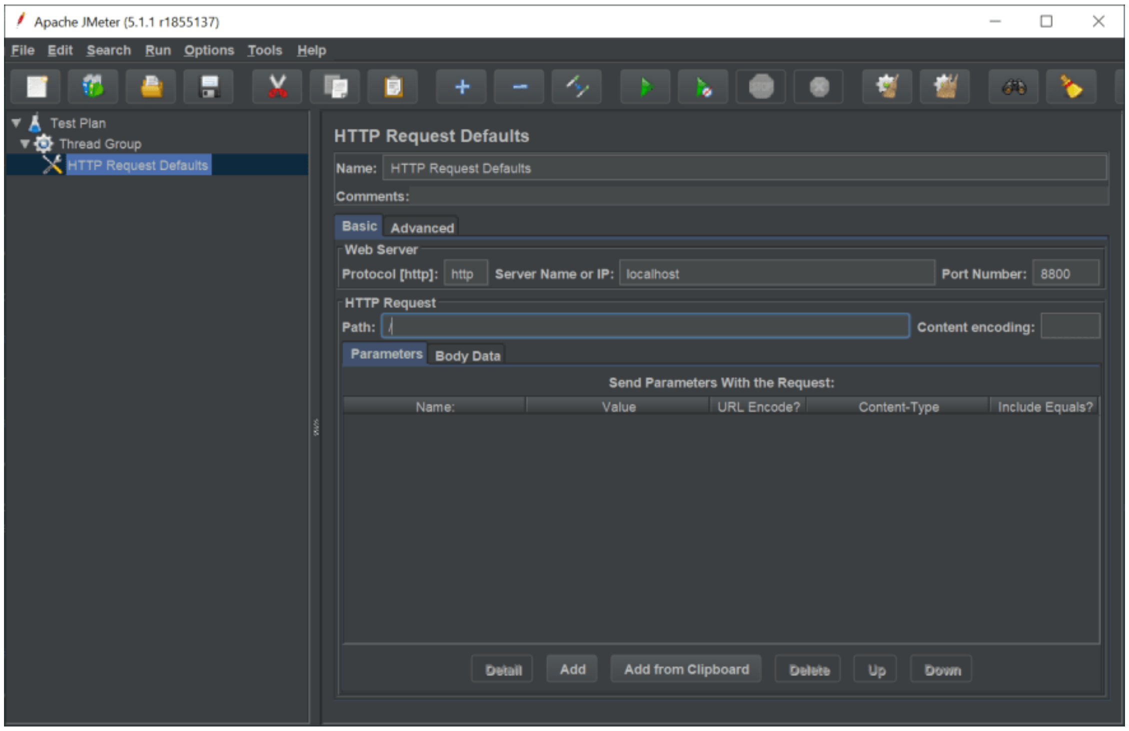
Startup delay (seconds)

To start load testing, we need create new thread group. Right-click on *Test Plan* and select `Add > Threads (Users) > Thread Group`. Here we can specify following parameters:

- Number of threads = 10
- Ramp-Up period = 1
- Loop count = 100

Setting up a configuration

(Image 11-3)


Apache JMeter (5.1.1 r1855137)
File Edit Search Run Options Tools Help
Test Plan
Thread Group
HTTP Request Defaults

HTTP Request Defaults

Name: HTTP Request Defaults

Comments:

Basic | Advanced

Web Server

Protocol [http]: http Server Name or IP: localhost Port Number: 8800

HTTP Request

Path: / Content encoding:

Parameters | Body Data

Send Parameters With the Request:

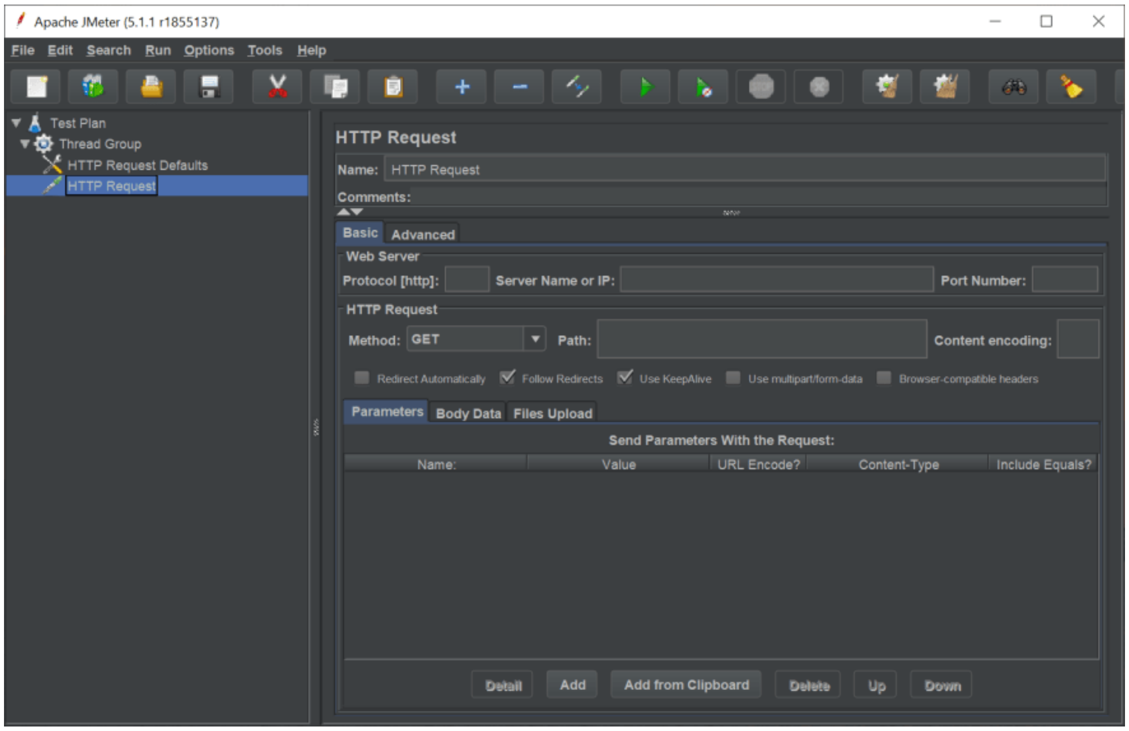
Name:	Value	URL Encode?	Content-Type	Include Equals?

Next step is to set up config for our load test. Our dummy api is running as *localhost:8800*, therefore we should tell JMeter where to find it. Right click on **Thread Group** and select `Add > Config Element > Http Request Default`. Here we add:

- Protocol = http
- Server name = localhost
- Port = 8800
- Path = /

Adding an HTTP request

(Image 11-4)


Apache JMeter (5.1.1 r1855137)
File Edit Search Run Options Tools Help
Test Plan
Thread Group
HTTP Request Defaults
HTTP Request

HTTP Request

Name: HTTP Request

Comments:

Basic | Advanced

Web Server

Protocol [http]: Server Name or IP: Port Number:

HTTP Request

Method: GET Path: Content encoding:

Redirect Automatically ✓ Follow Redirects ✓ Use KeepAlive Use multipart/form-data Browser-compatible headers

Parameters | Body Data | Files Upload

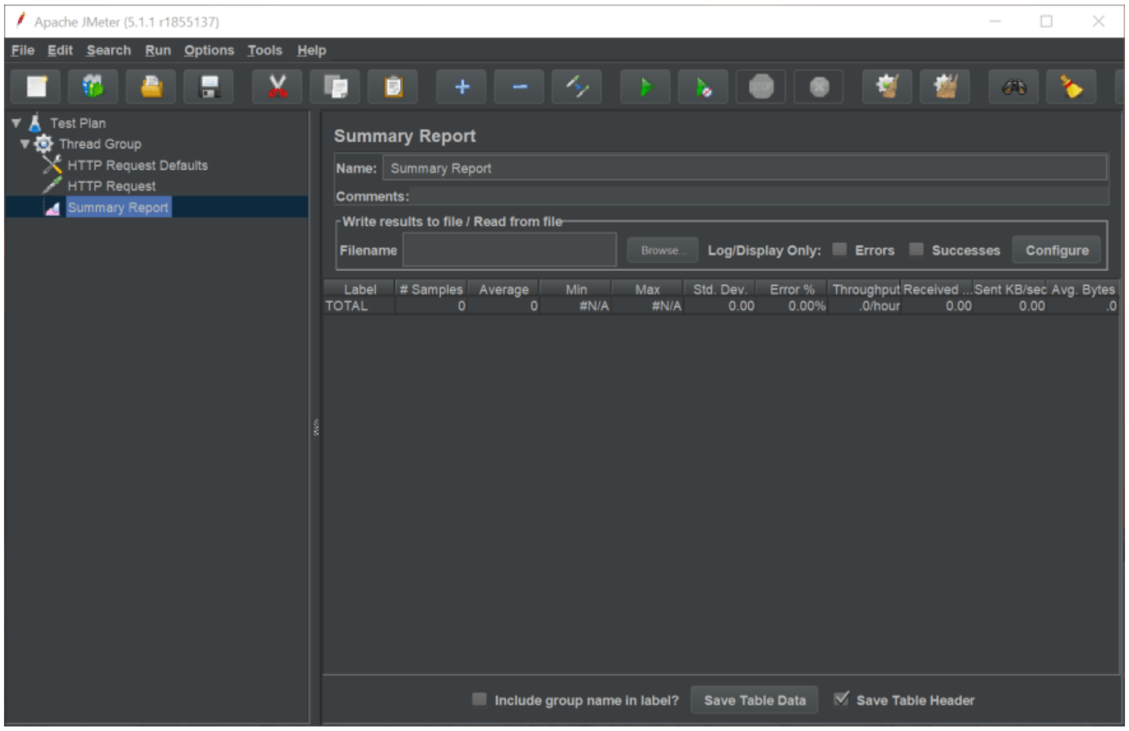
Send Parameters With the Request:

Name:	Value	URL Encode?	Content-Type	Include Equals?

We can now add a sampler - in our case HTTP Request. We predefined everything in the previous step, so nothing to add here. Although, in more complex tests, of course you have to specify it. Leave everything blank for now.

Adding a listener

(Image 11-5)



Summary Report

Name: Summary Report

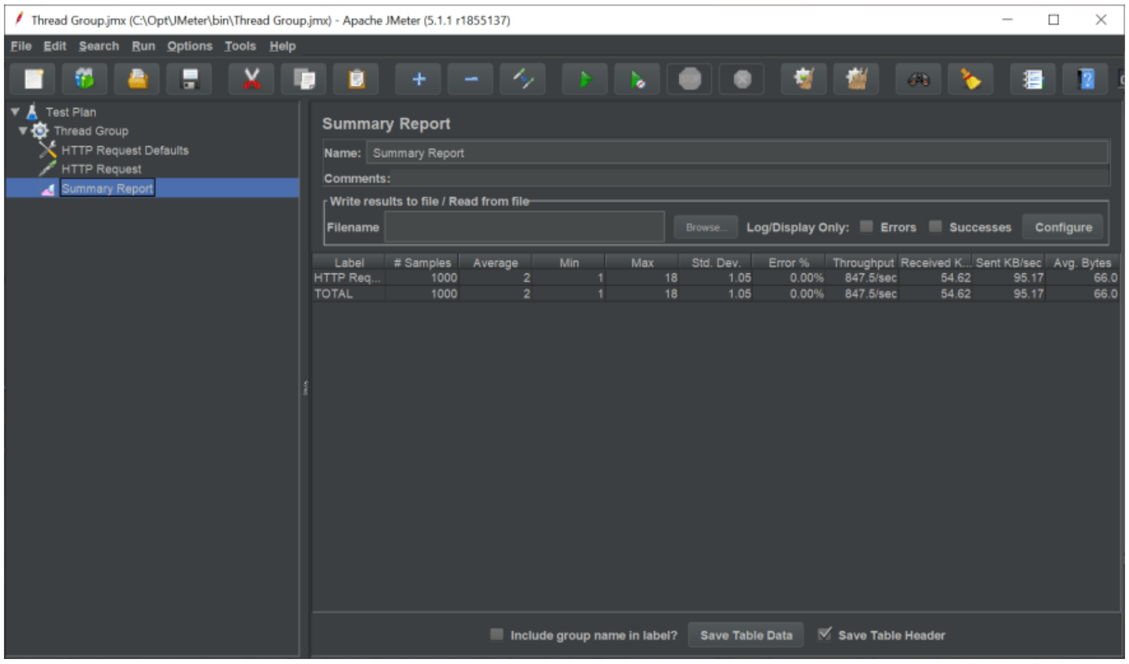
Comments:

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received ...	Sent KB/sec	Avg. Bytes
TOTAL	0	0	#N/A	#N/A	0.00	0.00%	.0/hour	0.00	0.00	.0

Finally, add a listener - an element to show results. Go and add `Add > Listener > Summary Report`. The summary report creates a table row for each differently named request in your test. With this component, you also can save data as CSV.

Running a test plan

(Image 11-6)


Thread Group.jmx (C:\Opt\JMeter\bin\Thread Group.jmx) - Apache JMeter (5.1.1 r1855137)

Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename

Log/Display Only: Errors Successes

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received K...	Sent KB/sec	Avg. Bytes
HTTP Req...	1000	2	1	18	1.05	0.00%	847.5/sec	54.62	95.17	66.0
TOTAL	1000	2	1	18	1.05	0.00%	847.5/sec	54.62	95.17	66.0

After all steps were completed, you can succesfully run a load test scenario. As an output, you will obtain a data table, that we contain following fields:

- Label = name of the HTTP Request sampler
- Samples = number of virtual users per request
- Average = average time taken by all the samples to execute specific label
- Min = the shortest time taken by a sample for specific label.
- Max = the longest time taken by a sample for specific label
- St.Dev = a set of exceptional cases which were deviating from the average value of sample response time.
- Error = a percentage of Failed requests per Label.
- Throughtput = a number of request that are processed per a time unit (seconds, minutes, hours) by the server

Summary

A load testing is an important part of a software quality assurance in general. It validates how the application would behave under the expected load and allows to highlight possible issues. There is a number of software products, that perform a load testing and a performance testing. Apache JMeter is an open source application, that perfectly does its job. In this chapter we observed basics of working with Apache JMeter in order to run simple test plans.

Chapter 12. Working with NoSQL databases

The data layer is an important part of any web application. It is mostly impossible to imagine a relatively complex application, that does not use any kind of a database to persist information. In this chapter we will focus on working with Spring Data and Mongo NoSQL databases using reactive repositories. First, we will observe a generic `ReactiveMongoRepository` contract, that abstracts most common CRUD tasks, and then we will learn how to extend it with a custom functionality.

Getting started with a ReactiveMongoRepository

Out of the box, Spring supplies us with repositories, that offer basic CRUD functionality, e.g. insertion, deletion, update and retrieving data. In case of reactive MongoDB access, we talk about `ReactiveMongoRepository` interface, which captures the domain type to manage as well as the domain type's id type. There are two ways to extend it functionality:

- Use *derived query methods*, which are generated from custom method signatures, you declare in the interface - like `findByFirstName` - this is out of the scope of this post.
- Create a custom implementation and combine it with `ReactiveMongoRepository` - this is what we will do in this article.

To start, let create a custom entity to use in our example. Let do a traditional example - user management. This code declares a `User` object that we will use as an example:

(Code snippet 12-1)

```
@Value
@Document(collection = "users")
public class User {
    @Id String userId;
    String email;
    String password;
    List<Role> roles;
}
```

The next step is to declare a repository interface, which extends `ReactiveMongoRepository`.

(Code snippet 12-2)

```
@Repository
public interface UserRepository extends ReactiveMongoRepository<User,
String> {}
```

This interface will provide us with basic CRUD functionality for `User` entities, including:

- Save
- Retrieve by id
- Update
- Retrieve all
- Remove

This, however does not include custom features we may need to do with `User` objects. For instance, we may need to update not **whole** record, but only specific fields, like `password`. Same way, we may need to handle changes for nested objects/arrays of the entity. To do this, we can't use derived query methods, but we have to declare a custom repository.

Build an extension for a ReactiveMongoRepository

The process to extend Spring's reactive Mongo repository with custom functionality takes 3 steps:

- Create an interface that defines custom data management methods, like a `CustomUserRepository`
- Provide an implementation using a `ReactiveMongoTemplate` to access data source. The name of the implementation interface, should have an `Impl` postfix, like a `CustomUsersRepositoryImpl`
- Extend the core repository with custom repository interface. For example, `UserRepository extends ReactiveMongoRepository<User, String>, CustomUserRepository`

Let have a practical example. We already mentioned `UsersRepository` as an entry point for data access operations. We will use this repositoryin services to handle data source access. However, Spring will create only implementations for CRUD operations, and for custom method updates we will need to provide own implementations. But this is not a rocket science with Spring.

The first thing is to create a custom interface to declare our own update methods.

(Code snippet 12-3)

```
public interface CustomUserRepository {

    Mono<User> changePassword (String userId, String newPassword);

    Mono<User> addNewRole (String userId, Role role);

    Mono<User> removeRole (String userId, String permission);

    Mono<Boolean> hasPermission (String userId, String permission);
}
```

Then, in the step 2, we need to provide a *custom implementation*. This is the step, where we will need to do most work. We will look on each method separately later, but for now we will provide an essential configuration. What do we need to do:

- Provide a dependency of a type `ReactiveMongoTemplate`. We will use it to access an underlaying data source.
- Add a constructor, where Spring will inject the required bean. Also, annotate the constructor using `@Autowired`

Take a look on the code snippet below:

(Code snippet 12-4)

```
public class CustomUserRepositoryImpl implements CustomUserRepository
{

    private final ReactiveMongoTemplate mongoTemplate;

    @Autowired
    public CustomUserRepositoryImpl(ReactiveMongoTemplate
mongoTemplate) {
        this.mongoTemplate = mongoTemplate;
    }

    @Override
    public Mono<User> changePassword(String userId, String
newPassword) {
        return null;
    }

    @Override
    public Mono<User> addNewRole(String userId, Role role) {
        return null;
    }
    @Override
    public Mono<User> removeRole(String userId, String permission) {
        return null;
    }

    @Override
    public Mono<Boolean> hasPermission(String userId, String
permission) {
        return null;
    }


}
```

Finally, we need to make Spring aware, that we want to use this repository alongside with the built-in one. For this, extend the `UserRepository` with the `CustomUserRepository` interface.

(Code snippet 12-5)

```
public interface UserRepository extends ReactiveMongoRepository<User,
String>, CustomUserRepository {
    //...
}
```

So, now we are ready to do implementations of custom update methods.

Implement custom update functionality

In this section we will look how to implement custom updates methods with Reactive MongoDB in Spring. Each method is described separately.

Update an entity's field

The one of the most common requirements is to change a value of an entity's field. In our example we use `confirmed` field to store if the user confirmed her/his account or not. Of course, in our example we concentrate only on database logic, and will not handle a confirmation workflow, so please assume that our `UserService` did everything correctly and now uses a repository to update user's status.

(Code snippet 12-6)

```
@Override
public Mono<User> changePassword(String userId, String newPassword) {
    // step 1
    Query query = new Query(Criteria.where("userId").is(userId));

    // step 2
    Update update = new Update().set("password", newPassword);

    // step 3
    return mongoTemplate.findAndModify(query, update, User.class);
}
```

Let examine what we do here:

- Create a query object that finds the concrete user with specific `userId`
- Create an update object to change the `confirmed` field's value to `true`. For this, use `set()` method that accepts as a first argument a field's name (key) and as a second one new value
- Finally execute an update using `mongoTemplate` object. The method `findAndModify` is used to apply provided update on documents that match defined criteria.

Note, that basically, the flow is same if we use "traditional" synchronous Spring MongoDB. The only difference is the return type - instead of plain `User` object, reacitve repository returns a `Mono`.

Add to a nested array

An another widespread operation you may need to handle in your applications is to add a new entity in a nested array. Let say, that we want to add a new role for the specific user. In our example we store permssions in the nested list `roles`.

Again, we don't focus on how actually permission management is done, and assume that `UserService` does everything as needed.

(Code snippet 12-7)

```
@Override
public Mono<User> addNewRole(String userId, Role role) {
    // step 1
    Query query = new Query(Criteria.where("userId").is(userId));

    // step 2
    Update update = new Update().addToSet("roles", role);

    // step 3
    return mongoTemplate.findAndModify(query, update, User.class);
}
```

Basically, steps are same. The only change is an `Update` object. We use here `addToSet` method which accepts a name of a nested array - `roles` and an object to insert in it.

There is an another approach to add an object to a nested collection - `push`:

(Code snippet 12-8)

```
// ..
Update updateWithPush = new Update().push("roles", role);
```

The difference between these two methods is following:

- `addToSet` method does not allow duplicates and inserts an object **only** if collection does not contain it already.
- `push` method allows duplicates and can insert the same entity several times

Remove from a nested array

Like we add something inside a nested collection, we may need to remove it. As we do in our examples role management, we may want to revoke given access rights from the user. To do this we actually need to remove an item from a nested collection. Take a look on the code snippet below:

(Code snippet 12-9)

```
@Override
public Mono<User> removeRole(String userId, String permission) {
    // step 1
    Query query = new Query(Criteria.where("userId").is(userId));

    // step 2
    Update update = new Update().pull("roles",
         new BasicDBObject("permission", permission));

    // step 3
    return mongoTemplate.findAndModify(query, update, User.class);
}
```

The update object uses `pull` method to remove a `role` with a given `permission` value. We assume that permissions are unique, but in a real life it is better to use some internal unique IDs for the nested object management. We can divide this step into two parts:

- The `pull` operator accepts as a first argument a name of a nested collection

- The second argument is a `BasicDBObject` which specifies a criteria of which role we want to delete. In our case we use `permission` field to specify it

Query by a nested object's field

We already mentioned, that Spring helps us to do querying with derived query methods, like `findByName`. Although, this will work with object's fields, but what if we want to query by a **nested object**?

For the illustration we declared a method `hasPermission` which returns a boolean value that indicates if a user has a specific access right, or in terms of our example - does a nested list `roles` contains an entity with a specific `permission` value.

Let see how we can solve this task:

(Code snippet 12-10)

```
@Override
// 1
public Mono<Boolean> hasPermission(String userId, String permission) {

    // 2
    Query query = new Query(Criteria.where("userId").is(userId))
            .addCriteria(Criteria.where("roles.permission").is(permiss
ion)); //3

    // 4
    return mongoTemplate.exists(query, User.class);
}
```

Let go step by step:

- Unlike previous methods, this method returns `Mono<Boolean>` result, which wraps a boolean value that indicates a presence/absence of the desired permission.
- We need to combine two criterias, because we are looking for the specific user with specific permission. `Query` object offers fluent interface and we can chain criterias with `addCriteria` method.
- We build a query object which looks for a `role` entity inside `roles` array with the needed `permission` value.
- Finally, we call `exists` method, which accepts a query and checks for an existence of the queried data

Summary

The Spring Data project is a handy tool, that simplifies a lot for developers the work with a data layer. It uses a concept of repositories - components, that abstract data operations from concrete database implementations. For Spring Webflux - due to it reactive nature - you need to use non-blocking reactive data access tools. To work with NoSQL MongoDB it is possible to use Reactive Mongo library. In this chapter we have observed a generic `ReactiveMongoRepository` that contains most common CRUD data operations, as well we have researched how to extend it in order to get a custom functionality for needs of your applications.

Chapter 13. Immutables and Spring Webflux

Throughout this book, we mainly used *immutable entities* to create model classes. We achieved this task with the `@Value` annotation, that comes with Lombok library. Some software developers don't like Lombok and, for sure, it has a number of drawbacks, yet they are not so critical. This is a fast approach, but it is not the only one. If you prefer to avoid Lombok, there is no reason to avoid a good software design, and immutable entities as one of its part. In this chapter we will talk about an importance of immutability and how to achieve it with Spring Webflux.

Why to use immutable objects?

First thing first is to define what do we understand as immutable objects? In a computer science, *an immutable object is an object, whose internal state remains unchanged after its creation*. In other words, all data is provided during the initialization. It allows to keep the instance same during its lifecycle. Generally, the immutable software design brings a number of important benefits:

- Thread safety
- Avoiding hidden side effects
- Protection of creating corrupted objects
- Protection against null reference errors

A classical example of an immutable class in Java is the `java.lang.String`. When we create a new instance, for example, from the `replace()` method, the original string value does not change. This is a simple case. In big enterprise applications and systems, the role of an immutable software design grows exponentialy. It is important in order to provide thread safety and as well to avoid creating invalid business entities with absent data.

Immutable objects with plain Java

Frankly speaking, you don't need to bring any additional dependencies in order to create immutable objects. It is possible to do its manually, just by following several important rules. In this case, an immutable object is one that has:

- No setter methods
- All fields private and final
- All arguments constructors

One should mark the class also with the `final` keyword. This is done in order to prevent a subclassing and as a result an overriding of methods. An another missing aspect, is to avoid references to *mutable* objects, that serve as fields. For example, if you class has a list as a field, the getter method `getList()` should not return a list directly, rather return a copy of the list. Take a look on the following example below:

(Code snippet 13-1)

```
final class Author {
    private final String name;
    private final List<Book> books;

    Author(String name, List<Book> books){
        this.name = name;
        this.books = books;
    }

    String getName(){
        return name;
    }

    List<Book> getBooks(){
        return List.copyOf(books);
    }
}
```

In this code snippet we use the `List.copyOf` static method. It creates an unmodifiable list instance, that contains the elements of the given `books` list, in its iteration order.

Immutable objects with Lombok

When your domain objects grow too big, it may be hard to define classes in the aforesaid manner. A good alternative is to use Lombok annotations to make them do "dirty work" instead. Basically, Lombok offers the `@Value` annotation that does same things, as we did manually. It has a following functionality:

- Marks all fields `private` and `final`
- Create an all arguments constructor
- Makes the class `final` to avoid sub classing
- No setters are generated

In this way, it is a counterpart of the `@Data` annotation, that provides functionality in order to create mutable entities. Likewise, the `@Data` annotation, the `@Value` annotation overrides basic `Object` methods:

- `hashCode`
- `equals`
- `toString`

We can refactor the previous example, with Lombok as following:

(Code snippet 13-2)

```
@Value
class Author {
    String name;
    List<Book> books;

    List<Book> getBooks(){
        return List.copyOf(books);
    }
}
```

Please note, that in the example, we also create a custom getter method for a list. It is due to the fact, that marking the `books` field private and final does not make unchangeable, because it can be modified with the `add()` method. So, we need to prevent any access to it outside the class.

A good practice is to use immutable collections, instead of mutables. This will simplify a code and it will not be necessary to create explicit getters. Acceptable solutions include Eclipse Collections or Google Guava.

Immutable objects with records

Since Java 14, was introduced an **experimental feature** that is called records. From a technical point of view, **records a compact syntax for declaring classes which are transparent holders for shallowly immutable data**. In other words, it is a shorthand approach to do what we did with "plain Java". A record object is just a Java entity class that is defined with the `record` keyword, instead of the `class` keyword and an all arguments constructor.

(Code snippet 13-3)

```
record Author (String name, List<Book> books){}
```

Like Lombok annotations, records also create hash code, equals and string methods. Each field is supplied its getter method.

Notes on using immutable objects with Jackson

When you are working with Spring Boot and Jackson, you may face the `InvalidDefinitionException` exception, especially during web layer tests. That is due to tge fact, that Jackson by default uses a no-args constructor and setters to deserialize an entity from a JSON payload. Certainly, this practice leads to bad software design, and you need to do some workarounds in order to mitigate that.

If such problem does occur, you need to modify your domain objects in order to tell Jackson, how they should be created during deserialization. Take a look on a modified version of the `Author` class:

(Code snippet 13-4)

```
@Value
class Author {
    String name;
    List<Book> books;

    @JsonCreator
    Author(@JsonProperty("name") String name, @JsonProperty("books")
List<Books> books){
        this.name = name;
        this.books = books;
    }

    List<Book> getBooks(){
        return List.copyOf(books);
    }
}
```

Here we need to utilize following Jackson annotations:

- `JsonCreator` marks a constructor to be explicitly used by Jackson for deserialization
- `JsonProperty` is used on constructor's arguments to help Jackson access concrete fields in runtime

Please note, that this solution perfectly works both with the `@Value` annotation, as well with immutable objects, that are created with plain Java.

Summary

Practices, that we observed so far, are not only used to create domain objects. They can be used for other types of components, like services, controllers or handlers. However, some Spring components do not accept that and it is important to remember. For example, using ORM like Hibernate requires data entities to be modifiable. However, when you implement data layer manually, for instance with R2DBC, it would make possible to use immutables.

Chapter 14. Client side caching

The front end caching is an important topic, as by providing a local copy of data, client side applications do not need to call a backend every time. That certainly increases an overall speed and improves an user experience. Moreover, a local caching is crucial when the client app need to store data, computed by itself. There are several caching options available, including local storage, state management libraries. The first option is simple, yet it is also a framework-independent choice and is supported by web browsers natively. While working with a local storage may be a pain, there are some libraries that make this task easier. One of them is a LocalForage library. In this chapter we will observe several architectural patterns in order to implement local caching. We will base our examples on Vue.js, however they can be used in a same way for web applications, built with React.js or Svelte.js.

What is caching and why to do it

Before we will progress with concrete caching patterns for web applications, it is important to review what do we understand as caching and what are reasons to bother with it. In this post by caching we define a storage of data locally, e.g. on user's side. While many frontend developers do not even want to hear about caching, I suggest you to try it, as it brings a number of advantages to you, including (but not limited to):

- Increase a speed and improve a user experience by having commonly requested data on a local device, rather than fetching it everytime from a server
- Improve data availability - as data has a local copy, it can be used, in situations when a server is not available

- Store computed data - when you need to save somewhere data, computed by a web application itself, you have to deal with a storage

The simplest form of local data persistance is Local Storage, included as a part of all modern web browsers. While it is straightforward and available for developers out-of-the-box, Local Storage may be not the best tool that fits all tasks. First of all, you need to understand that technically Local Storage saves data as a single JS object. Also it is synchronous and it is an important drawback for big and complex systems. Finally, you need to remember that Local Storage requires you to serialize your objects to store them, as API supports only string data type.

Other options include IndexedDB and state management libraries (Vuex etc). Also, they have an opposite disadvantage - they may be too complex for your apps.

Caching patterns

As it was mentioned before, in this chapter, we will present examples, built with Vue.js. We also will utilize the LocalForage library. There is possible to use either the LocalForage directly or use a wrapper, developed for your framework (like vlf for Vue.js). In order to make solutions portable to other frontend frameworks, such as React.js or Svelte.js we will stick with a direct usage of a LocalForage.

Pattern A: API first

The most popular approach is to prefer API data over a local storage. In this situation, we cache the last successful server response and revert to it, when API is not available (for instance, due to a network error). Such pattern is common in building offline first applications, yet does not provide a great improvement of performance.

(Code snippet 14-1)

```
import axios from 'axios'
import localForage from 'localforage'

data() {
    return {
        posts: []
    }
},
methods: {
    getEntities(){
        axios.get('https://jsonplaceholder.typicode.com/posts')
        .then(result=>{
            let data = result.data
            this.posts = data
            localForage.setItem('posts', data).then(r => {
                console.log('Posts saved locally')
            })
        })
        .catch(error=>{
            console.log(error)
            localForage.getItem('posts').then(data => {
                if (data!==null){
                    this.posts = data
                }
            })
        })
    }
},
created() {
    this.getEntities()
}
```

This pattern prefers server data over local persistance. We perform API call using Axios's `get` method. After the app have obtained data, we save it locally. In a case of errors we would rollback to the last saved copy.

Pattern B: Local first

Opposite to the first solution, where we prefer server data and only revert to a local cache in case of problems, this approach is considered as local first. In this case we save data locally after the initial successful retrivement and then consume it from local cache, rather call API everytime. Such approach sets some limitations so it usually used for data that does not change often. Let see how we can implement such pattern using Axios and LocalForage. Consider the following code snippet:

(Code snippet 14-2)

```
import axios from 'axios'
import localForage from 'localforage'

data() {
    return {
        posts: []
    }
},
methods: {
    getPostsFromServer(){
        axios.get('https://jsonplaceholder.typicode.com/posts')
            .then(result=>{
                let data = result.data
                this.posts = data
                localForage.setItem('posts', data)
            })
    },
    getPosts(){
        localForage.getItem('posts').then(data=>{
            if (data!==null){
                this.posts = data
            } else {
                this.getPostsFromServer()
            }
        }).catch(error=>{
            this.getPostsFromServer()
        })
    }
},
created() {
    this.getPosts()
}
```

In this case we store locally post entries, that is a data, which is not changed everytime, so we can cache it in local storage and retrieve it from there. On `created` hook we call `getPosts()`. This method retrieve data from local cache. In case of error (or if data is not available - is null), we call Axios to get data from server and store it to get available next time from local storage.

Pattern C: API after local storage

The last option we will implement combines both approaches. You can see, that it is similar in some aspects to the second one, however they are different. Consider its implementation in the code snippet below:

(Code snippet 14-3)

```
import axios from 'axios'
import localForage from 'localforage'

data() {
    return {
        posts: []
    }
},
methods: {
    getPosts(){
        localForage.getItem('posts').then(data => {
            this.posts = data
            axios.get('https://jsonplaceholder.typicode.com/posts')
            .then(result => {
                this.posts = result.data
                localForage.setItem('posts', result.data)
            })
        })
    }
},
created() {
    this.getPosts()
}
```

In the second example we fetched data from a server only when a local copy is not available (as a `catch` callback or in a case when a result from a local cache is null). Here we get data first from a local store, however then we still call the server to update local data copy.

This pattern combines best of two previous approaches: it preloads data on the `created` hook from a local cache and then update it once Axios completed an HTTP request.

Summary

In this chapter we observed strengths of using a client side caching for web applications. We also implemented three most common design patterns with the LocalStorage API and the LocalForage library. While these examples were done with Vue.js, they are not tighten with a particular framework and can be used also with applications, written in React.js and Svelte.js.

About the author

Iurii Mednikov (Yuri Mednikov) is a software architect and is a certified Java Programmer. He studied his bachelor degree in Computing in the Prague College (Czech Republic) and worked with a number of brands, such as UniCredit Bank, Raiffeisen Bank, Mercedes Benz and others. Iurii Mednikov has also published book "Principles of Vertx" and he writes for DZone, Medium and Dev.to.

Contacts

- Twitter
- Xing

Other books by Iurii Mednikov

- Iurii Mednikov *Principles of Vertx* (2020) 1st edition, view on leanpub