Reactive Programming

Reactive Streams specification, bringing the paradigm of Reactive Programming on the JVM

Why Reactive Programming

- declarative code (similar to functional programming) in order to build **asynchronous processing pipelines**. It is an event-based model where data is pushed to the consumer, as it becomes available: we deal with asynchronous sequences of events.
- Increases an application's capacity to serve large number of clients, without the headache of writing low-level concurrent or and/or parallelized code.
- Fully asynchronous and non-blocking, Reactive Programming is an alternative to the more limited ways of doing asynchronous code in the JDK: namely Callback based **APIs** and **Future**.
- composition, makes asynchronous code more readable and maintainable.

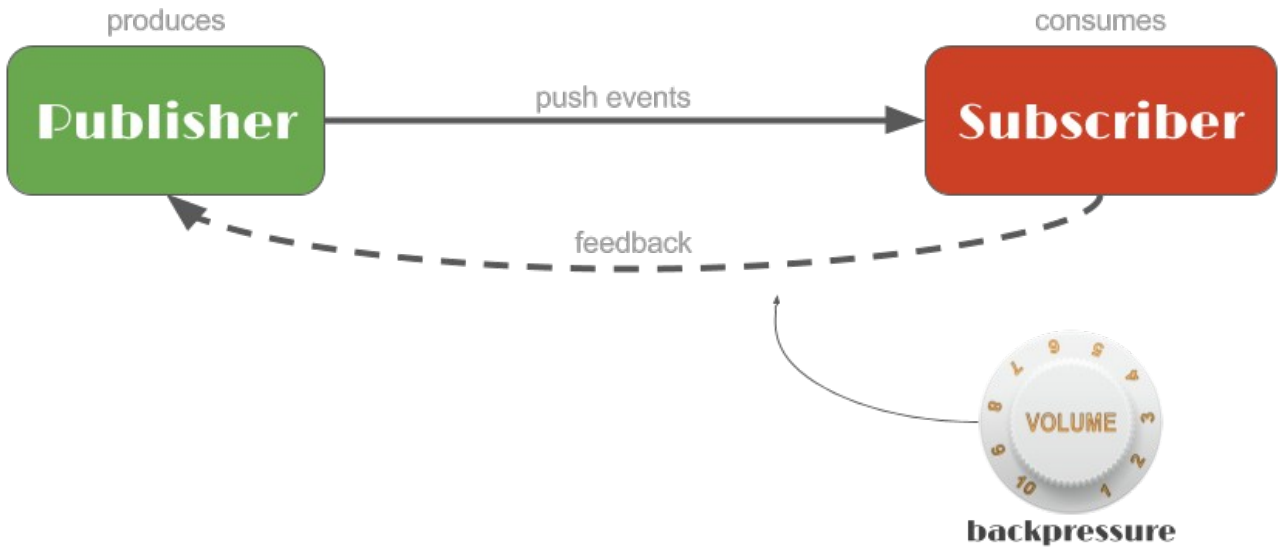
Reactive Streams

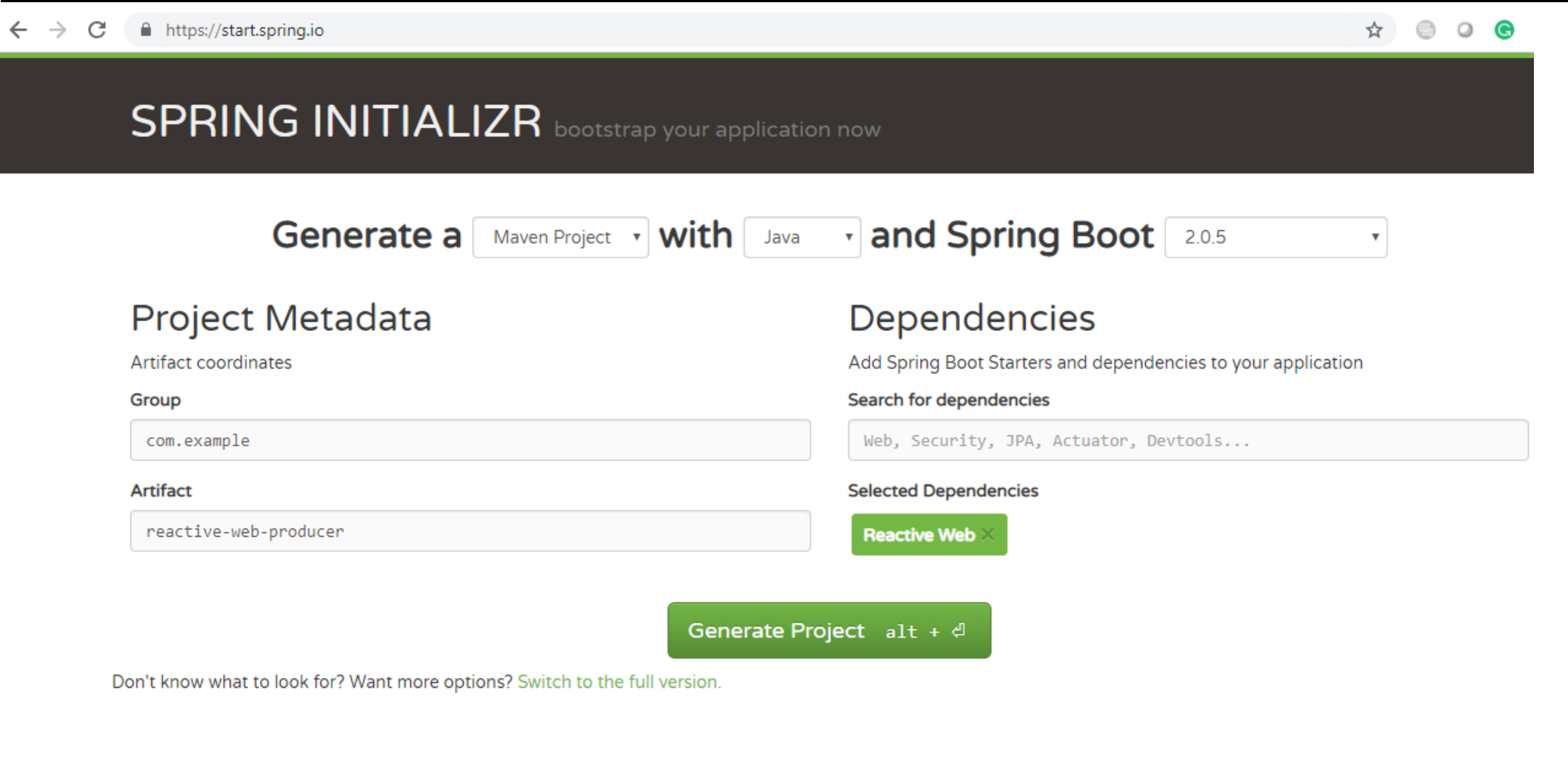
- An industry-driven effort to standardize Reactive Programming libraries on the JVM
- They are interoperable. Implementors include **Reactor 3** but also **RxJava** from version 2 and above, **Akka Streams**, **Vert.x** and **Ratpack**.
- It contains 4 very simple interfaces, which shouldn't be overlooked since it is the rules of the specification that bring the most value to it.
- It is fairly low-level. Reactor 3 aims at offering an higher level API that can be leverage in a large breadth of situations, building it on top of Reactive Streams Publisher.

Interactions

- In reactive stream sequences, the source Publisher produces data. But by default, it does nothing until a Subscriber has registered (subscribed), at which point it will push data to it.
- Reactor adds the concept of operators, which are chained together to describe what processing to apply at each stage to the data.

Interactions




https://start.spring.io
SPRING INITIALIZR bootstrap your application now
Generate a Maven Project with Java and Spring Boot 2.0.5
Project Metadata
Artifact coordinates
Group
com.example
Artifact
reactive-web-producer
Dependencies
Add Spring Boot Starters and dependencies to your application
Search for dependencies
Web, Security, JPA, Actuator, Devtools...
Selected Dependencies
Reactive Web
Generate Project alt + ⏎
Don't know what to look for? Want more options? Switch to the full version.

Person.java

```
public class Person {
    private Long pid;
    private String name, address, phone;
    private int salary;
    // Constructors, getters and setters
}
```

Defining Publisher

```
@RestController
public class GreetReactiveController {
    @GetMapping("/greetings")
    public Publisher<Greeting> greetingPublisher() {
        Flux<Greeting> greetingFlux = Flux.<Greeting>generate(sink -> sink.next(new Greeting("Hello"))).take(50);
        return greetingFlux;
    }
}
```

Defining Publisher

Calling **Flux.generate()** will create a never ending stream of the Greeting object.

The take() method, as the name suggests, will only take first 50 values from the stream.

It's important to note that the return type of the method is the asynchronous type Publisher<Greeting>.

To test this endpoint, navigate your browser to **http://localhost:8080/greetings** or use the curl client on your command line - **curl localhost:8080/greetings**

Defining Publisher

- This doesn't look like that big of a deal and we could have simply returned a List<Greeting> to achieve the same visual result.
- But again, notice that we are returning a Flux<Greeting>, which is an asynchronous type since that changes everything.
- Suppose we had a publisher that returned more than a thousand records, or even more. Think of what the framework has to do. It's given an object of type Greeting, which it has to convert to JSON for the end user.

Defining Publisher

- Had we used the traditional approach with Spring MVC, these objects would keep on accumulating in your RAM and once it collects everything it would return it to the client. This might exceed our RAM capacity and also blocks any other operation from getting processed in the meantime.
- When we use Spring Webflux, the whole internal dynamics get changed. The framework starts subscribing to these records from the publisher and it serializes each item and sends it back to the client in chunks.

Defining Publisher

- We do things asynchronously without creating too many threads and reusing the threads that are waiting for something.
- The best part it is that you don't have to do anything extra for this. In traditional Spring MVC, we could achieve the same by returning AsyncResult, DefferedResult, etc.
- To get some asynchronicity, but internally Spring MVC had to create a new Thread, which gets blocked since it has to wait.

Interactions

- Applying an operator returns a new intermediate Publisher (in fact it can be thought of as both a Subscriber to the operator upstream and a Publisher for downstream).
- The final form of the data ends up in the final Subscriber that defines what to do from a user perspective.

Spring Reactor

- This becomes more challenging when comparing it to the Java 8 Stream API, as they could be mistaken for being the same high-level abstractions.
- Reactive Streams is a specification for asynchronous stream processing.

Reactive Streams Specification

- This becomes more challenging when comparing it to the Java 8 Stream API, as they could be mistaken for being the same high-level abstractions.
- A system where lots of events are being produced and consumed asynchronously. Think about a stream of thousands of stock updates per second coming into a financial application, and for it to have to respond to those updates in a timely manner.

Reactive Streams Specification

- One of the main goals of this is to address the problem of backpressure. If we have a producer which is emitting events to a consumer faster than it can process them, then eventually the consumer will be overwhelmed with events, running out of system resources.
- Backpressure means that our consumer should be able to tell the producer how much data to send in order to prevent this, and this is what is laid out in the specification.

pom.xml

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    <version>3.3.9.RELEASE</version>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.1.3</version>
</dependency>
```

logging the output of Reactor in order to better understand the flow of data.

Producing a Stream of Data

- an application to be reactive, the first thing it must be able to do is to produce a stream of data.
- This could be something like the stock update, Without this data, we wouldn't have anything to react to, which is why this is a logical first step.
- Reactive Core gives us two data types that enable us to do this.

Flux

- It's a stream that can emit 0..n elements. Let's try creating a simple one:

```
Flux<Integer> just = Flux.just(1, 2, 3, 4);
```

- In this case, we have a static stream of four elements.

Mono

- a stream of 0..1 elements. Let's try instantiating one:

```
Mono<Integer> just = Mono.just(1);
```

- This looks and behaves almost exactly the same as the Flux, only this time we are limited to no more than one element.

Mono

- We did get the response this time for each person, though it took over 10 seconds. This defeats the purpose of the application being reactive.
- The way to fix all of these problems is simple: We make a list of type Mono and wait for all of them to complete, rather than waiting for each one:

```
List<Mono<Person>> list = Stream.of(1, 2, 3, 4, 5)
    .map(i -> client.get().uri("/person/{id}",
i).retrieve().bodyToMono(Person.class))
    .collect(Collectors.toList());
Mono.when(list).block();
```

Mono

- This is what we're aiming for. This time, it took just over two seconds, even with massive network lag. This increases the efficiency of our application drastically and really is a game-changer.
- If you look closely at the threads, Reactor is reusing them rather than creating new ones. This is really important if your application handles many requests in a short span of time.

Why Not Only Flux?

- it should be noted that both a Flux and Mono are implementations of the Reactive Streams Publisher interface. Both classes are compliant with the specification, and we could use this interface in their place:

```
Publisher<String> just = Mono.just("foo");
```

- But really, knowing this cardinality is useful. This is because a few operations only make sense for one of the two types, and because it can be more expressive (imagine findOne() in a repository).

Subscribing to a Stream

- use the subscribe() method to collect all the elements in a stream:

```
List<Integer> elements = new ArrayList<>();
Flux.just(1, 2, 3, 4)
  .log()
  .subscribe(elements::add);
assertThat(elements).containsExactly(1, 2, 3, 4);
```

- The data won't start flowing until we subscribe. Notice that we have added some logging as well, this will be helpful when we look at what's happening behind the scenes.

Server-Sent Events

- Another publisher that has been used ever since their arrival is Server-Sent Events.
- These events allow a web page to get updates from a server in real-time.

```
@GetMapping(value = "/greetings/sse", produces =
MediaType.TEXT_EVENT_STREAM_VALUE)
public Publisher<Greeting> sseGreetings() {
    Flux<Greeting> delayElements = Flux
            .<Greeting>generate(sink -> sink.next(new Greeting("Hello @" +
Instant.now().toString())))
            .delayElements(Duration.ofSeconds(1));
    return delayElements;
}
```

Server-Sent Events

```
@GetMapping(value = "/greetings/sse", produces =
MediaType.TEXT_EVENT_STREAM_VALUE)
Flux<Greeting> events() {
    Flux<Greeting> greetingFlux = Flux.fromStream(Stream.generate(()
-> new Greeting("Hello @" + Instant.now().toString())));
    Flux<Long> durationFlux = Flux.interval(Duration.ofSeconds(1));
    return Flux.zip(greetingFlux, durationFlux).map(Tuple2::getT1);
}
```

Server-Sent Events

These methods produce a TEXT_EVENT_STREAM_VALUE which esentially means that the data is being sent in the form of Server-Sent events.

Note that in the first example, we're using a Publisher and in the second example we're using a Flux

Server-Sent Events

- It's advised to use Flux and Mono over Publisher. Both of these classes are implementations of the Publisher interface originating from Reactive Streams. While you can use them interchangeably, it's more expressive and descriptive to use the implementations. two ways to create delayed server-sent events:
  - **.delayElements()**- This method delays each element of the Flux by the given duration
  - **.zip()** - We're defining a Flux to generate events, and a Flux to generate values each second. By zipping them together, we get a Flux generating events each second.
- Navigate to **http://localhost:8080/greetings/sse** or use a curl client on your command line

The Flow of Elements

- **onSubscribe()** – This is called when we subscribe to our stream
- **request(unbounded)** – When we call subscribe, behind the scenes we are creating a Subscription. This subscription requests elements from the stream. In this case, it defaults to unbounded, meaning it requests every single element available
- **onNext()** – This is called on every single element
- **onComplete()** – This is called last, after receiving the last element. There's actually a onError() as well, which would be called if there is an exception, but in this case, there isn't
-

The Flow of Elements

- This is the flow laid out in the Subscriber interface as part of the Reactive Streams Specification, and in reality, that's what's been instantiated behind the scenes in our call to onSubscribe().
- It's a useful method, but to better understand what's happening let's provide a Subscriber interface directly:
- each possible stage in the flow maps to a method in the Subscriber implementation. It just happens that the Flux has provided us with a helper method to reduce this verbosity.

The Flow of Elements

```
Flux.just(1, 2, 3, 4)
  .log()
  .subscribe(new Subscriber<Integer>() {
    @Override
    public void onSubscribe(Subscription s) {
      s.request(Long.MAX_VALUE);
    }
    @Override
    public void onNext(Integer integer) {
      elements.add(integer);
    }
    @Override
    public void onError(Throwable t) {}
    @Override
    public void onComplete() {}
});
```

Comparison to Java 8 Streams

- It still might appear that we have something synonymous to a Java 8 Stream doing collect:

```
List<Integer> collected = Stream.of(1, 2, 3, 4)
  .collect(toList());
```

- The core difference is that Reactive is a push model, whereas the Java 8 Streams are a pull model. In a reactive approach, events are pushed to the subscribers as they come in.

Comparison to Java 8 Streams

- Streams terminal operator is just that, terminal, pulling all the data and returning a result.
- With Reactive we could have an infinite stream coming in from an external resource, with multiple subscribers attached and removed on an ad hoc basis.
- We can also do things like combine streams, throttle streams and apply backpressure

Defining a Consumer

- Now let's see the consumer side of it. It's worth noting that you don't need to have a reactive publisher in order to use reactive programming on the consuming side:

```
public class Person {
    private int id;
    private String name;
    // Constructor with getters and setters
}
```

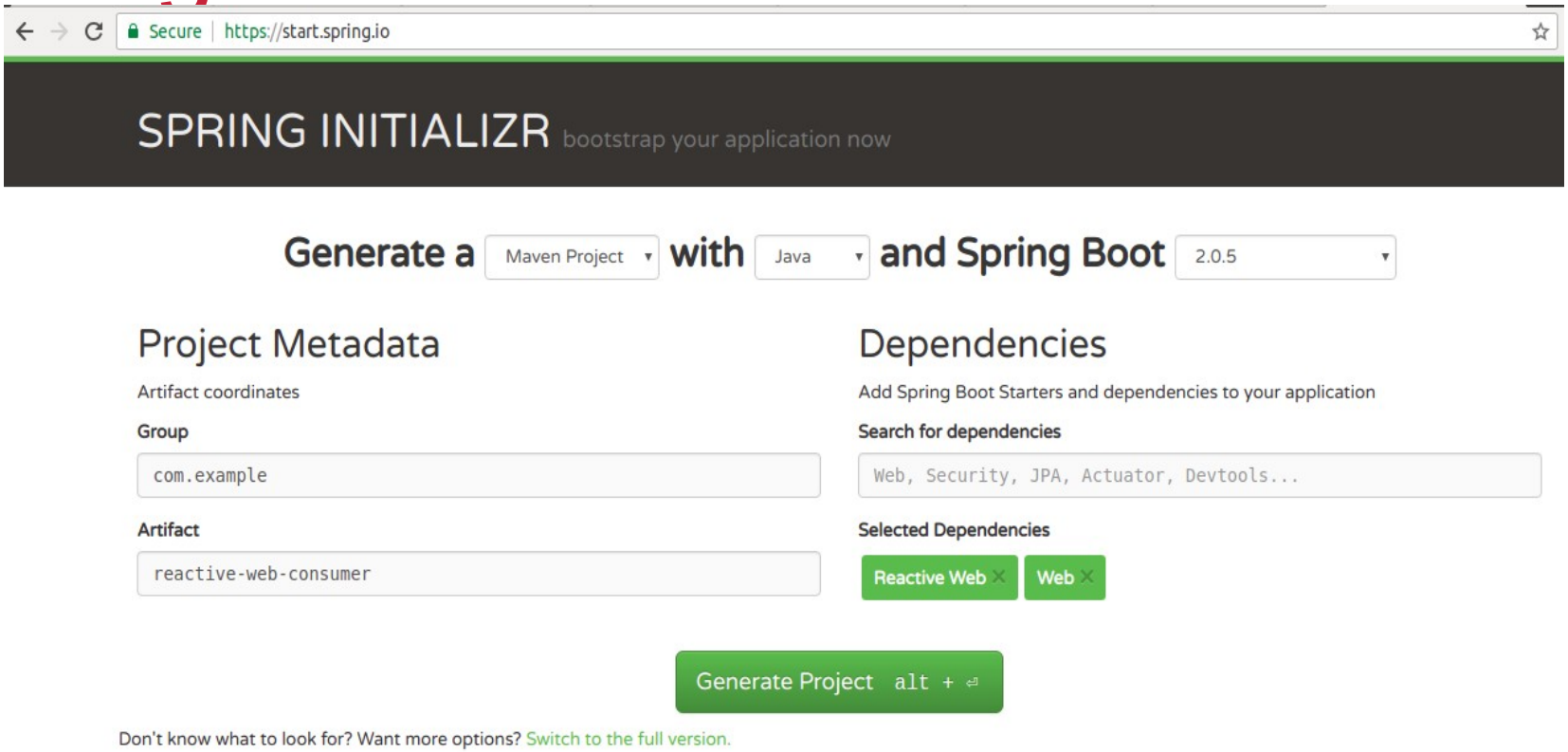
Defining a Consumer

```
@RestController
public class PersonController {
    private static List<Person> personList = new ArrayList<>();
    static {
        personList.add(new Person(1, "John"));
        personList.add(new Person(2, "Jane"));
        personList.add(new Person(3, "Max"));
        personList.add(new Person(4, "Alex"));
        personList.add(new Person(5, "Aloy"));
        personList.add(new Person(6, "Sarah"));
    }
    @GetMapping("/person/{id}")
    public Person getPerson(@PathVariable int id, @RequestParam(defaultValue = "2") int delay)
            throws InterruptedException {
        Thread.sleep(delay * 1000);
        return personList.stream().filter((person) -> person.getId() == id).findFirst().get();
    }
}
```

Defining Consumer

- We initialized a list of type Person and based on the id passed to our mapping, we filter that person out using a stream.
- You might be alarmed by the usage of **Thread.sleep()** here, though it's just used to simulate network lag of 2 seconds.

Defining Consumer



Defining Consumer

- Our producer app is running on port 8080. Now let's say that we want to call the **/person/{id}** endpoint 5 times. We know that, by default, each response takes a 2-second delay due to "network lag".
- Let's first do this using the traditional RestTemplate approach:

Defining Consumer

```
public class CallPersonUsingRestTemplate {
    private static final Logger logger = LoggerFactory.getLogger(CallPersonUsingRestTemplate.class);
    private static RestTemplate restTemplate = new RestTemplate();
    static {
        String baseUrl = "http://localhost:8080";
        restTemplate.setUriTemplateHandler(new DefaultUriBuilderFactory(baseUrl));
    }
    public static void main(String[] args) {
        Instant start = Instant.now();
        for (int i = 1; i <= 5; i++) {
            restTemplate.getForObject("/person/{id}", Person.class, i);
        }
        logTime(start);
    }
    private static void logTime(Instant start) {
        logger.debug("Elapsed time: " + Duration.between(start, Instant.now()).toMillis() + "ms");
    }
}
```

Defining Consumer

- As expected it took a little over 10 secs and this is how Spring MVC works by default.
- At this day and age, waiting for a little over 10 seconds for a result on a page is unacceptable. This is the difference between keeping a customer/client and losing it due to waiting for too long.
- Spring Reactor introduced a new web client to make web requests called WebClient. Compared to RestTemplate, this client has a more functional feel and is fully reactive. It's included in the **spring-boot-starter-weblux** dependency and it's build to replace **RestTemplate** in a non-blocking way.
- Let's rewrite the same controller, this time, using WebClient:

Polling Consumer

```
public class CallPersonUsingWebClient_Step1 {
    private static final Logger logger =
LoggerFactory.getLogger(CallPersonUsingWebClient_Step1.class);
    private static String baseUrl = "http://localhost:8080";
    private static WebClient client = WebClient.create(baseUrl);
    public static void main(String[] args) {
        Instant start = Instant.now();
        for (int i = 1; i <= 5; i++) {
            client.get().uri("/person/{id}", i).retrieve().bodyToMono(Person.class);
        }
        logTime(start);
    }
    private static void logTime(Instant start) {
        logger.debug("Elapsed time: " + Duration.between(start, Instant.now()).toMillis() +
"ms");
    }
}
```

Defining Consumer

- Here, we created a WebClient by passing the baseUrl. Then in the main method, we simply call the endpoint.
- get() indicates that we are making a GET request. We know that the response will be a single object, so we're using a Mono as explained before.
- Ultimately, we asked Spring to map the response to a Person class:

Defining Consumer

- And nothing happened, as expected.
- This is because we are not subscribing. The whole thing is deferred. It's asynchronous but it also doesn't kick off until we call the .subscribe() method. This is a common problem with people who are new to Spring Reactor, so keep an eye out for this
- Let's change our main method and add subscribe:

```
for (int i = 1; i <= 5; i++) {
    client.get().uri("/person/{id}", i).retrieve().bodyToMono(Person.class).subscribe();
}
```

Defining Consumer

- The request is sent but the .subscribe() method doesn't sit and wait for the response. Since it doesn't block, it finished before receiving the response at all.
- Could we counter this by chaining .block() at the end of the method calls?

```
for (int i = 1; i <= 5; i++) {
    client.get().uri("/person/{id}", i).retrieve().bodyToMono(Person.class).block();
}
```

Backpressure

- The subscriber is telling the producer to push every single element at once.
- This could end up becoming overwhelming for the subscriber, consuming all of its resources.
- Backpressure is when a downstream can tell an upstream to send it fewer data in order to prevent it from being overwhelmed.

```
Flux.just(1, 2, 3, 4)
  .log()
  .subscribe(new Subscriber<Integer>() {
    private Subscription s;
    int onNextAmount;
    @Override
    public void onSubscribe(Subscription s) {
        this.s = s;
        s.request(2);
    }
    @Override
    public void onNext(Integer integer) {
        elements.add(integer);
        onNextAmount++;
        if (onNextAmount % 2 == 0) {
            s.request(2);
        }
    }
    @Override
    public void onError(Throwable t) {}
    @Override
    public void onComplete() {}
});
```

Backpressure

- We can modify our Subscriber implementation to apply backpressure. Let's tell the upstream to only send two elements at a time by using request():

Backpressure

- the request(2) is called, followed by two onNext() calls, then request(2) again.
- this is reactive pull backpressure. We are requesting the upstream to only push a certain amount of elements, and only when we are ready.
- If we imagine we were being streamed tweets from twitter, it would then be up to the upstream to decide what to do. If tweets were coming in but there are no requests from the downstream, then the upstream could drop items, store them in a buffer, or some other strategy.

Mapping Data in a Stream

- A simple operation that we can perform is applying a transformation. In this case, let's just double all the numbers in our stream:

```
Flux.just(1, 2, 3, 4)
  .log()
  .map(i -> i * 2)
  .subscribe(elements::add);
```

- map() will be applied when onNext() is called.

Combining Two Streams

```
Flux.just(1, 2, 3, 4)
  .log()
  .map(i -> i * 2)
  .zipWith(Flux.range(0, Integer.MAX_VALUE),
    (one, two) -> String.format("First Flux: %d, Second Flux: %d", one, two))
  .subscribe(elements::add);
assertThat(elements).containsExactly(
  "First Flux: 2, Second Flux: 0",
  "First Flux: 4, Second Flux: 1",
  "First Flux: 6, Second Flux: 2",
  "First Flux: 8, Second Flux: 3");
```

Combining Two Streams

- creating another Flux that keeps incrementing by one and streaming it together with our original one. We can see how these work together by inspecting the logs
- one subscription per Flux. The onNext() calls are also alternated, so the index of each element in the stream will match when we apply the zip() function.

Hot Streams

- we've focused primarily on cold streams. These are static, fixed-length streams that are easy to deal with. A more realistic use case for reactive might be something that happens infinitely.
- For example, we could have a stream of mouse movements that constantly needs to be reacted to or a twitter feed. These types of streams are called hot streams, as they are always running and can be subscribed to at any point in time, missing the start of the data.

Creating a ConnectableFlux

- One way to create a hot stream is by converting a cold stream into one. Let's create a Flux that lasts forever, outputting the results to the console, which would simulate an infinite stream of data coming from an external resource:

```
ConnectableFlux<Object> publish = Flux.create(fluxSink -> {
    while(true) {
        fluxSink.next(System.currentTimeMillis());
    }
})
  .publish();
```

Creating a ConnectableFlux

- One way to create a hot stream is by converting a cold stream into one. Let's create a Flux that lasts forever, outputting the results to the console, which would simulate an infinite stream of data coming from an external resource:

```
ConnectableFlux<Object> publish = Flux.create(fluxSink -> {
    while(true) {
        fluxSink.next(System.currentTimeMillis());
    }
})
  .publish();
```

Creating a ConnectableFlux

- By calling publish() we are given a ConnectableFlux. This means that calling subscribe() won't cause it to start emitting, allowing us to add multiple subscriptions:

```
publish.subscribe(System.out::println);

publish.subscribe(System.out::println);
```

- If we try running this code, nothing will happen. It's not until we call connect(), that the Flux will start emitting:

```
publish.connect();
```

Throttling

- If we run our code, our console will be overwhelmed with logging. This is simulating a situation where too much data is being passed to our consumers. Let's try getting around this with throttling:

```
ConnectableFlux<Object> publish = Flux.create(fluxSink -> {
    while(true) {
        fluxSink.next(System.currentTimeMillis());
    }
})
  .sample(ofSeconds(2))
  .publish();
```

Throttling

- a sample() method with an interval of two seconds. Now values will only be pushed to our subscriber every two seconds, meaning the console will be a lot less hectic.
- There are multiple strategies to reduce the amount of data sent downstream, such as windowing and buffering.

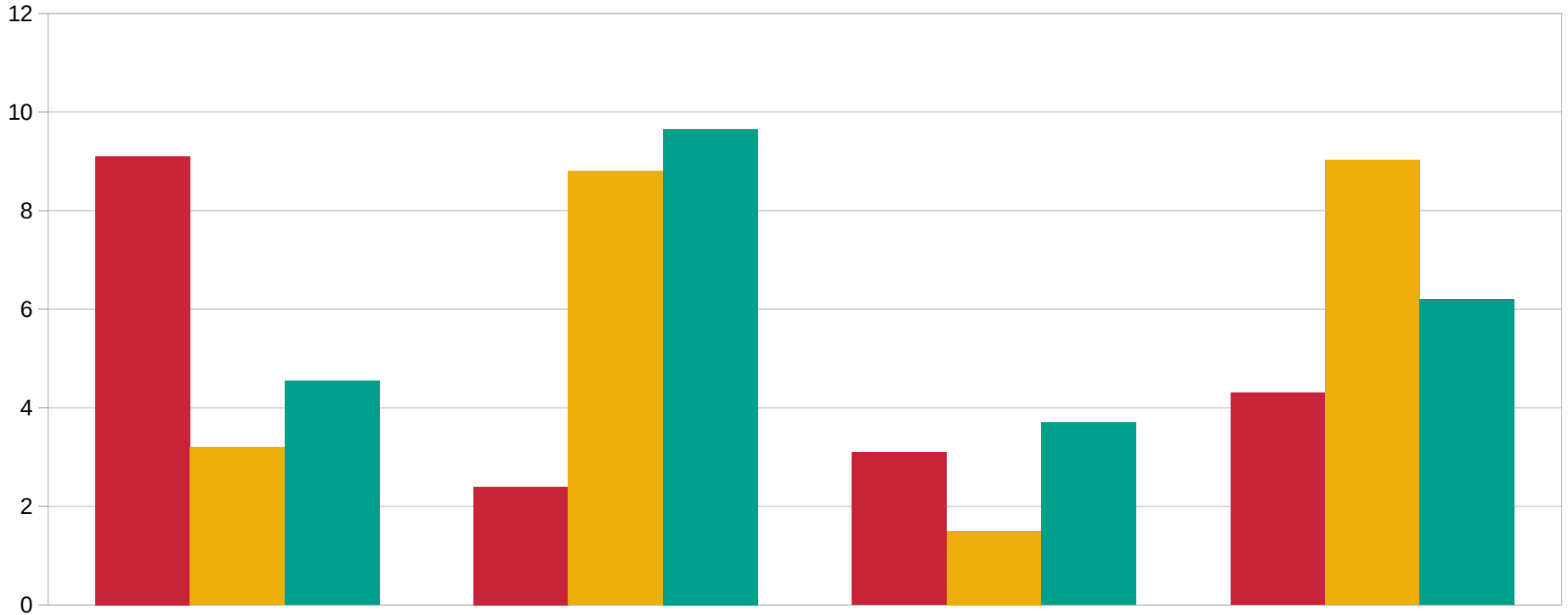
Concurrency

- All of our above examples have currently run on the main thread. However, we can control which thread our code runs on if we want. The Scheduler interface provides an abstraction around asynchronous code, for which many implementations are provided for us. Let's try subscribing to a different thread to main:

```
Flux.just(1, 2, 3, 4)
  .log()
  .map(i -> i * 2)
  .subscribeOn(Schedulers.parallel())
  .subscribe(elements::add);
```

Concurrency

- The Parallel scheduler will cause our subscription to be run on a different thread, which we can prove by looking at the logs.
- We see the first entry comes from the main thread and the Flux is running in another thread called parallel-1.
- Concurrency get's more interesting than this


12
10
8
6
4
2
0