Head-First Reactive Workshop

Stephane Maldini, Ben Hale, Madhura Bhave - Pivotal

Table of Contents

This repository hosts a complete workshop on Spring + Reactor. Just follow this README and create your first reactive Spring applications! Each step of this workshop has its companion commit in the git history with a detailed commit message.

At the end of the workshop, we will have created three applications:

- `stock-quotes` is a functional WebFlux app which streams stock quotes
- `stock-details` is an annotation-based WebFlux app using a reactive datastore
- `trading-service` is an annotation-based WebFlux app that consumes data from `stock-quotes` and `stock-details`

Reference documentations can be useful while working on those apps:

- Reactor Core documentation
- API documentation for Flux
- API documentation for Mono
- Spring WebFlux reference documentation and javadoc

Clone the repository:

git clone https://github.com/reactor/head-first-reactive-with-spring-and-reactor.git

We will start off by creating the Trading Service application which gets data from a pre-existing Stock Quotes application.

Trading Service application

Create this application

Go to `https://start.spring.io` and create a Maven project with Spring Boot 2.0.4.RELEASE, with groupId `io.spring.workshop` and artifactId `trading-service`. Select the `Reactive Web` and `Devtools` dependencies. Unzip the given file into a directory and import that application into your IDE.

If generated right, you should have a main `Application` class that looks like this:

trading-service/src/main/java/io/spring/workshop/tradingservice/TradingServiceApplication.java

```
package io.spring.workshop.tradingservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class TradingServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(TradingServiceApplication.class, args);
    }

}
```

Note that, by default, `spring-boot-starter-webflux` transitively brings `spring-boot-starter-reactor-netty` and Spring Boot auto-configures Reactor Netty as a web server.

Spring Boot supports Tomcat, Undertow and Jetty as well.

Use the WebClient to stream JSON to the browser

In this section, we'll call our remote `stock-quotes` service to get Quotes from it, so we first need to:

- copy over the `Quote` class to this application
- add the Jackson JSR310 module dependency

```
<dependency>
    <groupId>com.fasterxml.jackson.datatype</groupId>
    <artifactId>jackson-datatype-jsr310</artifactId>
    <version>2.9.8</version>
</dependency>
```

Create a `QuotesClient` annotated with `@Component` and inject a `WebClient.Builder`. Now, create a method in the `QuotesClient` called `quotesFeed` which will use the `webClient` to consume the stream of quotes via Server Sent Events (SSE).

HINT!...

```
package io.spring.workshop.tradingservice;

import java.time.Duration;
import java.util.stream.Stream;

import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;

import static org.springframework.http.MediaType.APPLICATION_STREAM_JSON;

@Component
public class QuotesClient {

  private final WebClient webClient;

  public QuotesClient(WebClient.Builder webclientBuilder) {
    this.webClient = webclientBuilder.build();
  }

  public Flux<Quote> quotesFeed() {
    return this.webClient.get()
                         .uri("http://localhost:8081/quotes")
                         .accept(APPLICATION_STREAM_JSON)
                         .retrieve()
                         .bodyToFlux(Quote.class);
  }
}
```

There are two ways to use a `WebClient`, directly via the static factory or by injecting the `WebClient.Builder`. The latter is used by libraries such as Spring Cloud Sleuth that enrich `WebClient` with extra features.

Now create a `QuotesController` annotated with `@Controller` and inject it with the `QuotesClient`. Add a method that responds to `"GET /quotes/feed"` requests with the `"text/event-stream"` content-type, with a `Flux<Quote>` as the response body. The data can be retrieved from the `quotesFeed` method on `QuotesClient`.

You can test it by starting both applications. First, start the Stock Quotes application. It can be started from your IDE or with `mvn spring-boot:run` and it should run a Netty server on port 8081. You should see in the logs something like:

HINT!!! ... The `QuotesController` should look like this :

```
package io.spring.workshop.tradingservice;

import reactor.core.publisher.Flux;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import static org.springframework.http.MediaType.APPLICATION_JSON_VALUE;
import static org.springframework.http.MediaType.TEXT_EVENT_STREAM_VALUE;

@Controller
public class QuotesController {

  private final QuotesClient quotesClient;

  public QuotesController(QuotesClient quotesClient) {
    this.quotesClient = quotesClient;
  }

  @GetMapping(path = "/quotes/feed", produces = TEXT_EVENT_STREAM_VALUE)
  @ResponseBody
  public Flux<Quote> quotesFeed() {
    return this.quotesClient.quotesFeed();
  }
}
```

```
INFO 2208 --- [  restartedMain] o.s.b.web.embedded.netty.NettyWebServer  : Netty
started on port(s): 8081
INFO 2208 --- [  restartedMain] i.s.w.s.StockQuotesApplication           : Started
StockQuotesApplication in 1.905 seconds (JVM running for 3.075)
```

Start the Trading Service application from your IDE or with `mvn spring-boot:run`. This should run a Netty server on port 8080.

```
INFO 2208 --- [  restartedMain] o.s.b.web.embedded.netty.NettyWebServer  : Netty
started on port(s): 8080
INFO 2208 --- [  restartedMain] i.s.w.t.TradingServiceApplication           : Started
TradingServiceApplication in 1.905 seconds (JVM running for 3.075)
```

You can hit http://localhost:8080/quotes/feed to consume the stream of quotes.

Now, let's create another application that can provide the details for a trading company.

Stock Details application

Create this application

Go to `https://start.spring.io` and create a Maven project with Spring Boot 2.0.4.RELEASE, with groupId `io.spring.workshop` and artifactId `stock-details`. Select the `Reactive Web`, `Devtools` and `Reactive Mongo` dependencies.

Unzip the given file into a directory and import that application into your IDE.

If generated right, you should have a main `Application` class that looks like this:

stock-details/src/main/java/io/spring/workshop/stockdetails/StockDetailsApplication.java

```
package io.spring.workshop.stockdetails;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class StockDetailsApplication {

    public static void main(String[] args) {
        SpringApplication.run(StockDetailsApplication.class, args);
    }

}
```

Edit your `application.properties` file to start the server on port 8082.

stock-details/src/main/resources/application.properties

```
server.port=8082
spring.application.name=stock-details
```

Use a reactive datastore

In this application, we'll use a MongoDB datastore with its reactive driver; for this workshop, we'll use an in-memory instance of MongoDB. So add the following:

stock-details/pom.xml

```
<dependency>
      <groupId>de.flapdoodle.embed</groupId>
      <artifactId>de.flapdoodle.embed.mongo</artifactId>
</dependency>
```

We'd like to manage `TradingCompany` with our datastore.

stock-details/src/main/java/io/spring/workshop/stockdetails/TradingCompany.java

```
package io.spring.workshop.stockdetails;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Document
public class TradingCompany {

  @Id
  private String id;

  private String description;

  private String ticker;

  public TradingCompany() {
  }

  public TradingCompany(String id, String description, String ticker) {
    this.id = id;
    this.description = description;
    this.ticker = ticker;
  }

  public TradingCompany(String description, String ticker) {
    this.description = description;
    this.ticker = ticker;
  }

  public String getId() {
    return id;
  }

  public void setId(String id) {
    this.id = id;
  }

  public String getDescription() {
    return description;
  }

  public void setDescription(String description) {
    this.description = description;
  }

  public String getTicker() {
    return ticker;
```

```
  }

  public void setTicker(String ticker) {
    this.ticker = ticker;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    TradingCompany that = (TradingCompany) o;

    if (!id.equals(that.id)) return false;
    return description.equals(that.description);
  }

  @Override
      public int hashCode() {
          int result = id.hashCode();
          result = 31 * result + description.hashCode();
          return result;
      }
}
```

Now create a `TradingCompanyRepository` interface that extends `ReactiveMongoRepository`. Add a `findByTicker(String ticker)` method that returns a single `TradingCompany` in a reactive fashion.

HINT:

```
package io.spring.workshop.stockdetails;

import reactor.core.publisher.Mono;

import org.springframework.data.mongodb.repository.ReactiveMongoRepository;

public interface TradingCompanyRepository extends
ReactiveMongoRepository<TradingCompany, String> {

    Mono<TradingCompany> findByTicker(String ticker);

}
```

We'd like to insert trading companies in our datastore when the application starts up. For that, create a `TradingCompanyCommandLineRunner` component that implements Spring Boot's `CommandLineRunner`. In the `run` method, use the reactive repository to insert `TradingCompany` instances in the datastore.

Since the `run` method returns void, it expects a blocking implementation. This is why you should use the `blockLast(Duration)` operator on the `Flux` returned by the repository when inserting data. You can also `then().block(Duration)` to turn that `Flux` into a `Mono<Void>` that waits for completion.

HINT :

```
package io.spring.workshop.stockdetails;

import java.time.Duration;
import java.util.Arrays;
import java.util.List;

import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class TradingCompanyCommandLineRunner implements CommandLineRunner {

    private final TradingCompanyRepository repository;

    public TradingCompanyCommandLineRunner(TradingCompanyRepository repository) {
        this.repository = repository;
    }

    @Override
    public void run(String... strings) {
        List<TradingCompany> companies = Arrays.asList(
                new TradingCompany("Pivotal Software", "PVTL"),
                new TradingCompany("Dell Technologies", "DELL"),
                new TradingCompany("Google", "GOOG"),
                new TradingCompany("Microsoft", "MSFT"),
                new TradingCompany("Oracle", "ORCL"),
                new TradingCompany("Red Hat", "RHT"),
                new TradingCompany("Vmware", "VMW")
        );
        this.repository.insert(companies).blockLast(Duration.ofSeconds(30));
    }

}
```

Create a JSON web service

We're now going to expose `TradingCompany` through a Controller. First, create a `TradingCompanyController` annotated with `@RestController` and inject the `TradingCompanyRepository`. Then add two new Controller methods in order to handle:

- GET requests to `"/details"`, returning all `TradingCompany` instances, serializing them with

content-type `"application/json"`

- GET requests to `"/details/{ticker}"`, returning a single `TradingCompany` instance, serializing it with content-type `"application/json"`

Partial (!!) HINT :

```
@GetMapping( path = "/details/{ticker}", produces = "application/json")
    public Mono<TradingCompany> getTicker(@PathVariable String ticker) {
        return repo.findByTicker( ticker );
    }
```

It can be started from your IDE or with `mvn spring-boot:run` and it should run a Netty server on port 8082. You should see in the logs something like:

```
INFO 2208 --- [  restartedMain] o.s.b.web.embedded.netty.NettyWebServer  : Netty
started on port(s): 8082
INFO 2208 --- [  restartedMain] i.s.w.s.StockDetailsApplication           : Started
StockDetailsApplication in 1.905 seconds (JVM running for 3.075)
```

Now that we have an application that can return the details for a company with a given ticker, we can update the Trading Service application to use those details and return a combination of the latest quote for that ticker along with the company's details.

Update Trading Service application

We will first create a service that will use a `WebClient` to get data from the Stock Details application. Create a `TradingCompanyClient` annotated with `@Component`. Then add two methods:

- `findAllCompanies` will return a `Flux<TradingCompany>` by using the webClient to get data from the `/details` endpoint from the Stock details application
- `getTradingCompany` will return a `Mono<TradingCompany>` by using the webClient to get data from the `/details/{ticker}` endpoint from the Stock details application
    - If no trading company is found for the given ticker, the `Mono` will complete without any data. Instead, we will emit a `TickerNotFoundException` error using the `switchIfEmpty` operator.

Let's expose the two `TradingCompanyClient` methods with a local `TradingCompanyController`.

You might have updated an already running `TradingServiceApplication`. Since DevTools is present, you can just recompile to automatically restart the application.

You can test the new endpoints by hitting:

- http://localhost:8080/details to list all trading companies
- http://localhost:8080/details/MSFT to get details for a particular ticker
- http://localhost:8080/details/PIZZA to see what happens for an unknown ticker

Let's add a method called `getLatestQuote(String ticker)` on the `QuotesClient` which will get the latest quote for a given ticker from the quotes feed.

- Reuse `quotesFeed` to get the actual feed
- Filter the stream of quotes with quotes matching the given ticker
- Take the next matching quote
- If no matching quote is found within 15 seconds, then return a fallback `Quote` for the ticker.

Now, we want to combine data produced by `stock-quotes` and `stock-details` as a `TradingCompanySummary`.

Copy the following class to your project.

trading-service/src/main/java/io/spring/workshop/tradingservice/TradingCompanySummary.java

```
package io.spring.workshop.tradingservice;

public class TradingCompanySummary {

    private final Quote latestQuote;

    private final TradingCompany tradingCompany;

    public TradingCompanySummary(TradingCompany tradingCompany, Quote latestQuote) {
        this.latestQuote = latestQuote;
        this.tradingCompany = tradingCompany;
    }

    public Quote getLatestQuote() {
        return latestQuote;
    }

    public TradingCompany getTradingCompany() {
        return tradingCompany;
    }
}
```

Now, add another method to the `QuotesController` which can handle requests to `/quotes/summary/{ticker}`.

- Use the `TradingCompanyClient` to get the details for the company with the given ticker
- Create a `TradingCompanySummary` by composing the details with the latest quote from the `QuotesClient`

You can compose two reactive results using `Mono.zip(monoA, monoB, biFunction)` or `monoA.zipWith(monoB, biFunction)`.

We need to handle the `TickerNotFoundException` emitted by the `TradingCompanyClient` as an HTTP 404. You will need to create a method annotated with `@ExceptionHandler`.

Again, because of DevTools we can just recompile and test by hitting:

- http://localhost:8080/quotes/summary/MSFT to get the summary for a particular ticker
- http://localhost:8080/quotes/summary/PIZZA to test the 404 NOT FOUND an unknown ticker

Now, we will look at creating a functional-style WebFlux application by re-creating the Stock Quotes Application. Fasten your seatbelt and remove the stock-quotes directory!

Stock Quotes application

Create this application

Go to `https://start.spring.io` and create a Maven project with Spring Boot 2.0.4.RELEASE, with groupId `io.spring.workshop` and artifactId `stock-quotes`. Select the `Reactive Web` and `Devtools` dependencies. Unzip the given file into a directory and import that application into your IDE.

If generated right, you should have a main `Application` class that looks like this:

stock-quotes/src/main/java/io/spring/workshop/stockquotes/StockQuotesApplication.java

```
package io.spring.workshop.stockquotes;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class StockQuotesApplication {

    public static void main(String[] args) {
        SpringApplication.run(StockQuotesApplication.class, args);
    }
}
```

Edit your `application.properties` file to start the server on port 8081.

stock-quotes/src/main/resources/application.properties

```
server.port=8081
```

Launching it from your IDE or with `mvn spring-boot:run` should start a Netty server on port 8081. You should see in the logs something like:

```
INFO 2208 --- [  restartedMain] o.s.b.web.embedded.netty.NettyWebServer  : Netty
started on port(s): 8081
INFO 2208 --- [  restartedMain] i.s.w.s.StockQuotesApplication           : Started
StockQuotesApplication in 1.905 seconds (JVM running for 3.075)
```

Create a Quote Generator

To simulate real stock values, we'll create a generator that emits such values at a specific interval. Copy the following classes to your project.

stock-quotes/src/main/java/io/spring/workshop/stockquotes/Quote.java

```
package io.spring.workshop.stockquotes;

import java.math.BigDecimal;
import java.math.MathContext;
import java.time.Instant;

public class Quote {

    private static final MathContext MATH_CONTEXT = new MathContext(2);

    private String ticker;

    private BigDecimal price;

    private Instant instant = Instant.now();

    public Quote() {
    }

    public Quote(String ticker, BigDecimal price) {
        this.ticker = ticker;
        this.price = price;
    }

    public Quote(String ticker, Double price) {
        this(ticker, new BigDecimal(price, MATH_CONTEXT));
    }

    public String getTicker() {
        return ticker;
    }

    public void setTicker(String ticker) {
        this.ticker = ticker;
    }

    public BigDecimal getPrice() {
        return price;
    }

    public void setPrice(BigDecimal price) {
        this.price = price;
    }
```

```
    public Instant getInstant() {
        return instant;
    }

    public void setInstant(Instant instant) {
        this.instant = instant;
    }

    @Override
    public String toString() {
        return "Quote{" +
                "ticker='" + ticker + '\'' +
                ", price=" + price +
                ", instant=" + instant +
                '}';
    }
}
```

stock-quotes/src/main/java/io/spring/workshop/stockquotes/QuoteGenerator.java

```
package io.spring.workshop.stockquotes;

import java.math.BigDecimal;
import java.math.MathContext;
import java.time.Duration;
import java.time.Instant;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.stream.Collectors;

import reactor.core.publisher.Flux;

import org.springframework.stereotype.Component;

@Component
public class QuoteGenerator {

    private final MathContext mathContext = new MathContext(2);

    private final Random random = new Random();

    private final List<Quote> prices = new ArrayList<>();

    private final Flux<Quote> quoteStream;

    /**
     * Bootstraps the generator with tickers and initial prices
     */
    public QuoteGenerator() {
        initializeQuotes();
```

```
        this.quoteStream = getQuoteStream();
    }

    public Flux<Quote> fetchQuoteStream() {
        return quoteStream;
    }

    private void initializeQuotes() {
        this.prices.add(new Quote("CTXS", 82.26));
        this.prices.add(new Quote("DELL", 63.74));
        this.prices.add(new Quote("GOOG", 847.24));
        this.prices.add(new Quote("MSFT", 65.11));
        this.prices.add(new Quote("ORCL", 45.71));
        this.prices.add(new Quote("RHT", 84.29));
        this.prices.add(new Quote("VMW", 92.21));
    }


    private Flux<Quote> getQuoteStream() {
        return Flux.interval(Duration.ofMillis(200))
                .onBackpressureDrop()
                .map(this::generateQuotes)
                .flatMapIterable(quotes -> quotes)
                .share();
    }

    private List<Quote> generateQuotes(long i) {
        Instant instant = Instant.now();
        return prices.stream()
                .map(baseQuote -> {
                    BigDecimal priceChange = baseQuote.getPrice()
                            .multiply(new BigDecimal(0.05 * this.random.nextDouble()),
this.mathContext);

                    Quote result = new Quote(baseQuote.getTicker(),
baseQuote.getPrice().add(priceChange));
                    result.setInstant(instant);
                    return result;
                })
                .collect(Collectors.toList());
    }
}
```

Because we're working with `java.time.Instant` and Jackson, we should import the dedicated module in our app.

The `QuoteGenerator` instantiates a `Flux<Quote>` that emits a `Quote` every 200 msec and can be **shared** between multiple subscribers (look at the `Flux` operators for that). This instance is kept as an attribute for reusability.

stock-quotes/pom.xml

```
        <dependency>
            <groupId>com.fasterxml.jackson.datatype</groupId>
            <artifactId>jackson-datatype-jsr310</artifactId>
        </dependency>
```

Functional web applications with "WebFlux.fn"

Spring WebFlux comes in two flavors of web applications: annotation based and functional. For this first application, we'll use the functional variant.

Incoming HTTP requests are handled by a `HandlerFunction`, which is essentially a function that takes a ServerRequest and returns a `Mono<ServerResponse>`. The annotation counterpart to a handler function would be a Controller method.

But how those incoming requests are routed to the right handler?

We're using a `RouterFunction`, which is a function that takes a `ServerRequest`, and returns a `Mono<HandlerFunction>`. If a request matches a particular route, a handler function is returned; otherwise it returns an empty `Mono`. The `RouterFunction` has a similar purpose as the `@RequestMapping` annotation in `@Controller` classes.

Take a look at the code samples in the Spring WebFlux.fn reference documentation

Create your first HandlerFunction + RouterFunction

First, create a `QuoteHandler` class and mark is as a `@Component`;this class will have all our handler functions as methods. Let's inject our `QuoteGenerator` instance in our `QuoteHandler`.

Now create a `streamQuotes` handler that streams the generated quotes with the `"application/stream+json"` content type.

To route requests to that handler, you need to expose a `RouterFunction` to Spring Boot. Create a `QuoteRouter` configuration class (i.e. annotated with `@Configuration`) that creates a bean of type `RouterFunction<ServerResponse>`.

Modify that class so that GET requests to `"/quotes"` are routed to the handler you just implemented.

Since `QuoteHandler` is a component, you can inject it in `@Bean` methods as a method parameter.

Your application should now behave like this:

```
$ curl http://localhost:8081/quotes -i -H "Accept: application/stream+json"
HTTP/1.1 200 OK
transfer-encoding: chunked
Content-Type: application/stream+json

{"ticker":"CTXS","price":84.0,"instant":1494841666.633000000}
{"ticker":"DELL","price":67.1,"instant":1494841666.834000000}
{"ticker":"GOOG","price":869,"instant":1494841667.034000000}
{"ticker":"MSFT","price":66.5,"instant":1494841667.231000000}
{"ticker":"ORCL","price":46.13,"instant":1494841667.433000000}
{"ticker":"RHT","price":86.9,"instant":1494841667.634000000}
{"ticker":"VMW","price":93.7,"instant":1494841667.833000000}
```

Integration tests with WebTestClient

Spring WebFlux (actually the `spring-test` module) includes a `WebTestClient` that can be used to test WebFlux server endpoints with or without a running server. Tests without a running server are comparable to MockMvc from Spring MVC where mock request and response are used instead of connecting over the network using a socket. The WebTestClient however can also perform tests against a running server.

You can check that your last endpoint is working properly with the following integration test:

stock-quotes/src/test/java/io/spring/workshop/stockquotes/StockQuotesApplicationTests.java

```
package io.spring.workshop.stockquotes;

import java.util.List;

import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.reactive.server.WebTestClient;

import static org.assertj.core.api.Assertions.assertThat;

@RunWith(SpringRunner.class)
//  We create a `@SpringBootTest`, starting an actual server on a `RANDOM_PORT`
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class StockQuotesApplicationTests {

    // Spring Boot will create a `WebTestClient` for you,
    // already configure and ready to issue requests against "localhost:RANDOM_PORT"
    @Autowired
    private WebTestClient webTestClient;

    @Test
    public void fetchQuotes() {
        List<Quote> result =
                webTestClient
                // We then create a GET request to test an endpoint
                .get().uri("/quotes")
                .accept(MediaType.APPLICATION_STREAM_JSON)
                .exchange()
                // and use the dedicated DSL to test assertions against the response
                .expectStatus().isOk()
                .expectHeader().contentType(MediaType.APPLICATION_STREAM_JSON)
                .returnResult(Quote.class)
                .getResponseBody()
                .take(20)
                .collectList()
                .block();

        assertThat(result).allSatisfy(quote ->
assertThat(quote.getPrice()).isPositive());
    }

}
```

Additional Resources

Talks on Spring Reactive:

- Designing, Implementing, and Using Reactive APIs (Ben Hale, Paul Harris)
- Reactive Spring (Josh Long, Mark Heckler)

Hands-on Reactor:

- Reactor Hands-on