Creating Enterprise Reactive Apps Part One

1. Project Reactor and Spring WebFlux
2. Implementing a reactive data layer
3. Implementing a reactive service layer
4. Testing the reactive service layer

Demo project: **`demo-webflux`**

1. Project Reactor and Spring WebFlux

- Overview of Project Reactor
- Overview of Spring WebFlux
- Overview of the demo application
- Adding support for Spring WebFlux
- Adding support for Reactive MongoDB
- Test dependencies

Overview of Project Reactor

- The previous chapter covered Reactive Streams, available since Java 9
  - Specifies 4 interfaces…
  - `Publisher`, `Subscriber`, `Subscription`, `Processor`

- Project Reactor is an open-source library from Pivotal that builds on Reactive Streams, see https://projectreactor.io/
  - Provides 2 specializations of `Publisher<T>`…
  - `Mono<T>` is a publisher that produces 0 or 1 value
  - `Flux<T>` is a publisher that produces 0 or more values

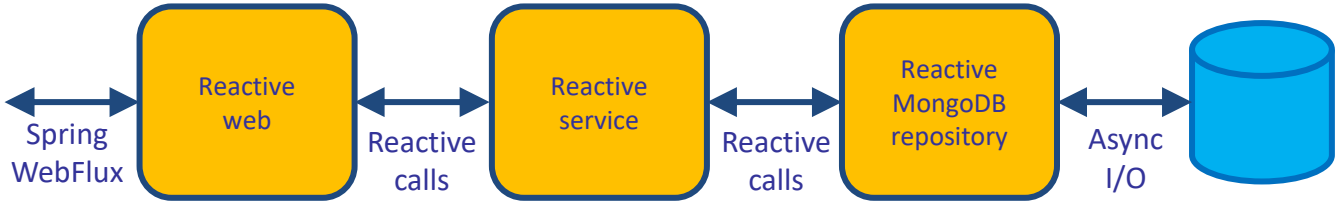
- Recommended usage:
  - Method params - use `Publisher<T>` for substitutability
  - Method results - use `Mono<T>` or `Flux<T>` for specificity

Overview of Spring WebFlux

- Spring Boot 2 introduced support for Spring WebFlux
  - Enables you to create reactive web applications
  - Produce and consume HTTP resources, WebSockets, SSE
- Should I use Spring WebFlux or Spring MVC?
  - Use Spring WebFlux for apps that create huge/streaming data
  - Use Spring MVC for CRUD-style applications
- Spring WebFlux doesn't depend on the Servlet APIs
  - It has a new reactive web runtime
  - There are adapters to the Servlet API, if you still want to use it

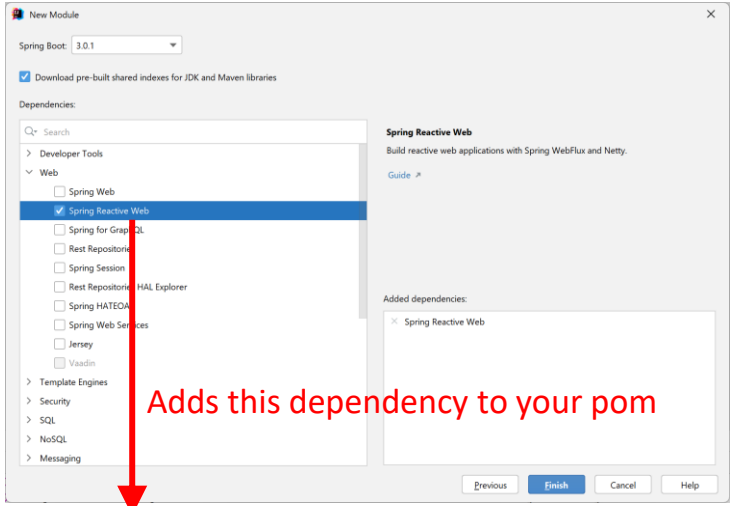
Overview of the Demo Application

- We're going to see how to build a complete reactive app using Spring WebFlux
  - We'll define a reactive data repository for a MongoDB database
  - We'll define a reactive service layer, to consume the repository
  - We'll define a reactive Web layer, to consume the service
  - The reactive Web layer will be implemented using Spring WebFlux



Adding Support for Spring WebFlux

- When you create a Spring project using Spring Initializr, add the Spring Reactive Web dependency
  - Uses Netty by default, but you can switch to use Tomcat, Jetty

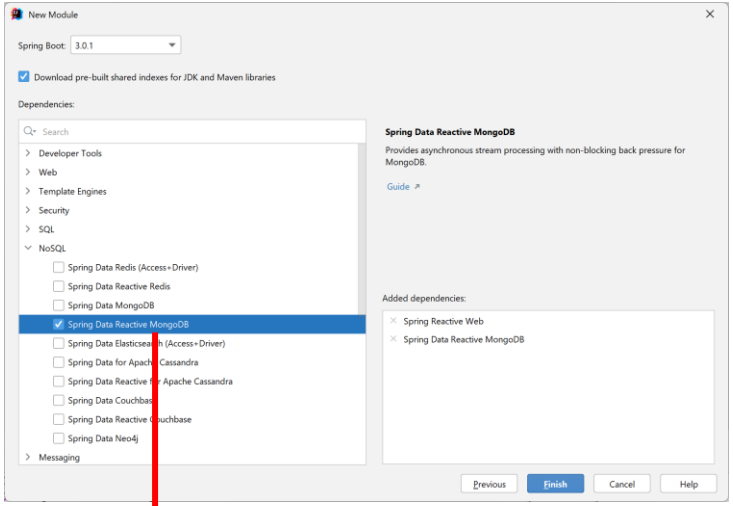



```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

pom.xml

Adding Support for Reactive MongoDB

- If you want to use reactive data access, e.g. a MongoDB database, then add the appropriate dependency



Adds this dependency to your pom

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
</dependency>
```

pom.xml

Test Dependencies

- Your pom file will contain two sets of test dependencies
  - JUnit tests
  - Project Reactor tests

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-test</artifactId>
    <scope>test</scope>
</dependency>
```

pom.xml

2. Implementing a Reactive Data Layer

- Overview
- MongoDB documents
- Defining a reactive data repository
- Seeding the MongoDB database
- Setting the active profile
- Running the application
- Reminder of the big picture

Overview

- To obtain the true value of reactive systems, it really helps if the database driver itself supports async I/O ☺
  - Otherwise you won't be able to scale-out reads without scaling-out threads, which is exactly what you want to avoid

- Spring Data has several reactive data repositories
  - Reactive MongoDB
  - Reactive Redis
  - Reactive Cassandra
  - Etc.

- You just need to add the appropriate Spring Boot starter to your project - we added Spring Data Reactive MongoDB

MongoDB Documents (1 of 2)

- MongoDB is a document-oriented database
  - A MongoDB document is a BSON object (effectively binary JSON)
  - MongoDB documents contain fieldname/value pairs

```
var financialTransaction = {
    _id:    ObjectId("21aa914e0405a59ce30a94a2"),   // Unique ID for this object.
    amount: 5000,
    when:   new Date('Jul 2, 1997')
}
```

MongoDB Documents (2 of 2)

- In work with MongoDB documents in Java:
  - Define a class and annotate with `@Document`
  - Define fields as appropriate, plus an ID field annotated with

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

@Document
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Tx {

    @Id
    private String id;

    private double amount;
    private LocalDateTime when;
}
```

Tx.java

- Conceptually similar to defining entity classes in JPA

Defining a Reactive Data Repository (1 of 4)

- Spring Data is a data-access abstraction mechanism in Spring Boot
  - Makes it much easier to access a wide range of data stores

- It provides the `CrudRepository` interface that specifies synchronous I/O operations
  - Sub-interfaces available for many types of data source
  - E.g. `MongoRepository`

- It provides the `ReactiveCrudRepository` interface that specifies asynchronous I/O operations
  - Sub-interfaces available for many types of data source
  - E.g. `ReactiveMongoRepository`

Defining a Reactive Data Repository (2 of 4)

- **`ReactiveCrudRepository`**
  - Specifies CRUD operations in an agnostic manner
- All methods are reactive
  - They return a `Publisher`…
  - Either `Mono` (0-1 result)
  - Or `Flux` (* results)
- In your client code:
  - Call a method
  - Subscribe to returned publisher
  - Obtain result(s) reactively

Modifier and Type	Method and Description
`reactor.core.publisher.Mono<Long>`	`count()` Returns the number of entities available.
`reactor.core.publisher.Mono<Void>`	`delete(T entity)` Deletes a given entity.
`reactor.core.publisher.Mono<Void>`	`deleteAll()` Deletes all entities managed by the repository.
`reactor.core.publisher.Mono<Void>`	`deleteAll(Iterable<? extends T> entities)` Deletes the given entities.
`reactor.core.publisher.Mono<Void>`	`deleteAll(org.reactivestreams.Publisher<? extends T> entityStream)` Deletes the given entities supplied by a `Publisher`.
`reactor.core.publisher.Mono<Void>`	`deleteById(ID id)` Deletes the entity with the given id.
`reactor.core.publisher.Mono<Void>`	`deleteById(org.reactivestreams.Publisher<ID> id)` Deletes the entity with the given id supplied by a `Publisher`.
`reactor.core.publisher.Mono<Boolean>`	`existsById(ID id)` Returns whether an entity with the given id exists.
`reactor.core.publisher.Mono<Boolean>`	`existsById(org.reactivestreams.Publisher<ID> id)` Returns whether an entity with the given id, supplied by a `Publisher`, exists.
`reactor.core.publisher.Flux<T>`	`findAll()` Returns all instances of the type.
`reactor.core.publisher.Flux<T>`	`findAllById(Iterable<ID> ids)` Returns all instances of the type `T` with the given IDs.
`reactor.core.publisher.Flux<T>`	`findAllById(org.reactivestreams.Publisher<ID> idStream)` Returns all instances of the type `T` with the given IDs supplied by a `Publisher`.
`reactor.core.publisher.Mono<T>`	`findById(ID id)` Retrieves an entity by its id.
`reactor.core.publisher.Mono<T>`	`findById(org.reactivestreams.Publisher<ID> id)` Retrieves an entity by its id supplied by a `Publisher`.
`<S extends T>` `reactor.core.publisher.Mono<S>`	`save(S entity)` Saves a given entity.
`<S extends T>` `reactor.core.publisher.Flux<S>`	`saveAll(Iterable<S> entities)` Saves all given entities.
`<S extends T>` `reactor.core.publisher.Flux<S>`	`saveAll(org.reactivestreams.Publisher<S> entityStream)` Saves all given entities.

Defining a Reactive Data Repository (3 of 4)

- **`ReactiveMongoRepository`**
  - Inherits the basic methods from the previous slide and adds a few more…

Modifier and Type	Method and Description
`<S extends T> reactor.core.publisher.Flux<S>`	`findAll(Example<S> example)`
`<S extends T> reactor.core.publisher.Flux<S>`	`findAll(Example<S> example, Sort sort)`
`<S extends T> reactor.core.publisher.Flux<S>`	`insert(Iterable<S> entities)` Inserts the given entities.
`<S extends T> reactor.core.publisher.Flux<S>`	`insert(org.reactivestreams.Publisher<S> entities)` Inserts the given entities.
`<S extends T> reactor.core.publisher.Mono<S>`	`insert(S entity)` Inserts the given entity.

Methods inherited from interface org.springframework.data.repository.reactive.ReactiveSortingRepository

`findAll`

Methods inherited from interface org.springframework.data.repository.reactive.ReactiveCrudRepository

`count, delete, deleteAll, deleteAll, deleteAll, deleteById, deleteById, existsById, existsById, findAll, findAllById, findAllById, findById, findById, save, saveAll, saveAll`

Methods inherited from interface org.springframework.data.repository.query.ReactiveQueryByExampleExecutor

`count, exists, findOne`

Defining a Reactive Data Repository (4 of 4)

- To define a reactive MongoDB repository in your app:
  - Define an interface that extends `ReactiveMongoRepository`
  - Specify the document type and the ID type
  - Optionally define custom finder methods

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

interface TxRepository extends ReactiveMongoRepository<Tx, String> {
}
```

TxRepository.java

- This is what a custom finder method might look like
  - This method finds all documents in the MongoDB database that have a `when` attribute equal to the specified value

```
Flux<Tx> findByWhen(LocalDateTime when);
```

Seeding the MongoDB Database (1 of 3)

- For demo purposes, it's handy to seed the MongoDB database with some sample `Tx` documents
  - See following slides for explanations of this code…

```
@Log4j2
@Component
@Profile("onlyForDemoPurposes")
public class SeedDb implements ApplicationListener<ApplicationReadyEvent> {

    private final TxRepository repo;

    public SeedDb(TxRepository repo) { this.repo = repo; }

    @Override
    public void onApplicationEvent(ApplicationReadyEvent event) {
        repo.deleteAll()
            .thenMany(Flux.just(100.0, 200.0, 300.0)
                          .map(amount -> new Tx(UUID.randomUUID().toString(),
                                                amount,
                                                LocalDateTime.now()))
                          .flatMap(tx -> repo.save(tx))
            )
            .thenMany(repo.findAll())
            .subscribe(tx -> log.info("Tx document successfully inserted: " + tx));
    }
}
```

SeedDb.java

Seeding the MongoDB Database (2 of 3)

```
@Log4j2
@Component
@Profile("onlyForDemoPurposes")
public class SeedDb implements ApplicationListener<ApplicationReadyEvent> {

    private final TxRepository repo;

    public SeedDb(TxRepository repo) { this.repo = repo; }

    …
}
```

- We only want database seeding when we're in "demo" mode, not when we're in production mode
- The `ApplicationListener` interface enables us to handle lifecycle events (`ApplicationReadyEvent` is the final event)
- If a component has only a single constructor, you can omit `@Autowired` to inject dependencies into the constructor

Seeding the MongoDB Database (3 of 3)

```
repo.deleteAll()
    .thenMany(Flux.just(100.0, 200.0, 300.0)
                  .map(amount -> new Tx(UUID.randomUUID().toString(),
                                        amount,
                                        LocalDateTime.now()))
                  .flatMap(tx -> repo.save(tx))
    )
    .thenMany(repo.findAll())
    .subscribe(tx -> log.info("Tx document successfully inserted: " + tx));
```

- `deleteAll()` returns a `Mono` (i.e. a publisher), and publishers support chained processing via `thenMany()`
- `Flux.just()` is a factory method that creates a new publisher with a static list of items (numbers here)
- `map()` receives each number and maps it to a `Tx` object
- `flatMap()` saves each `Tx` object to the MongoDB database
- `thenMany()` finds all the docs, as a sanity check
- `subscribe()` is necessary because publishers are lazy, i.e. you must subscribe to them to trigger their execution

Setting the Active Profile

- The `SeedDb` component only kicks in if the active profile includes "`onlyForDemoPurposes`"

```
@Log4j2
@Component
@Profile("onlyForDemoPurposes")
public class SeedDb implements ApplicationListener<ApplicationReadyEvent> {
    …
}
```

SeedDb.java

- There are lots of ways to set an active profile
  - E.g. in the `application.properties` file

```
spring.profiles.active=onlyForDemoPurposes
```

application.properties

Running the Application

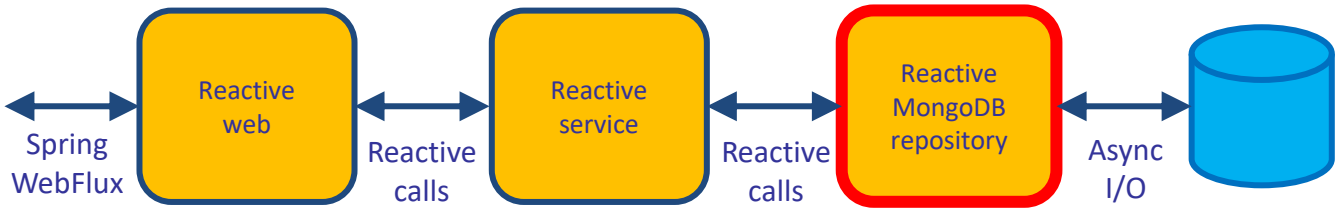
- Run the Spring Boot application class
  - i.e. `Application`

- See the following log messages in the console

```
2020-02-06 23:02:39.884  INFO 15004 --- [           main] demo.webflux.DemowebfluxApplication      : Started DemowebfluxApplication in 3.882 seconds
2020-02-06 23:02:40.007  INFO 15004 --- [localhost:27017] org.mongodb.driver.connection          : Opened connection [connectionId{localValue:1,
2020-02-06 23:02:40.011  INFO 15004 --- [localhost:27017] org.mongodb.driver.cluster             : Monitor thread successfully connected to serve
2020-02-06 23:02:40.086  INFO 15004 --- [ntLoopGroup-2-2] org.mongodb.driver.connection          : Opened connection [connectionId{localValue:3,
2020-02-06 23:02:40.238  INFO 15004 --- [ntLoopGroup-2-3] org.mongodb.driver.connection          : Opened connection [connectionId{localValue:4, s
2020-02-06 23:02:40.266  INFO 15004 --- [ntLoopGroup-2-4] org.mongodb.driver.connection          : Opened connection [connectionId{localValue:5, se
2020-02-06 23:02:40.376  INFO 15004 --- [ntLoopGroup-2-4] demo.webflux.SeedDb                    : Tx document successfully inserted: Tx(id=1fdaa86
2020-02-06 23:02:40.378  INFO 15004 --- [ntLoopGroup-2-4] demo.webflux.SeedDb                    : Tx document successfully inserted: Tx(id=ed7eb1
2020-02-06 23:02:40.379  INFO 15004 --- [ntLoopGroup-2-4] demo.webflux.SeedDb                    : Tx document successfully inserted: Tx(id=86ca87
```

Reminder of the Big Picture

- Here's a reminder of the structure of the demo app
  - This section has shown how to implement the reactive MongoDB repository layer, to do async I/O to a MongoDB database



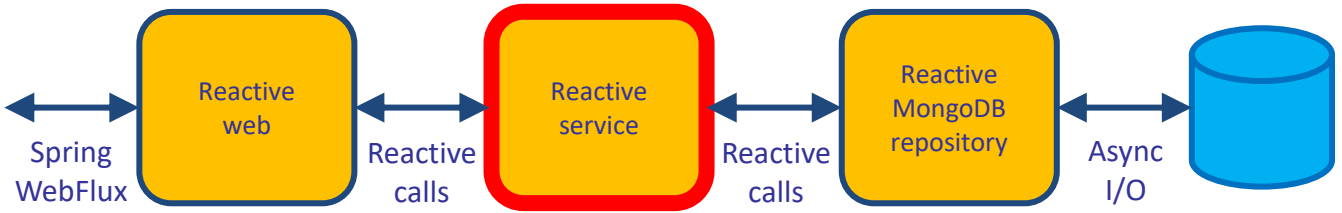
3. Implementing a Reactive Service Layer

- Overview
- Defining a service class
- Implementing service operations

Overview

- In the previous section we implemented a reactive MongoDB repository
  - Performs async (non-blocking) I/O on a MongoDB database

- In this section we implement a reactive Spring service
  - A business component that consumes the reactive repository



Defining a Service Class

- The service class is a regular Spring component, and we autowire two components:
  - `TxRepository` - to do reactive data access
  - `ApplicationEventPublisher` - to publish Spring events

```
import org.springframework.context.ApplicationEventPublisher;

@Log4j2
@Service
public class TxService {

    private final TxRepository repo;
    private final ApplicationEventPublisher pub;

    TxService(TxRepository repo, ApplicationEventPublisher pub) {
        this.repo = repo;
        this.pub = pub;
    }
    …
}
```

TxService.java

Implementing Service Operations (1 of 4)

- In our demo, the service is a skinny wrapper over the repo
  - It shows how to consume reactive repository operations

- Our service has a couple of "getter" methods
  - `getAll()` returns many items reactively, i.e. a `Flux<Tx>`
  - `getById()` returns one item reactively, i.e. a `Mono<Tx>`

```
public class TxService {

    public Flux<Tx> getAll() {
        return repo.findAll();
    }

    public Mono<Tx> getById(String id) {
        return repo.findById(id);
    }
    …
```

TxService.java

Implementing Service Operations (2 of 4)

- Our service has a method to create a financial transaction
  - Calls repository's `save()` method, which returns `Mono<Tx>`
  - `Mono` and `Flux` have many hook methods, e.g. `doOnSuccess()`
  - `doOnSuccess()` takes a `Consumer<Tx>` to process the result

```
public Mono<Tx> create(double amount) {
    checkAmount(amount);
    return repo.save(new Tx(null, amount, LocalDateTime.now()))
               .doOnSuccess(tx -> pub.publishEvent(new TxCreatedEvent(tx)));
}
```

- Our service also publishes an event if the tx value is high

```
private void checkAmount(double amount) {
    if (amount > 1_000_000) {
        pub.publishEvent(new TxHighValueEvent(amount));
    }
}
```

TxService.java

Implementing Service Operations (3 of 4)

- Our service has a method to update a financial transaction
  - Calls repository's `findById()` method, returns a `Mono<T>`
  - We call `map()` on the `Mono<Tx>` to update it locally
  - Then we call `flatMap()` to save it to the database reactively

```
public Mono<Tx> update(String id, double amount, LocalDateTime when) {
    checkAmount(amount);
    return repo.findById(id)
               .map(t -> new Tx(t.getId(), amount, when))
               .flatMap(t -> repo.save(t).thenReturn(t));
}
```

TxService.java

- Use `map()` for sync operation, returns result immediately
  - `map( Function<T,R> )`
- Use `flatMap()` for async operation, returns a publisher
  - `flatMap( Function<T,Publisher<R>> )`

Implementing Service Operations (4 of 4)

- Our service has a method to delete a financial transaction
  - Similar idea to previous slide
  - What does it return?

```
public Mono<Tx> delete(String id) {
    return repo.findById(id)
               .flatMap(t -> repo.deleteById(t.getId()).thenReturn(t));
}
```

TxService.java

4. Testing the Reactive Service Layer

- Overview
- Test dependencies
- Defining a test class
- Autowiring dependencies into the test class
- How to define reactive tests
- Defining reactive tests
- Exercise

Overview

- In this section we'll see how to test the functions in our reactive service class

- The service methods are reactive
  - i.e. they return publishers (Mono or Flux)
  - So we need a way to test results asynchronously

- Project Reactor provides a test API that allows us to test reactive publishers
  - We can subscribe to a publisher and specify expectations about the results it will publish

Test Dependencies

- Here's a reminder of the test dependencies in the pom file
  - JUnit 5
  - Project Reactor tests

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-test</artifactId>
    <scope>test</scope>
</dependency>
```

pom.xml

Defining a Test Class

- Here's a test class for our reactive service

```
@Log4j2
@DataMongoTest
@Import(TxService.class)
public class TxServiceModificationsTest {
    …
}
```

TxServiceModificationsTest.java

- `@DataMongoTest` and `@Import` create a *test slice*
  - They add components into the application context

- Specifically…
  - `@DataMongoTest` adds MongoDB-related components
  - `@Import` adds a `TxService` component

Autowiring Dependencies into the Test Class

- You can autowire dependencies into the test class

```
@Log4j2
@DataMongoTest
@Import(TxService.class)
public class TxServiceModificationsTest {

    private final TxService service;
    private final TxRepository repo;

    public TxServiceModificationsTest(@Autowired TxService service,
                                      @Autowired TxRepository repo) {
        this.service = service;
        this.repo = repo;
    }
    …
}
```

TxServiceModificationsTest.java

How to Define Reactive Tests

- To define a reactive test:
  - Invoke a reactive method-under-test (i.e. it returns a publisher)
  - Then use `StepVerifier` to verify it publishes expected results

- Here's the general way to use a `StepVerifier`:
  - Call `StepVerifier.create()` to set up a `StepVerifier`
  - Call `expectXxx()` to specify what you expect to be published
  - Call `verifyXxx()` to trigger the expected results were published

```
Publisher<XXX> resultPublisher = someReactiveMethod();

StepVerifier.create(resultPublisher)
            .expectNext(next-expected-published-result)
            .expectNextMatches(predicate-to-test-next-published-result)
            .expectNextCount(expected-number-of-published-items)
            .verifyComplete();
```

general syntax

Defining Reactive Tests (1 of 3)

- Here's a reactive test for the service `create()` method

```
@Test
public void createTx_returnsCreatedTxWithId() {

    Mono<Tx> createdTx = service.create(1234);

    StepVerifier
        .create(createdTx)
        .expectNextMatches(tx -> tx.getId() != null && tx.getId().length() != 0)
        .verifyComplete();
}
```

TxServiceModificationsTest.java

Defining Reactive Tests (2 of 3)

- Here's a reactive test for the service `update()` method

```
@Test
public void updateTx_txUpdated() {

    Mono<Tx> updatedTx = service.create(4321)
                                .flatMap(tx -> service.update(tx.getId(),
                                                              8888,
                                                              LocalDateTime.now()));


    StepVerifier
        .create(updatedTx)
        .expectNextMatches(tx -> tx.getAmount() == 8888)
        .verifyComplete();
}
```

TxServiceModificationsTest.java

Defining Reactive Tests (3 of 3)

- Here's a reactive test for the service `delete()` method

```
@Test
public void deleteTx_txDeleted() {

    Mono<Tx> deletedTx = service.create(5678)
                               .flatMap(tx -> service.delete(tx.getId()));

    StepVerifier
        .create(deletedTx)
        .expectNextMatches(tx -> tx.getAmount() == 5678)
        .verifyComplete();
}
```

TxServiceModificationsTest.java

Exercise

- Take a look at the test in here:
  - `TxServiceQueryTest.java`

- What does it do?

Summary

- Project Reactor and Spring WebFlux
- Implementing a reactive data layer
- Implementing a reactive service layer
- Testing the reactive service layer