



Reactive Spring Application

Introducing Reactive Programming

Objectives

After completing this lesson, you should be able to

- Describe the basic concepts of Reactive Programming
- Write a “reactive” Spring application

Agenda

- **What is Reactive Programming?**
- Reactive Features
- Reactive Stream Implementations
- Reactive Spring Features
- Lab

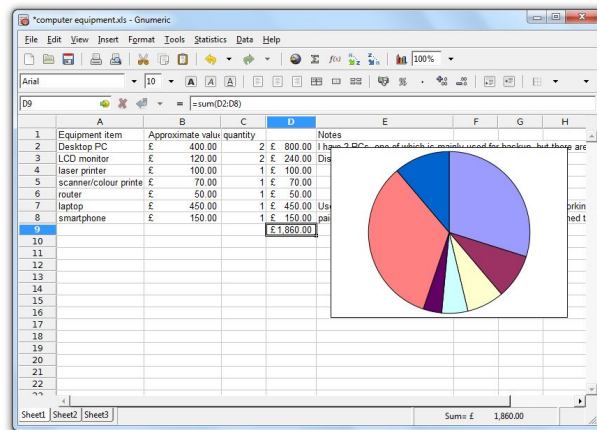


What is Reactive Programming?

- Programming with *asynchronous* data streams
 - Everything (almost) can be seen as a stream
 - Messages, variables, user input, data structures, ...
 - **Stream:** Ongoing events ordered in time
 - Series of user clicks through our web app
 - Calls to a web server over time
 - Separate values returned from a function
 - Rows returned from a DB query
 - **Event:** (almost) anything
 - User clicks, calls to a web server, function returning a result, row returned from a DB query

Familiar Example 1: Spreadsheets

- Spreadsheets
 - Formula cells in a spreadsheet automatically “react” to changes in the cells used by the formula
 - Spreadsheet recalculates whenever such cells are modified



Familiar Example 2: User Interfaces

- Implementing a GUI
 - Must respond to mouse/keyboard events
 - Setup *Listeners*
 - Respond to events by running handlers *asynchronously*
- JavaScript (AJAX, SPA) web-pages
 - Make a REST request to back-end server
 - Define a *call-back* for when data is returned
 - Call-back invoked *asynchronously*

Why Reactive?

- New applications and environments
 - Distributed multi-process applications
 - Cloud, PaaS, Microservices
- Challenges
 - Latency inevitable
 - Redundancy and recovery
 - Scale out, not up
- Imperative (traditional) logic becomes *very* complicated
 - Too many nested callbacks



projectreactor.io



Reactive Programming Examples



- Reactive is good for
 - Time series processing without storing state
 - Non-blocking processing - threads do not need to wait for calls to complete
 - Speed / Throughput - Far more efficient use of threads and memory
 - Building complex event-driven systems without “callback-hell”
- It is not appropriate or necessary for all applications
 - Considerable learning-curve

Agenda

- What is Reactive Programming?
- **Reactive Features**
- Reactive Stream Implementations
- Reactive Spring Features
- Lab
- Optional/Advanced ...



Reactive Programming

- Non-blocking applications
 - Asynchronous, event-driven
 - Scale using a *minimal* number of threads
 - Flow control (*backpressure*)
 - Stream processing
- Implications
 - Major shift from *imperative* style logic to a *declarative pipeline* of asynchronous logic
 - Intelligent routing and consumption of events
 - Comparable to **CompletableFuture** in Java 8 and composing follow-up actions via lambda expressions

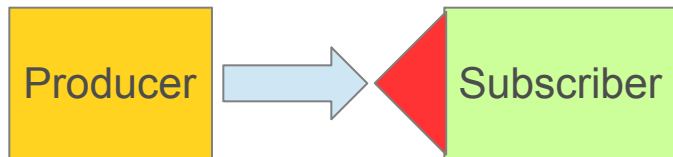
Asynchronous Components

- Similar to messaging systems
 - Independent components (tasks)
 - Respond to incoming events
 - Pass on results by generating events
 - *But components do not get their own thread*
 - Thread selects and runs components that are ready
 - Then switches to next ready-to-run component
 - Switching components *much* cheaper than thread-switching
- Analogous concepts
 - Actors, Coroutines, C.S.P.

*CSP = Communicating Sequential
Processes*

Back-Pressure

- Controls data flow through the reactive pipeline
 - Ensures producers don't overwhelm subscribers (consumers)
 - *Options:* Ignore excess events or block until ready
- *Example*
 - Pipeline of reactive components from the HTTP socket to database
 - Too many HTTP requests, data repository slows down or stops until capacity frees up



Back
Pressure

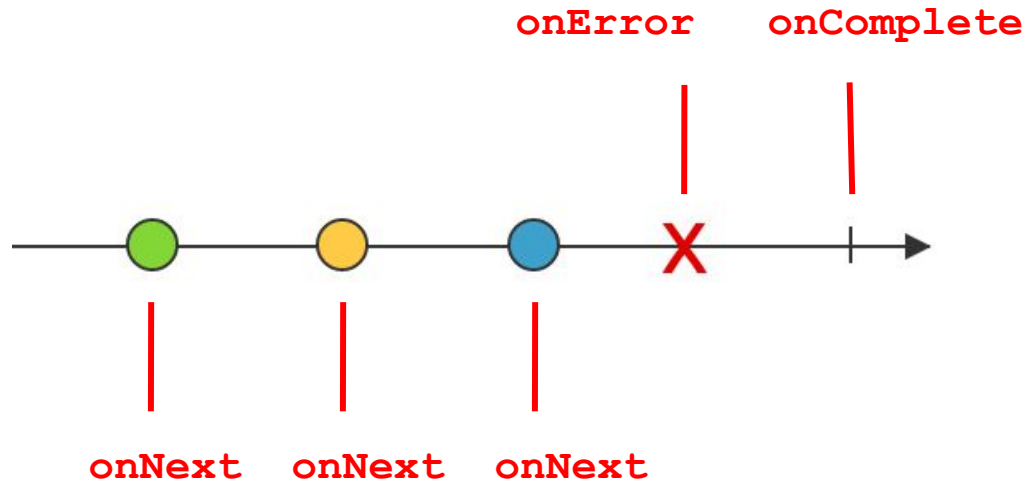
Agenda

- What is Reactive Programming?
- Reactive Features
- **Reactive Stream Implementations**
- Reactive Spring Features
- Lab
- Optional/Advanced ...



Handle Stream of Events Asynchronously

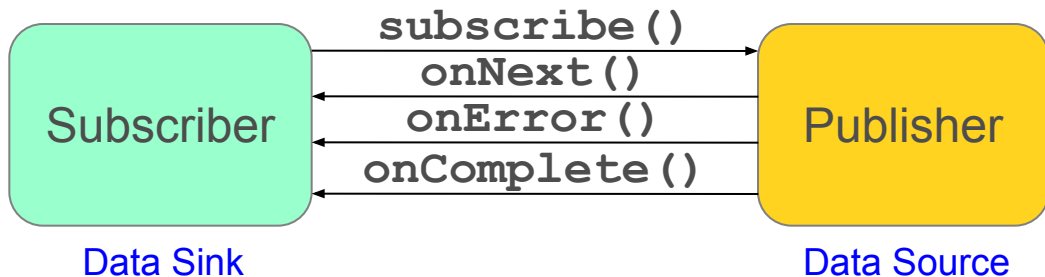
- Reactive Streams
 - Specification of *Interfaces*
 - Needs an implementation



Reactive Streams



- Require a publisher and a subscriber
 - Like messaging
- Publisher provides data & specifies how to process it
 - Produces a “stream” of 0, 1 or more data items
- Subscriber actually processes the events in the stream
 - *Nothing happens without a subscriber*



Reactive Stream Implementations

- Flow classes
 - Part of Java 9 JDK
- Project Reactor from Pivotal
 - Supported by Spring 5
 - <http://projectreactor.io>
- RxJava
 - Rx has several implementations
- Spring supports Reactor and RxJava
 - *Doesn't require Java 9*



PROJECT **REACTOR**



RxJava

Stream Types

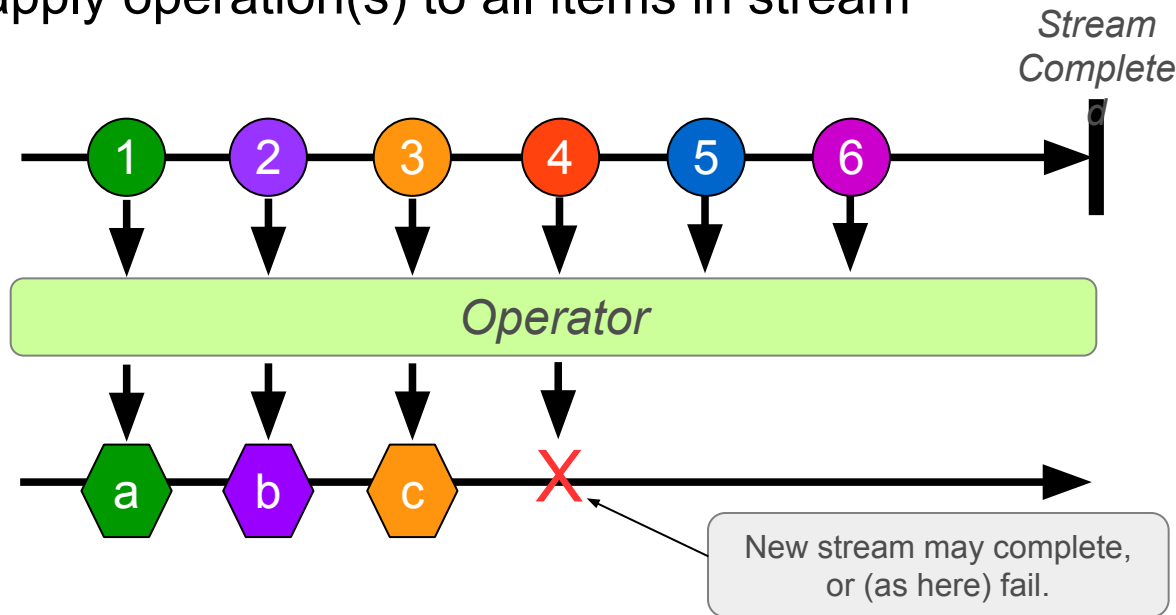


- A sequence of zero or more events
 - *Reactor* calls this *Flux*
 - *RxJava* calls this *Observable*
- Common special case
 - Stream of 0 or 1 events
 - *Reactor* calls this *Mono*
 - *RxJava* calls this *Single*
- *All are publishers of events*

Flux: Sequence of [0 .. N]

Flux, Observable
are Publishers

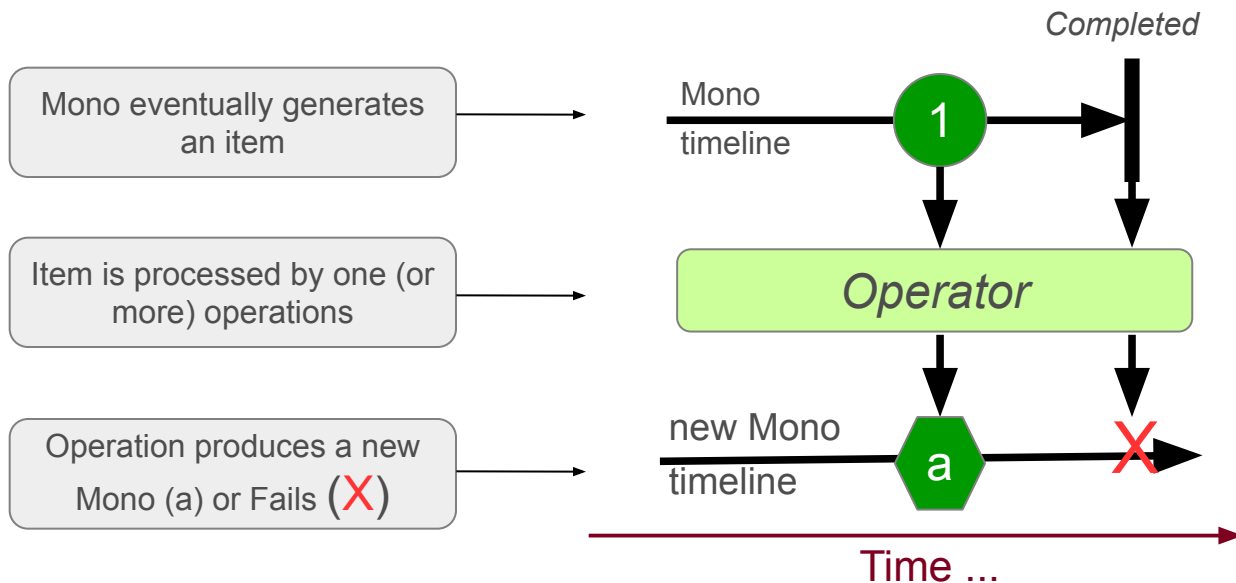
- Return continuous stream of events with failure handling
 - *Flux* (Reactor) or *Observable* (RxJava)
 - Can apply operation(s) to all items in stream



Mono: Sequence of [0 .. 1]

Mono and Single
are Publishers

- Used to return a single result (or fail doing so)
 - Mono* (Reactor) or *Single* (RxJava)



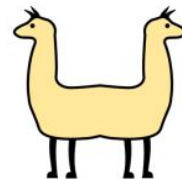
Subscribing to a Stream

Reactor Syntax

- Two options
 - Implement a **Subscriber<T>**
 - **onSubscribe**, **onNext**, **onError**, **onComplete**
 - Provide a **Consumer**
 - Has a single **accept(T)** – equivalent to **onNext**
 - Can pass a lambda

```
Flux.just( "red", "green", "blue" ) // The producer
    .log()                          // Stream operator: log item
    .map(String::toUpperCase)       // Stream operator: convert to upper case
    .subscribe(System.out::println); // Subscribe, Consumer prints each item
```

Java Streams are *not* Reactive



pushmi-pullyu

- Java Streams
 - Subscriber actively *pulls* in data, blocks if not available

```
List<Shop> shops = customer.getVendors();  
List<BigDecimal> discountedPrices =  
    shops.stream() // Fetches data from list  
        .map(Shop::getPrice)  
        .map(Discount::applyDiscount)  
        .collect(Collectors.toList());
```

- Reactive Streams
 - Data is sent (pushed) to subscriber via *callbacks*
 - By “pushing-back” we get back-pressure

Agenda

- What is Reactive Programming?
- Reactive Features
- Reactive Stream Implementations
- **Reactive Spring Features**
- Lab
- Optional/Advanced ...



Reactive Features in Spring

- Spring Data Reactive Repositories
 - Returns query results as a Stream
- Web Client
 - Reactive alternative to `RestTemplate`
- **WebFlux**
 - Reactive `@Controllers`

Reactive Spring Data Repository

Return immediately, even if
data is not available

Using
Reactor API

```
public interface CustomerRepository
    extends ReactiveCrudRepository<Customer, Long> {
    Mono<Customer> findBySocialSecurityNumber(String ssn);
    Flux<Customer> findBySuburb(String suburb);
}
```

Reactor defines **Mono** and **Flux**

Using
RxJava API

```
public interface CustomerRepository
    extends RxJavaCrudRepository<Customer, Long> {
    Single<Customer> findBySocialSecurityNumber(String ssn);
    Observable<Customer> findBySuburb(String suburb);
}
```

RxJava defines **single** and **Observable** instead

WebClient – 1

The Mono is processed
in the *same* thread

- Asynchronous alternative to RestTemplate
 - HTTP response is handled by *different* thread to HTTP request

```
WebClient client = WebClient.create(ACCOUNT_SERVER_URL);  
Mono<Account> result = client.get()  
    .uri("/accounts/{id}", id)           ← HTTP GET for account {id}  
    .accept(MediaType.APPLICATION_JSON)  
    .retrieve()    // Send request  
    .bodyToMono(Account.class);        ← Response (a Mono) so  
                                         block until available  
  
Account account = result.block();        ← // Wait for account to be returned
```

WebClient – Processing Alternative 1

Mono or Flux is processed
in a *different* thread

- Alternative ways of processing the Mono (or Flux)
 - Note you *must* **subscribe()** to do *anything*

```
// Option 1: Subscribe using lambda
result.subscribe(a -> {
    // For each item returned, log it
    logger.info("Account: " + a.getName())
});
```

Pass lambda to
subscribe()

WebClient – Processing Alternative 2

- Provide success & error callback processing
 - Note you still *must* **subscribe()** to do *anything*

// Option 2: Define success/failure callbacks

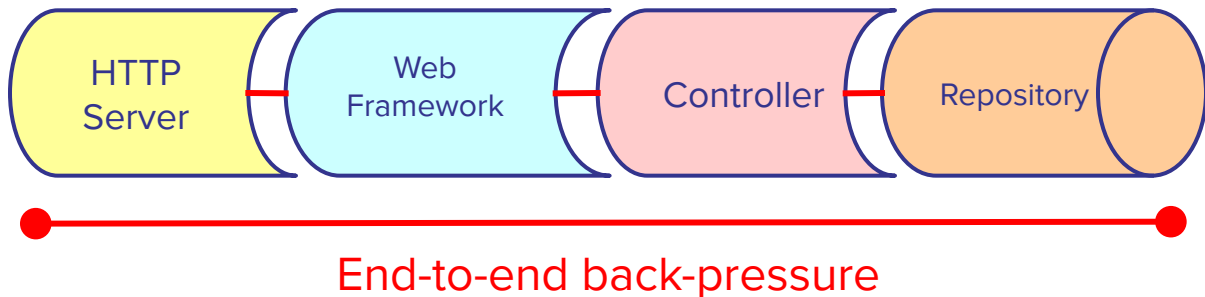
```
result.doOnSuccess(a -> {  
    // For each item returned  
    a -> logger.info("Account: " + a.getName());  
}).doOnError(e -> {  
    System.out.println(e.getMessage());  
}).subscribe();
```

These are just
callbacks

Must subscribe()

Spring WebFlux

- Consider incoming HTTP Requests as a stream
 - Process in usual way
 - Controllers return Reactive Streams
- *A Reactive Web Pipeline*



Reactive Web Controller

Using
Reactor API

```
@Controller
public class CustomerController {
    private final CustomerRepository customerRepository; // Reactive repository

    @Autowired
    public CustomerController (CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    @GetMapping("/customers/{id}")
    public Mono<Customer> getCustomer(@PathVariable Long id) {
        return customerRepository.findById(id); // Or return Single<Customer>
    }

    @GetMapping("/customers")
    public Flux<Customer> getCustomer() { // Or return Observable<Customer>
        return customerRepository.findAll();
    }
}
```

Getting Started

- Support in Spring Boot 2.0 and Spring Initializr

SPRING INITIALIZR

bootstrap your application now

Generate a

Maven Project

 with Spring Boot

2.0.0

Project Metadata

Artifact coordinates

Group

com.example

Artifact

demo

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

reactive Web

Reactive Web
Reactive web development with Tomcat and Spring Reactive (experimental)

Actuator
Production ready features to help you monitor and manage your application

Generate Project

Don't know what to look for? Want more options? [Switch to the full version.](#)

References



- Reactive Programming
 - Spring Blog article by Dave Syer
 - [Part 1](#)Part 1, [Part 2](#)Part 1, Part 2, [Part 3](#)
 - An [Introduction to Reactive](#) by André Staltz
 - Project Reactor [documentation](#)
- WebFlux
 - [Article](#) by Rossen Stoyanchev (Spring MVC lead)
 - [WebFlux](#) in Spring reference documentation



Summary

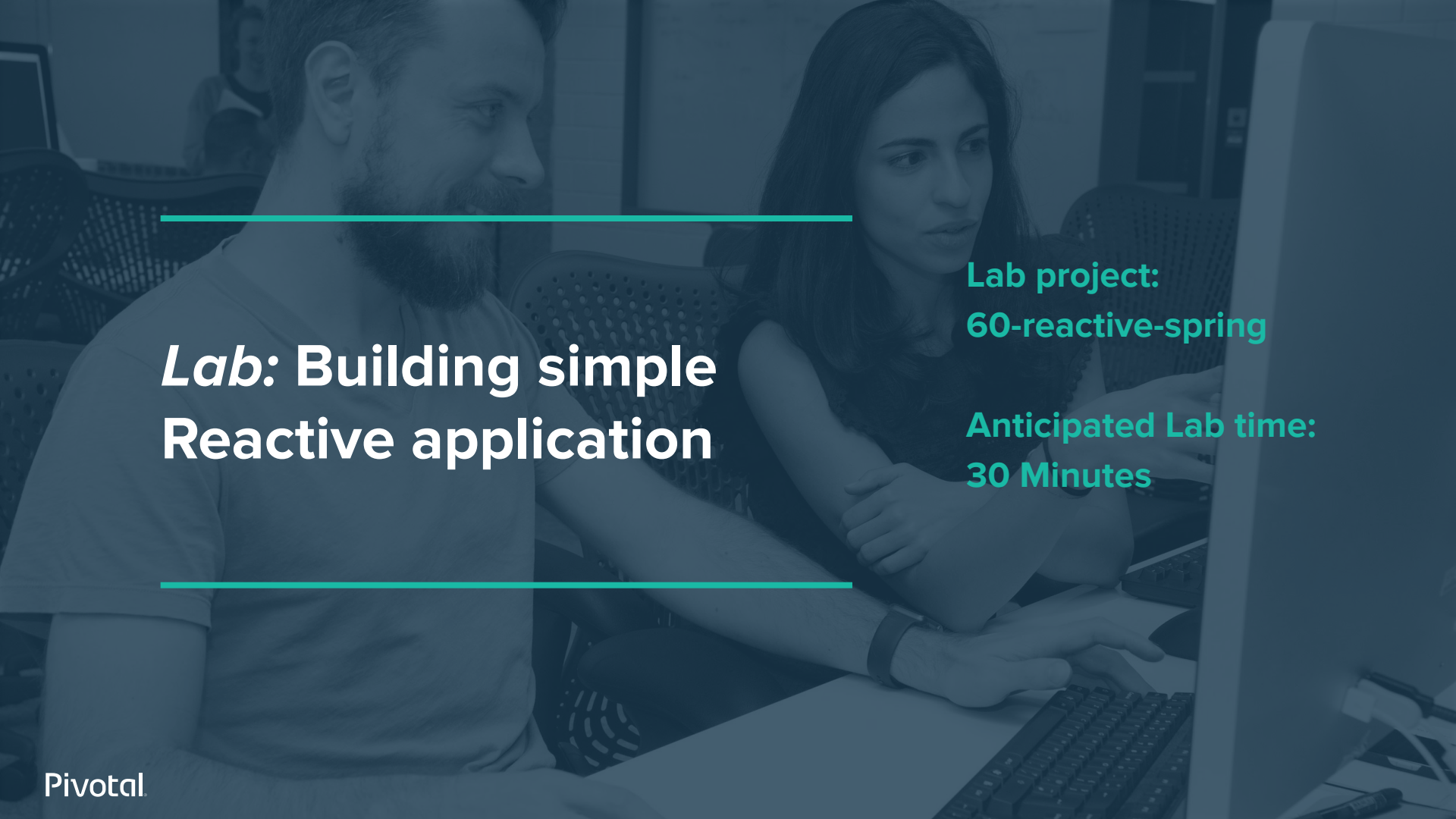
- What is Reactive Programming?
 - What are
 - Mono, Flux, Single, Observable?
- How does Spring incorporate Reactive Features?
 - Reactive Repositories
 - Web Client
 - Web Flux



projectreactor.io



RxJava

A man with a beard and a woman are sitting at a desk, looking at a computer monitor. The man is on the left, wearing a light-colored t-shirt, and the woman is on the right, wearing a dark top. They are both looking at the screen with interest. The background is slightly blurred, showing other office equipment and a person in the distance. The overall tone is professional and collaborative.

***Lab:* Building simple Reactive application**

Lab project:
60-reactive-spring

Anticipated Lab time:
30 Minutes

Spring 5 – Web Stack

