Writing Reactive Application using Angular/RxJS, Spring WebFlux and Couchbase

Naresh Chintalcheru

Who is Naresh

- Technology professional for 18+ years
- Currently, Technical Architect at Cars.com
- Lecturer at Illinois State University

Agenda

- Introduction
- Reactive Principles
- Why Reactive
- Design Patterns
- Reactive Application Architecture
  - Frontend
  - Backend APIs
  - Data Access
- Demo Code
- Q & A

Programming puzzle

Programming puzzle

```
a = 1;
b = a + 1;

a = 2;

print a; ---> 2
print b; ---> ?
```

Programming puzzle

```
a = 1;
b = a + 1;

a = 2;

print a; ---> 2
print b; ---> ?
```

2 or 3

Programming puzzle

```
a = 1;
b = a + 1;

a = 2;

print a; ---> 2
print b; ---> ?
```

2 ✓ 3 ✓

Programming puzzle

```
a = 1;
b = a + 1;

a = 2;

print a; ---> 2
print b; ---> ?
```

2

Imperative

3

Reactive

Reactive Programming Interface

	A	B	C
1	a	1	
2	b	=B1+1	
3			

Reactive Programming

Programming paradigm oriented around data flows and propagation of change ***

-Wiki

Functional programming with asynchronous data streams

Reactive Programming

- Imperative programming is a Pull model (hasNext, Next)
- Reactive programming is Push model, the elements will be pushed down the stream as soon as they are available
- Difference is the direction of the data flow

Reactive Systems

Reactive Systems

Reactive Systems are built on four guiding principles

Reactive Principles

Responsive

Resilient

Scalable

Message-driven

Reactive Manifesto 2.0

Reactive Principles

Reactive Systems are built on four guiding principles ***

- **Responsive** - React to Users, System responds in a timely manner
- **Resilient** - React to Failures, Failure expected and embraced
- **Elastic** - React to varying Load
- **Message-Driven** - Asynchronous message passing

Message-Driven

- A message-driven architecture provides the overall foundation for a responsive system
- Asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation and location transparency
- Boundary also provides the means to delegate failures as messages

Reactive Programming vs Reactive Systems

Reactive Programming vs Reactive Systems ***

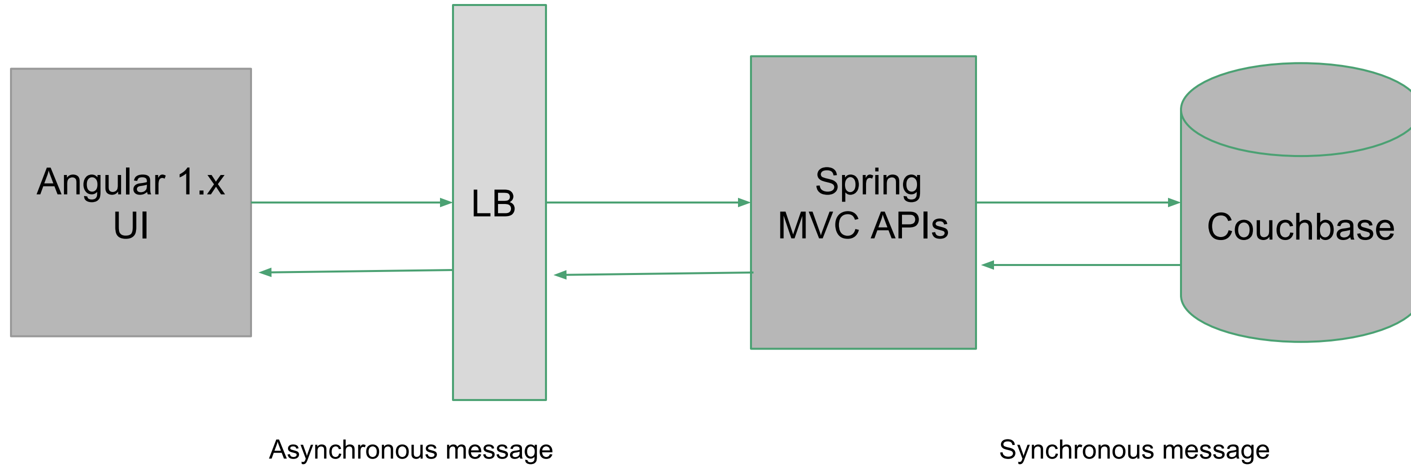
Reactive programming is about Data Flows and Change propagation

Reactive System is about Asynchronous Message-driven Control Flows and System responsiveness even under load (Scalability) or partial failure (Resilient)

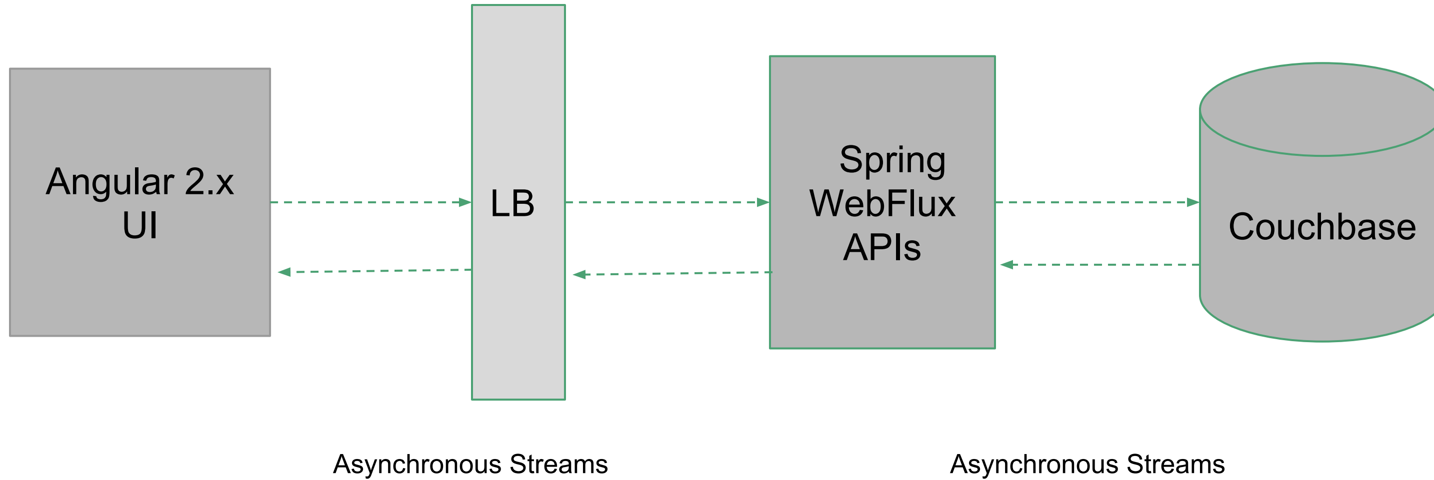
Reactive Design Patterns

- Message Control Patterns
- Flow Control Patterns
- Fault-tolerance and Recovery Patterns
- Replication and Resource Management Patterns

App Design



Reactive App Design



Reactive-Stream Specification ***

Reactive Streams standard for Asynchronous Stream processing with Non-blocking Back Pressure

Runtime environments (JVM and JavaScript) as well as network protocols

www.reactive-streams.org

Reactive stream Example

- Mouse and Keyboard interaction
- Stock Prices
- System Events

Why Reactive Systems ?

Why Reactive

- Functional reactive programming promotes immutability, concurrent programming and parallel execution, process higher loads with fewer threads

- Efficient resource utilization of multicore cpu using asynchronous Non-Blocking I/O

- Cost effective in Cloud

- Reactive streams are designed to solve the central issue with asynchronous programming Backpressure

Beyond Promises

- Asynchronous task management with Future<T> Objects and Callback-hell but Promises to rescue CompletableFuture<T> in Java 8
- Promises are containers for single future computation
- Promises tring to connect multiple asynchronous steps together is difficult

Linux Kernel Support for Non-Blocking I/O

- Linux Kernel 2.6 release in 2006
  - Support Asynchronous I/O
  - POSIX AIO API

- AIO is to allow a process to initiate a number of I/O operations without having to block or wait for any to complete

- Process can retrieve the results of the I/O at some later time, or after being notified of I/O completion

How Linux Non-Blocking Works ?

- The proc file system contains two virtual files that store AIO data can be tuned for asynchronous I/O performance

- The /proc/sys/fs/aio-nr file provides the current number of system-wide asynchronous I/O requests

- The /proc/sys/fs/aio-max-nr file is the maximum number of allowable concurrent requests. The maximum is commonly 64KB

Java NIO

Non-Blocking I/O Support

- JDK5
- java.nio.*

Reactive App Design

Client-side benefits ?

Polyglot Reactive Libraries

RxJava, RxJS, RxScala, Rx.Net, RxSwift, RxRuby, RxPHP and WebFlux

Reactive-Streams Libraries on JVM

- RxJava
- Spring WebFlux
- Redhat Vert.x
- Lightbend Akka Streams
- Java v9 Flow APIs

How Reactive Libraries Work

- Graph Processing algorithm that captures the dependencies among the reactive values and the language runtime uses the graph to keep track of which computations must be executed again when one of the inputs changes

- Change propagation techniques Pull, Push or Hybrid

Observable

- Reactive applications use observable models, event streams and stateful clients.

- Observable models enable other systems to receive events when state changes.

  E.g., Street lights

Observable

Observable is the combination of

- The ability of a producer to push elements to consumer
- The ability for a producer to signal to the consumer that there is no more data available.
- The ability for a producer to signal to the consumer that an error has occurred.

Four Quadrants Of Data Flow

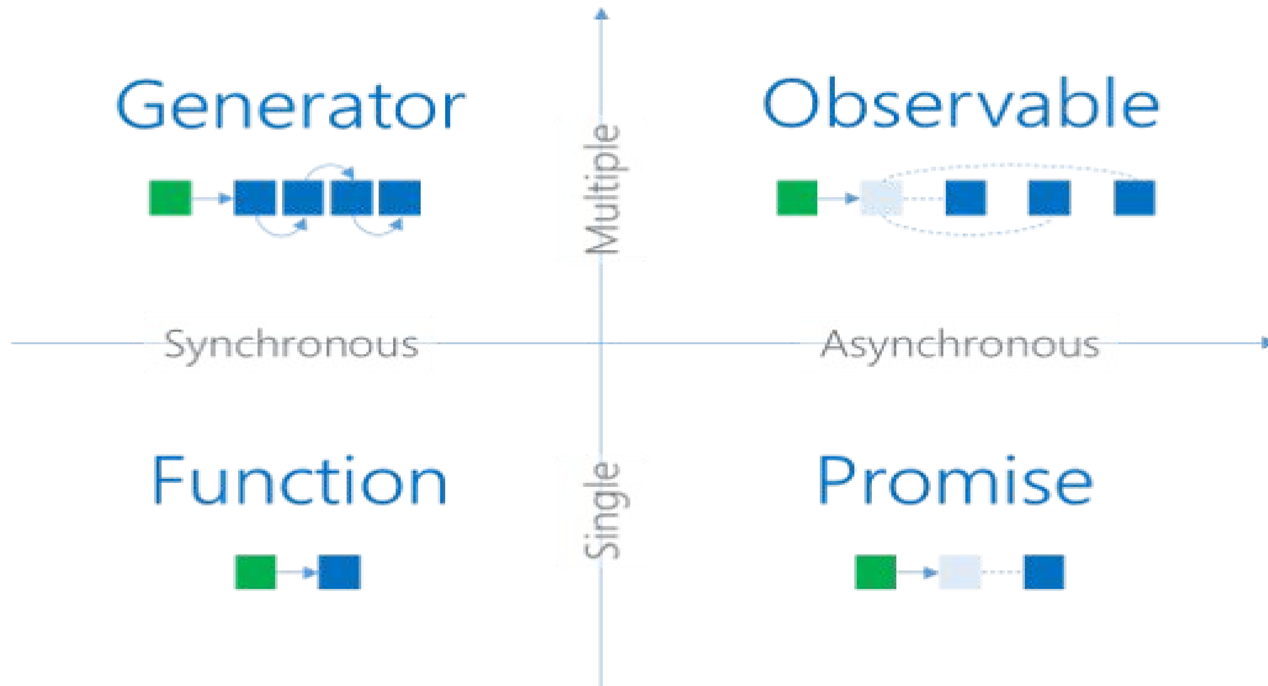
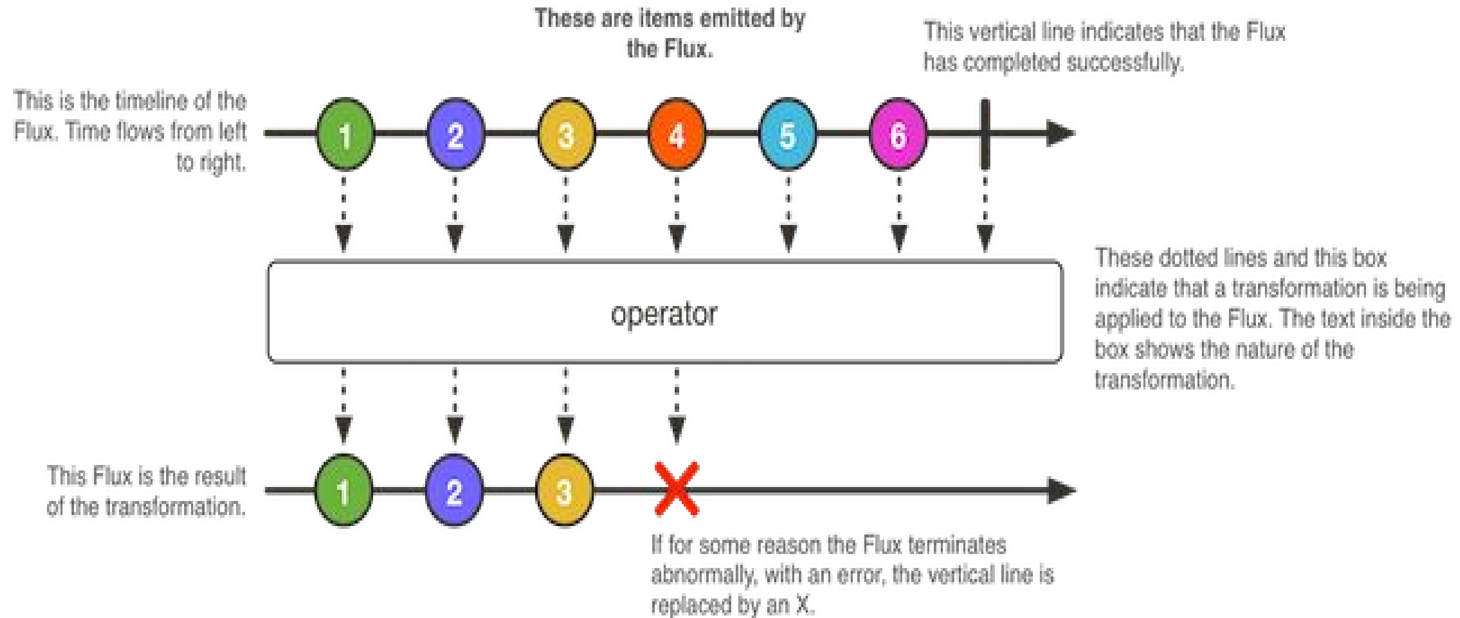


Image Source

Reactive Programming

Rx = Observable + Observer + Subscriber

Data Flow



[Image Source]

Hot & Cold Observable

Hot Observable may begin emitting items as soon as it is created, and so any observer who later subscribes to that Observable may start observing the sequence somewhere in the middle

E.g., Stocks quotes, Keyboard/mouse events, Street Lights, News websites

Cold Observable waits until an observer subscribes to it before it begins to emit items, and so such an observer is guaranteed to see the whole sequence from the beginning. Cold observable does not guarentee the sequence.

E.g., Netflix Streaming, Wed requests

Observable code

```
List<String> words = Arrays.asList("hello", "good", "morning");


Observable.from(words)
   .flatMap(word -> Observable.from(word.split("")))
   .distinct()
   .sorted()
   .zipWith(Observable.range(1, Integer.MAX_VALUE), (string, count) -> String.format("%2d. %s", count, string))
```

Subscribe Code

Observables are lazy and cancellable

```
observable.subscribe(
        new Subscriber<Person>() {

           @Override
           public void onCompleted() {
               System.out.println(" ********** Completed ********");
           }

           @Override
           public void onError(Throwable throwable) {
               System.err.println(" ********** Whoops: " + throwable.getMessage());
           }

           @Override
           public void onNext(Person person) {
              System.out.println(" ******** First Name= " + person.getFirstname() + " LastName= " + person.getLastname() );
           }
          }
      );
```

Functional Reactive Programming

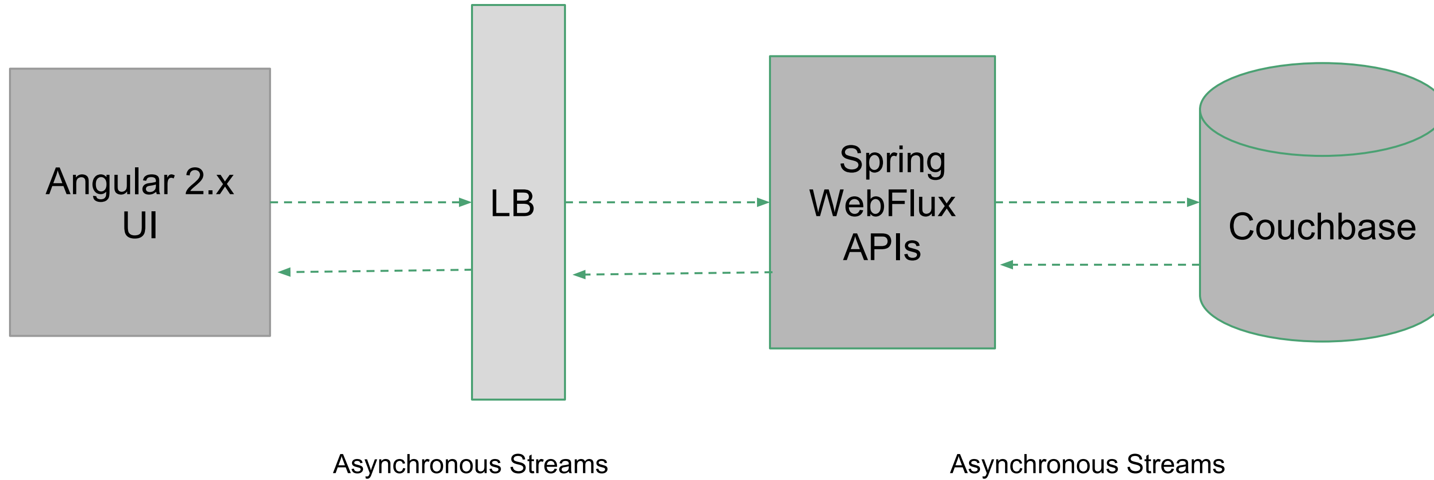
- Reactive programming is based on the functional constructs
- Functional programming principles are essential in understanding RP
- Functional programming provides Declarative, Composable and Concurrent code
- Learn to replace loops with recursion, traverse data structures without iterators, do input/output using monads, and dealing with Immutable data sources.
- Exceptions are side-effects that undermine type system and functional purity

Functional Programming

- Pure Functions, Higher-Order Functions, Monads, Currying and Immutable data

- Pure functions does not have side-effects, for a given inputs it always have same output, e.g., mathematical functions
- Higher-Order Functions, functions which takes at least one more functions as parameters and return a function (e.g., Lambds's)
- Monads wraps data value and provide series of computational operators (e.g., Streams/Optional)
- Currying is technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument

Reactive App Design



Angular 2 Building Blocks

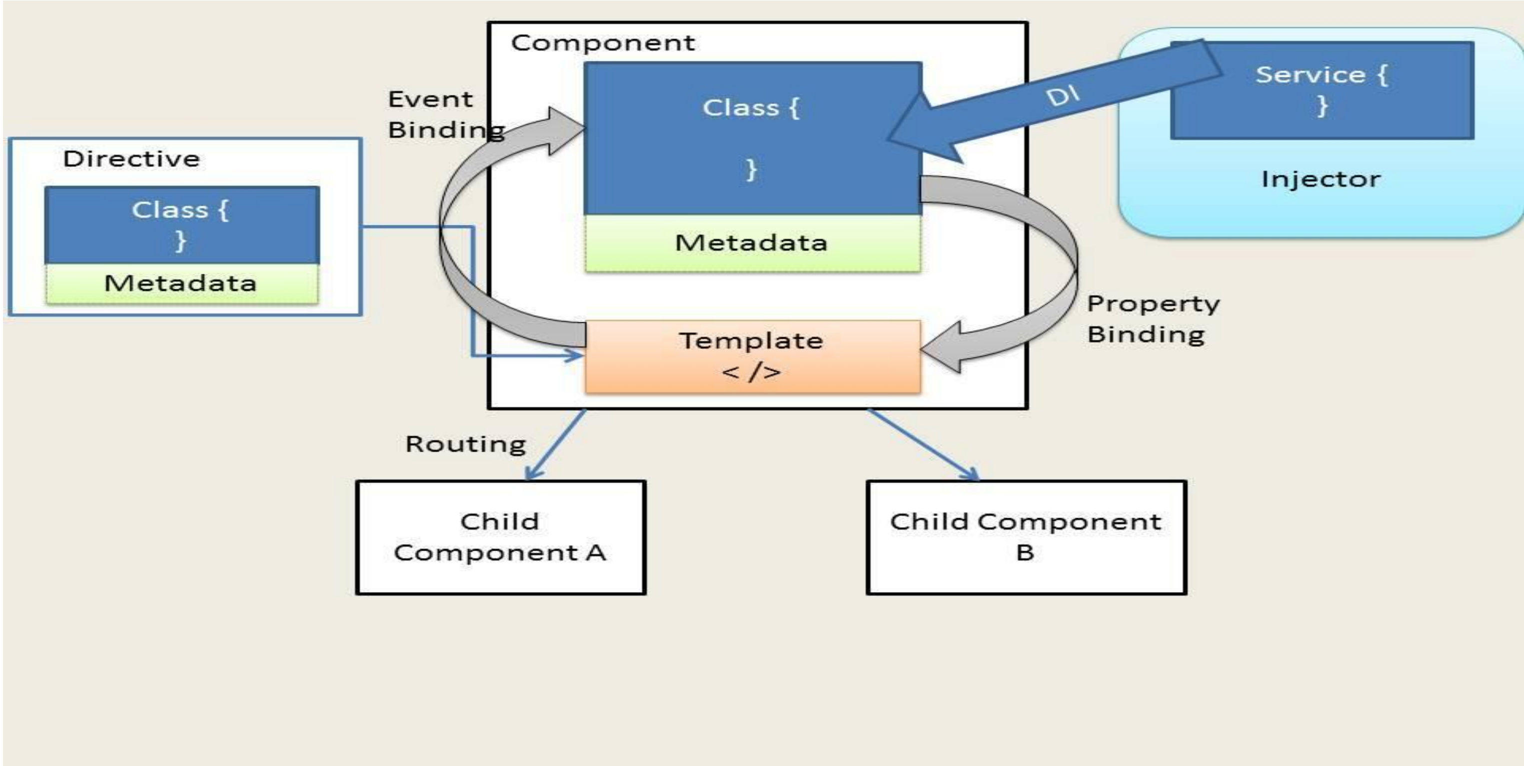


Image Source

Observable Data Service

The $http service was based on promises and deferrals but 2.0 rely on RxJS Observable pattern

<table>
<tr><th>Angular 1.x</th><th>Angular 2.x</th></tr>
<tr><td>

```
export class HeroService {

    getHeroes(): Promise<Hero[]> {

        return Promise.resolve(HEROES);

    }
}
```

</td><td>

```
export class HeroService {


    getHeroes(): Observable<Hero[]> {

        return this.http.get(this.heroesUrl)

            .map(this.extractData)

            .catch(this.handleError);

    }

}
```

</td></tr>
</table>

Observables are lazy

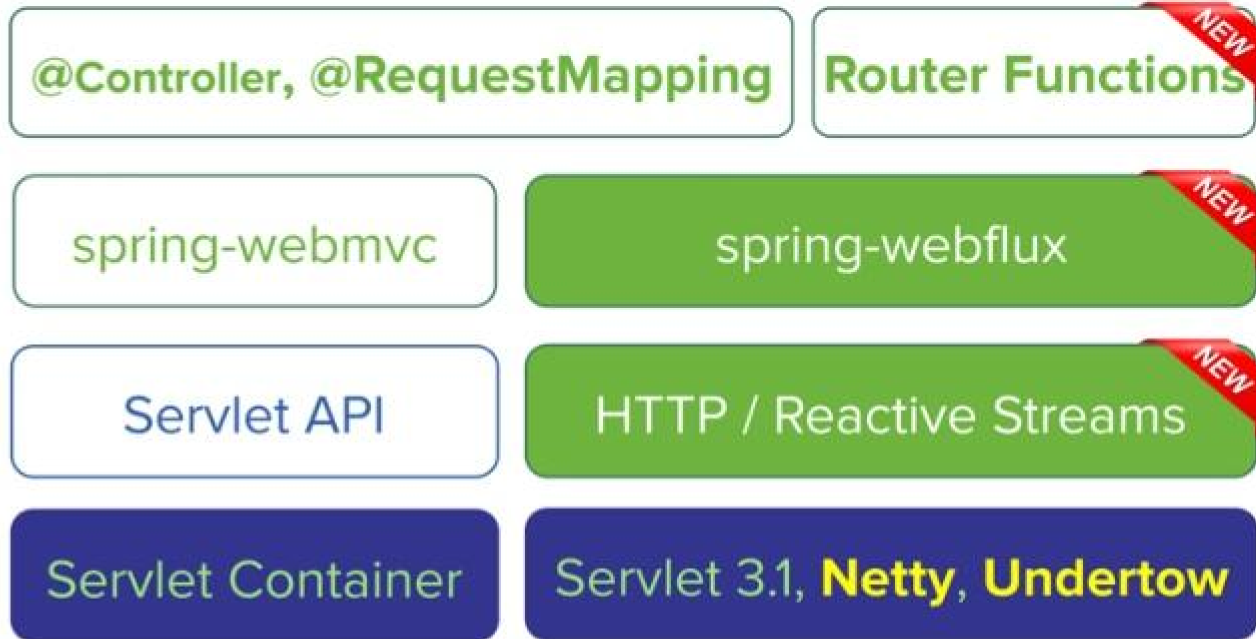
```
export class HeroListComponent implements OnInit {
    constructor (private heroService: HeroService) {}
     ngOnInit() { this.getHeroes(); }


      getHeroes() {
        this.heroService.getHeroes()
              .subscribe(
                heroes => this.heroes = heroes,
                error =>  this.errorMessage = <any>error);
     }
 }
```

Promises vs Observables

Observables	**Promise**
Observables handle multiple values over time	Promises are only called once and will return a single value
Observables are cancellable	Promises are not cancellable

Spring WebFlux (Reactor)

Spring WebFlux Architecture



Reactor Types

Reactor supports two types

- Flux<T>
- Mono<T>

Flux (Publisher/Observable)

Flux is a Reactive Streams Publisher with rx operators that emits 0 to N elements, and then completes (successfully or with an error).

Observer subscribes to an Observable, Observer reacts/operates to item or sequence of items the Observable emits

Mono (Publisher/Observable)

Mono is a Reactive Streams Publisher with rx operators that emits 0 to 1 elements, and then completes (successfully or with an error)

Reactor code

```
public interface UserRepository {

    Mono<User> findById(Long id);

    Flux<User> findAll();

    Mono<Void> save(User user);

}
```

Reactor code

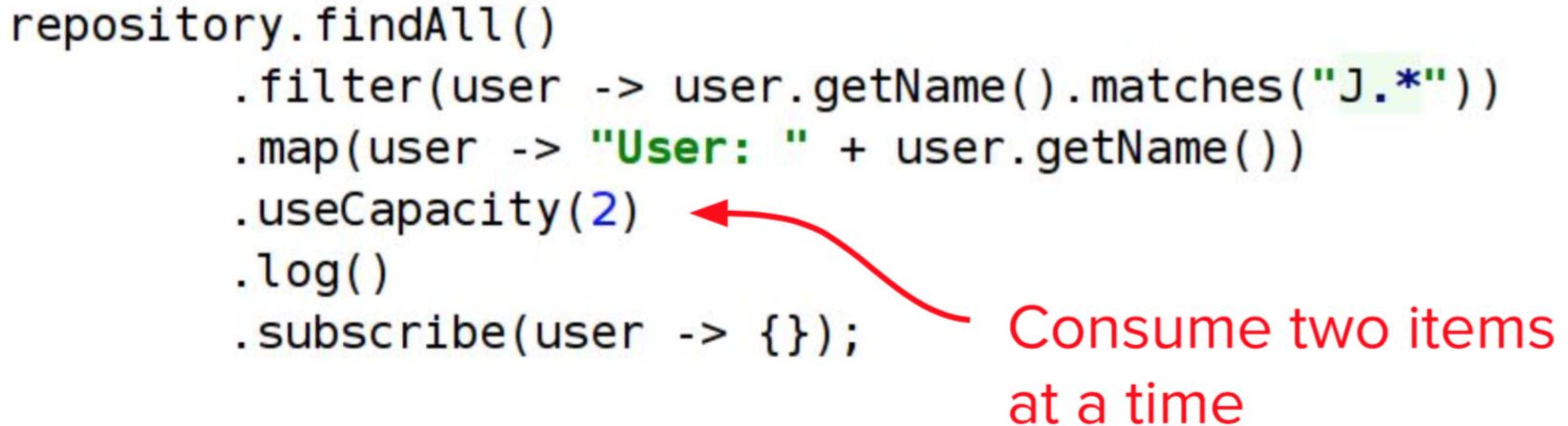
```
repository.findAll()
        .filter(user -> user.getName().matches("J.*"))
        .map(user -> "User: " + user.getName())
        .log()
        .subscribe(user -> {});
```

Subscriber triggers flow of data

Reactor code

```
onSubscribe
request(unbounded)
onNext(User: Jason)
onNext(User: Jay)
...
onComplete()
```

Reactor code



Reactor code

```
onSubscribe
request(2)
onNext(User: Jason)
onNext(User: Jay)
request(2)
onNext(User: Joe)
onNext(User: John)
...
```

Non-Blocking

ParallelFlux<T>   parallel()

Prepare to consume this Flux on number of 'rails' matching number of CPU Cores in round-robin fashion.

ParallelFlux<T>   parallel(int parallelism)

Prepare to consume this Flux on parallelism number of 'rails' in round-robin fashion.

ParallelFlux<T>   parallel(int parallelism, int prefetch)

Prepare to consume this Flux on parallelism number of 'rails' in round-robin fashion and use custom prefetch amount and queue for dealing with the source Flux's values.

Operators

Operators operate on Observables

Operators By Category

- Creating Observables
- Transforming Observables
- Filtering Observables
- Combining Observables
- Backpressure Observables

Stateful/Stateless Operators

- Stateless Operators - filter, flatMap
- Stateful Operators - Sort, Distinct
- Terminal vs Intermediate Operators

Backpressure

What is Backpressure ?

What is Backpressure

Observable is emitting items more rapidly than an operator or observer can consume.

Backpressure Overflow Strategy

FluxSink.OverflowStrategy.BUFFER
Buffer all signals if the downstream can't keep up

FluxSink.OverflowStrategy.DROP
Drop the incoming signal if the downstream is not ready to receive it

FluxSink.OverflowStrategy.ERROR
Signal an IllegalStateException when the downstream can't keep up

FluxSink.OverflowStrategy.IGNORE
Completely ignore downstream backpressure requests

FluxSink.OverflowStrategy.LATEST
Downstream will get only the latest signals from upstream

Reactive Database Access

Reactive Data Access

- Couchbase Reactive Streams Java SDK Driver, providing asynchronous stream processing with non-blocking backpressure.

- Fully implements the Reactive Streams API RxJava for providing interop with other reactive streams within the JVM ecosystem.

Reactive Data Access

```
Observable<Comments> comments =

    bucket
      .async()
      .get("post::1")
      .flatMap(post -> Observable.from(post.content().getArray("comments").toList()))
      .flatMap(commentId -> bucket.async().get((String) commentId))
      .filter(comment -> comment.content().getBoolean("published"));
```

Reactive Data Access Support

- Postgres
- Couchbase
- MongoDB
- Redis
- Cassandra
- Elastic Search
- Apache Kafka

Demo

Ref

https://www.ibm.com/developerworks/library/l-async/

https://projectreactor.io/docs/core/release/api/

http://reactivex.io/documentation/operators/backpressure.html

https://en.wikipedia.org/wiki/Reactive_programming

https://www.sitepoint.com/functional-reactive-programming-rxjs/

https://www.manning.com/books/reactive-design-patterns

https://github.com/poutsma/web-function-sample