Reactive programming with Spring

Reactive system

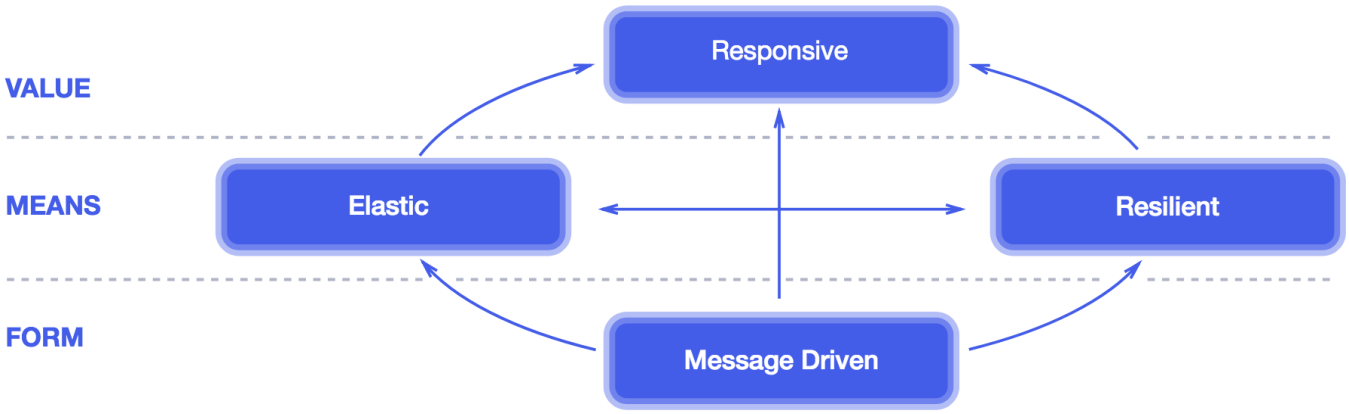
Any application should react to changes:

- any changes in demand (load)
- any changes in the availability of services

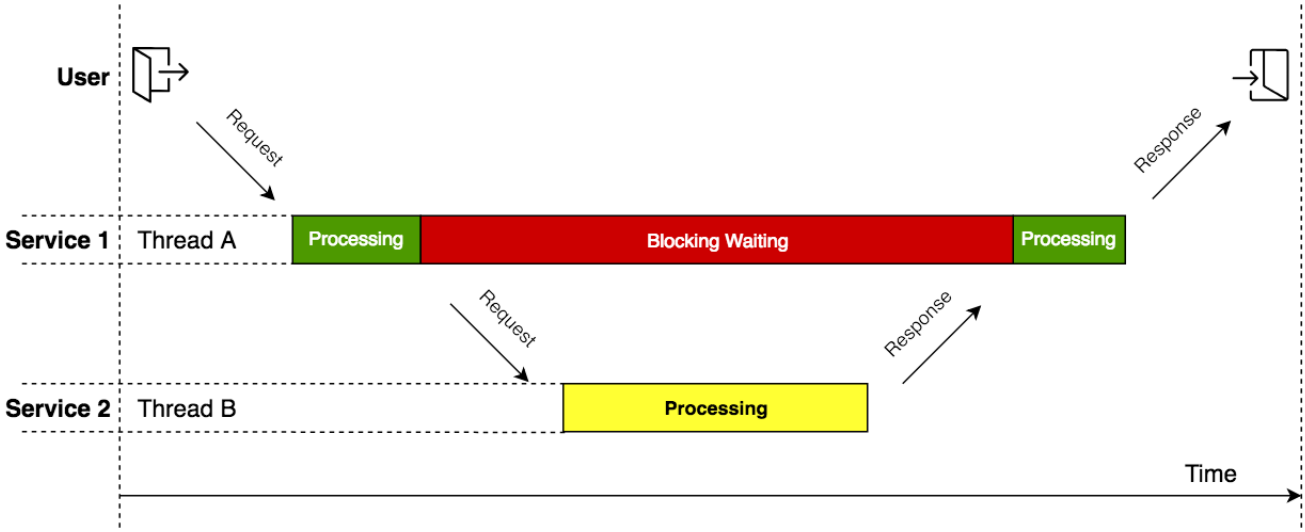
these kinf of app should be **reactive** to any changes that affects the system ability to response to user request

Main features of the **reactive** app :

- ability to stay responsive under a varying workload
- throughput of the system should increase automatically when more users start using it
- should decrease automatically when the demand goes down.

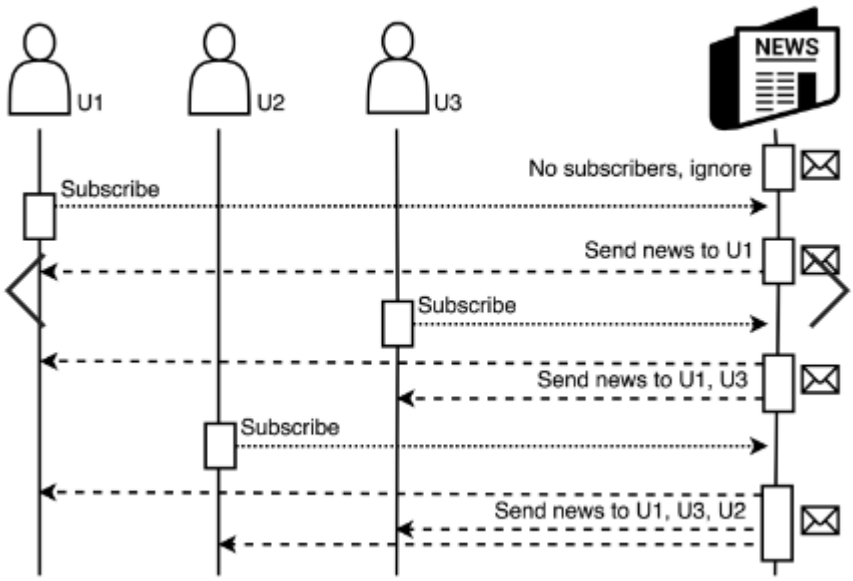


In the interconnected world of microservices one possible botleneck could be the blocking communication among microservices with ex Spring Boot + http based communication.

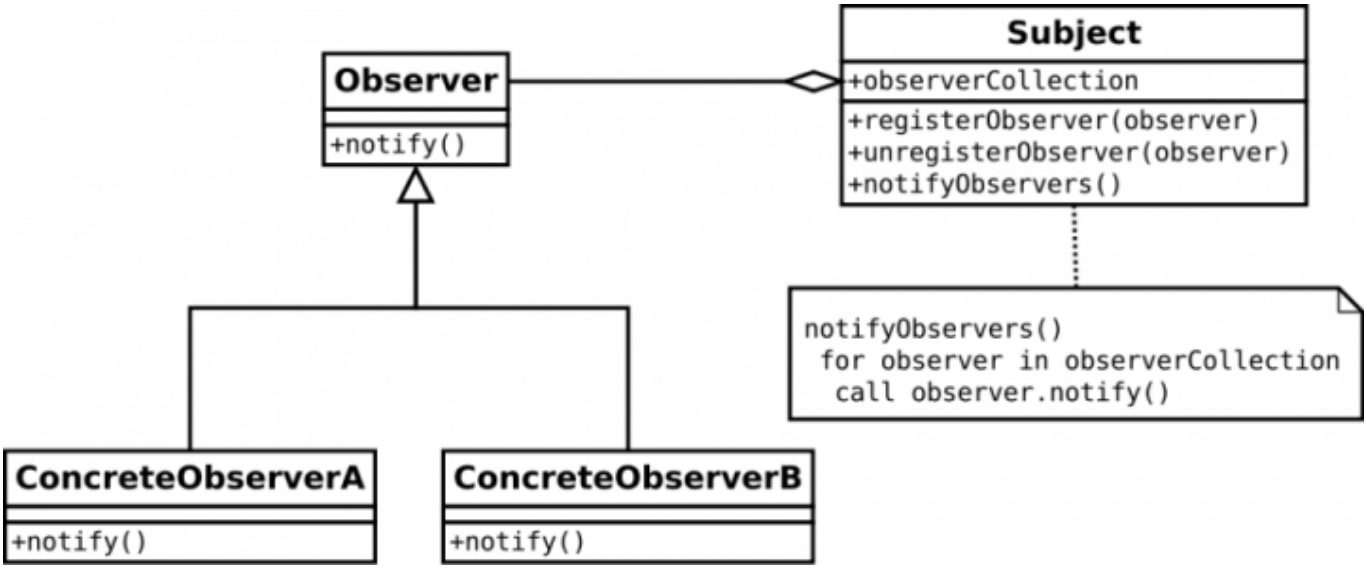


Reactive Programming

Observer real life situation



Observer design pattern as a solution



Reactive solution with java libraries

Reactive solution main building blocks: Observer + Iterator design pattern implemented in

- RxJava 1.X
- RxJava 2.x

However not fully succesfull.

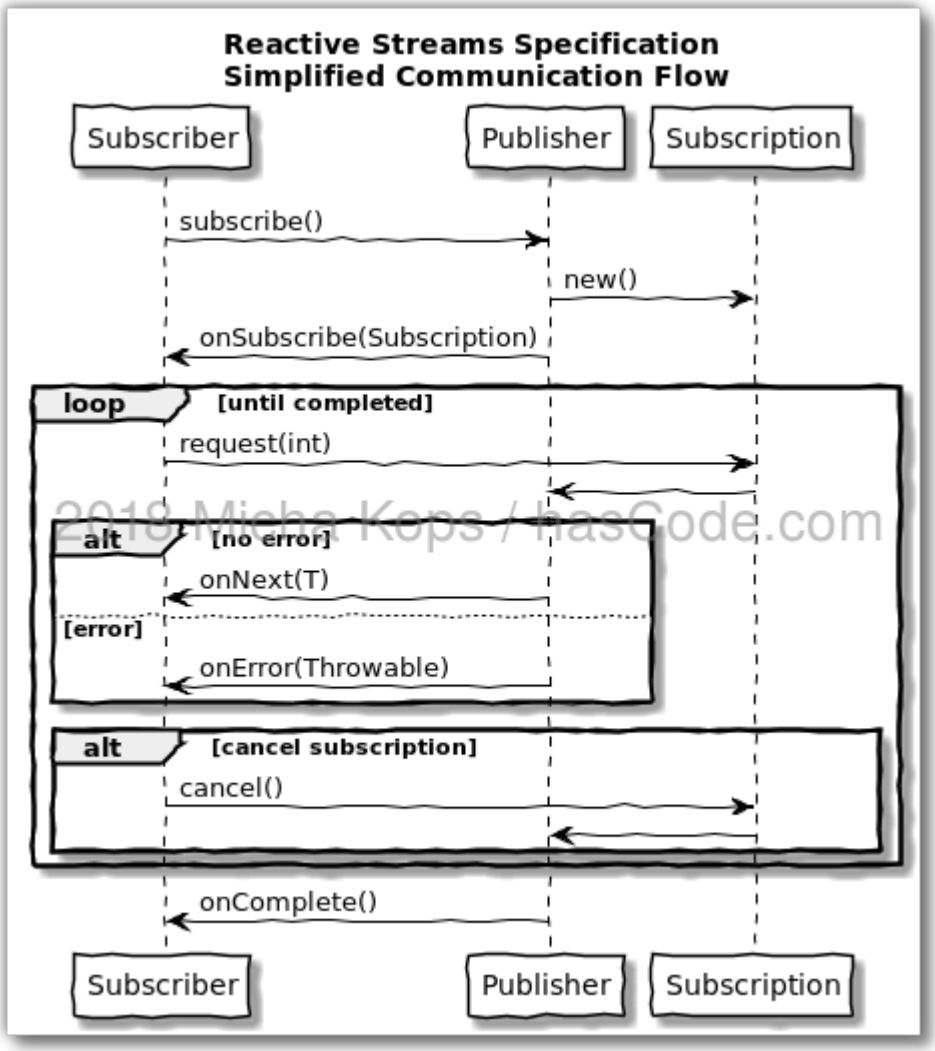
Reactive Stream spec

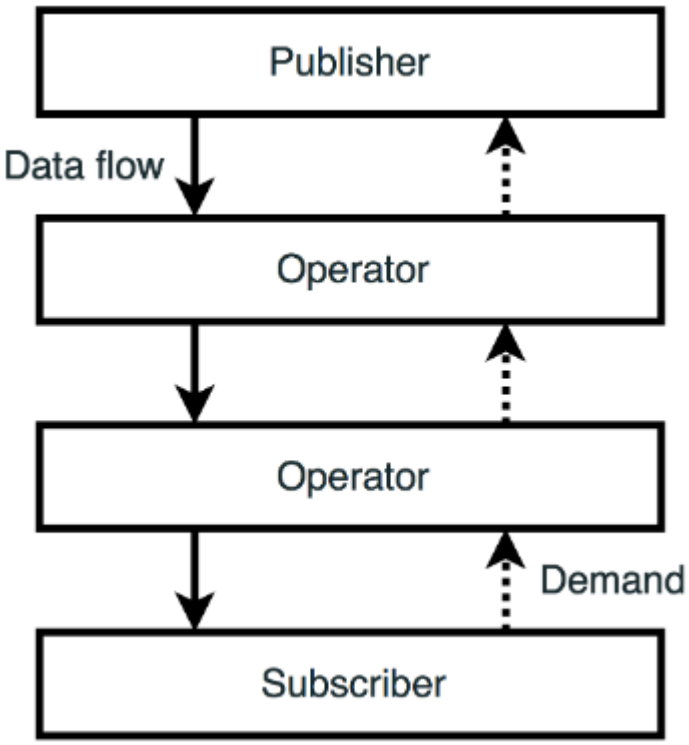
The Reactive Streams **specification** defines the following interfaces:

- `Publisher<T>` (start point of the communication)
- `Subscriber<T>` (end point of the communication),
- `Subscription` (handle the start - end points relation),

- and `Processor <T, R>` (some transformation logic).

by introducing the *pull-push* data exchange model resolves the *backpressure* issue.





Projector Reactor as implementation of the Reactive Stream Spec

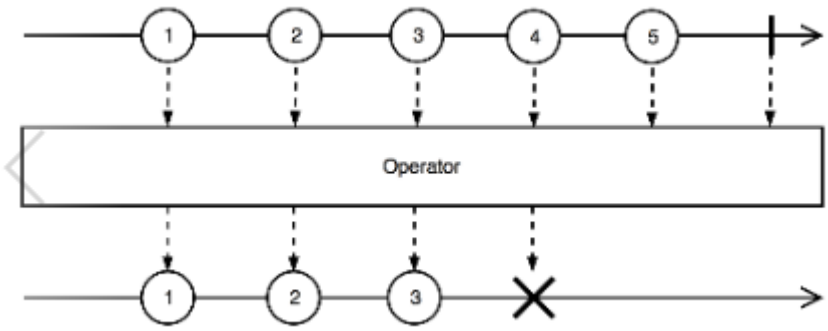
- Project Reactor 1.x
- Project Reactor 2.x

Adding Reactor to the Spring (5.x) project

```
compile("io.projectreactor:reactor-core:3.2.0.RELEASE")
//...
testCompile("io.projectreactor:reactor-test:3.2.0.RELEASE")
```

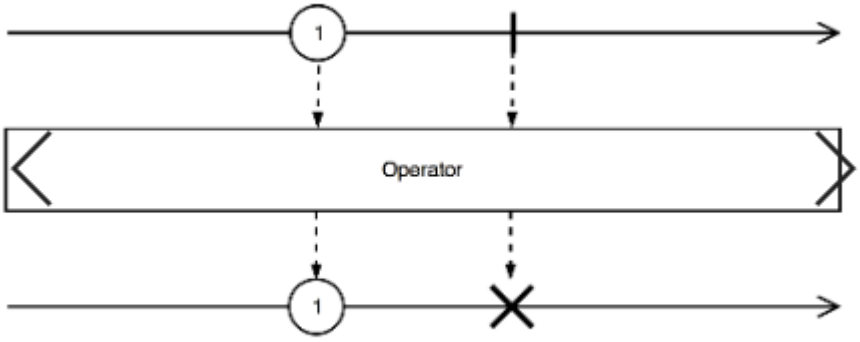
Reactive types – Flux and Mono as `Publisher<T>` implementation

- `Flux` defines a reactive stream that can produce zero, one, or many elements



Flux stream transformed into another Flux stream

`Mono` defines a reactive stream that can produce zero, or one element



Example of Reactive Flux streams

```
Flux<String>  stream1 = Flux.just("Hello","world");
Flux<Integer> stream2 = Flux.fromArray(newInteger[]{1,2,3});
Flux<Integer> stream3 = Flux.fromIterable(Arrays.asList(9, 8, 7));
//....
Flux<Integer> stream4 = Flux.range(2019, 9); // starting with 2019 generate 9
elements
```

Example of Reactive Mono streams

```
Mono<String> stream5 = Mono.just("One");
Mono<String> stream6 = Mono.justOrEmpty(null);
Mono<String> stream7 = Mono.justOrEmpty(Optional.empty());
//...
Mono<String> stream8 = Mono.fromCallable(()->httpRequest()); // handling a
asynchronous requests http|db
// shorthener way
Mono<String> stream8 = Mono.fromCallable(this::httpRequest);
```

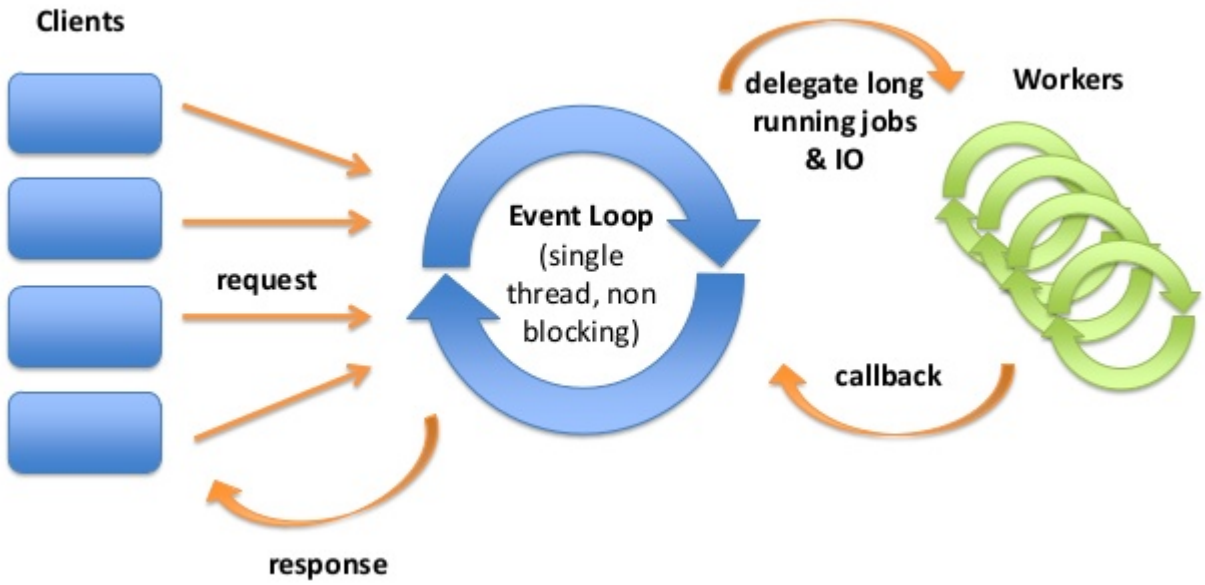
Use cases with different `subscribe` functions.

```
Flux.just("A","B","C")
  .subscribe(
        data -> log.info("onNext: {}", data),
        err ->{ /* ignored  */ },
        ()-> log.info("onComplete")
      );
```

```
   Flux.just("A","B","C")
               .subscribe(
                       data -> log.info("onNext: {}", data),
                       err ->{ /* ignored  */ },
                       () -> log.info("onComplete")
               );
```

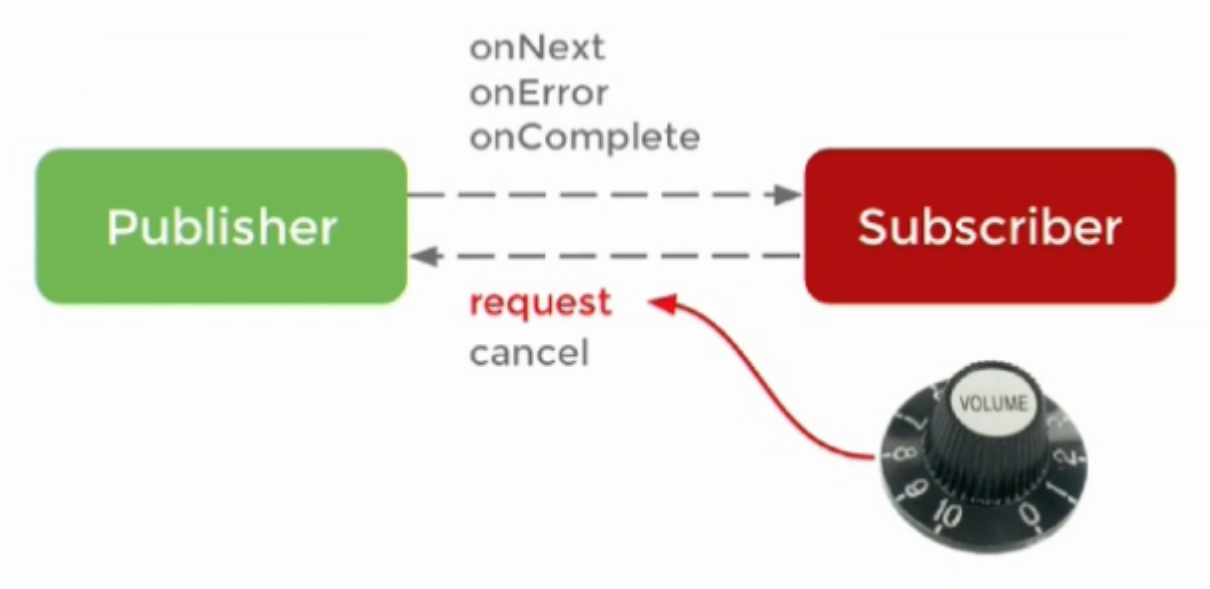
```
  Flux.range(1, 100)
               .subscribe(
                       data -> log.info("onNext: {}", data),
                       err -> { /* ignore */ },
                       () -> log.info("onComplete"),
                       subscription -> {
                           subscription.request(4);
                           //subscription.cancel();
                           log.info("Request end for 4 ");
                           subscription.request(4);
                       }
               );
```

Reactive Spring

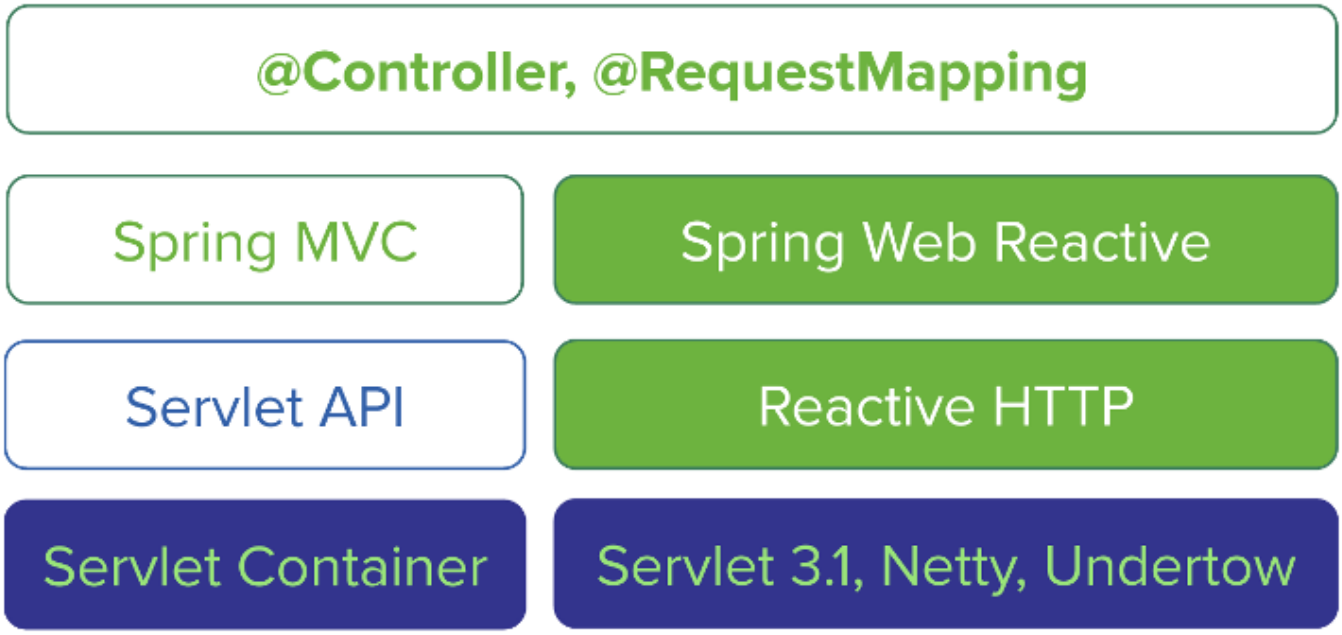


The core features are :

- handle the backpressure `backpressure` is a mechanism that permits a receiver to ask how much dat it wants to receive from the emitter.
- implements the push-pull strategy|model

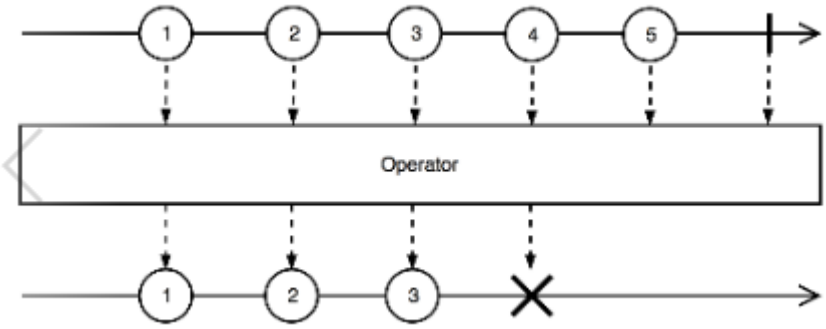


- reactive `stream` based controllers , and alternatives to different handler design
- functional programming (lambda oriented routing, processing)
- non-blocking: make asynchronous calls and respond as the results of those calls are returned



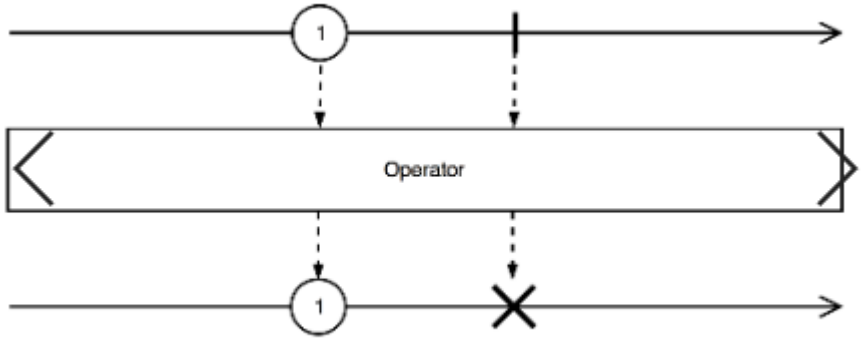
Reactive types – Flux and Mono as `Publisher<T>` implementation

- `Flux` defines a reactive stream that can produce zero, one, or many elements



Flux stream transformed into another Flux stream

- `Mono` defines a reactive stream that can produce zero, or one element



Initializing Spring with reactive features (Use Case)

- Web -> Reactive Web (Includes Spring WebFlux)

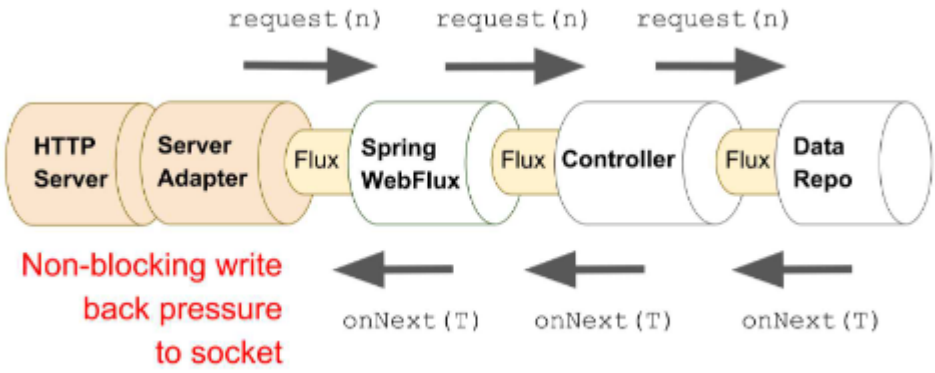
- NoSQL -> Reactive MongoDB (includes the drivers)
- NoSQL -> Embedded MongoDB (run embedded version of MongoDB)
- Core -> Lombok (special annotation will generate getters, setters. etc...)

Application dependencies (starters)

```
dependencies {
        compile('org.springframework.boot:spring-boot-starter-data-mongodb-
reactive')
        compile('org.springframework.boot:spring-boot-starter-webflux')
        compile('org.projectlombok:lombok')
        compile('de.flapdoodle.embed:de.flapdoodle.embed.mongo')
        runtime('org.springframework.boot:spring-boot-devtools')
        testCompile('org.springframework.boot:spring-boot-starter-test')
        testCompile('io.projectreactor:reactor-test')
}
```

(Webflux) application main elements

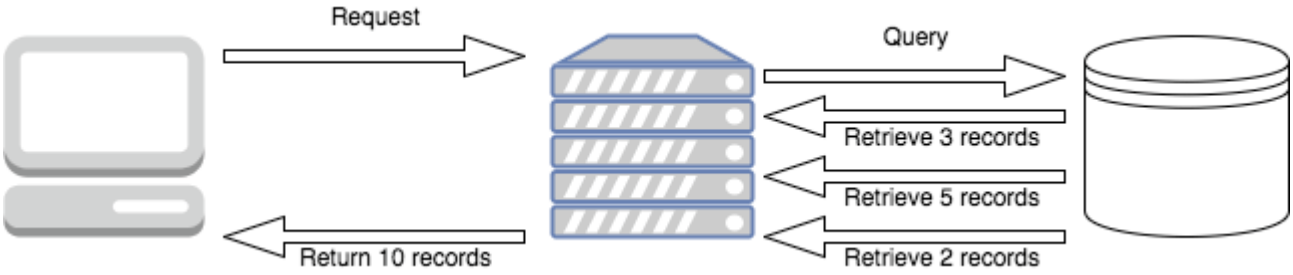
- DeliveryModel - define the model returned by the repository
- DeliveryRepository - define the interface of the repository, to persist to and from the DB (reactive DB!!!)
- DeliveryService (Interface and Implementation) - implement the service logic (interact with the repository)
- DeliveryController - receives the requests and return reactive responses (Mono and FLuxes)



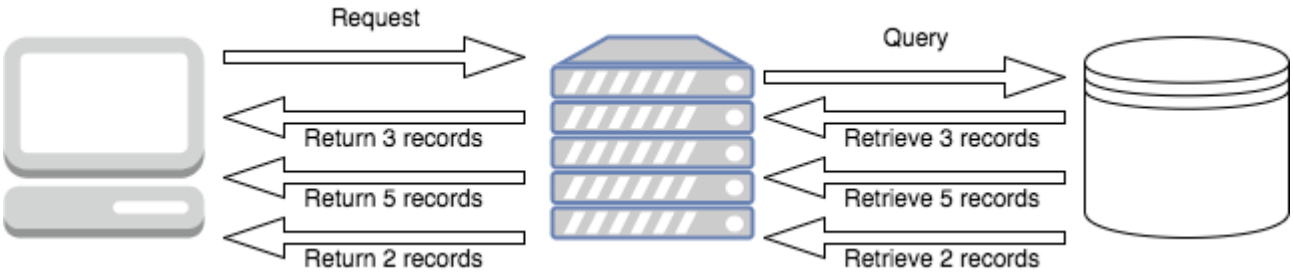
Important notes regarding the DB

```
@EnableReactiveMongoRepositories
@SpringBootApplication
public class AnnotationdemoApplication {

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

}
```

Spring with blocking (nonreactive) DB connection



Spring with nonblocking (reactive) DB connection



sending a POST request to the Web

```
POST /delivery HTTP/1.1
Host: localhost:8080
User-Agent: PostmanRuntime/7.13.0
Accept: */*
Host: localhost:8080

{
        "toAddress": "Fo utca 3. kincseskolozsvar",
        "customer": "Matyas kiraly"
}
```

receiving GET request

```
GET /deliveries HTTP/1.1
Host: localhost:8080
Content-Type: application/json
User-Agent: PostmanRuntime/7.13.0
Accept: */*
```

Functional reactive services with Spring WebFlux

Functional Spring WebFlux application is based on

- a router responsible for routing HTTP requests to handler functions.

- handler functions are responsible for executing business functionality and building responses.

In the handler functions

- The handler functions return Mono.
- each method is passed a `ServerRequest` argument
- the `ok()` method returns a `BodyBuilder` with an HTTP status code of 200;
- the `body()` method sets the contents to be returned to the caller and returns a `Mono<ServerResponse>`.
-