Reactive programming with Spring

Reactive system

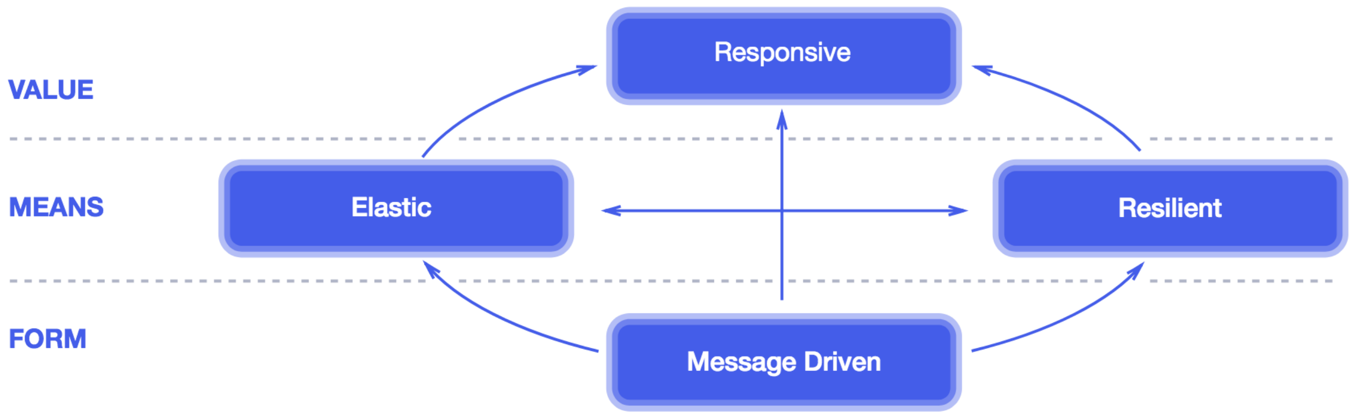
Any application should react to changes:

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

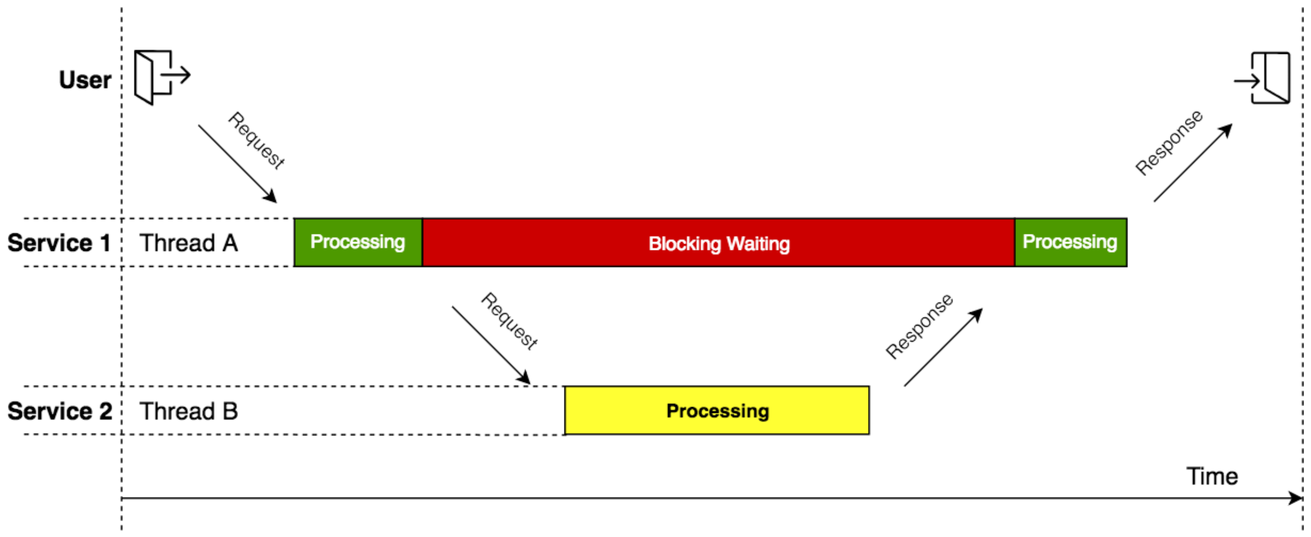
these kinf of app should be **reactive** to any changes that affects the system ability to response to user request

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.

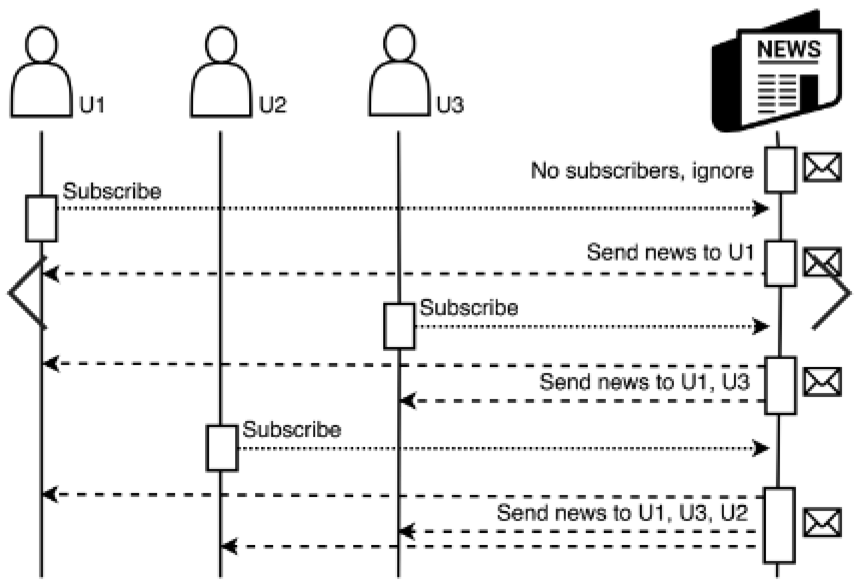


In the interconnected world of microservices one possible botleneck could be the blocking communication among microservices with ex Spring Boot + http based communication.

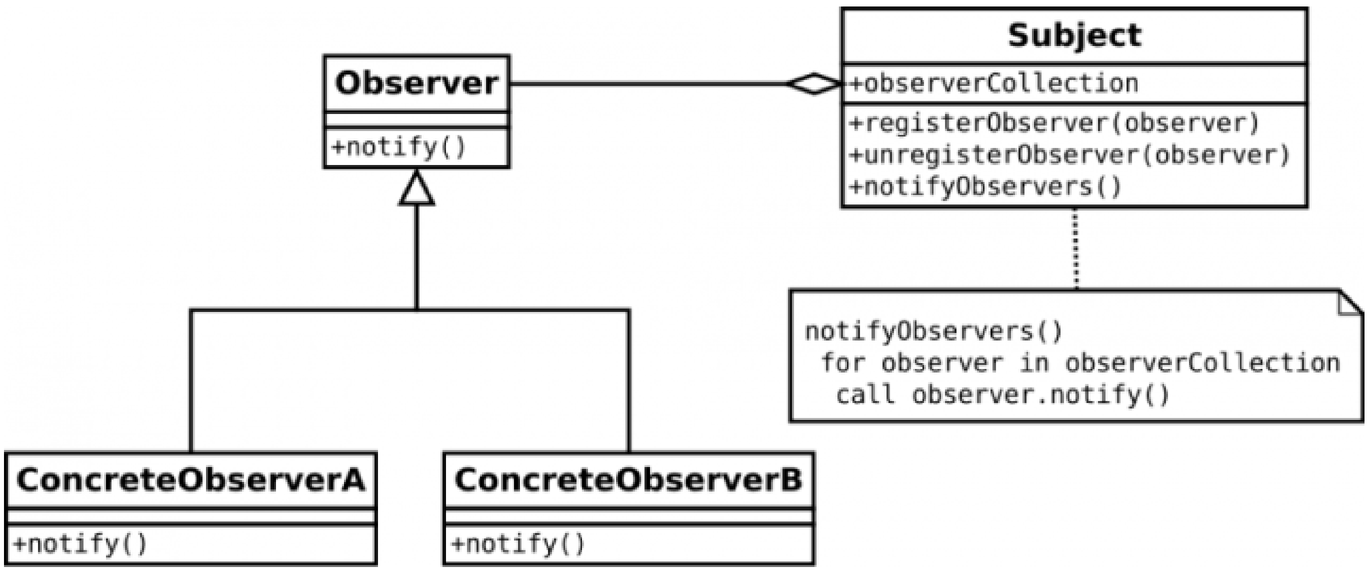


Reactive Programming

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.

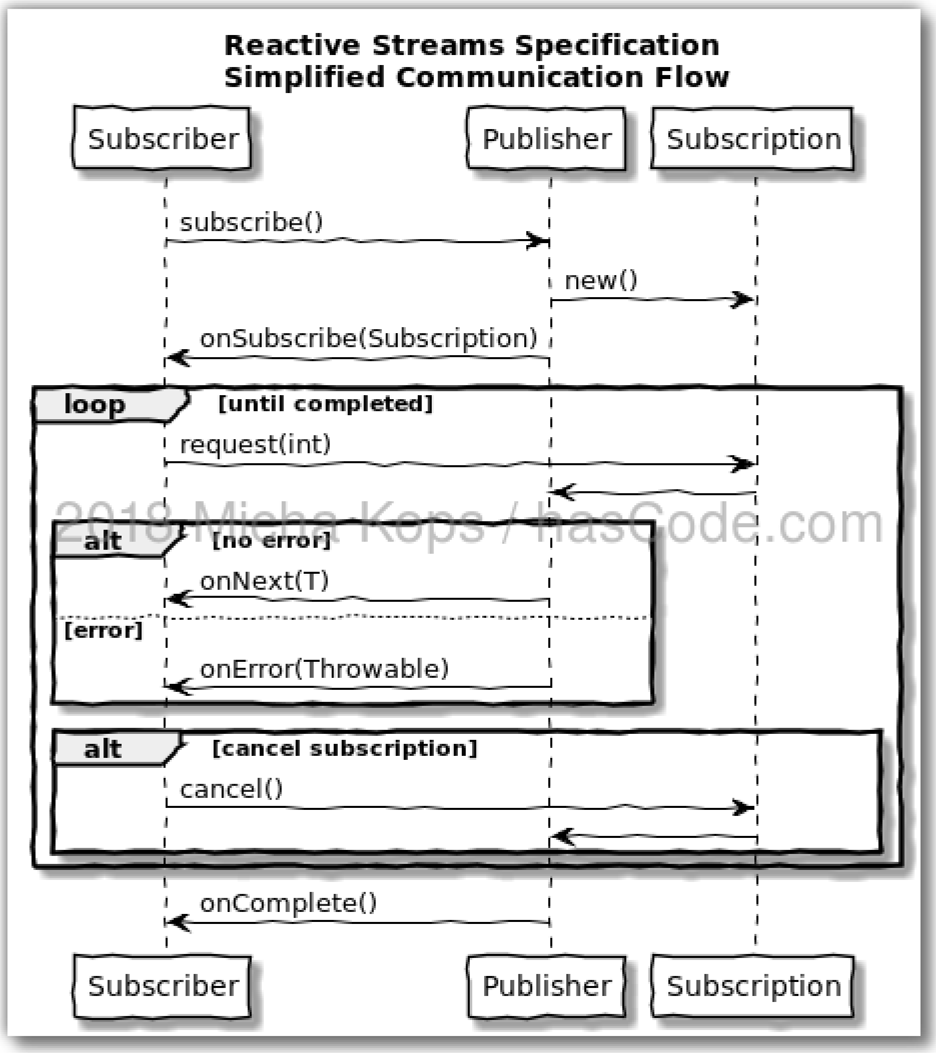
Reactive Stream spec

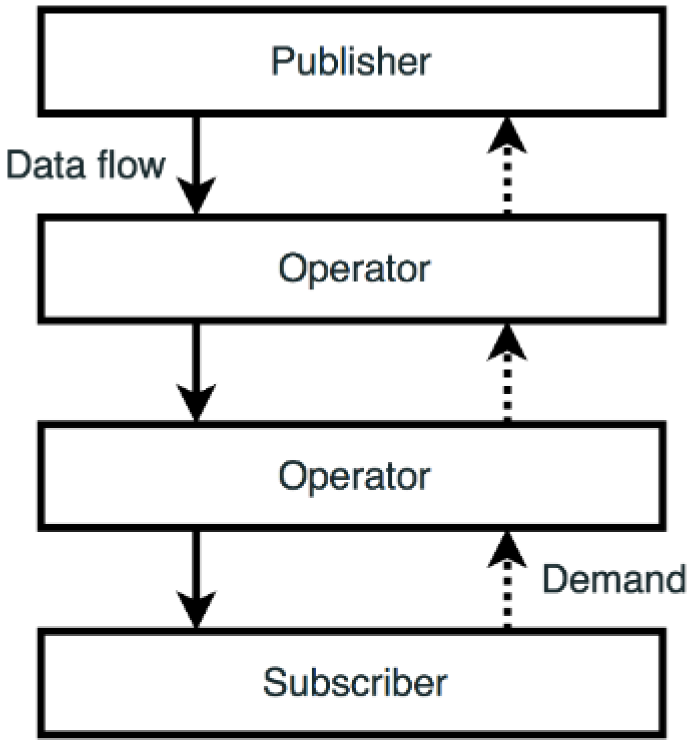
The Reactive Streams **specification** defines the following interfaces:

- `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

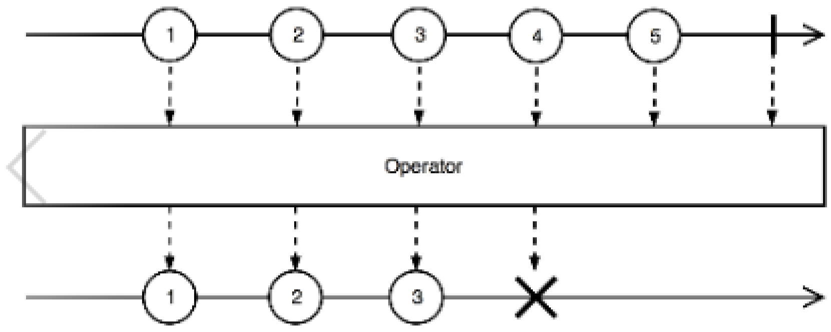
- Project Reactor 1.x
- Project Reactor 2.x

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")
```

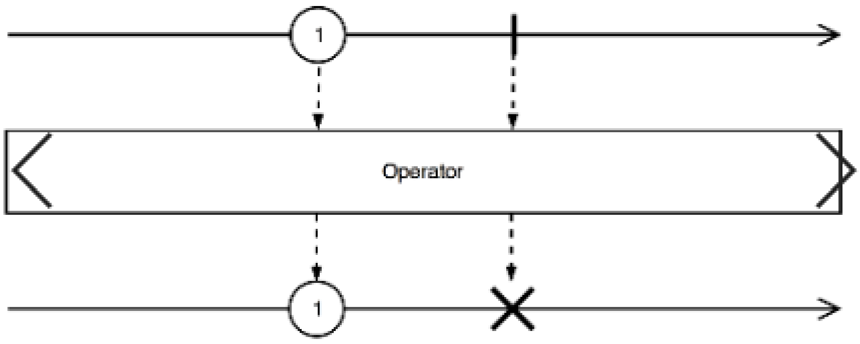
Reactive types – Flux and Mono as `Publisher<T>` implementation

- `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.

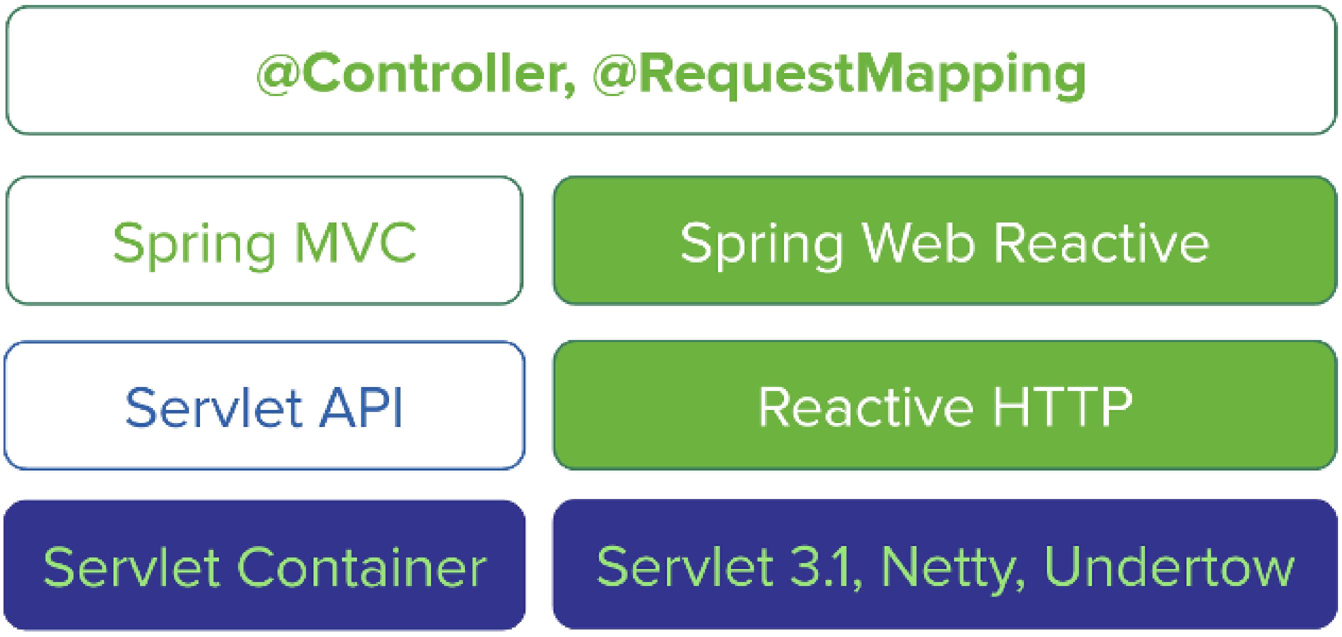
```
Flux.just("A","B","C")
  .subscribe(
        data -> log.info("onNext: {}", data),
        err ->{ /* ignored  */ },
        ()-> log.info("onComplete")
      );
```

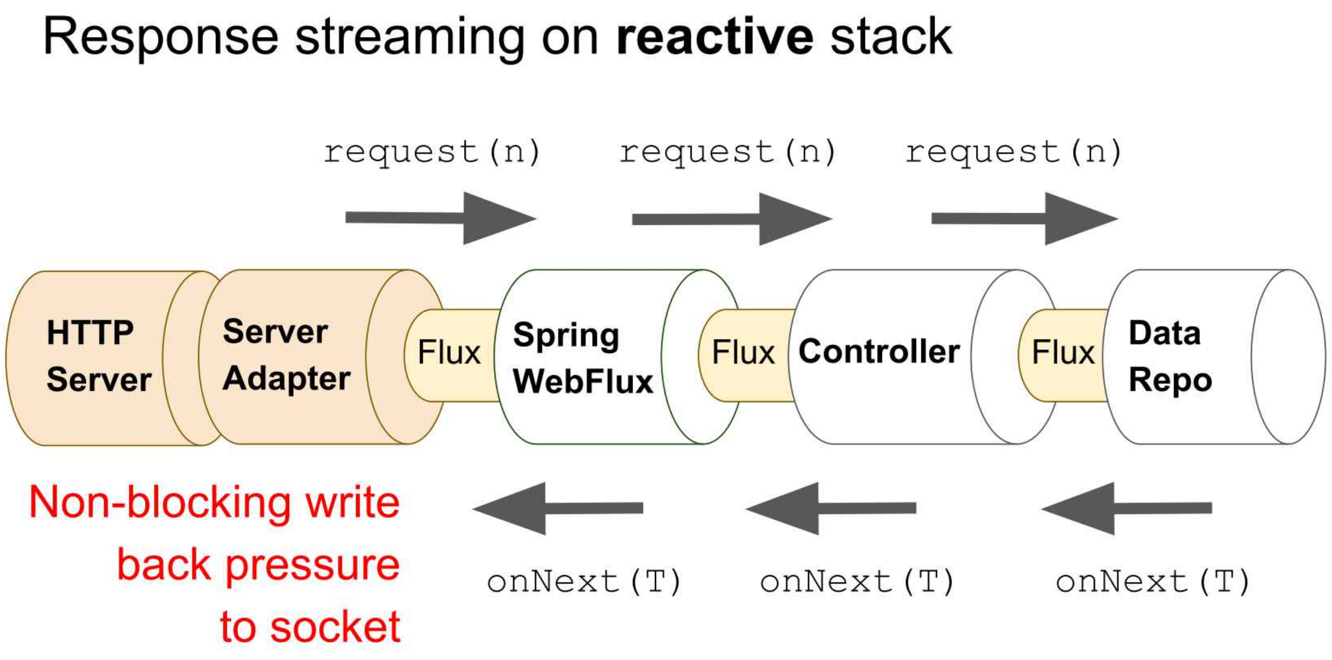
```
   Flux.just("A","B","C")
                .subscribe(
                        data -> log.info("onNext: {}", data),
                        err ->{ /* ignored  */ },
                        () -> log.info("onComplete")
                );
```

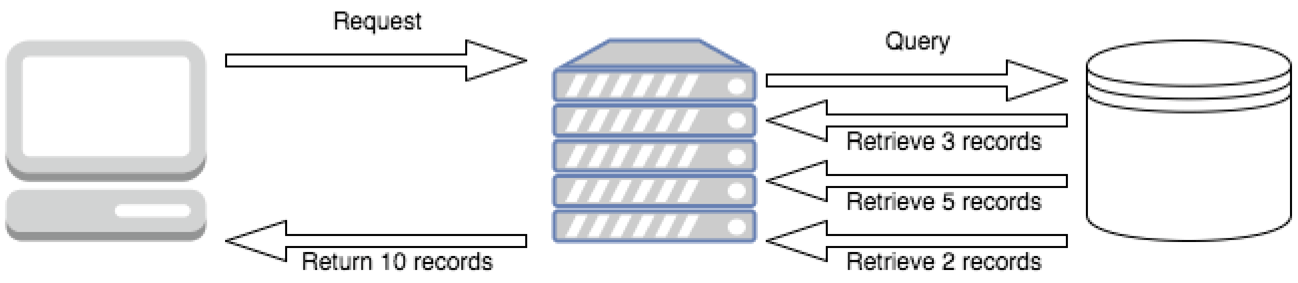
```
  Flux.range(1, 100)
                .subscribe(
                        data -> log.info("onNext: {}", data),
                        err -> { /* ignore */ },
                        () -> log.info("onComplete"),
                        subscription -> {
                            subscription.request(4);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

Reactive Spring

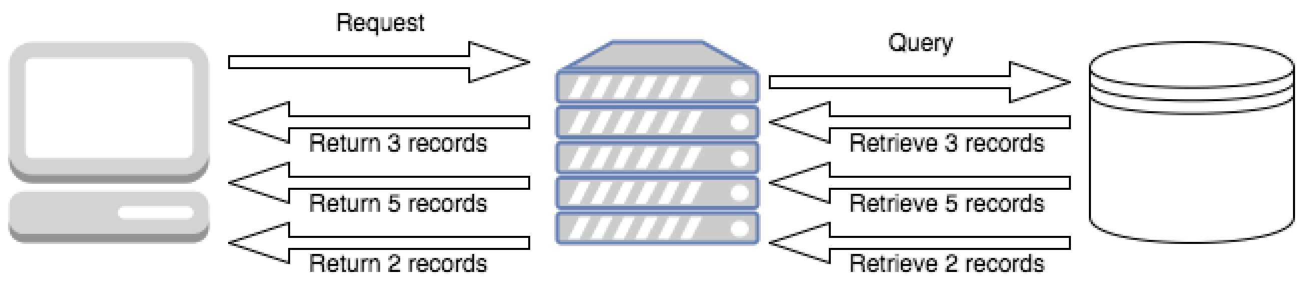
Spring with blocking (non reactive) DB connection







Spring with nonblocking (reactive) DB connection


Request
Return 3 records
Return 5 records
Return 2 records
Query
Retrieve 3 records
Retrieve 5 records
Retrieve 2 records