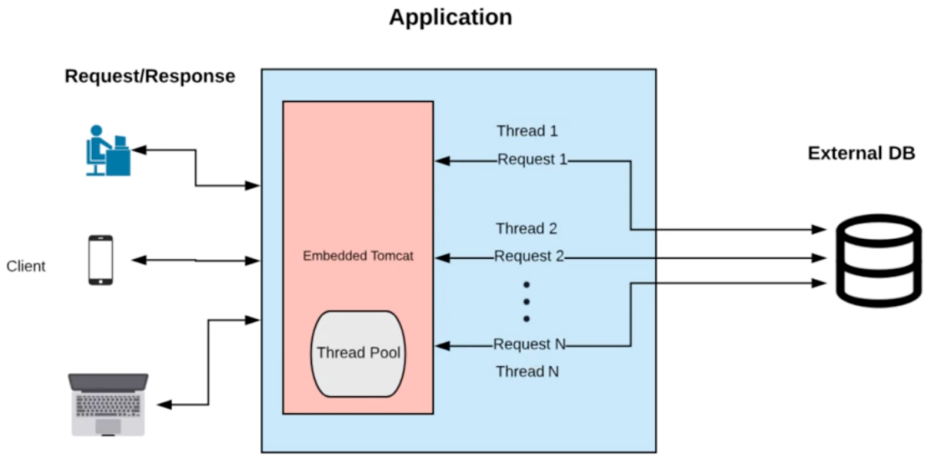
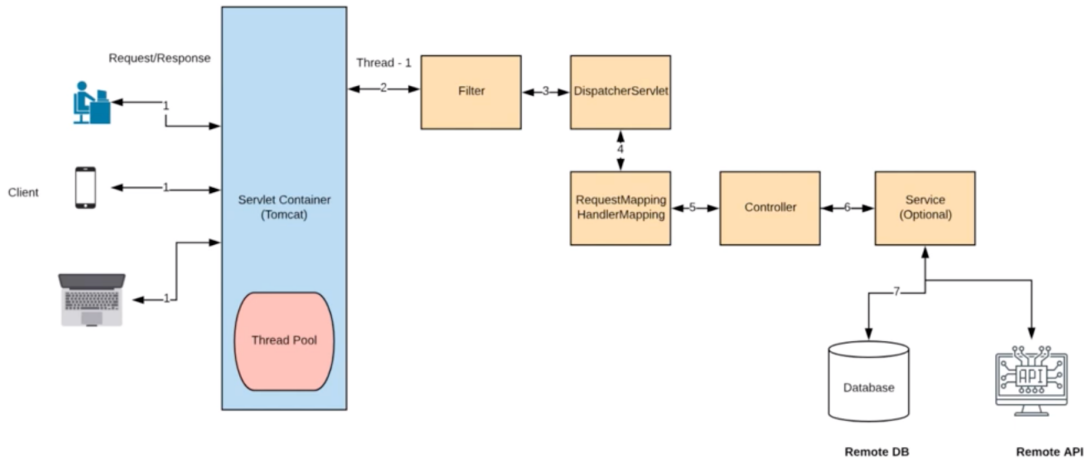
Reactive Programming:

Thread Per Request Model:



Spring MVC Request/Response Flow – Type 3



Handling Concurrent Requests

- Managed by the below property.

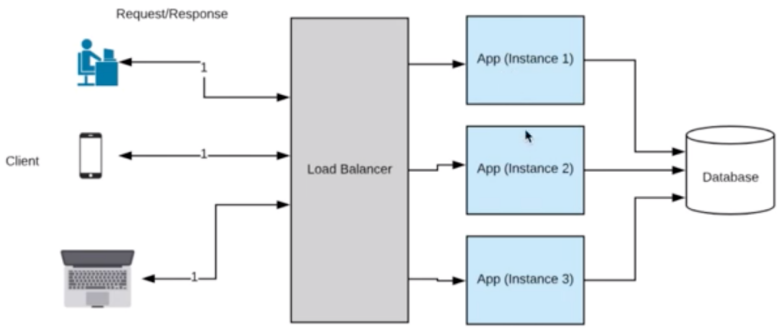
  server.tomcat.max-threads

- By default it can handle **200** connections.

- Can be overridden in **application.properties** or **application.yml file**

How is it handled today ?

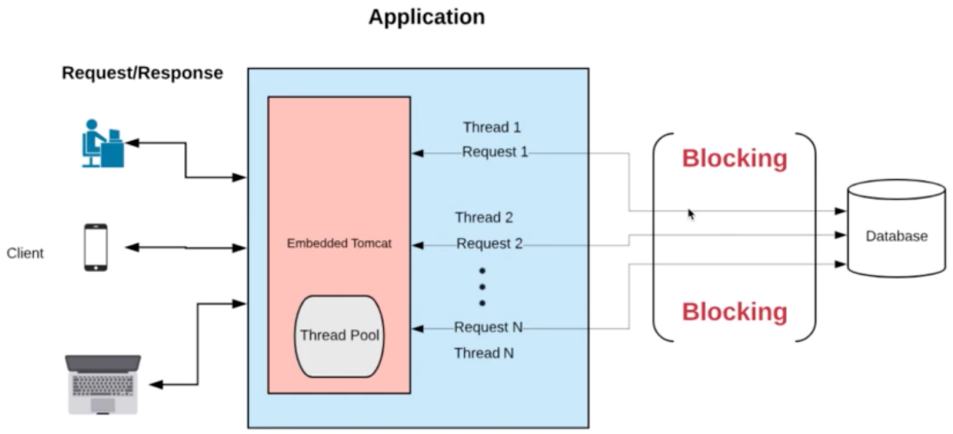
- Load is handled today **horizontal scaling**.



Handling Concurrent Requests

- Limitation on handling many concurrent users.
- Move away from **“Thread Per Request Model” .**

Traditional REST API Design:



Better API Design:

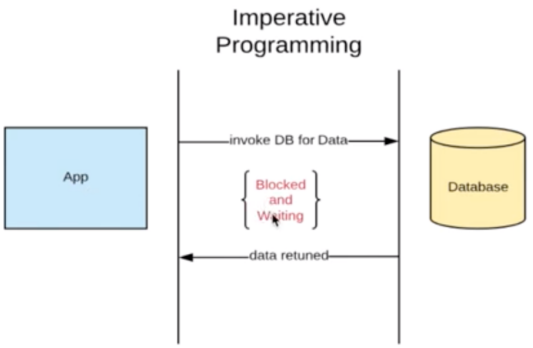
- **Asynchronous** and **Non Blocking**.
- Move away from **Thread per request** model.
- Use fewer threads.
- **Back Pressure** compatible.

What is Reactive Programming ?

- New programming paradigm.
- Asynchronous and Non Blocking.
- Data flow as an **Event/Message Driven** stream.
- Functional Style Code.
- Back Pressure on Data Streams.

Imperative Programming:

```
List<Item> items = itemRepository.getAllItems();
```



Data flow as an Event Driven stream

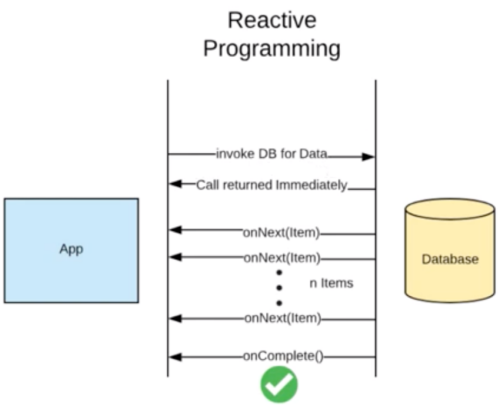
- One **Event or Message** for a every result item from Data Source.

- Data Sources:
  - Data Base
  - External Service
  - File etc.,

- One **Event or Message** for **completion** or **error**.

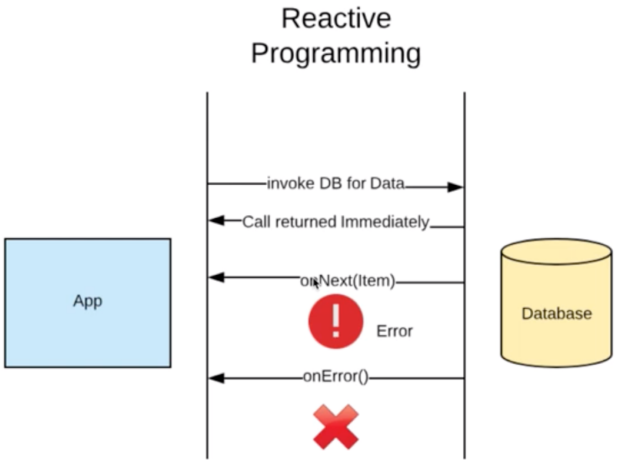
Data flow as an Event Driven stream

```
List<Item> items = itemRepository.getAllItems();
```



Data flow as an Event Driven stream

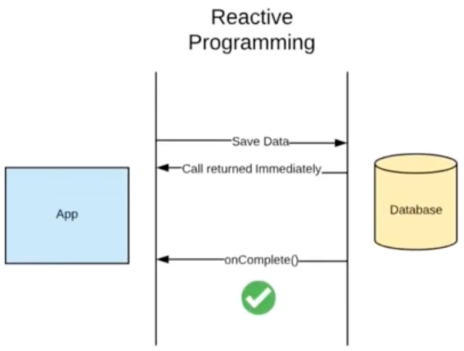
- Error Flow



Data flow as an Event Driven stream

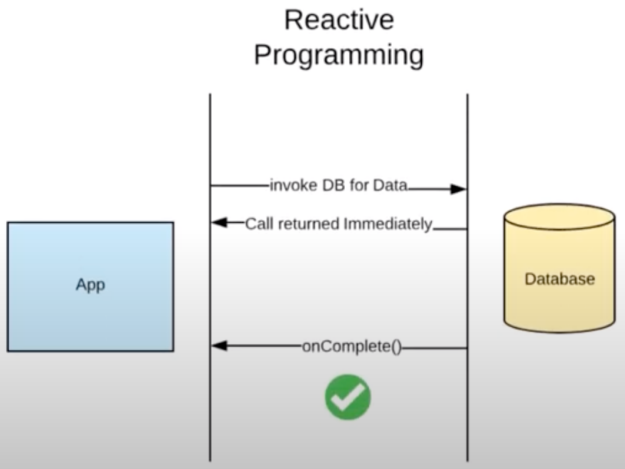
- Save Data

```
itemReactiveRepository.save(item);
```



Data flow as an Event Driven stream

- No Data



Summary - Data flow as an Event Driven stream

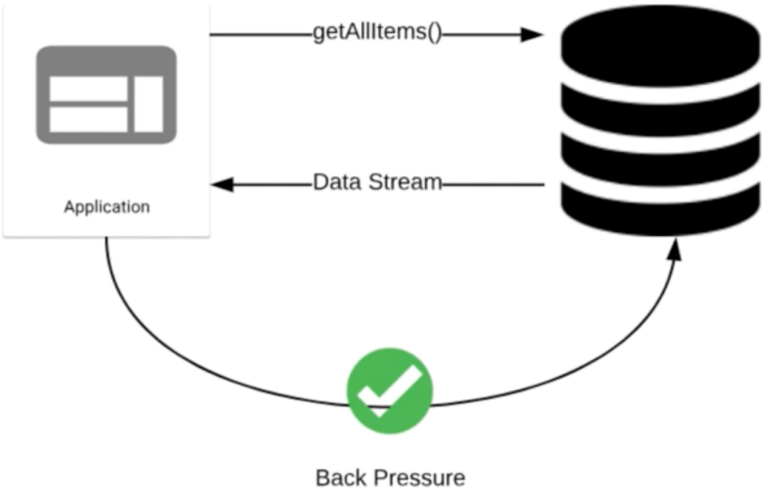
- OnNext(item) – Data Stream events.
- OnComplete() -> Completion/Success event.
- OnError() -> Error Event

Functional Style Code

- Similar to **Streams API**.
- Easy to work with **Lambdas**.

```
@PutMapping("{id}")
public Mono<ResponseEntity<Item>> updateOfficer(@PathVariable(value = "id") String id,
                                                @RequestBody Item item) {
    return itemRepository.findById(id) Mono<Item>
            .flatMap(currentItem -> {
                currentItem.setPrice(item.getPrice());
                return itemRepository.save(currentItem);
            }) Mono<Item>
            .log() Mono<Item>
            .map(updateItem -> new ResponseEntity<>(updateItem, HttpStatus.OK)) Mono<ResponseEntity<Item>>
            .log() Mono<ResponseEntity<Item>>
            .defaultIfEmpty(new ResponseEntity<>(HttpStatus.NOT_FOUND));
}
```

Back Pressure on Data Streams



Reactive Stream Specification

- Publisher
- Subscriber
- Subscription
- Processor

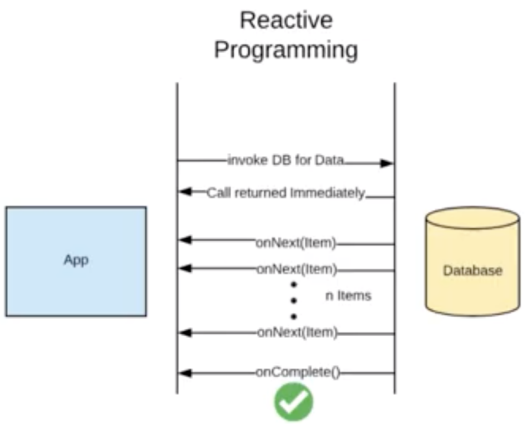
Publisher

```
public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
}
```

- Represents the **Data Source**.
  - Data Base
  - External Service etc.,

Subscriber:

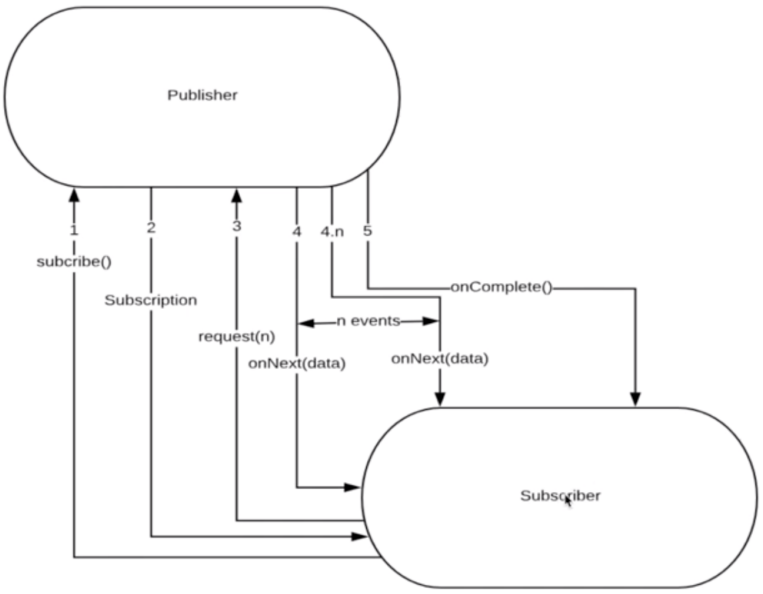
```
public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```



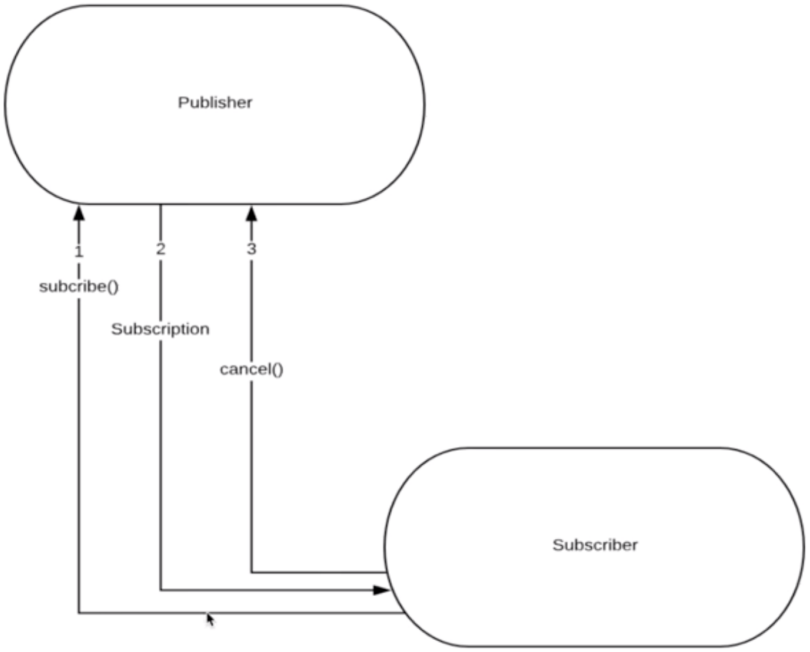
Subscription

```
public interface Subscription {
    public void request(long n);
    public void cancel();
}
```

Publisher/Subscriber Event Flow



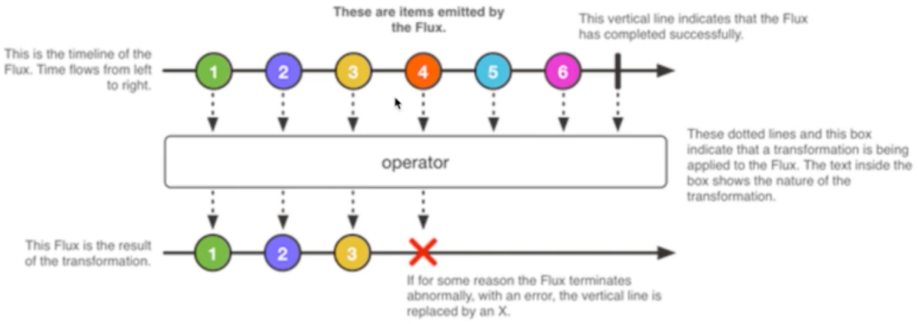
Publisher/Subscriber Event Flow



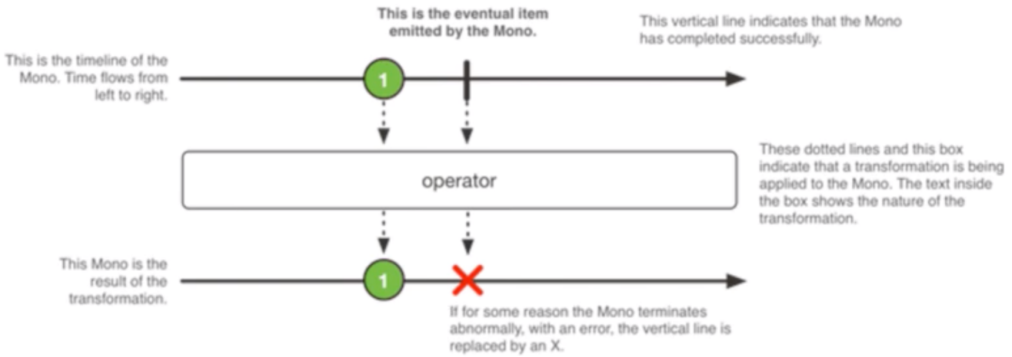
Processor

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
}
```

Flux – 0 to N elements



Mono – 0 to 1 elements



Spring WebFlux - Functional Web: