Build Reactive RESTFUL APIs using Spring Boot/WebFlux

- Dilip

What’s Covered ?

- Reactive Programming.
- Why Reactive Programming ?
- Project Reactor
- Reactive APIs using Spring Webflux
- Automated tests using Junit.

Targeted Audience

Code

YouTube Channel

“CODE WITH DILIP”

Thank You!

Prerequisites:

- Knowledge of Spring Boot.
- Experience building RESTFUL APIs.
- Knowledge of Java8 Lambdas, Streams.

Gift This Course Wishlist

Modern Java - Learn Java 8 features by coding it

Learn Lambdas, Streams , new Date APIs, Optionals and Parallel programming in Java 8 by coding it.

4.5 (158 ratings) 800 students enrolled

Created by Dilip S Last updated 8/2018 English

$28.99 ~~$99.99~~ 71% off

Add to cart

Buy now

30-Day Money-Back Guarantee

Includes

- 11 hours on-demand video
- 1 article
- 80 downloadable resources
- Full lifetime access
- Access on mobile and TV
- Certificate of Completion

What you'll learn

- Complete understanding of Lambdas, Streams , Optional via code.
- Learn to build complex Streams Pipeline.
- Learn to use Method Reference , Constructor reference syntax.
- Student will be able to upgrade their Java knowledge with the new Functional Features.
- Students will be able to implement the new Java 8 concepts in real time
- Learn the new Date/Time Libraries in Java 8.
- Learn and understand Parallel Programming with the Streams.
- This course will be continuously updated.

Prerequisites:

- Knowledge of Spring Boot.
- Experience building RESTFUL APIs
- Knowledge of Java8 Lambdas, Streams.
- At least JDK 8 is needed.
- IntelliJ or any IDE (Eclipse, NetBeans etc.,)

Prerequisites:

- Knowledge of Spring Boot.
- Experience building RESTFUL APIs
- Knowledge of Java8 Lambdas, Streams.
- At least JDK 8 is needed.
- IntelliJ or any IDE (Eclipse, NetBeans etc.,)

Why Reactive Programming ?

Evolution of Programming

Past (10 -15 Years ago)

- Monolith Applications.
- Run in App Server.
- Does not embrace Distributed Systems.

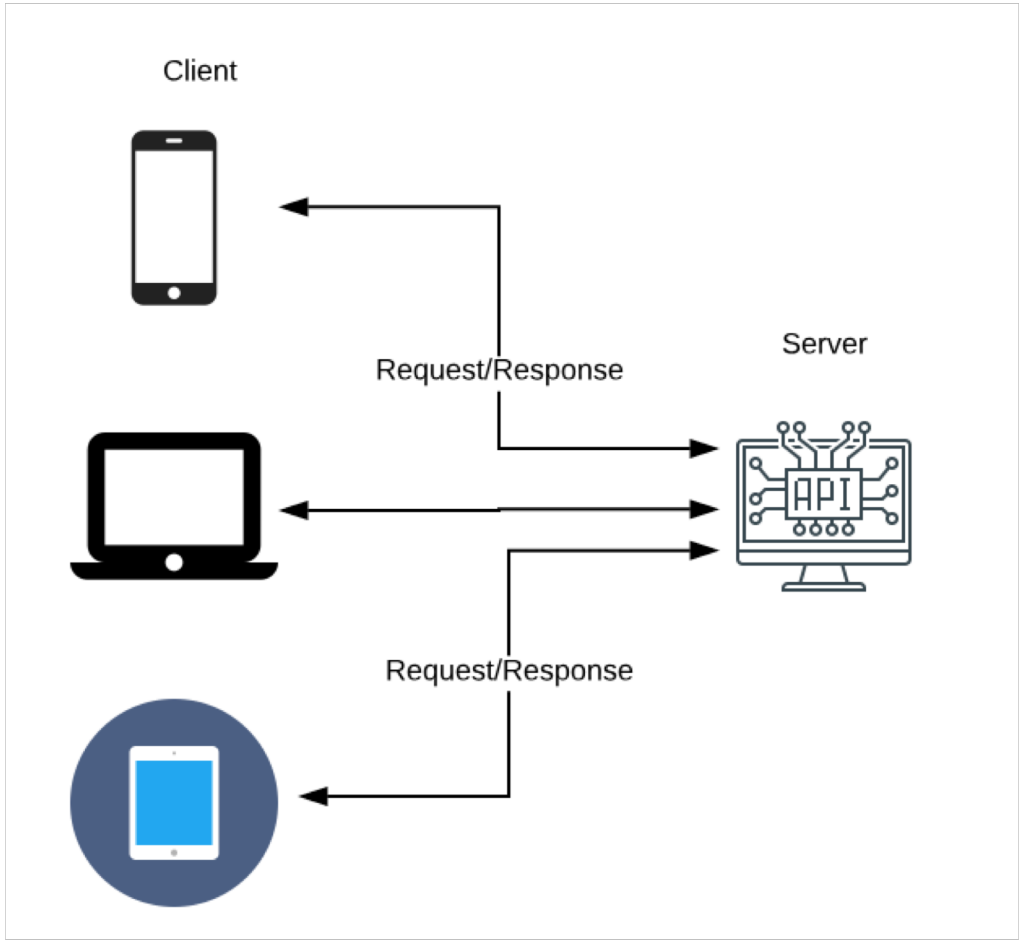
Today

- Micro Services.
- Run it in cloud.
- Embraces Distributed systems.

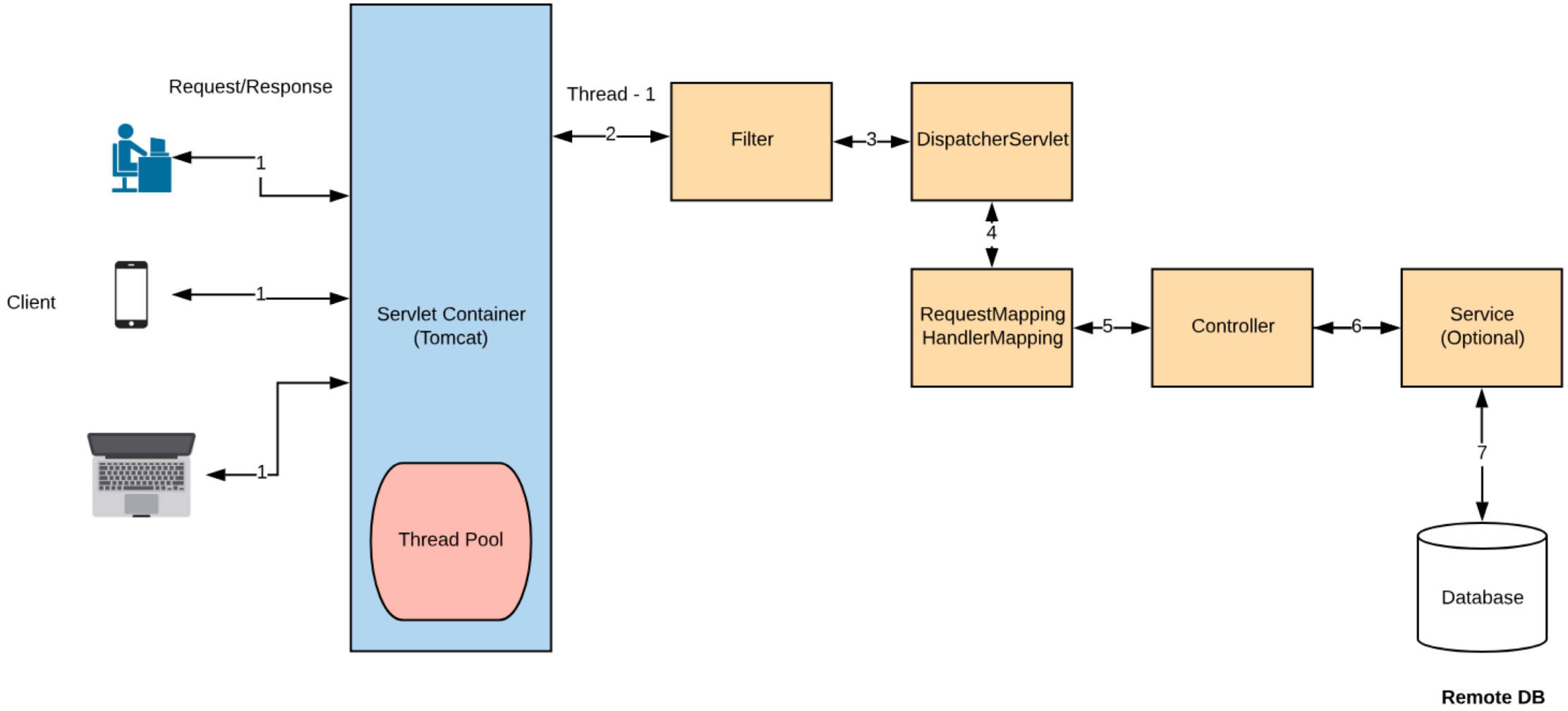
Expectations of the App:

- Scale based on the load.
- Use resources efficiently.
- Latency or Response Time should be faster.

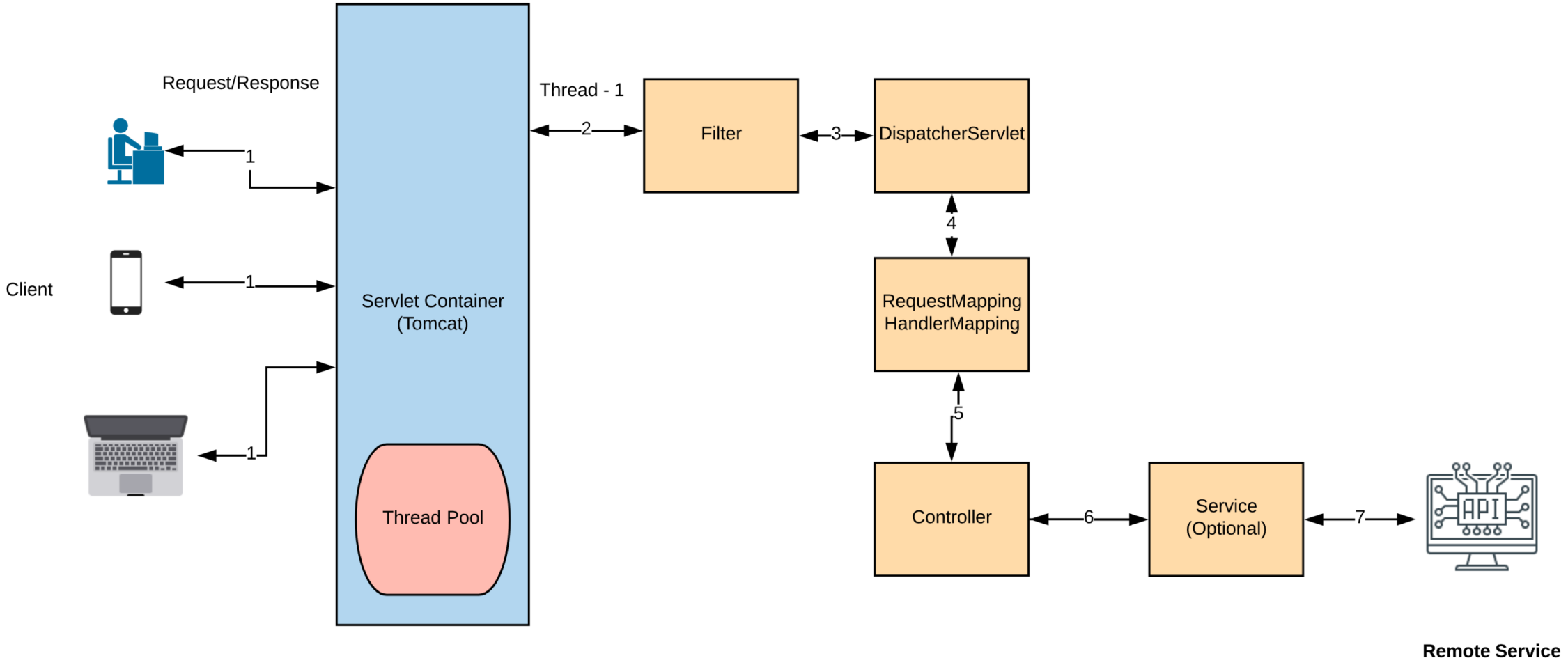
Traditional Rest API Request/Response using Spring MVC/Spring Boot



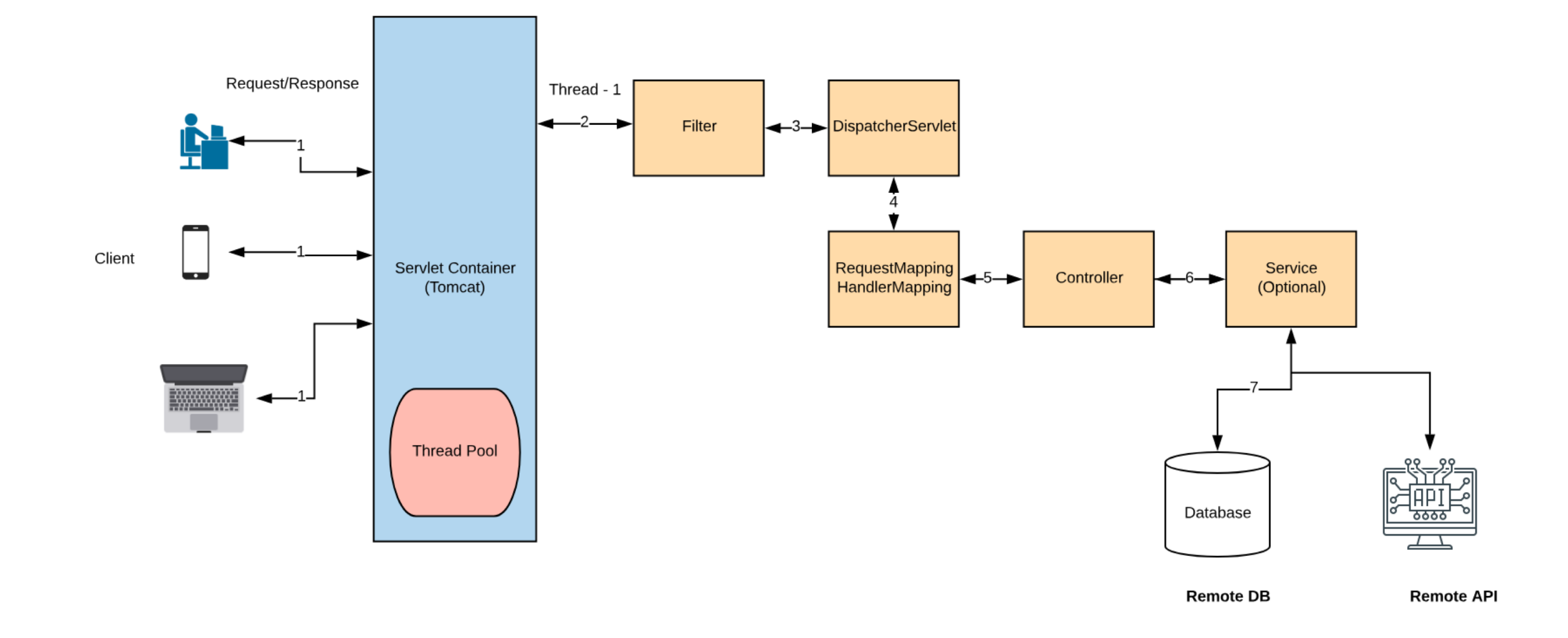
Spring MVC Request/Response Flow – Type 1



Spring MVC Request/Response Flow – Type 2

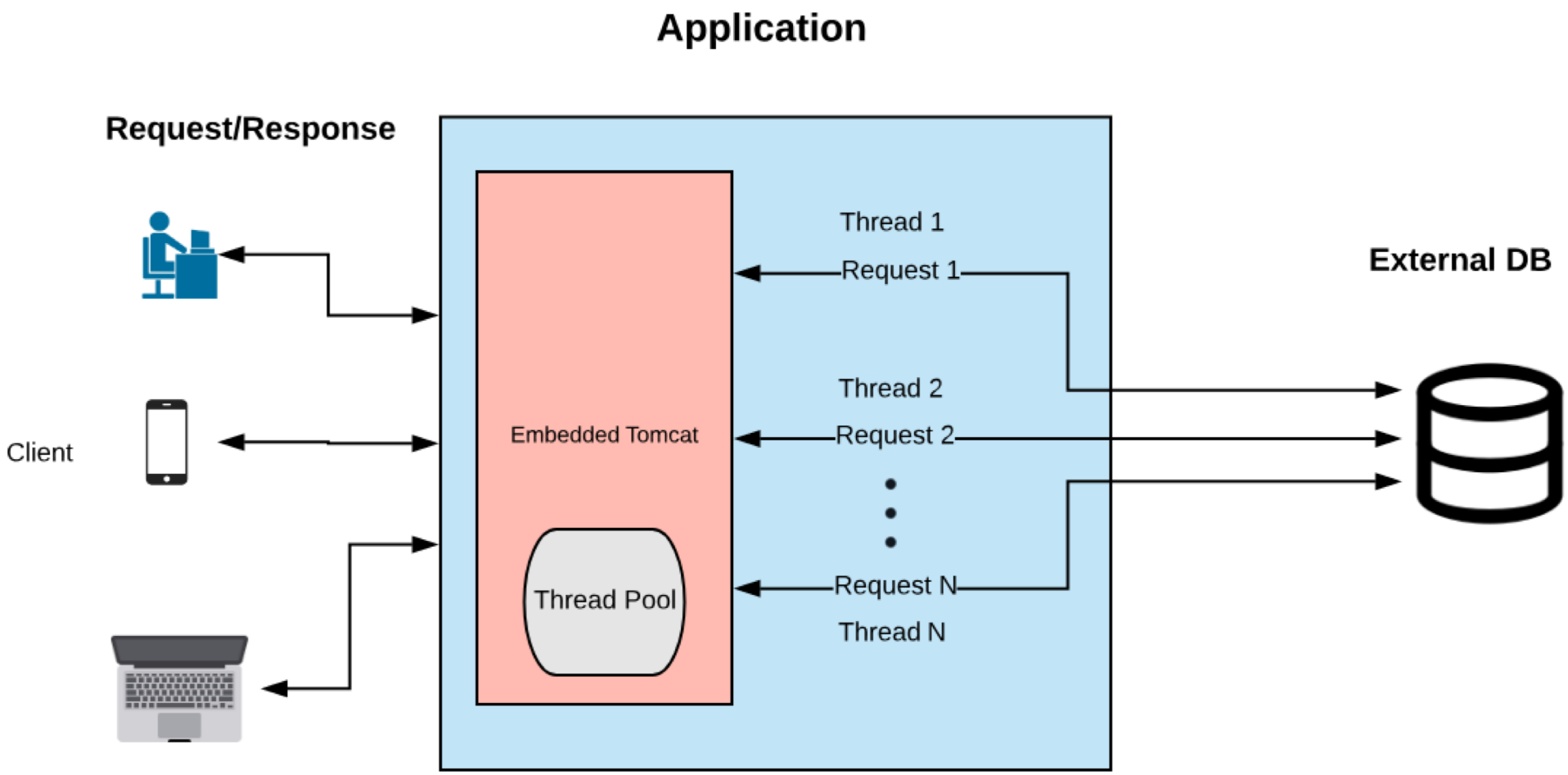


Spring MVC Request/Response Flow – Type 3



Handling Concurrent Requests

Thread Per Request Model:



Handling Concurrent Requests

- Managed by the below property.

  server.tomcat.max-threads

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

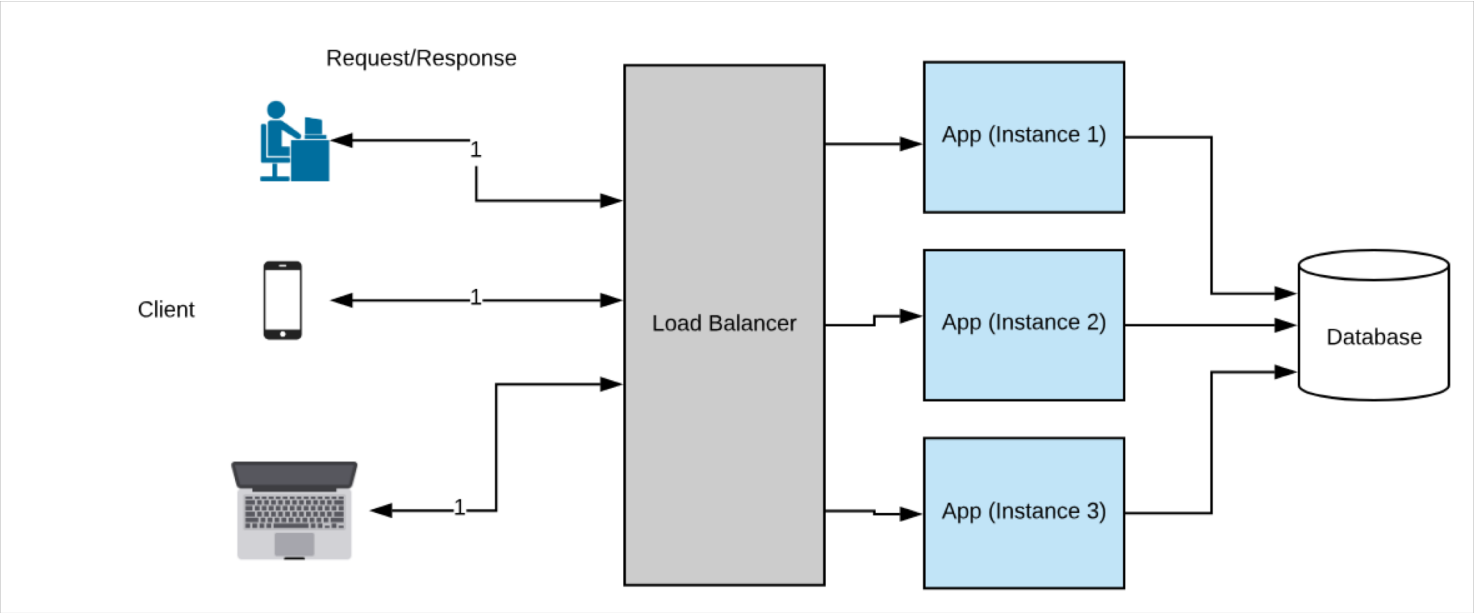
- Can be overridden in **application.properties** or **application.yml file**

Handling Concurrent Requests

- Each thread takes some memory.
- Common Stack size is 1MB.
- Higher the thread pool size, Higher the memory consumption.
- Applications really perform poor with less memory available.

How is it handled today ?

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



- **Kubernetes** or some container orchestration.

Handling Concurrent Requests

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

Traditional Rest APIs:

- Does it mean **Traditional Rest API** will be obsolete ?
  - No
- Does it mean **Traditional Rest API** should moved to the new model ?
  - No
- Works Perfect for many use cases.
- Many APIs in the future will be built with the same design.

Traditional Rest API Design using Spring MVC

Traditional REST API Design:

```
@GetMapping("/v1/items/{id}")
public ResponseEntity<Item> getItemFromExternalServices(@PathVariable Integer id){

    //1st db call
    Price price = priceRepository.findById(id).get();  // 1.db-call  2.blocking

    //2nd rest call
    ResponseEntity<Location> locationEntity =
            restTemplate.getForEntity(locationUrl,Location.class);  // 1.rest call synchronous  2.blocking

    Item item = buildItem(price,locationEntity.getBody());

    return ResponseEntity.ok(item);

}
```

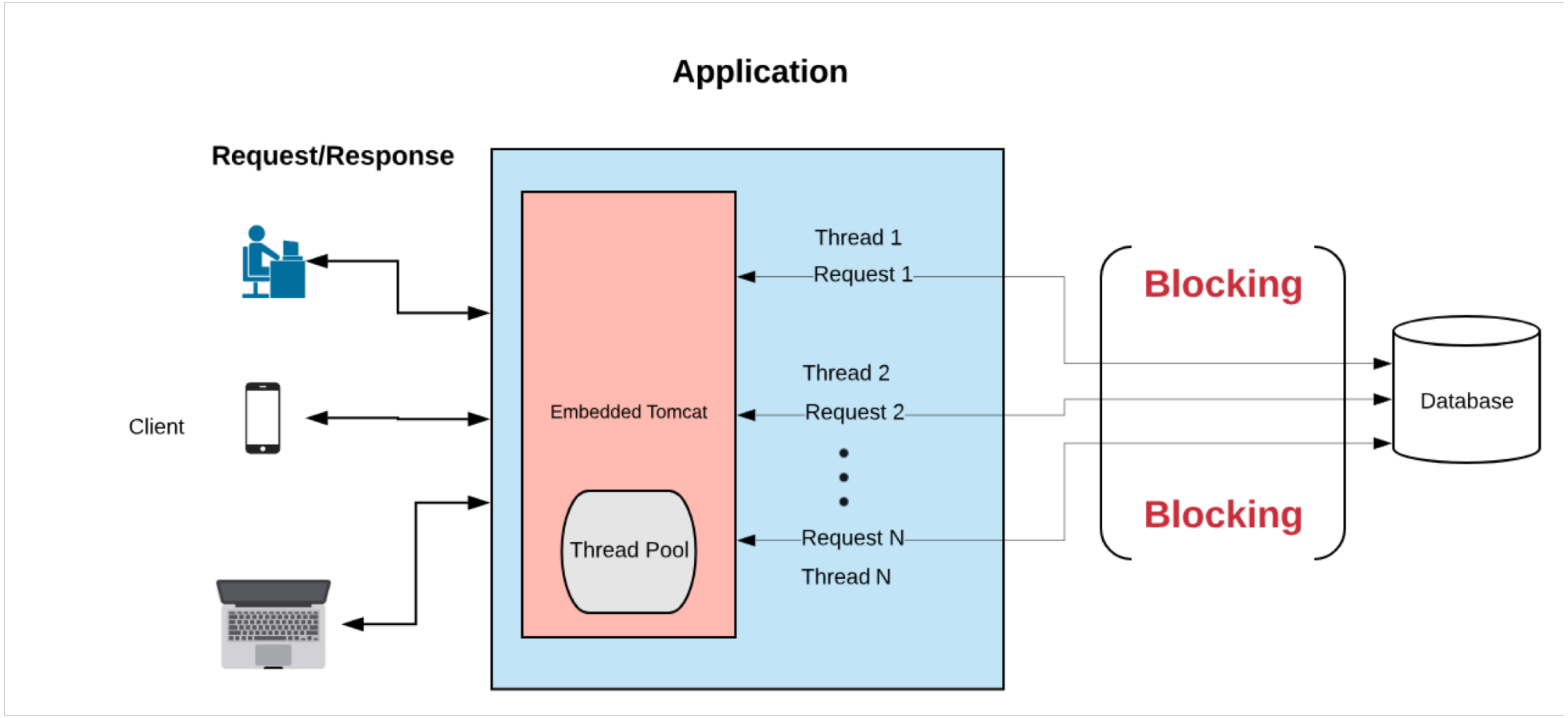
- Imperative Style APIs
  - Top-down approach

Traditional REST API Design:

- **Blocking** and **Synchronous**.

```
@GetMapping("/v1/items/{id}")
public ResponseEntity<Item> getItemFromExternalServices(@PathVariable Integer id){

    //1st db call
    Price price = priceRepository.findById(id).get();  // 1.db-call  2.blocking

    //2nd rest call
    ResponseEntity<Location> locationEntity =
            restTemplate.getForEntity(locationUrl,Location.class);  // 1.rest call synchronous  2.blocking

    Item item = buildItem(price,locationEntity.getBody());

    return ResponseEntity.ok(item);

}
```

Traditional REST API Design:



Traditional REST API Design:

Synchronous:

```
@GetMapping("/v1/items/{id}") // 2 milli sec
public ResponseEntity<Item> getItemFromExternalServices1(@PathVariable Integer id){

    //1st db call
    Price price = priceRepository.findById(id).get();  // 1 milli sec

    //2nd rest call
    ResponseEntity<Location> locationEntity =
            restTemplate.getForEntity(locationUrl,Location.class);  // 1 milli sec

    Item item = buildItem(price,locationEntity.getBody());

    return ResponseEntity.ok(item);

}
```

Traditional REST API Design:

- Need to make calls asynchronous, basically non blocking.

- Currently in Java we have:
  - Callbacks
  - Futures

Callbacks:

- Complex.
- No Return Value.
- Code is hard to read and maintain.
- Leads to **CallBack** hell.

Future

- Another alternative to write **asynchronous** code in Java.
- Returns **Future** instance.
- Hard to compose multiple asynchronous operations.

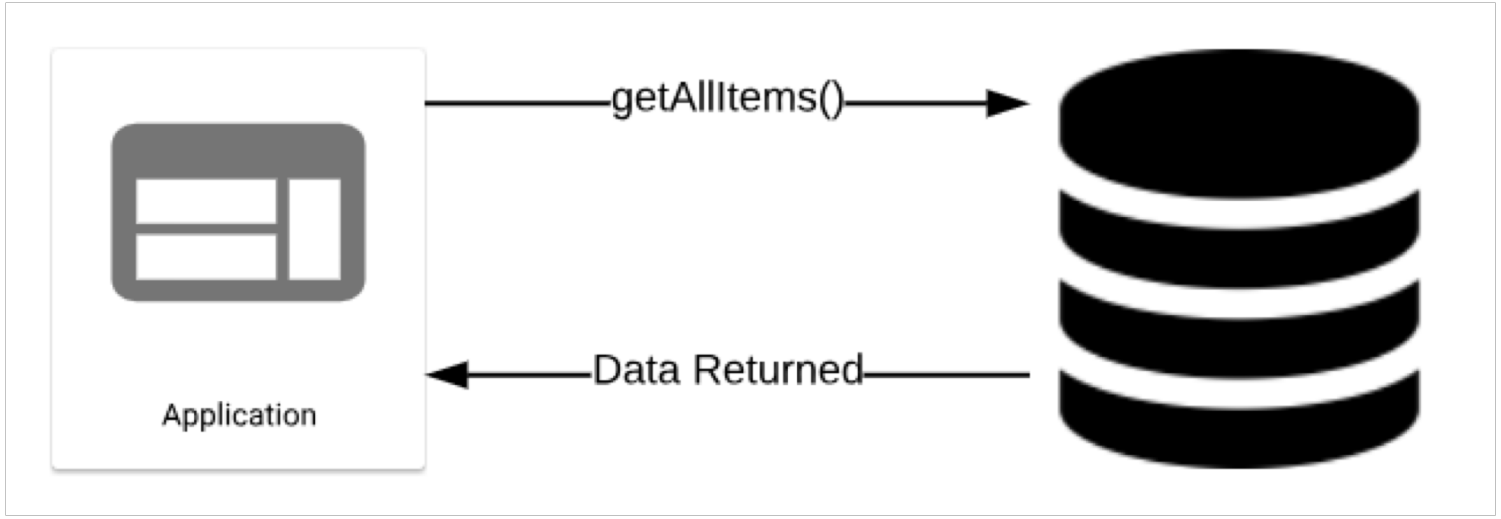
Completable Future

- Introduced as part of **Java8**.
- Supports functional style API.
- Easy to compose multiple Asynchronous operations.
- Not a great fit **asynchronous call** with multiple items.

Traditional REST API Design:

```
@GetMapping("/v1/items")
public ResponseEntity<List<Item>> getAllItems(){

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

    return ResponseEntity.ok(items);

}
```

Traditional REST API Design:

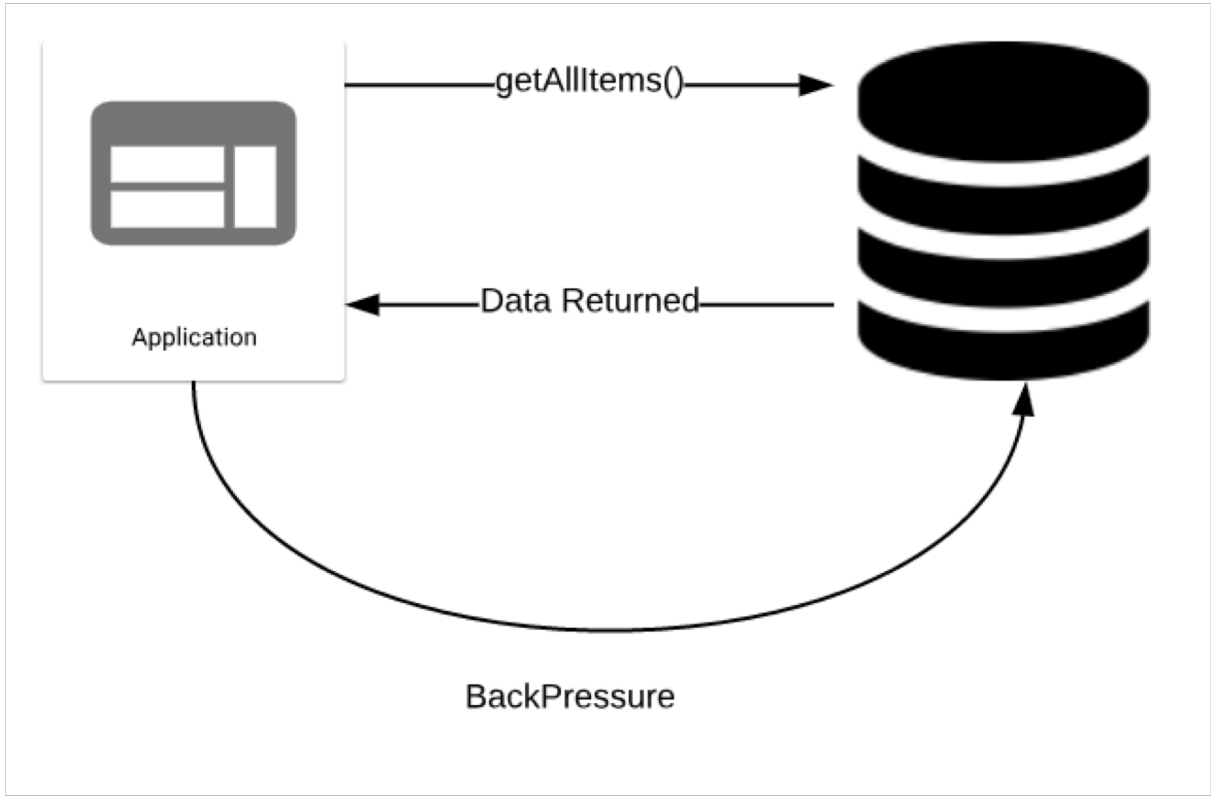


- Application crash with **Out Of Memory** error.

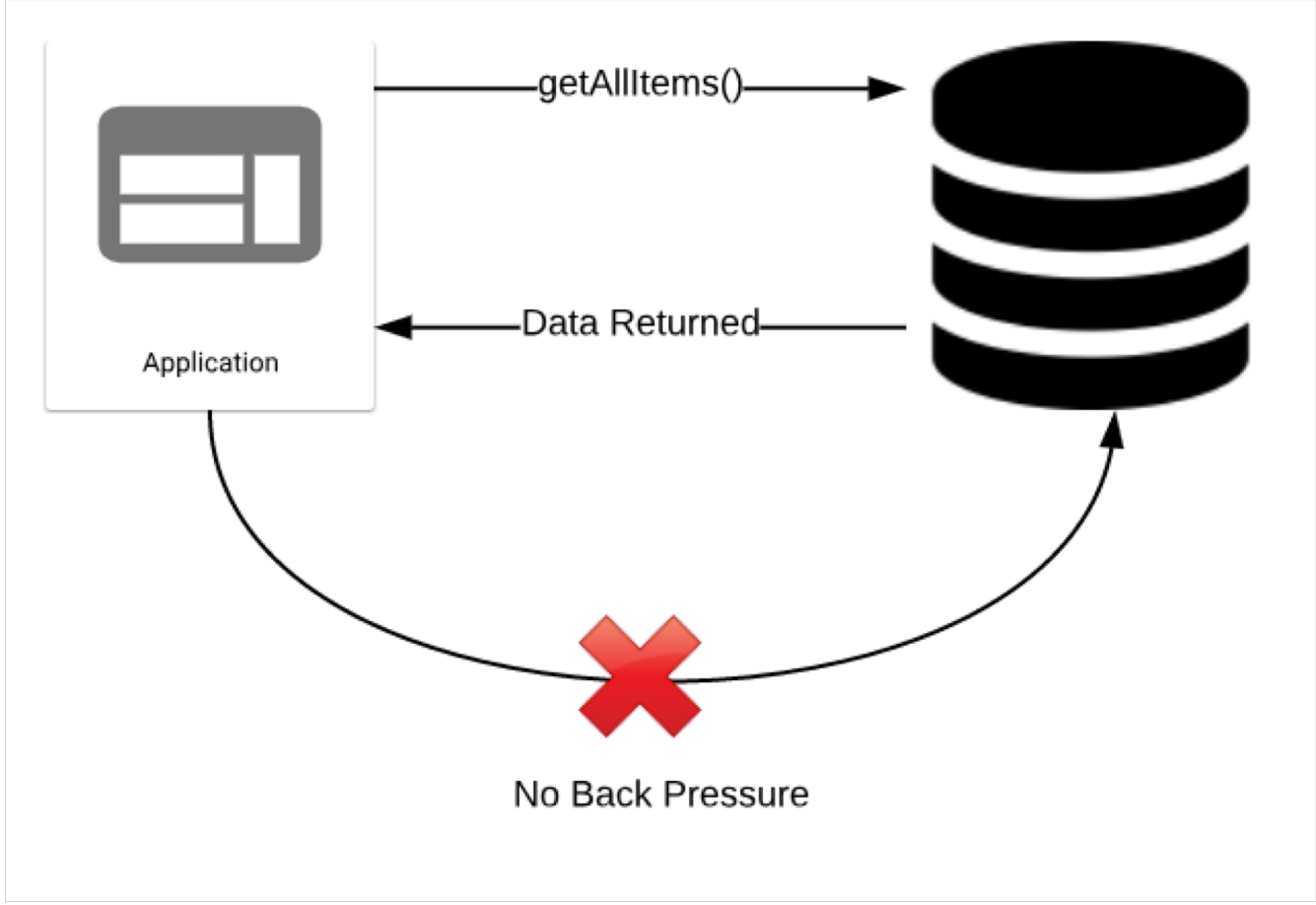
- Client might be overwhelmed with huge data.

Traditional REST API Design:

- How to avoid this from happening ?



Traditional REST API Design:



Traditional Rest API Design:

- Limit on the number of Concurrent users.
- Synchronous and Blocking.
- Imperative Style APIs
- No Back Pressure support.

Traditional Rest API Design:

- Does it mean **Traditional Rest API Design** is not recommended ?
  - No
  - **Spring MVC** module will still be part of Spring Eco system and continue to be supported and enhanced.
  - **Traditional API design** fits for many use cases.

Better API Design:

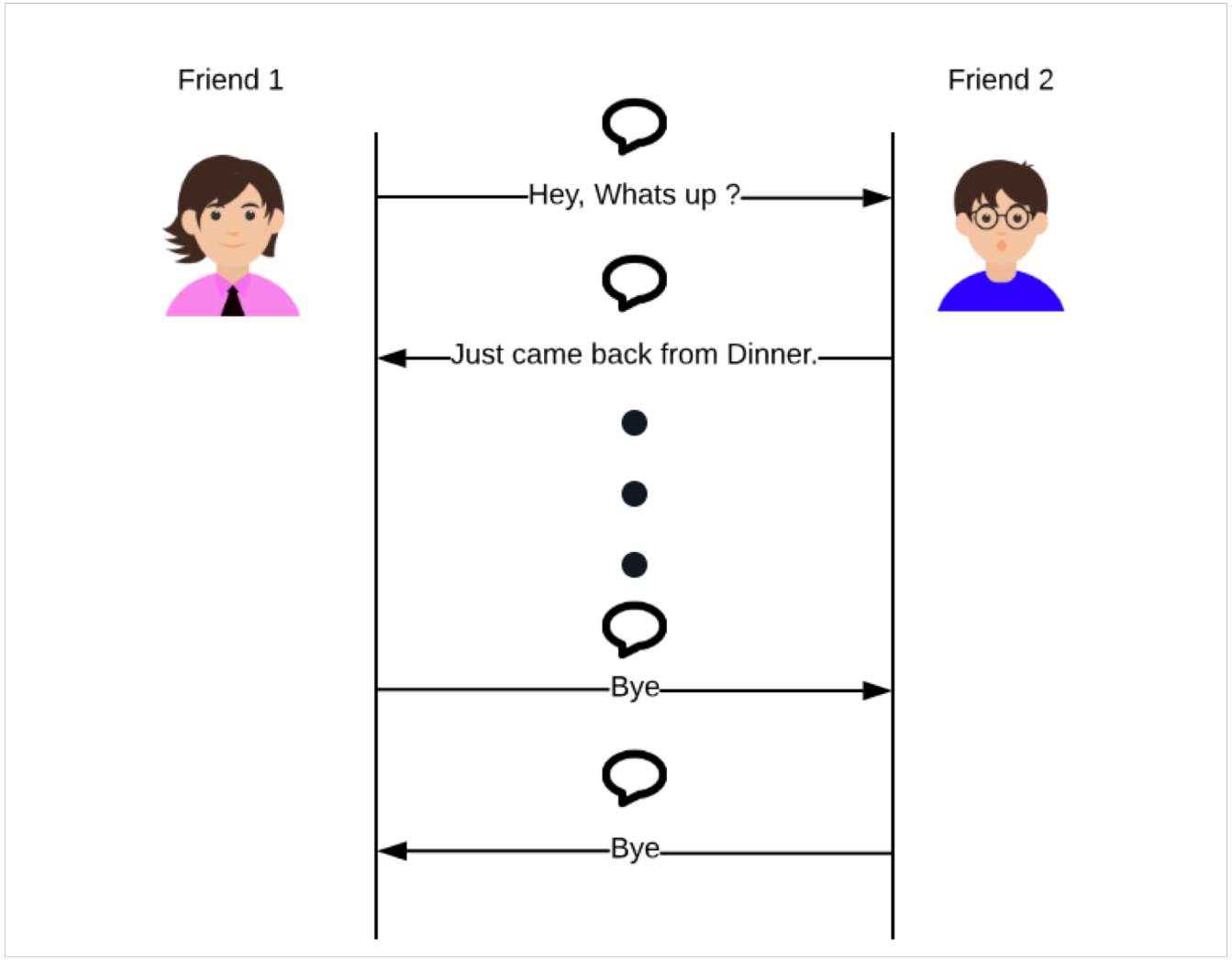
- **Asynchronous** and **Non Blocking**.
- Move away from **Thread per request** model.
- Use fewer threads.
- **Back Pressure** compatible.

Reactive Programming

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.

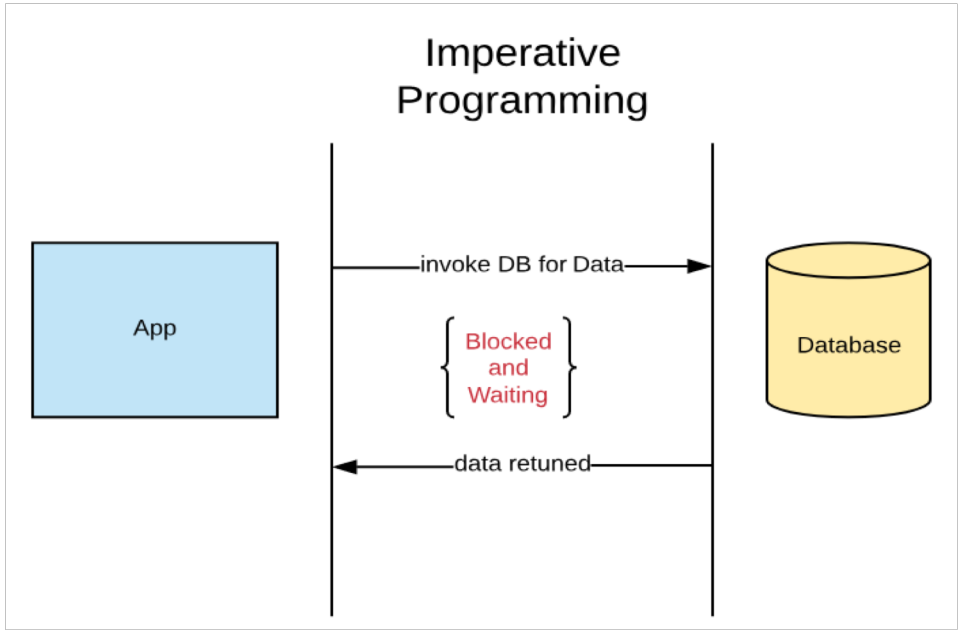
Reactive Programming



Data flow as an Event/Message Driven stream

Imperative Programming:

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



- Synchronous and blocking communication Model.

Data flow as an Event Driven stream

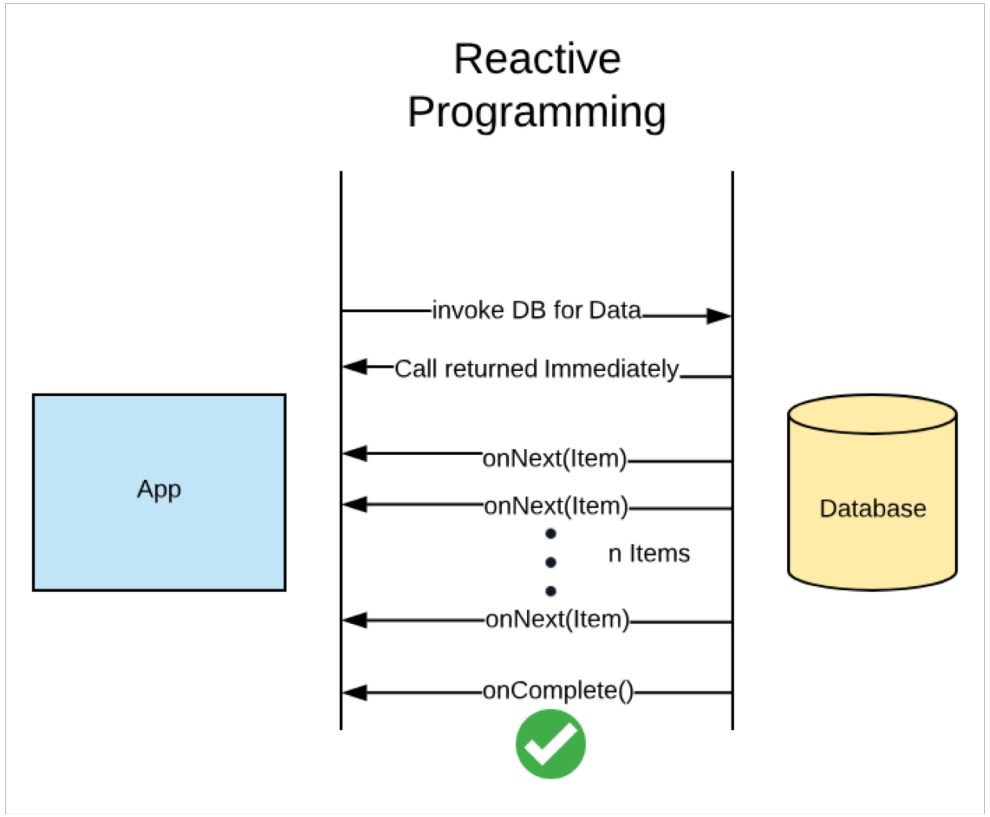
- One **Event or Message** for a every result item from Data Source.

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

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

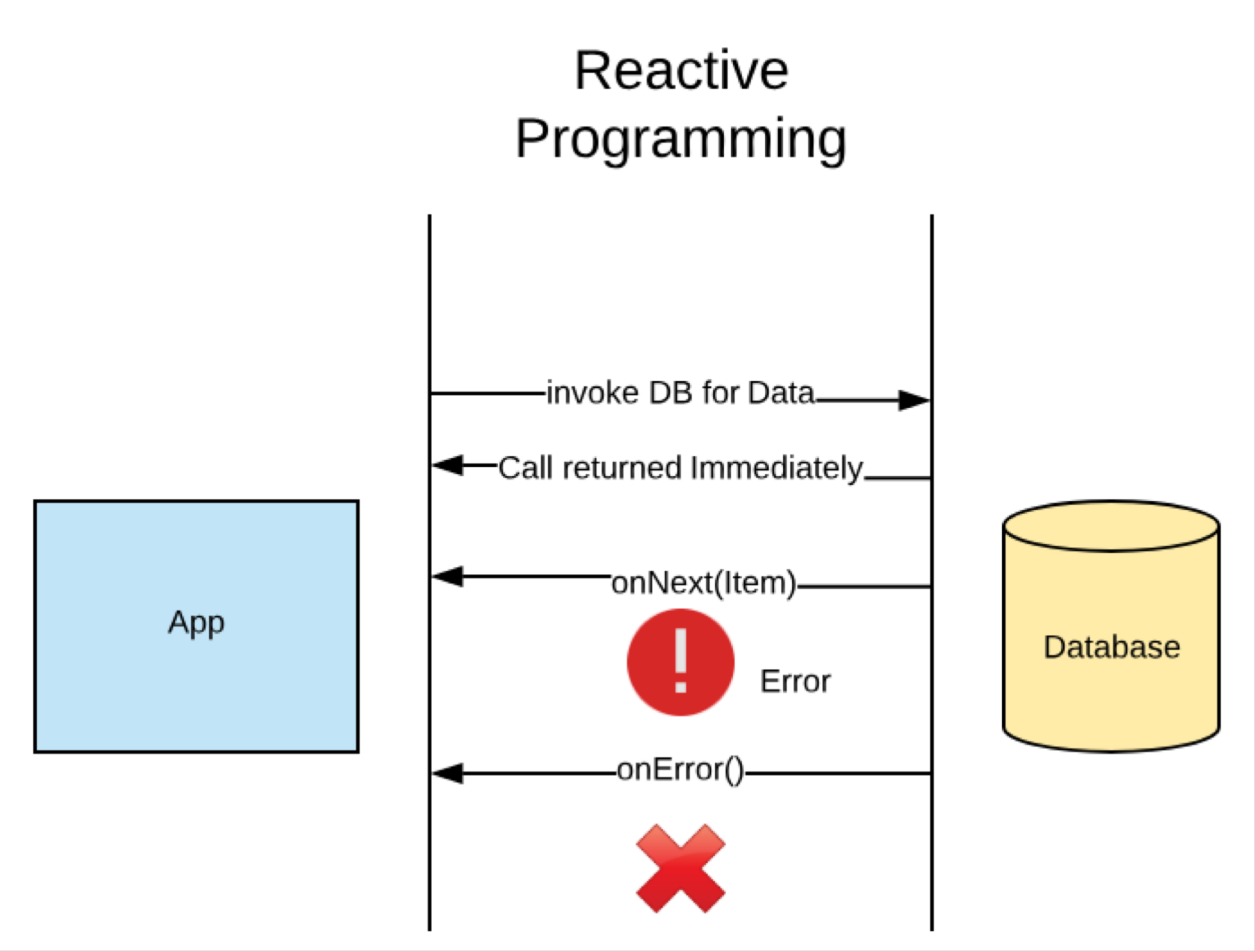
Data flow as an Event Driven stream

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



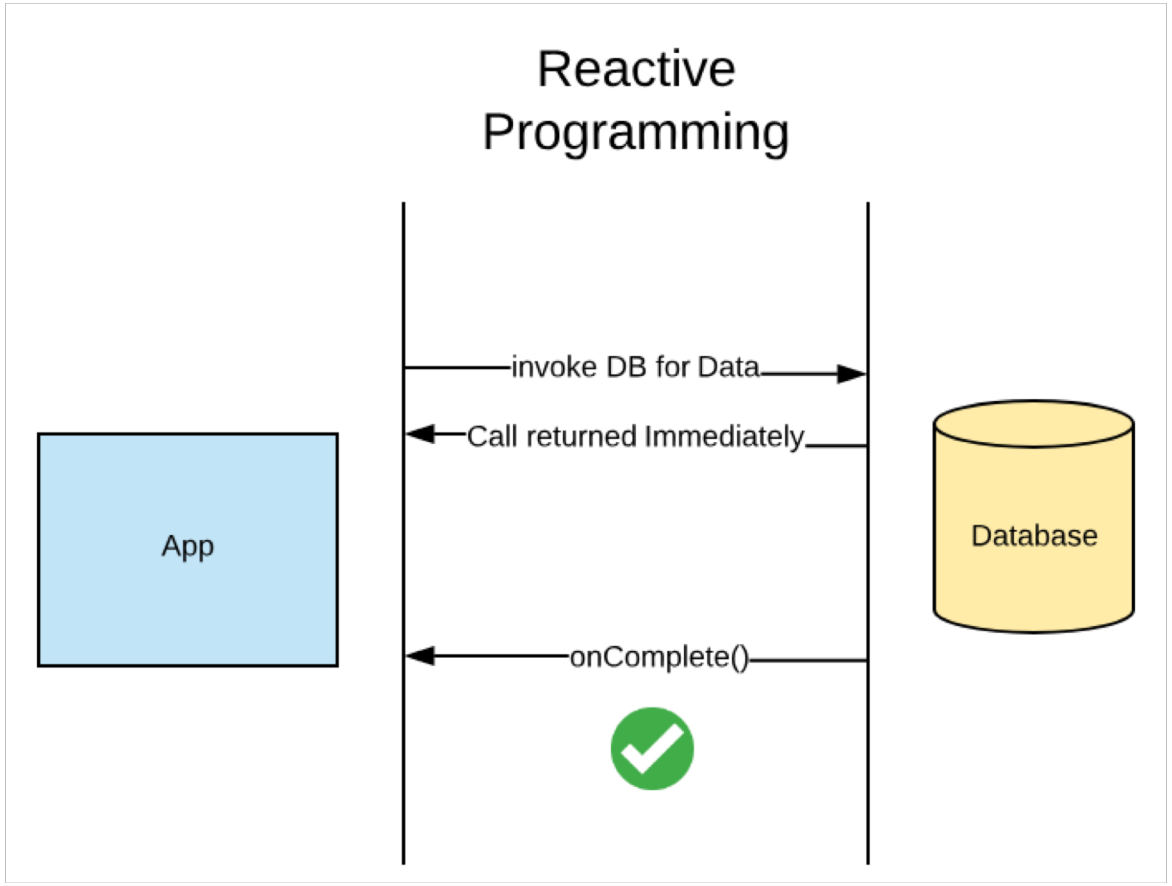
Data flow as an Event Driven stream

- Error Flow



Data flow as an Event Driven stream

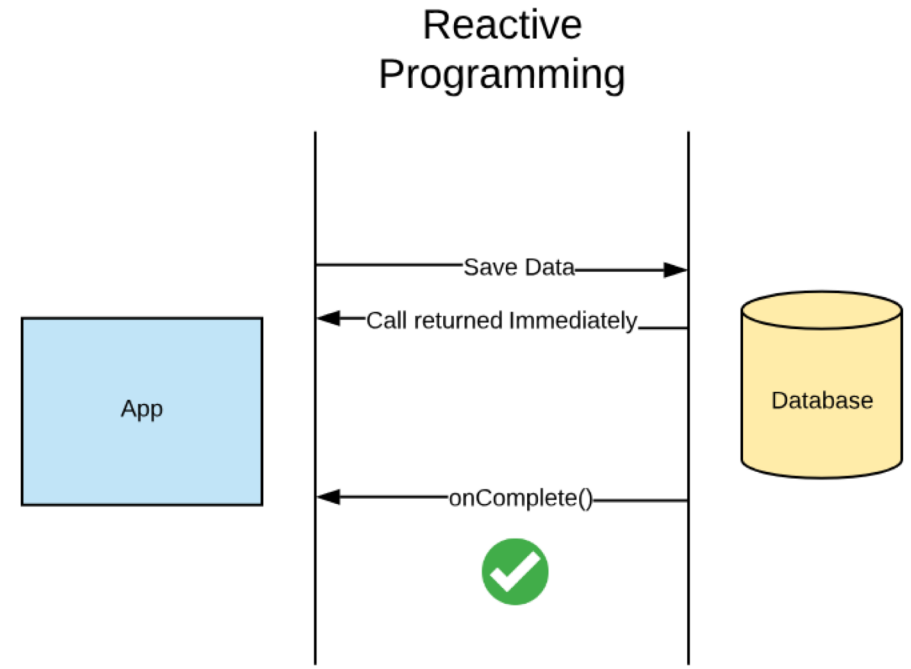
- No Data



Daa flow as an Event Driven stream

- Save Data

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



Summary - Data flow as an Event Driven stream

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

Functional Style Code

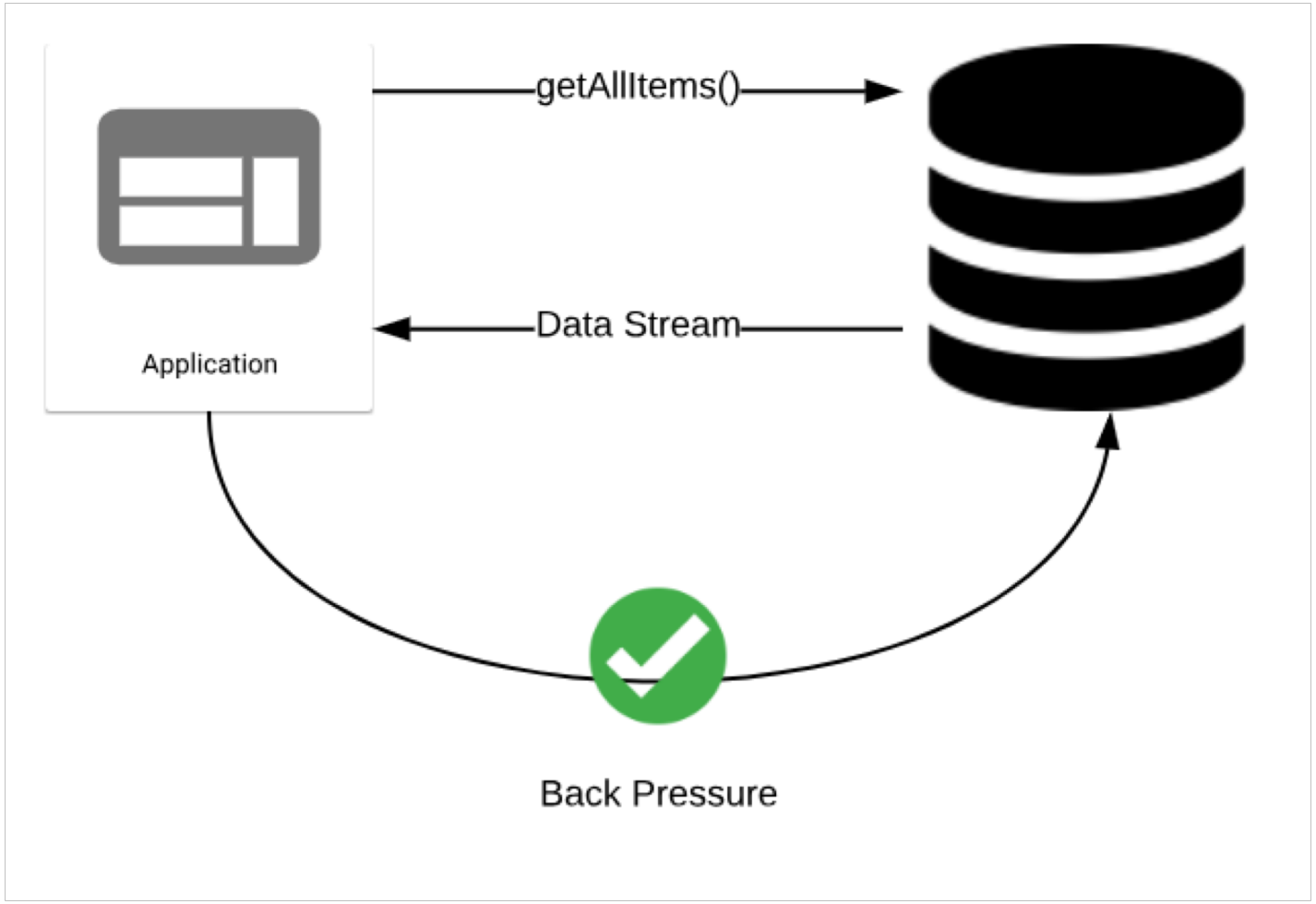
Functional Style Code

- Similar to **Streams API**.

- Easy to work with **Lambdas**.

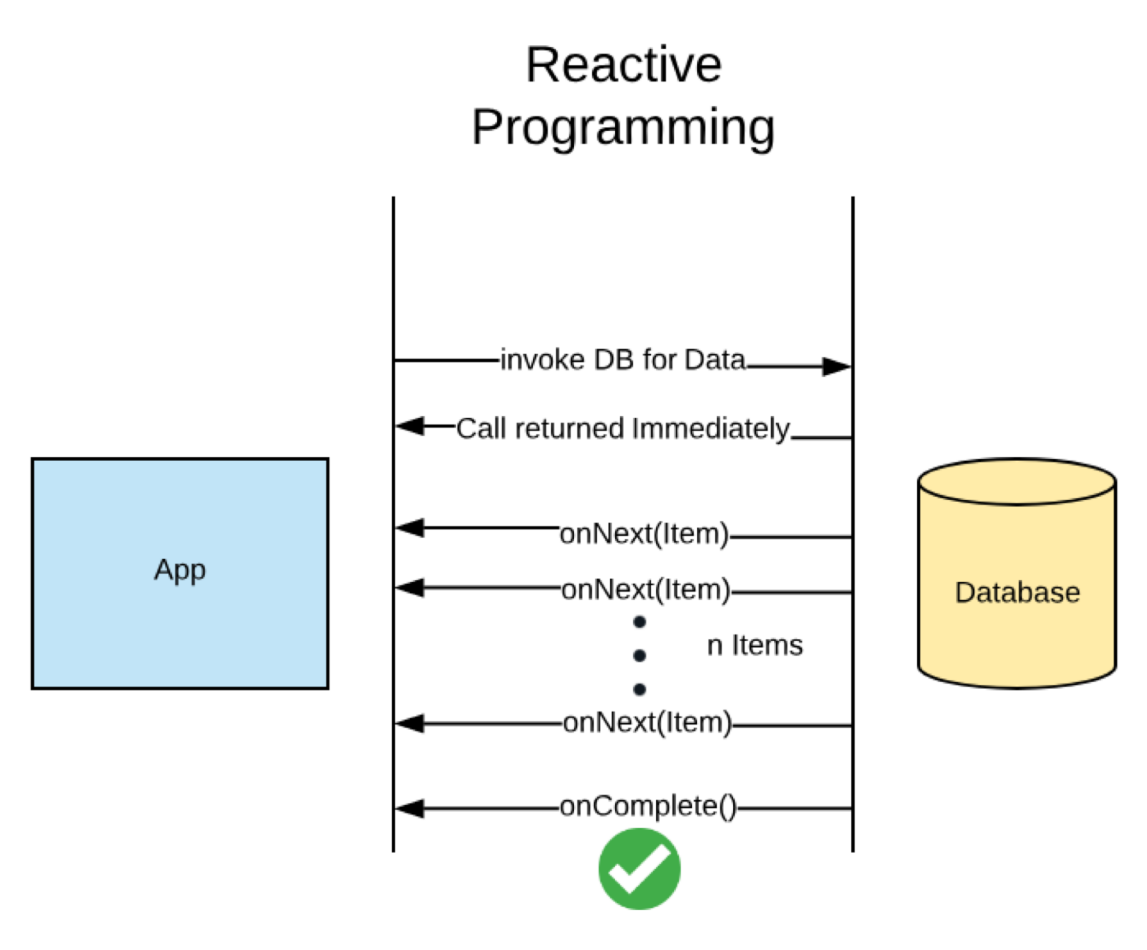
```
@PutMapping("{id}")
public Mono<ResponseEntity<Item>> updateOfficer(@PathVariable(value = "id") String id,
                                                @RequestBody Item item) {
    return itemRepository.findById(id) Mono<Item>
            .flatMap(currentItem -> {
                currentItem.setPrice(item.getPrice());
                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 Streams specification

What is a Reactive Stream ?



What is a Reactive Stream Specification ?

- **Specification** or **Rules** for a Reactive Stream.

- Who created the Specification?
  - Pivotal , Netflix, LightBend and Twitter and etc.

Reactive Stream Specification

- Publisher
- Subscriber
- Subscription
- Processor

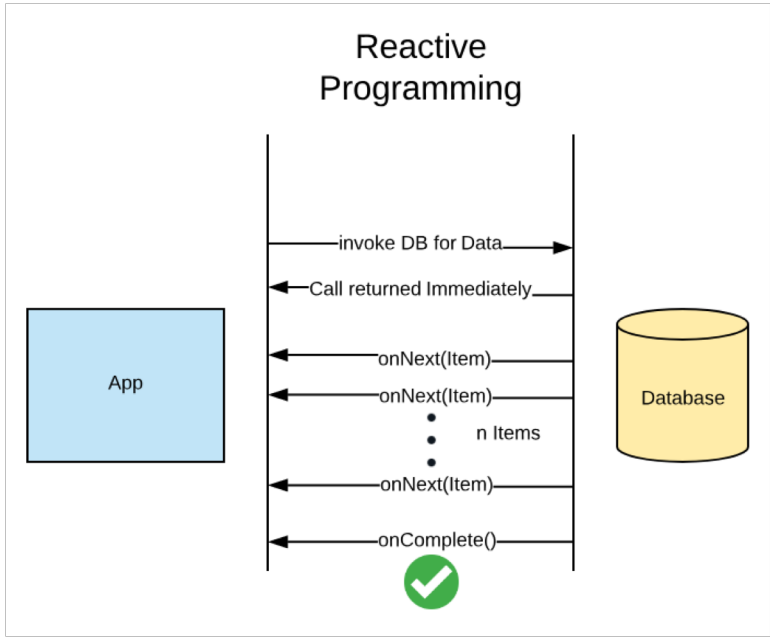
Publisher

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

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

Subscriber:

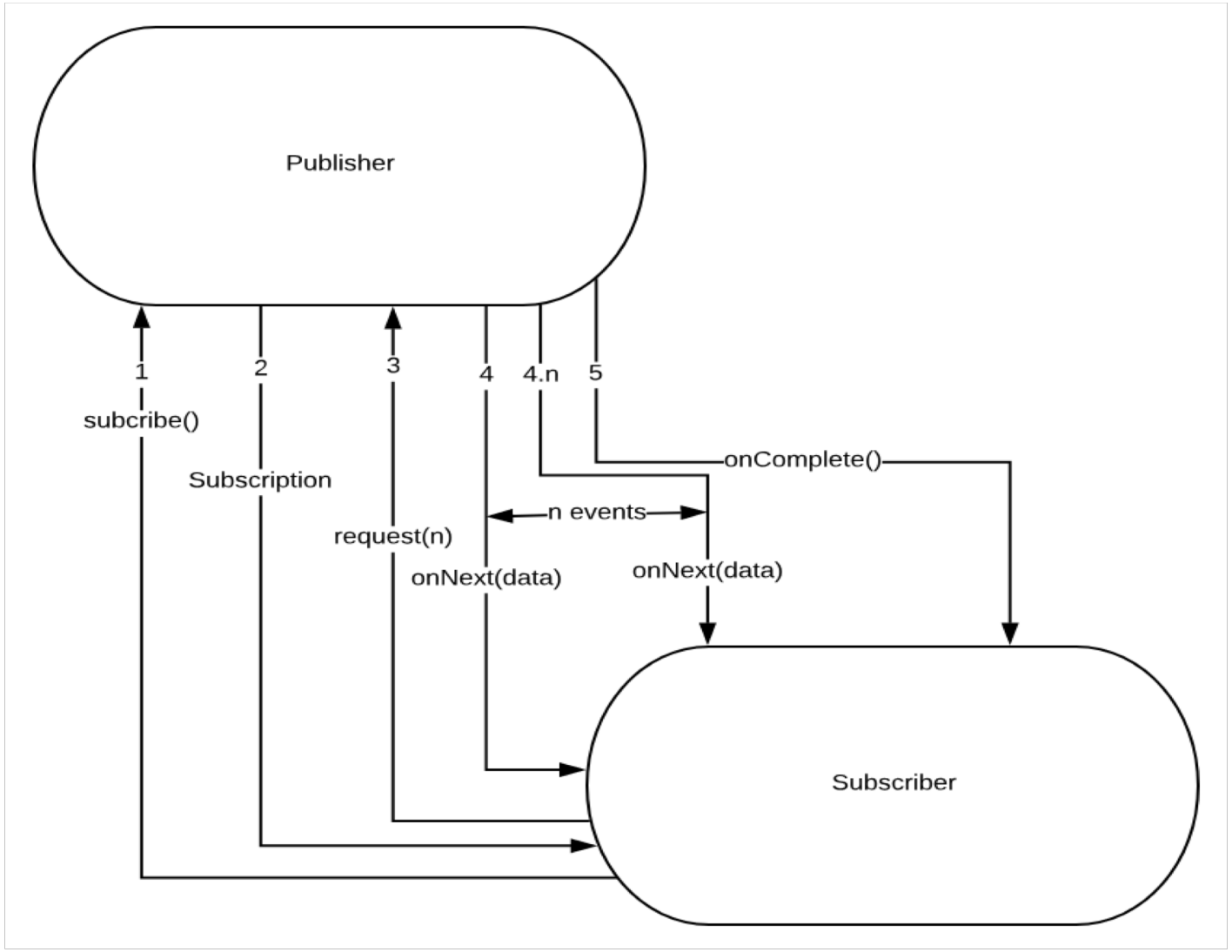
```
public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```



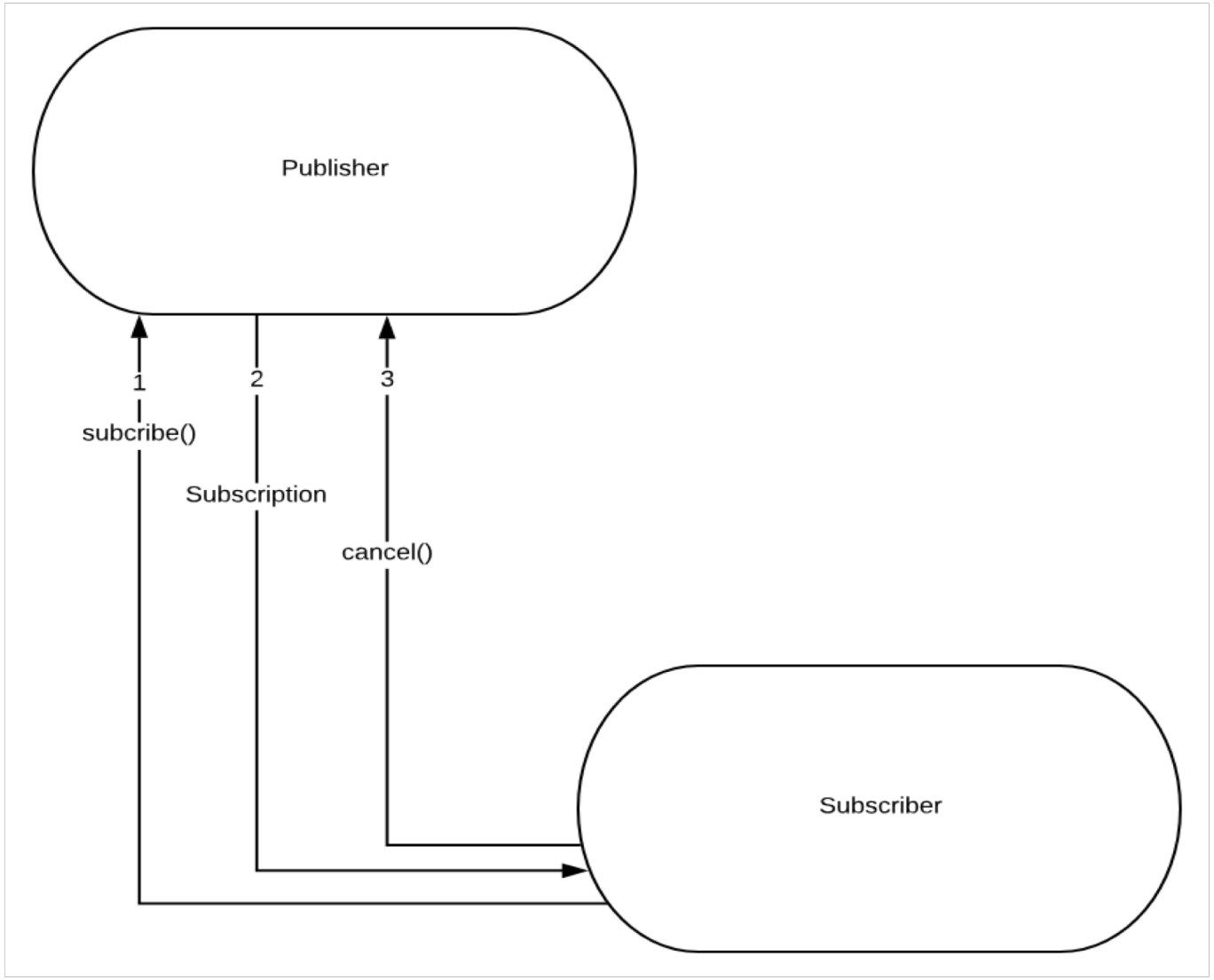
Subscription

```
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> {
}
```

Reactive Libraries

What is a Reactive Library ?

- Implementation of **Reactive Streams Specification.**
  - Publisher
  - Subscriber
  - Subscription
  - Processor

Reactive Libraries:

- RxJava
- Reactor
- **Flow** Class - JDK 9

Reactor or Project Reactor

- Built and maintained by **Pivotal**.

- Recommended Library to work with **Spring Boot.**

Project Reactor

Project Reactor

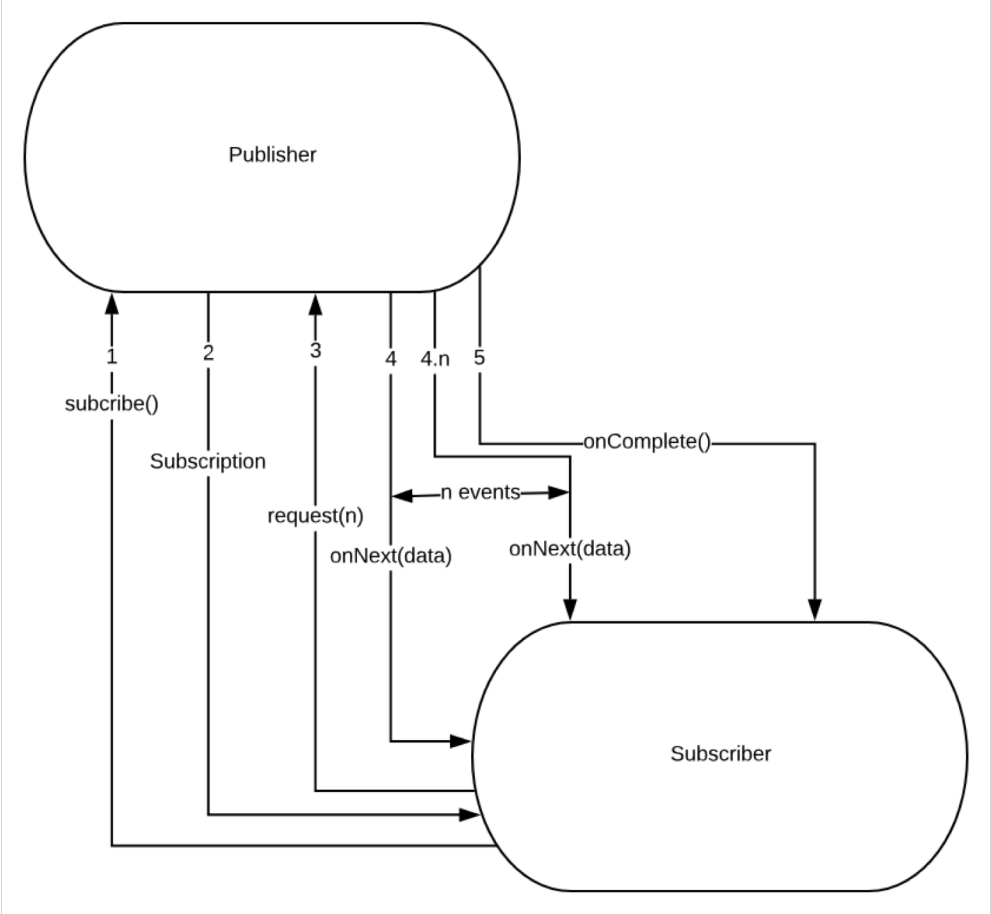
- reactor-core
- reactor-test
- reactor-netty

reactor-core

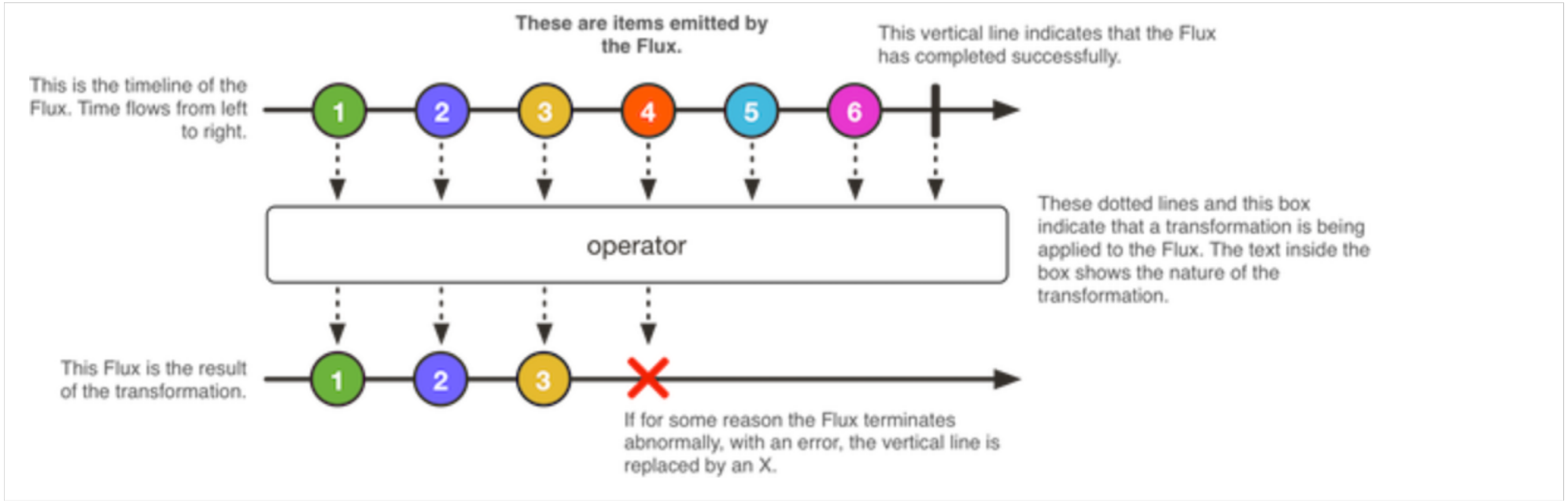
- Core library for Project Reactor.
- Implementation of **Reactive Streams Specification**.
- Java 8 (Minimum)

reactor-core

- **Flux** and **Mono**
- Reactive Types of project reactor.
- **Flux –** Represents 0 to N elements
- **Mono –** Represents 0 to 1 element.



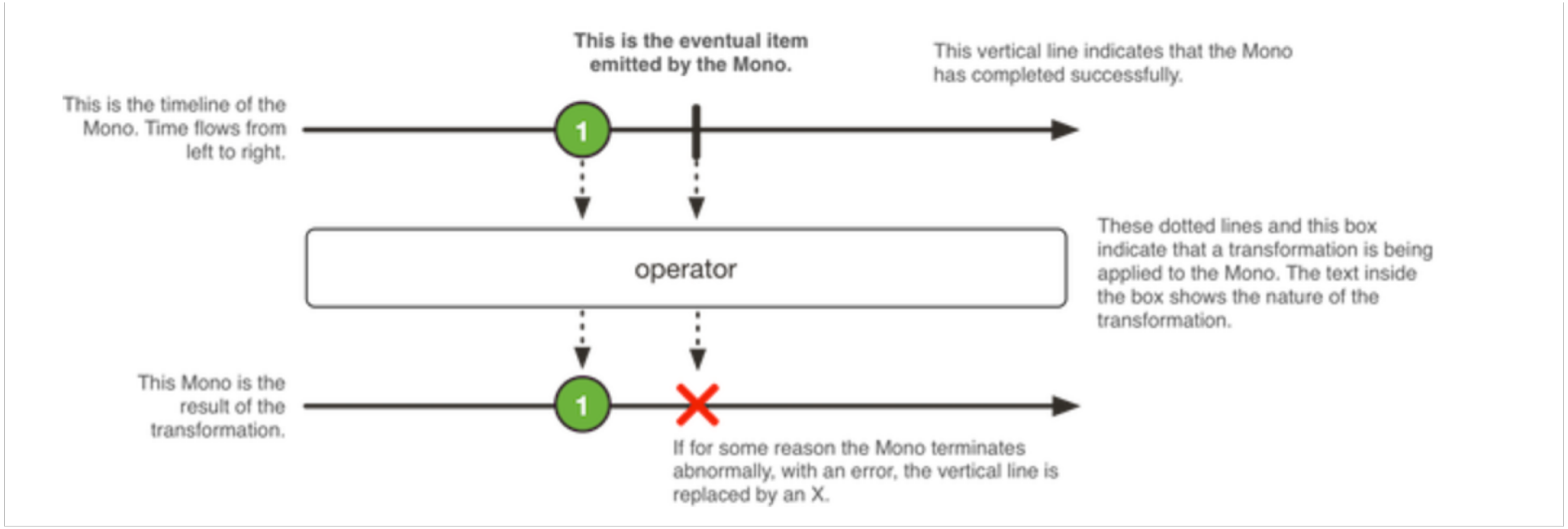
Flux – 0 to N elements



Flux – 0 to N elements

```
Flux.just("Spring", "Spring Boot", "Reactive Spring Boot")
        .map(s -> s.concat("flux"))
.subscribe(System.out::println);
```

Mono – 0 to 1 elements

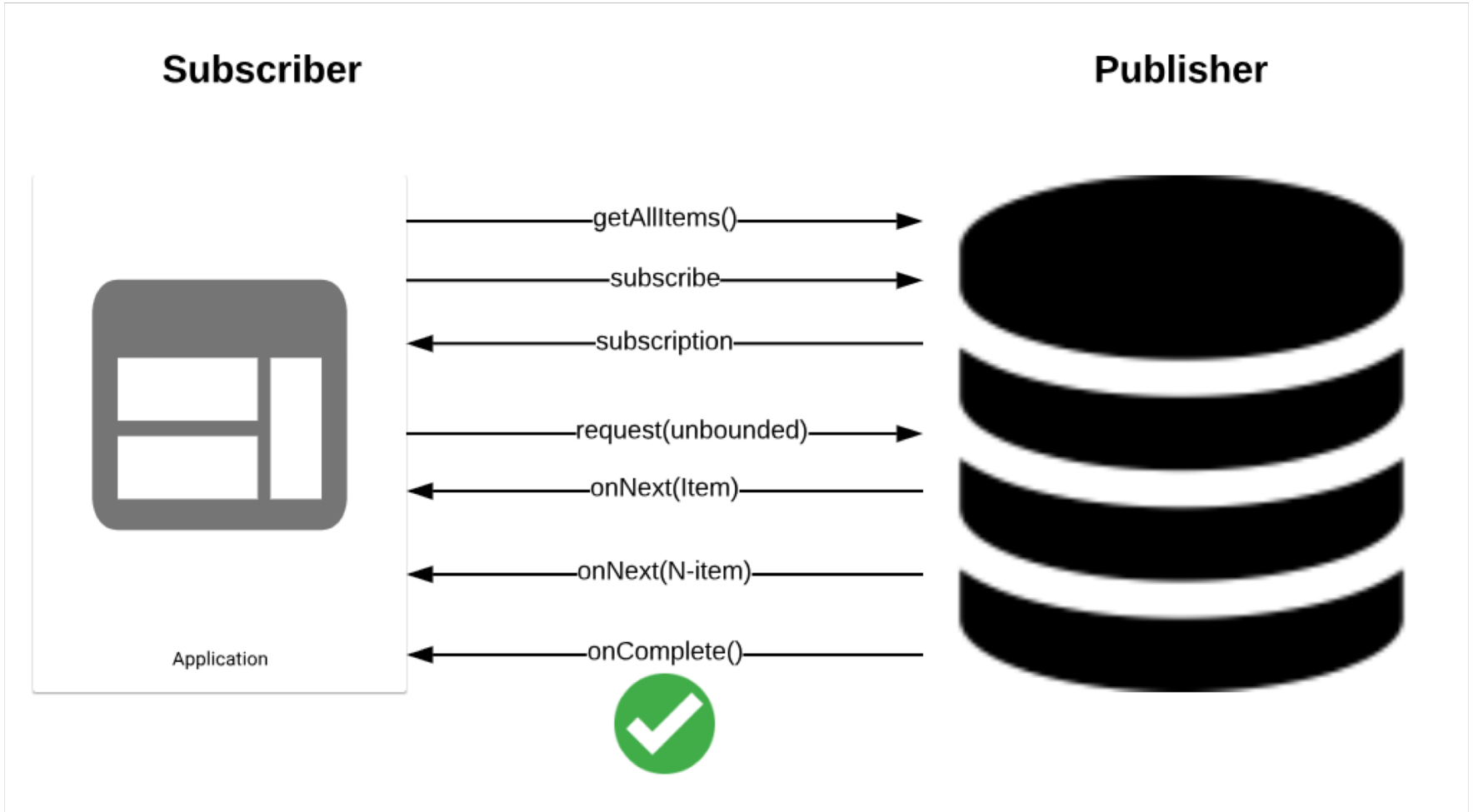


Mono – 0 to 1 elements

```
Mono.just("Spring")
        .map(s -> s.concat("flux"))
        .subscribe(System.out::println);
```

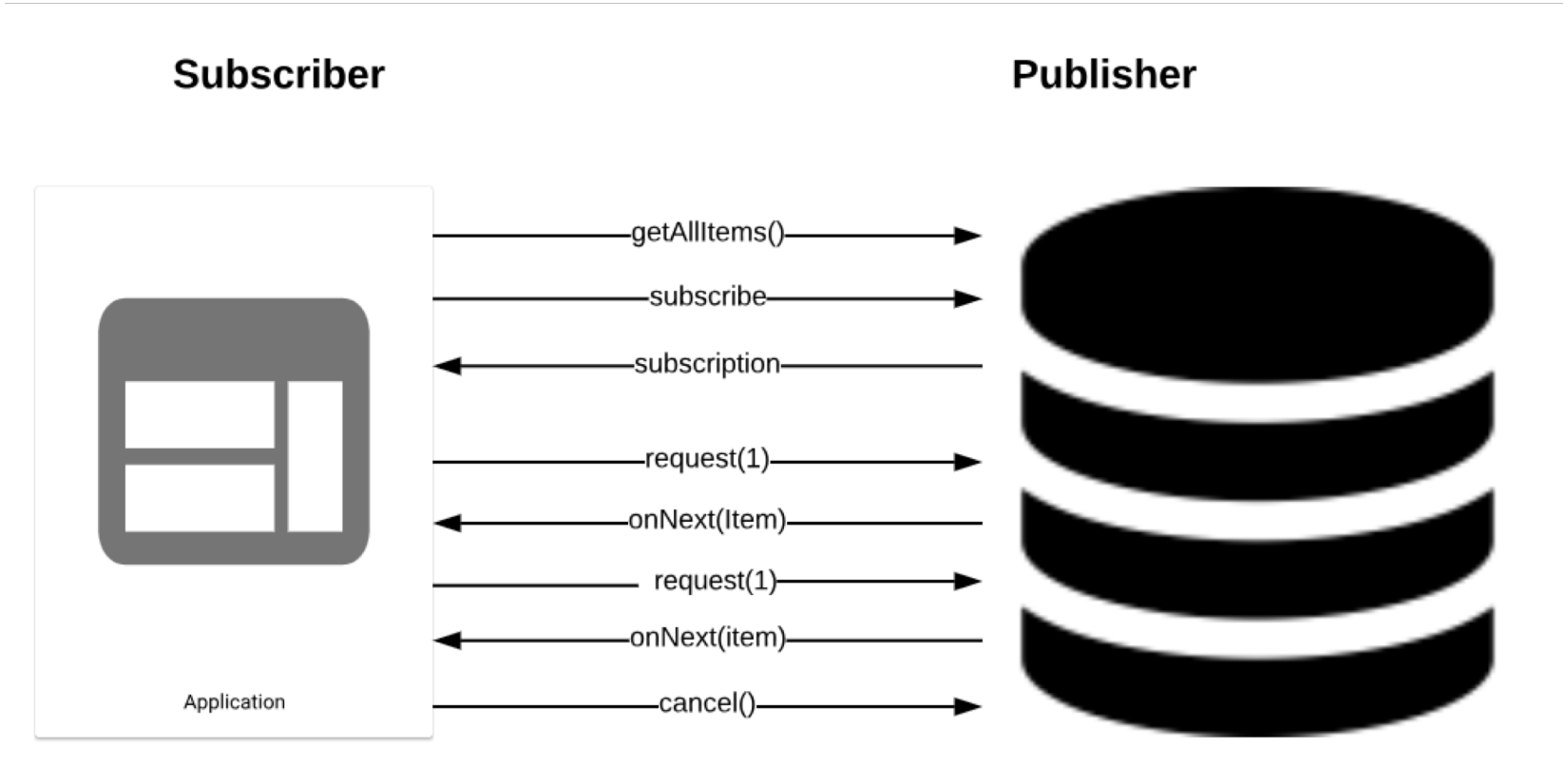
Back Pressure on Reactive Data Streams

Default Data Flow – Project Reactor



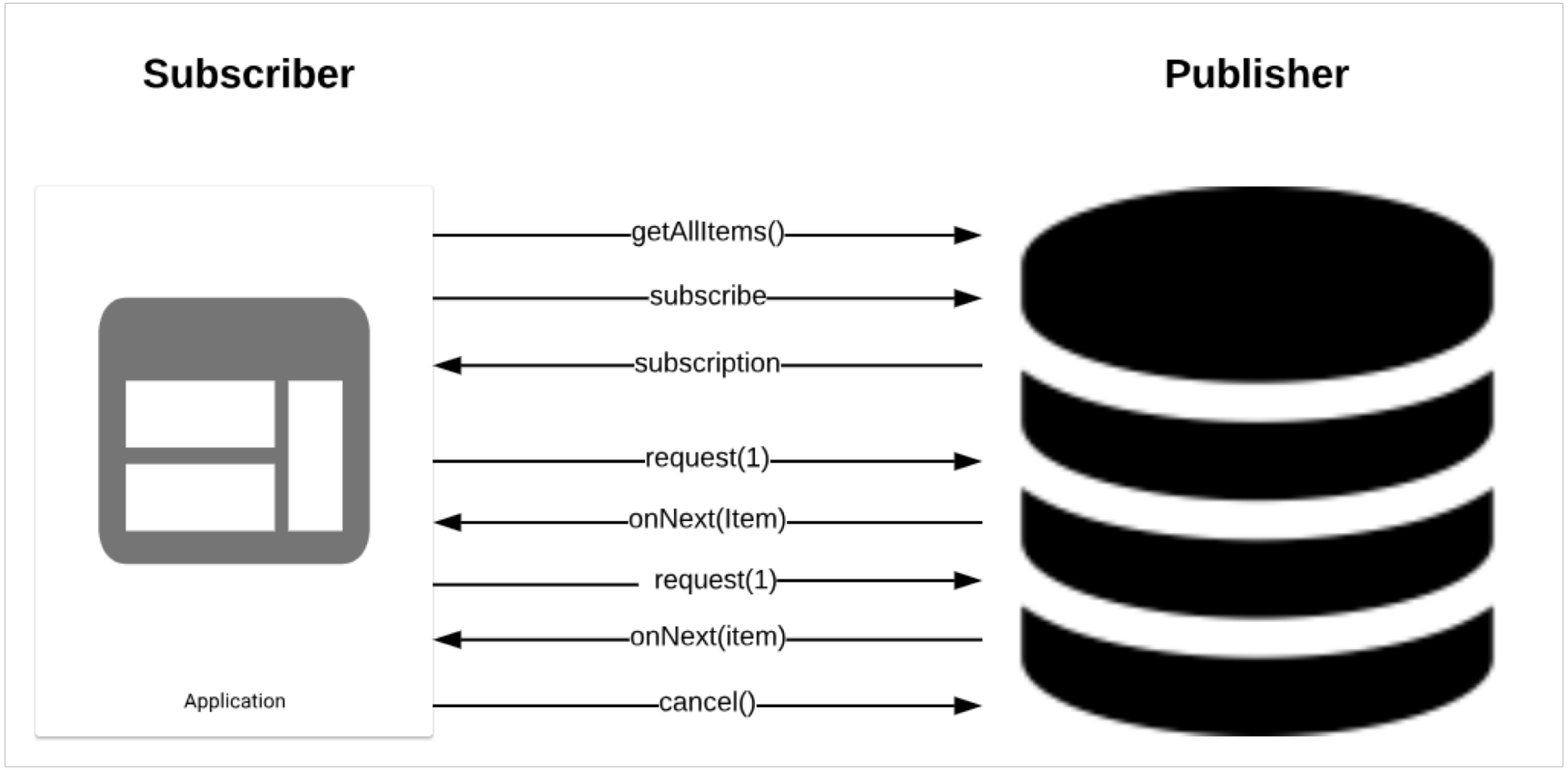
What is Backpressure?

- Subscriber controls the data flow from the Publisher.



Project Reactor Communication Model

Project Reactor Communication Model



Introduction to Spring Boot 2

Spring WebFlux

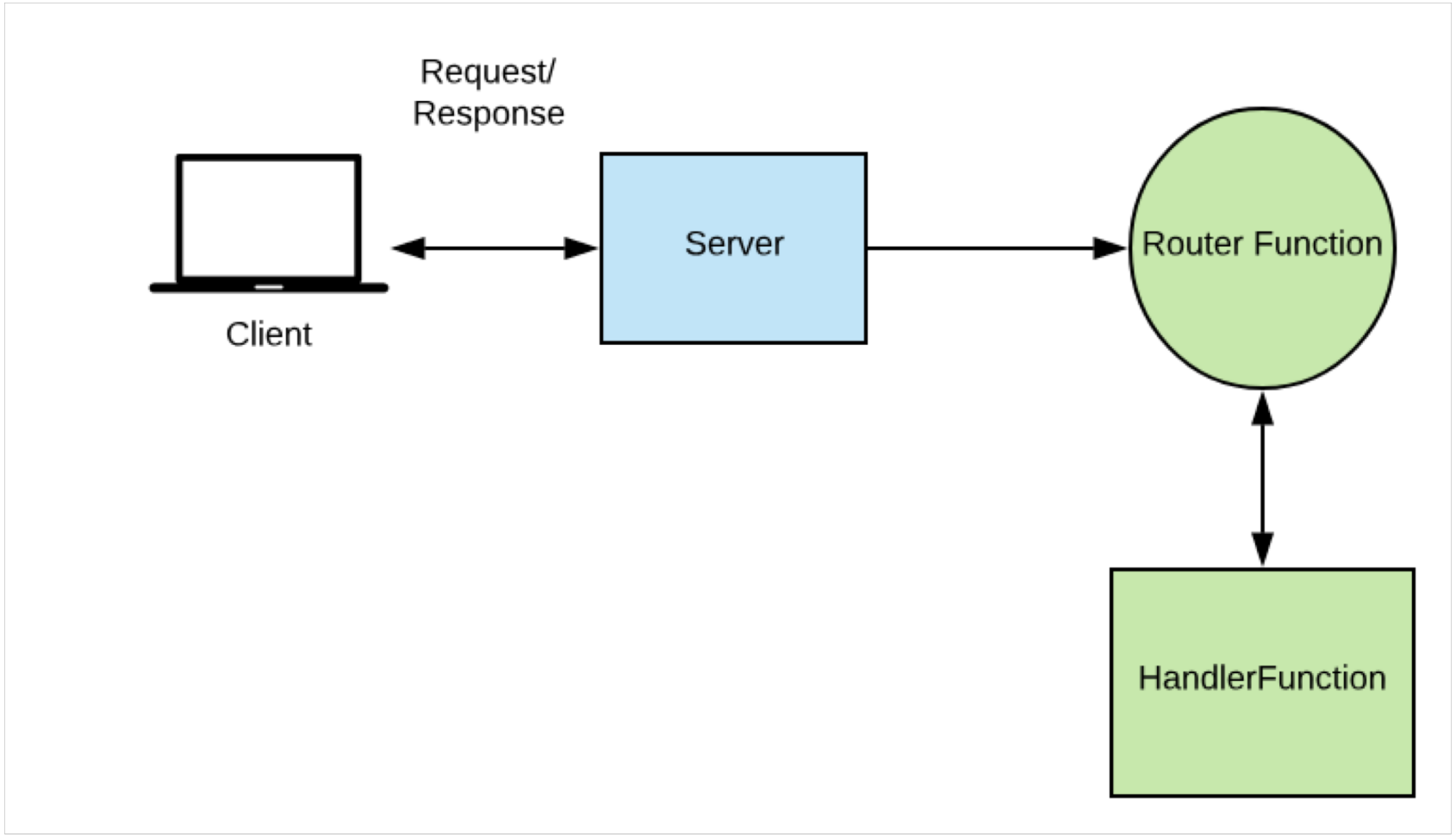
- Supports two patterns for building nonblocking **RESTFUL** APIs.
  - Annotated Controllers
  - Functional Web

Spring WebFlux - Functional Web:

- Use **Functions** to route the request and response.

- **RouterFunction** and **HandlerFunction.**

Spring WebFlux - Functional Web:



RouterFunction:

- Use to route the incoming request.

- Similar to the functionality of **@RequestMapping** annotation.

```
@GetMapping("/flux")
public Flux<Integer> returnFlux(){

    return Flux.just(1,2,3,4)
            .delayElements(Duration.ofSeconds(1))
            .log();

}
```

Handler Function:

- Handles the request and response.

- Similar to the body of the **@RequestMapping** annotation.

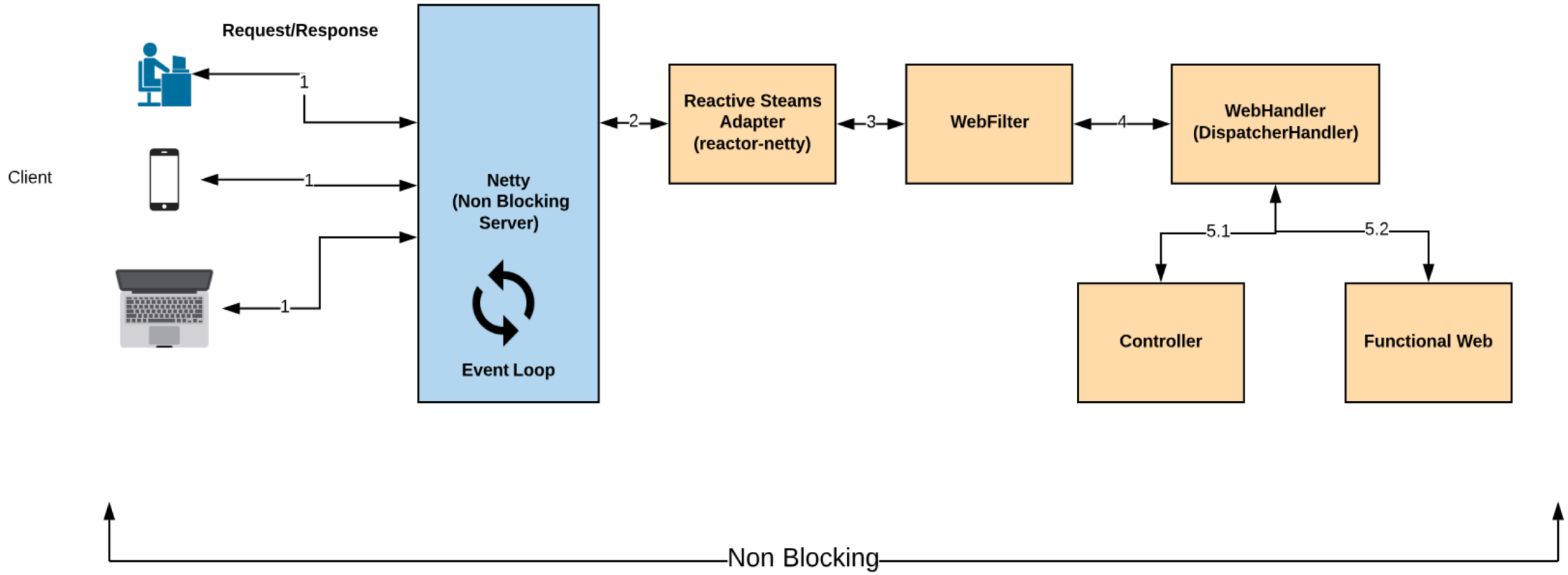
```
@GetMapping("/flux")
public Flux<Integer> returnFlux(){

    return Flux.just(1,2,3,4)
            .delayElements(Duration.ofSeconds(1))
            .log();

}
```

Handler Function:

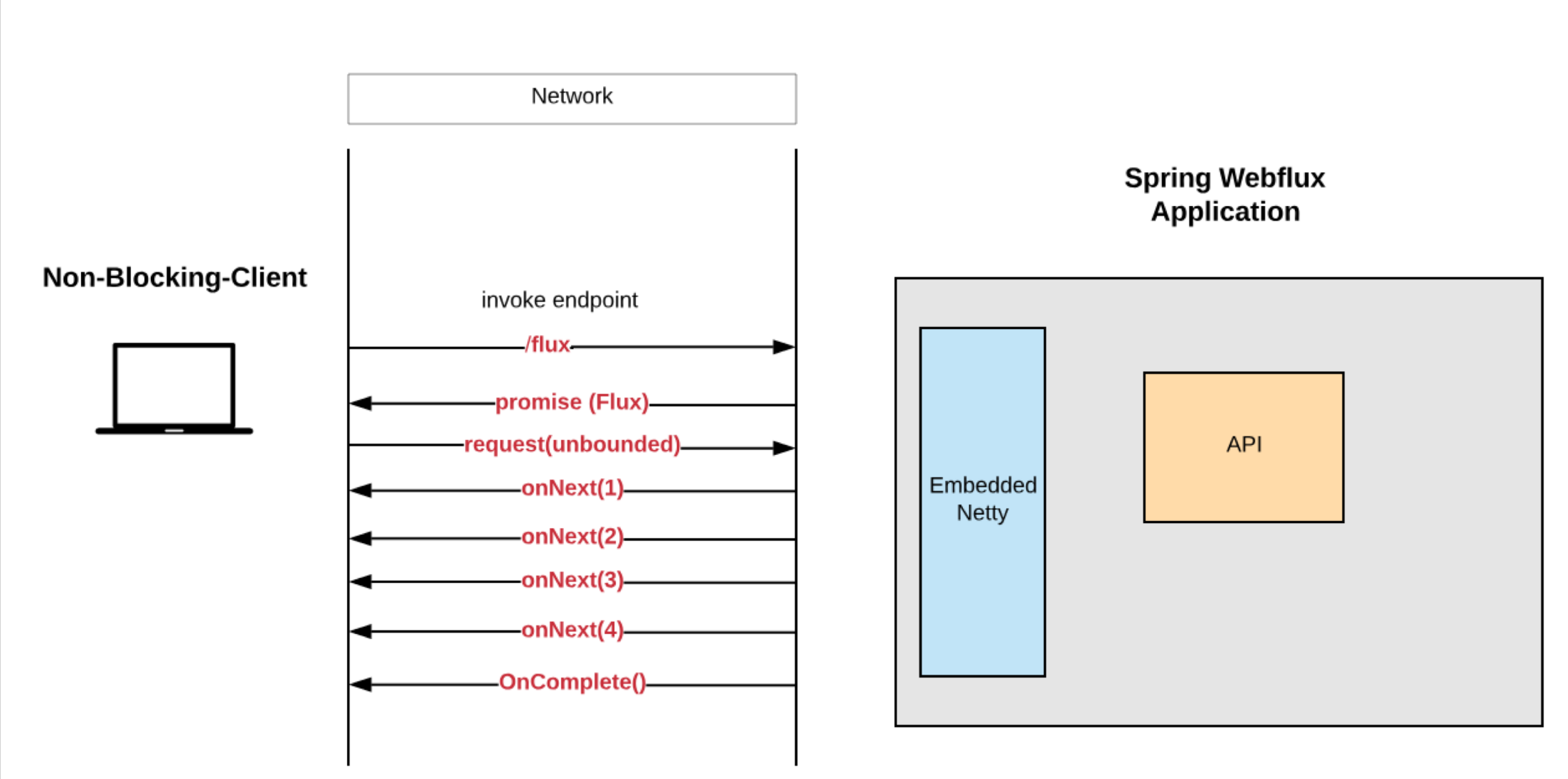
- ServerRequest and ServerResponse.
- **ServerRequest** represents the **HttpRequest.**
- **ServerResponse** represents the **HttpResponse**.

Spring Webflux – Behind the Scenes

Spring WebFlux - Non Blocking Request/Response



Spring WebFlux – Request/Response



Flux Endpoint

```
@GetMapping("/flux")
public Flux<Integer> returnFlux(){

    return Flux.just(1,2,3,4)
            .delayElements(Duration.ofSeconds(1))
            .log();

}
```

Netty

Netty

- Netty is an asynchronous event-driven **network** application framework for rapid development of maintainable **high performance** protocol servers & clients.

- Netty is built on top of **Java**.

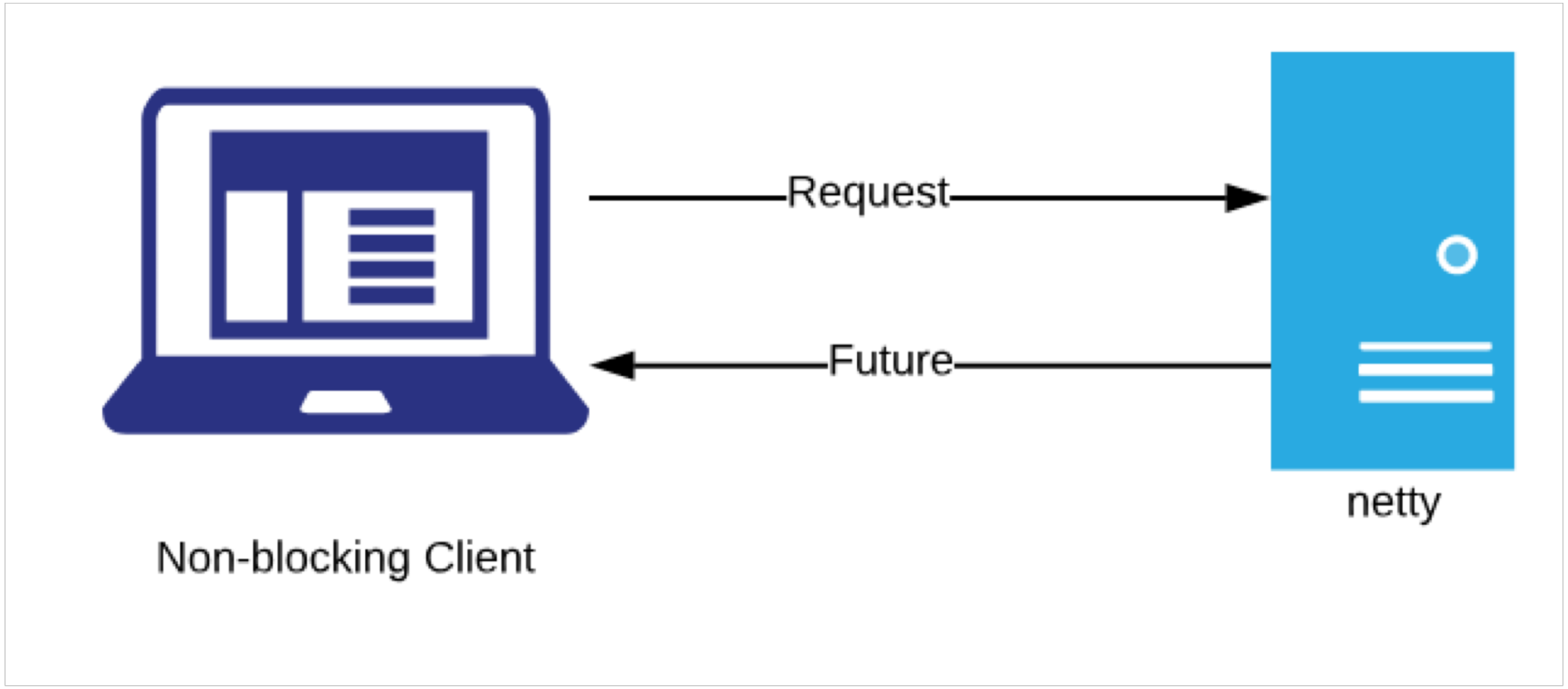
Who uses Netty ?

- Facebook
- Google
- Instagram
- Twitter
- Apache Cassandra and etc.,

Protocols Supported

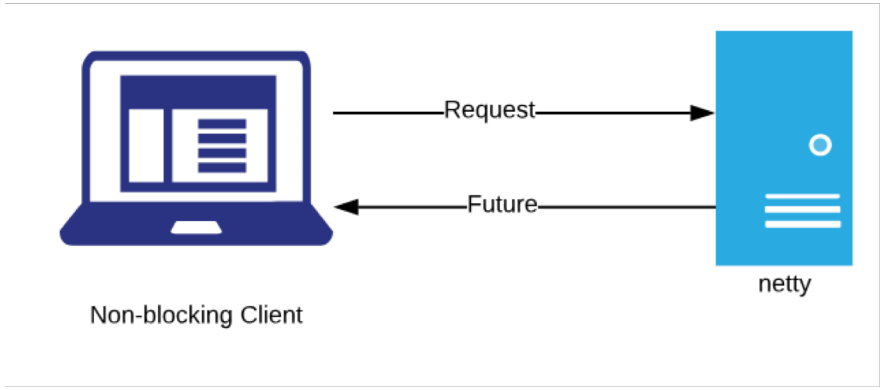
- FTP
- HTTP
- SMTP
- WebSocket and etc.,

Netty – Asynchronous

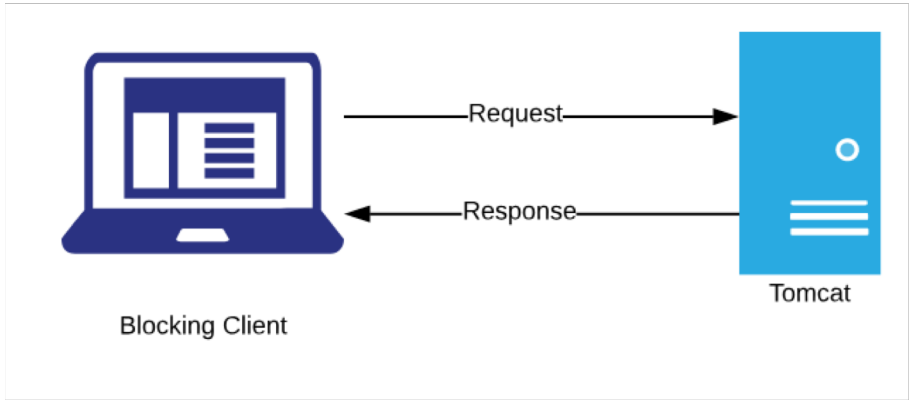


Spring WebFlux vs Spring MVC

Spring Webflux + Netty



Spring MVC + Tomcat



Netty – Asynchronous

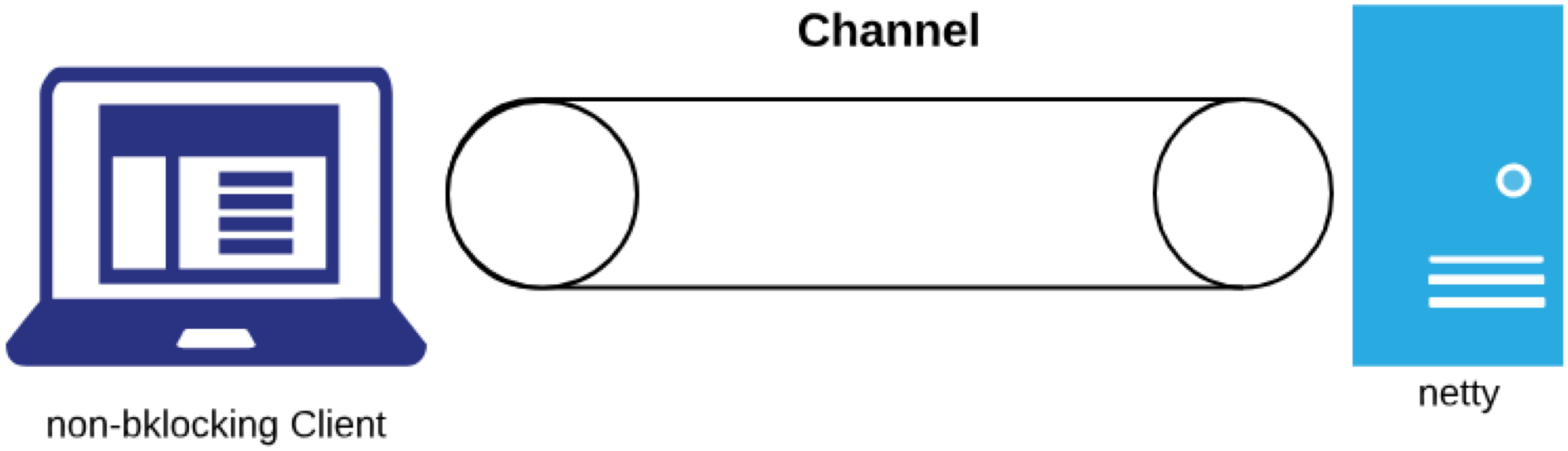
- Being Asynchronous – Frees us from the blocking calls.
- Handles large number of connections.

Events in Netty

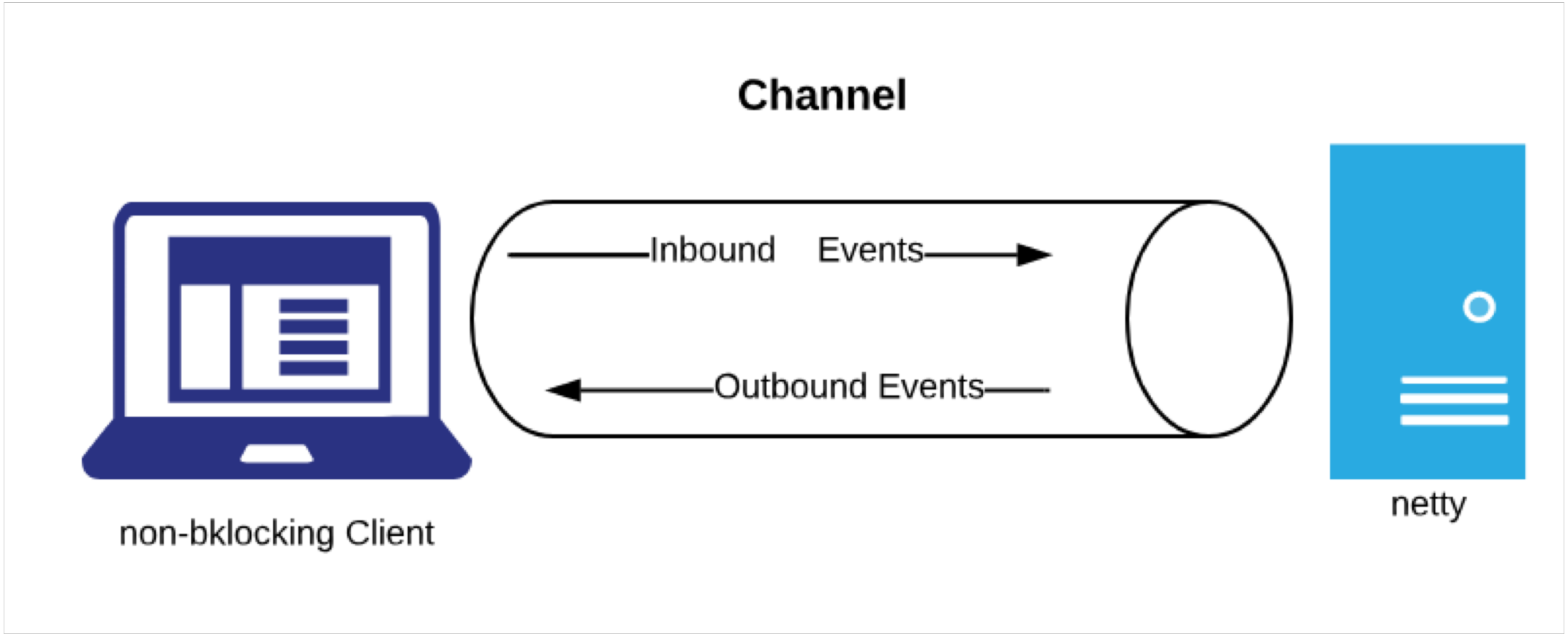
- Client requesting for a new connection is treated as an event.
- Client requesting for data is treated as an event.
- Client posting for data is treated as an event.
- Errors are treated as event.

Netty – Channel

- Channel – Represents the connection between the client and server.

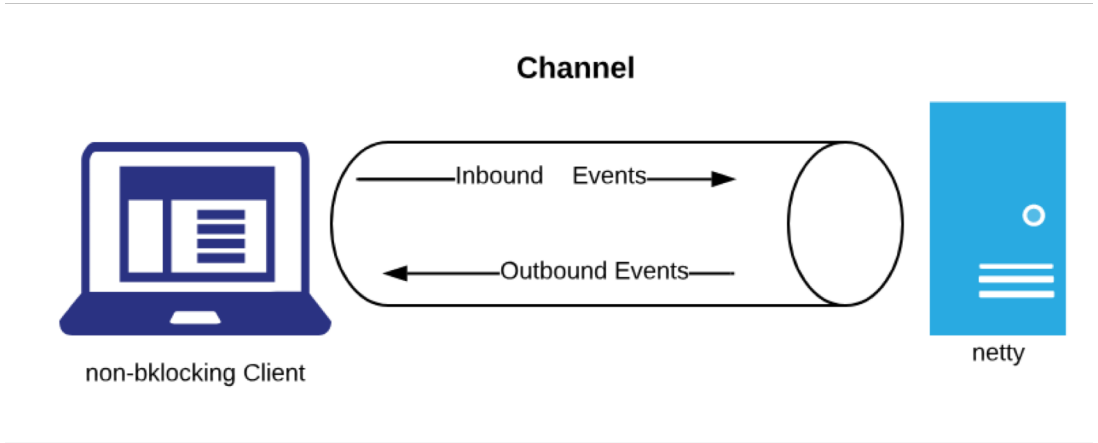


Netty – Channel + Events



Netty – Channel + Events

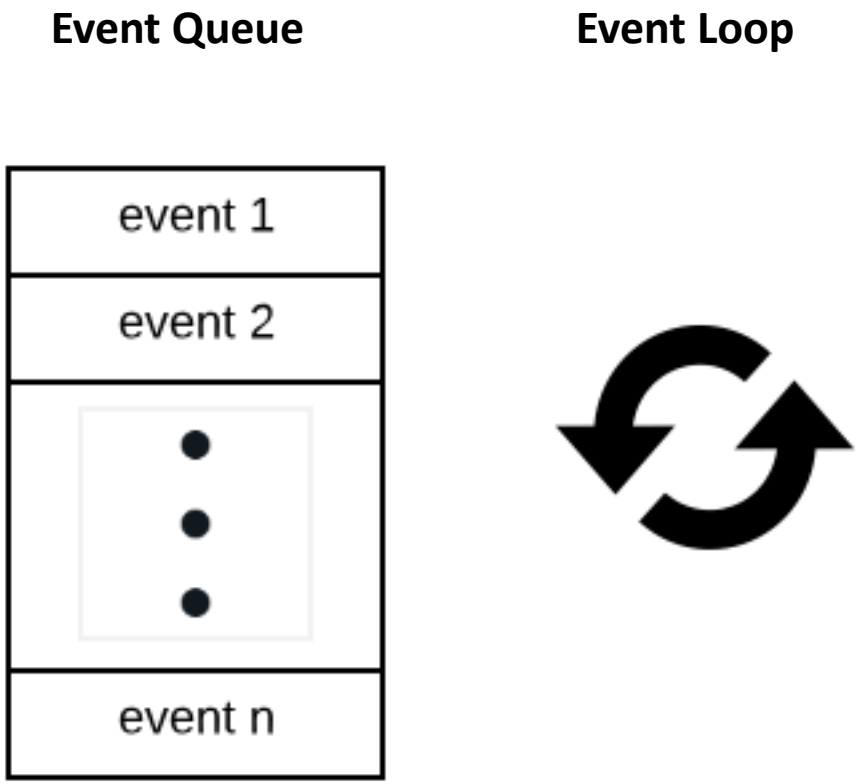
- Inbound events:
  - Requesting for Data
  - Posting Data and etc.,
- Outbound events
  - Opening or Closing a connection.
  - Sending response to the client.



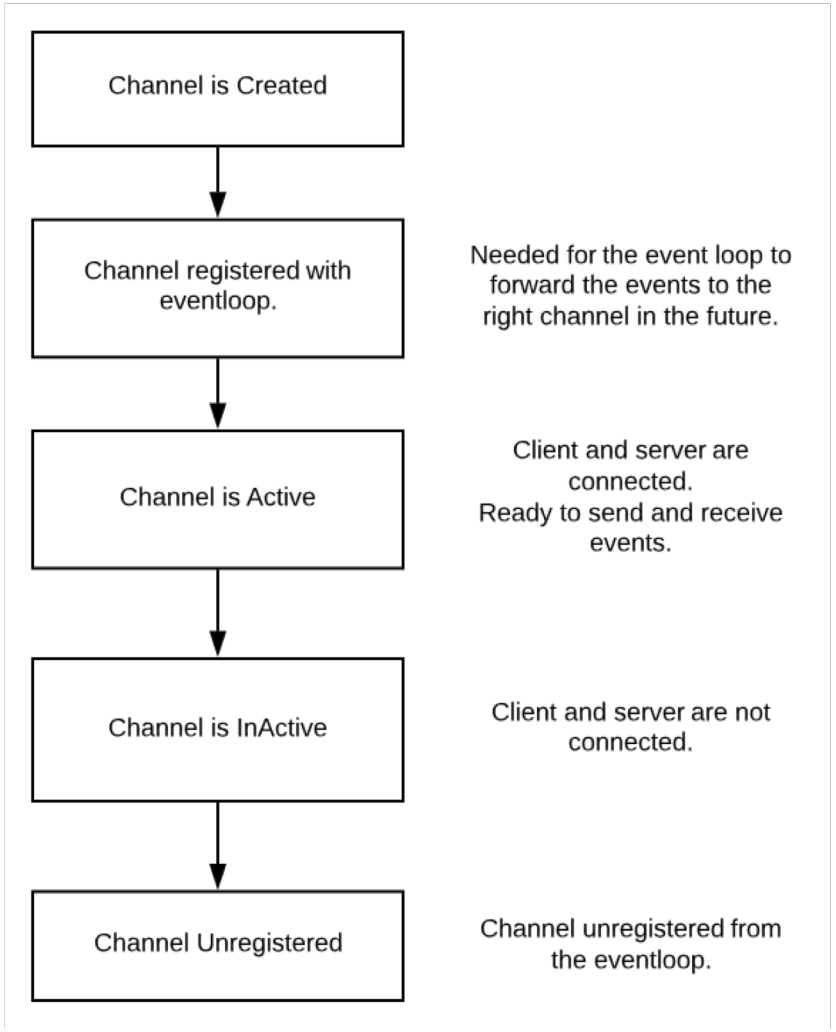
Netty - Event Loop

- Loop the looks for events.

- EventLoop is registered with a **single** dedicated thread.

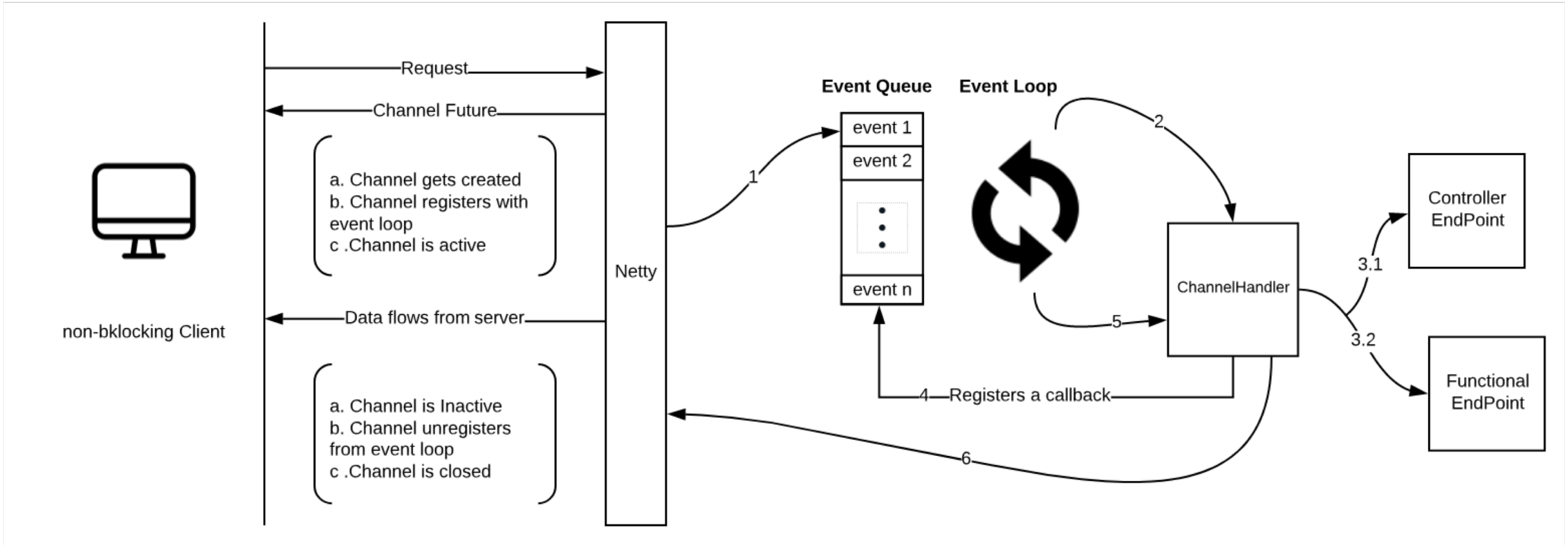


Channel LifeCycle

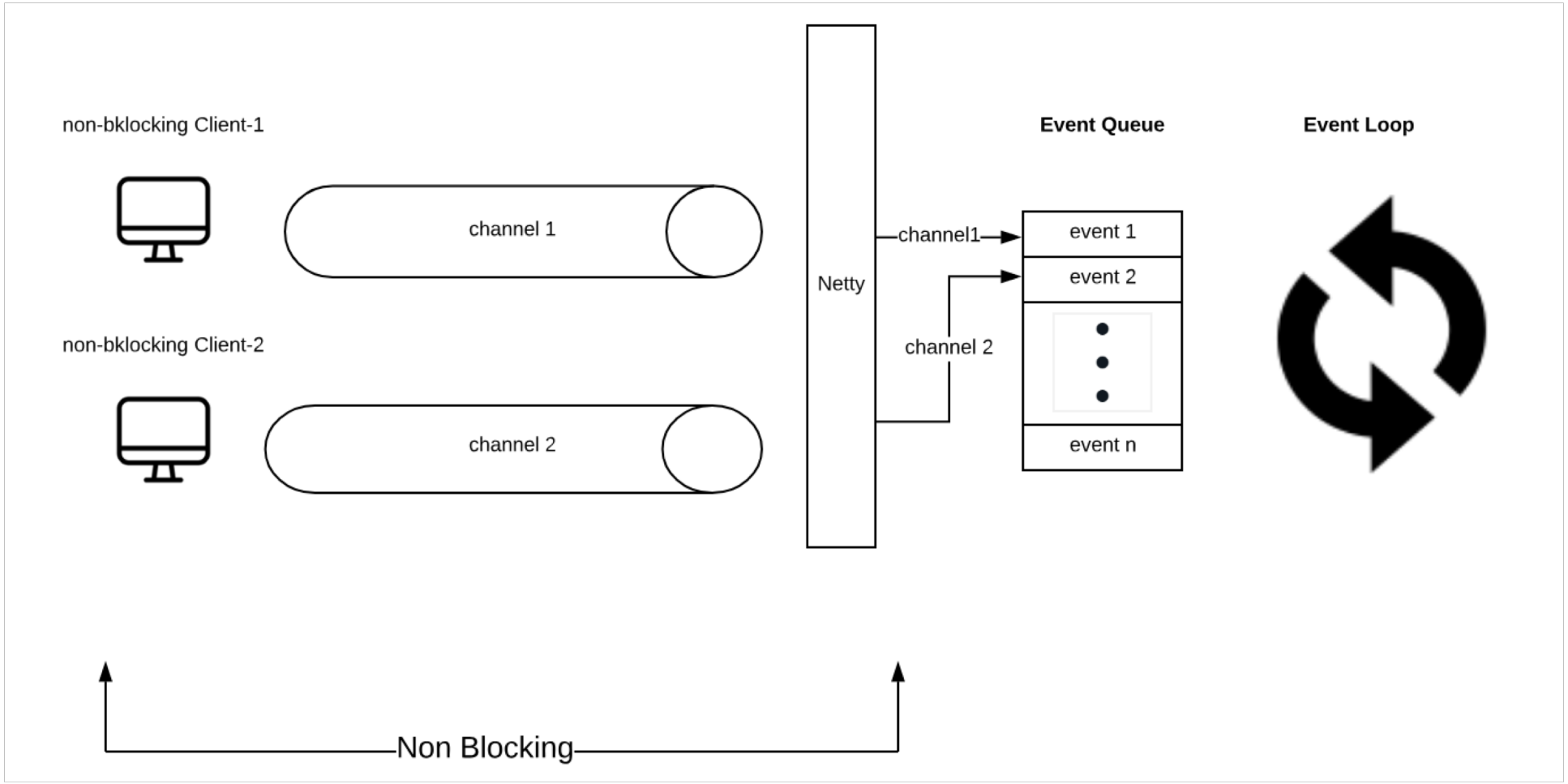


Netty – End to End, Threads & Execution Model

EventLoop + Channel



EventLoop + Multiple Channel



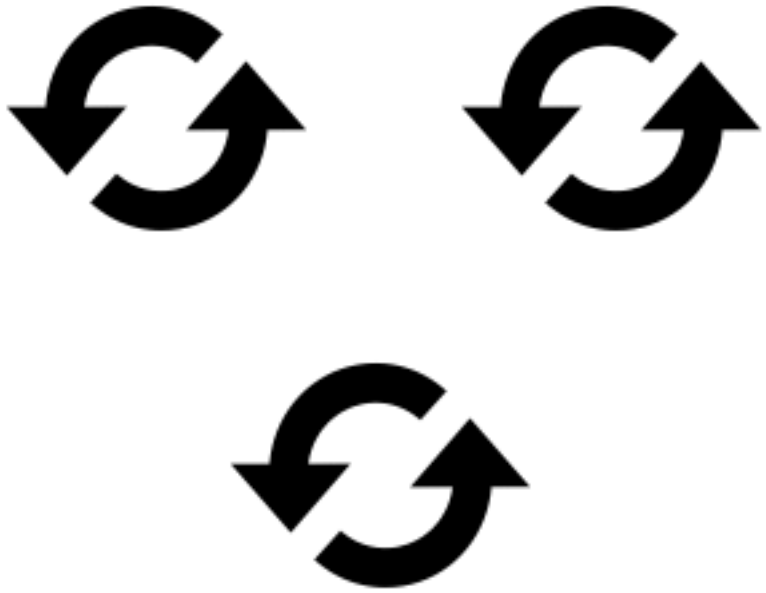
EventLoop

- Is it possible to have more than one event loop ?
  - Yes
- No of EventLoops == 2 X no of processors for the virtual machine.
- How to calculate the number of processors ?

  Runtime.*getRuntime*().availableProcessors()

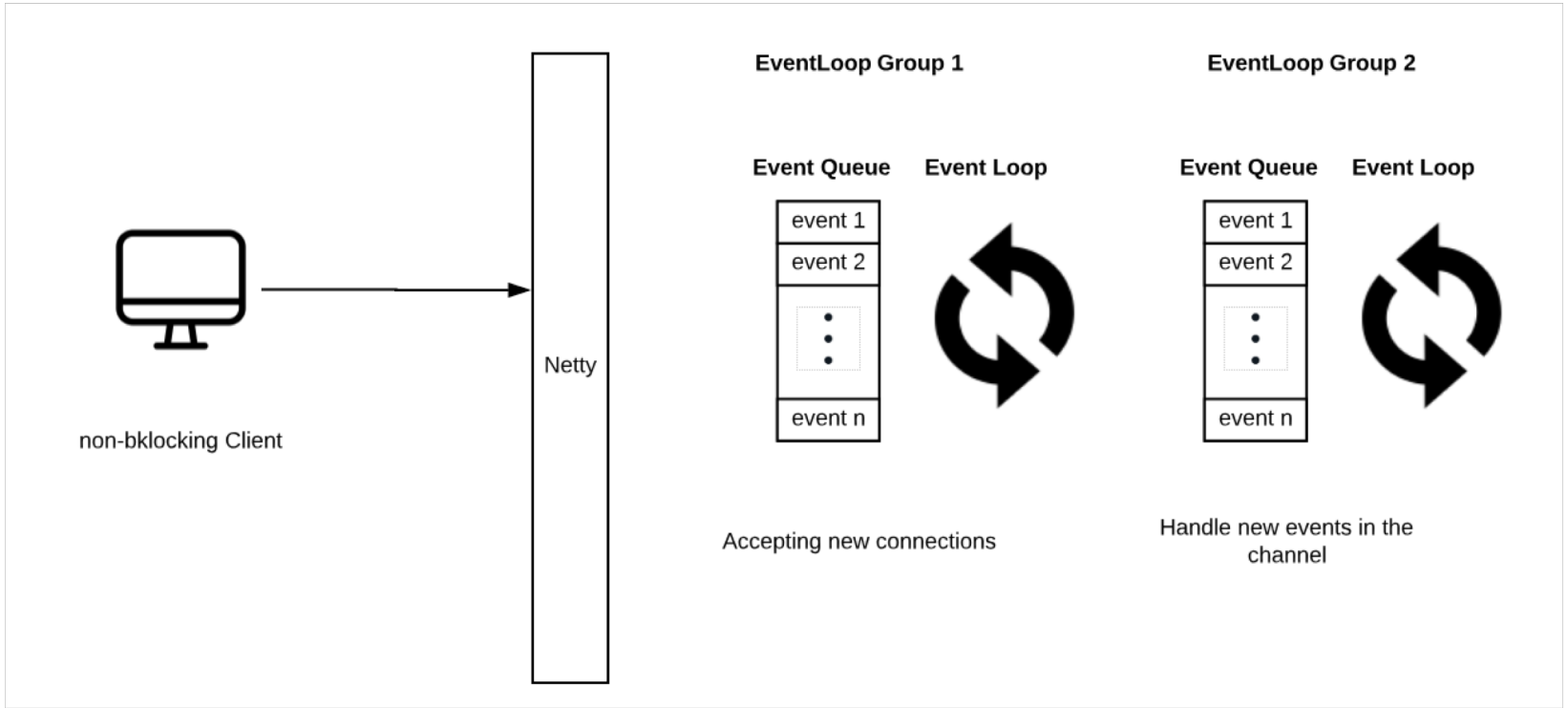
EventLoopGroup

- EventLoops are grouped together to form **EventLoopGroup**.



EventLoopGroup

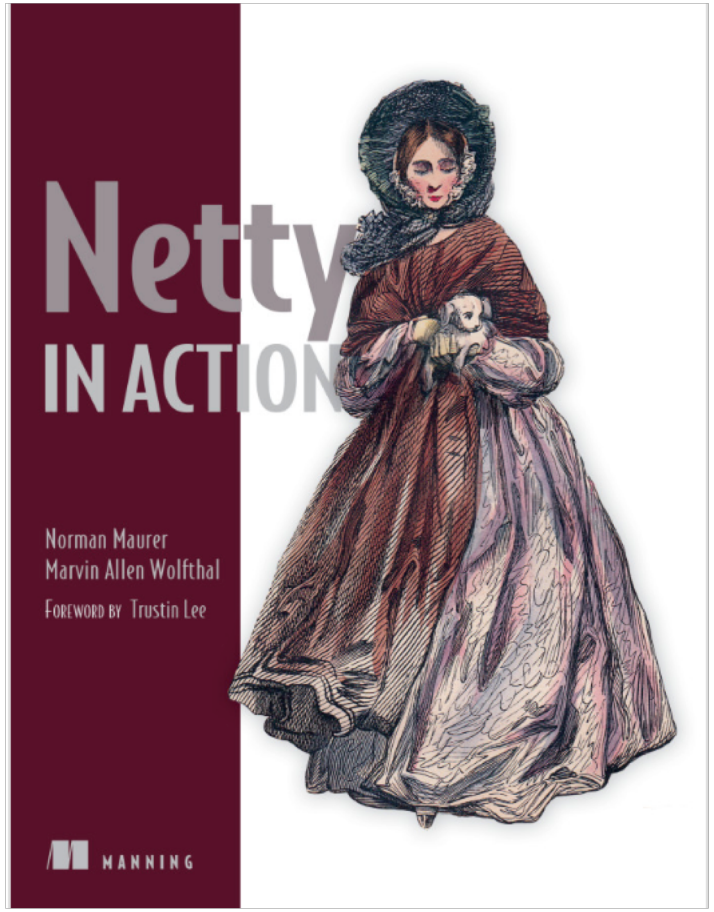
- How many EventLoopGroup are there in Netty?
  - 2 EventLoop Groups.



Threads

- How many threads ?
  - Number of threads = = 2 X no of processors for the virtual machine.

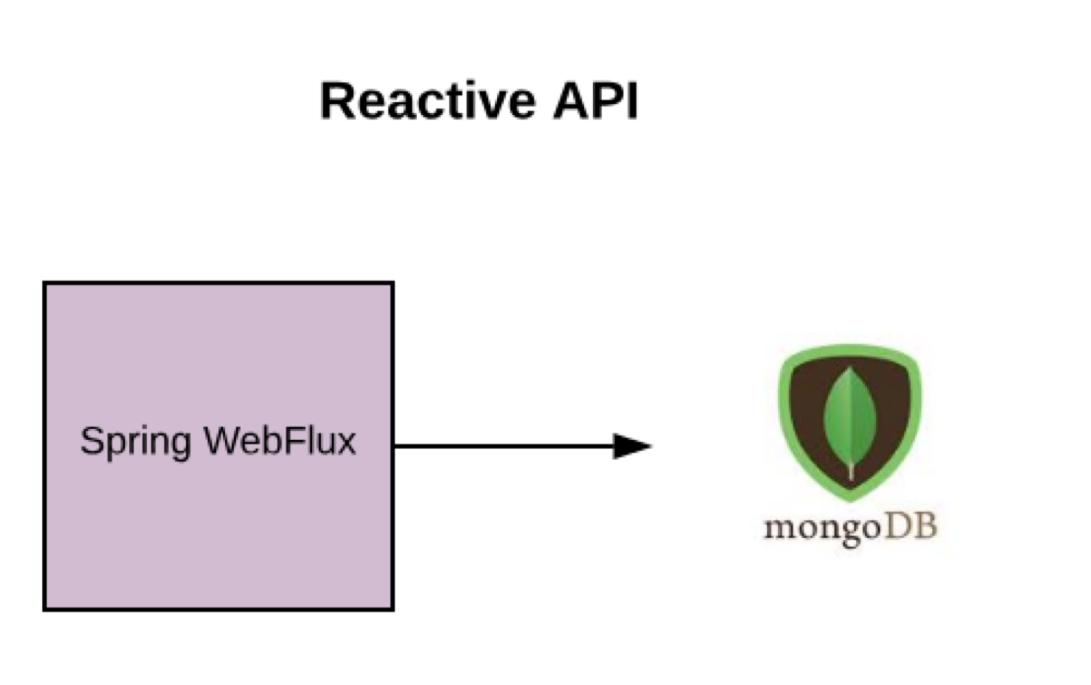
Netty In Action - Book



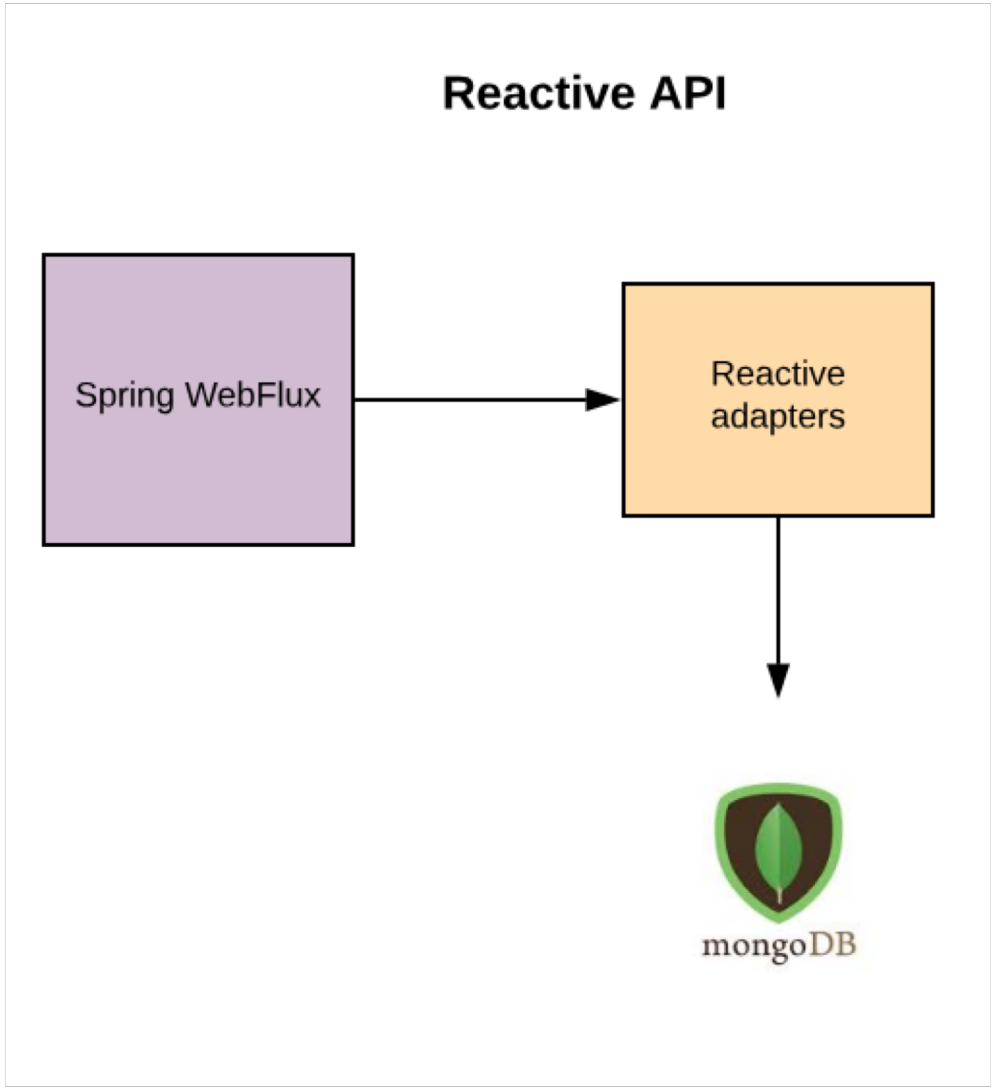
Application Overview

Application Overview

- Reactive API for the retail Industry.
- API to manage Items in the inventory.
- Basically reactive CRUD API.



Application Overview

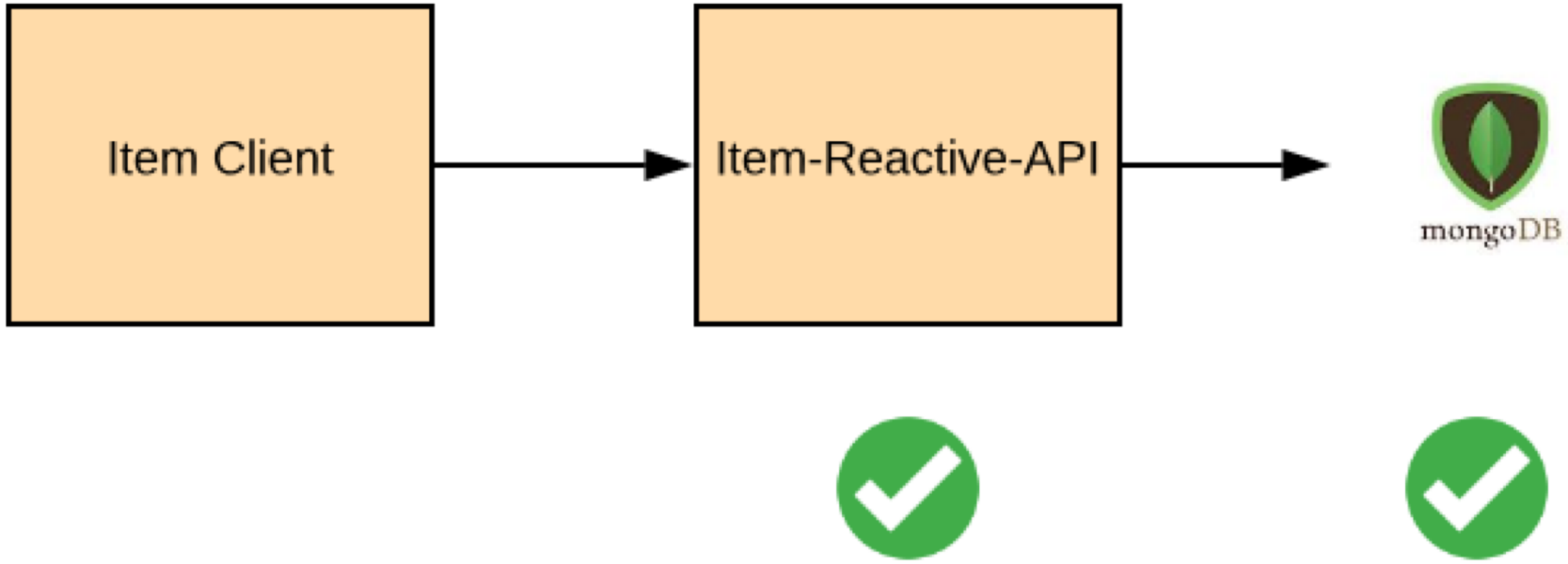


Setup MongoDB

Spring Profiles

WebClient

WebClient

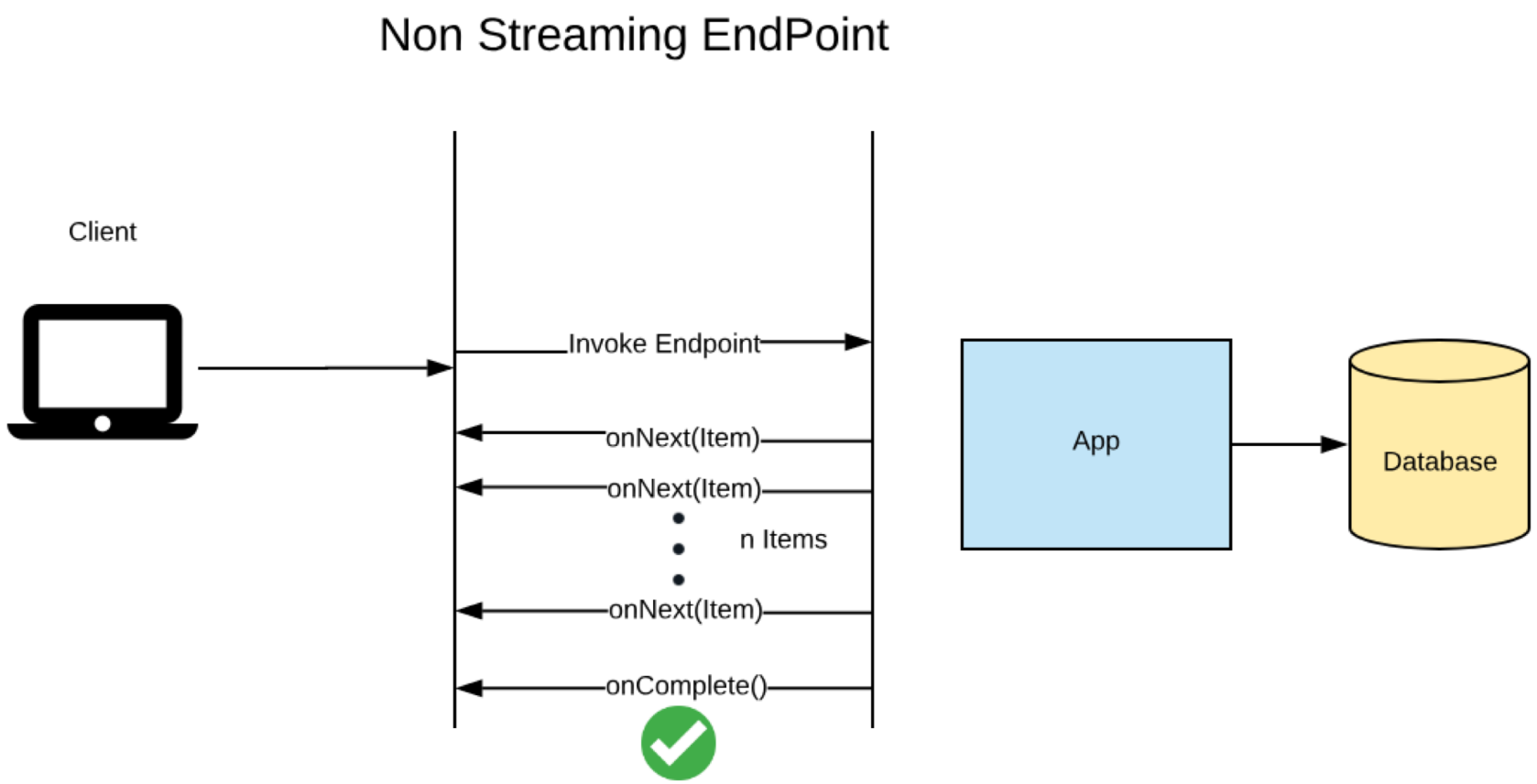


Streaming EndPoint/Server Sent Events(SSE)

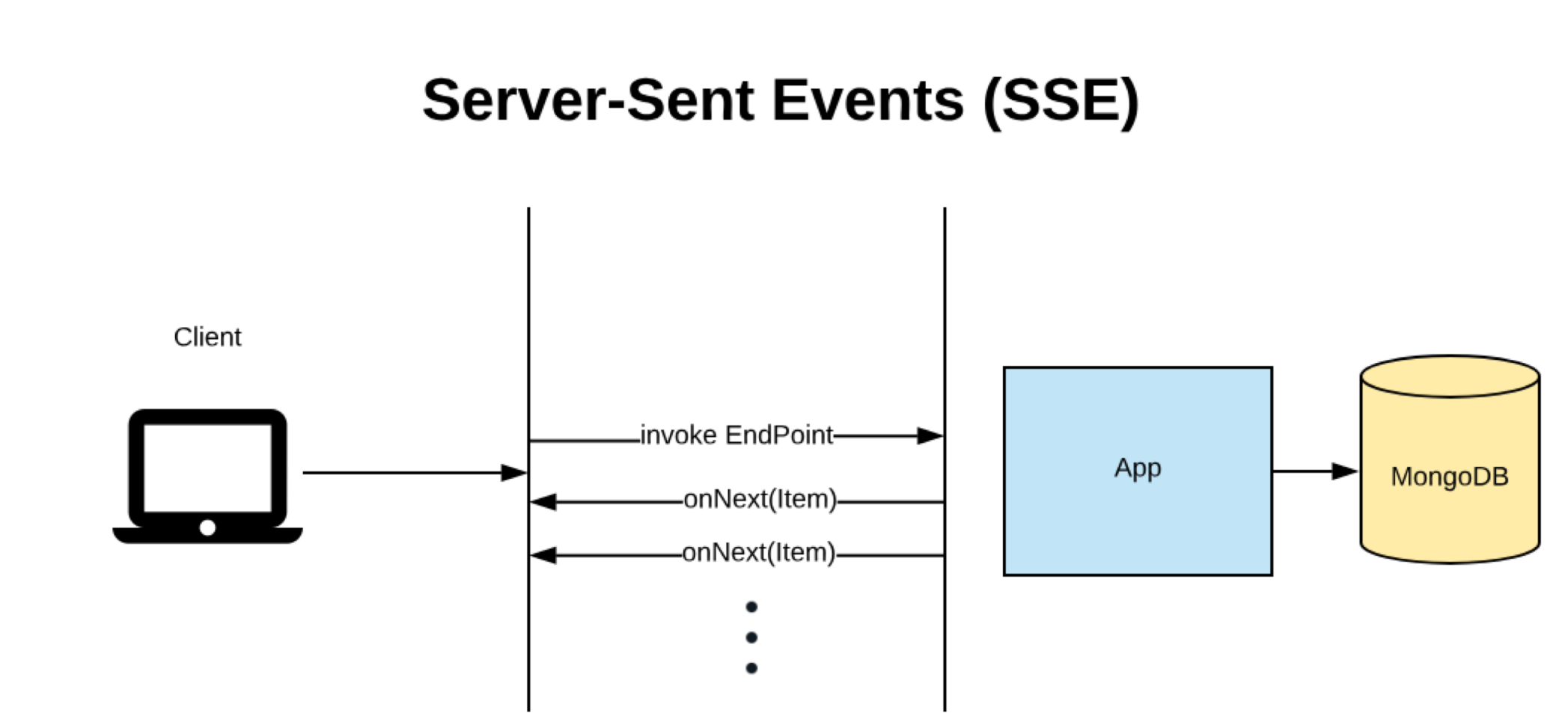
Streaming End Point or SSE?

- It is an endpoint once the connection is made its going to keep pushing the data to the client as the new data is available.

Non Streaming EndPoint



Streaming EndPoint - SSE



Use Cases

- Stock Tickers
- Weather Updates
- Flight Arrival/Departure/Delay updates in airports.

MongoDB

- **Tailable Cursor**
  - Connections remains open after all the results are retrieved.
  - https://docs.mongodb.com/manual/core/tailable-cursors/index.html

- **Capped Collections**
  - Collection of fixed-size in MongoDB.
  - Preserves the insertion Order.
  - https://docs.mongodb.com/manual/core/capped-collections/

USE CASES OF SPRING WEBFLUX

Use Cases of Spring WebFlux

- Perfect for **Micro-Services** Environment
- Performs really good for Slow Clients
- Perfect for **Streaming Real time data** use cases.