

Spring WebFlux: Getting Started

Introducing Spring Webflux



Esteban Herrera

Author | Developer | Consultant

@eh3rrera eherrera.net

What does “reactive” mean?

The Reactive Manifesto

Published on September 16 2014. (v2.0)

Organisations working in disparate domains are independently discovering patterns for building software that look the same. These systems are more robust, more resilient, more flexible and better positioned to meet modern demands.

These changes are happening because application requirements have changed dramatically in recent years. Only a few years ago a large application had tens of servers, seconds of response time, hours of offline maintenance and gigabytes of data. Today applications are deployed on everything from mobile devices to cloud-based clusters running thousands of multi-core processors. Users expect millisecond response times and 100% uptime. Data is measured in Petabytes. Today's demands are simply not met by yesterday's software architectures.

We believe that a coherent approach to systems architecture is needed, and we believe that all necessary aspects are already recognised individually: we want systems that are Responsive, Resilient, Elastic and Message Driven. We call these Reactive Systems.

Systems built as Reactive Systems are more flexible, loosely-coupled and [scalable](#). This makes them easier to develop and amenable to change. They are significantly more

Languages

- Java: [RxJava](#)
- JavaScript: [RxJS](#)
- C#: [Rx.NET](#)
- C#(Unity): [UniRx](#)
- Scala: [RxScala](#)
- Clojure: [RxClojure](#)
- C++: [RxCpp](#)
- Lua: [RxLua](#)
- Ruby: [Rx.rb](#)
- Python: [RxPY](#)
- Go: [RxGo](#)
- Groovy: [RxGroovy](#)
- JRuby: [RxJRuby](#)
- Kotlin: [RxKotlin](#)
- Swift: [RxSwift](#)
- PHP: [RxPHP](#)
- Elixir: [reaxive](#)
- Dart: [RxDart](#)

ReactiveX for platforms and frameworks

- [RxNetty](#)
- [RxAndroid](#)
- [RxCocoa](#)

Course Overview



Introducing Spring WebFlux

Reactive programming with Reactor

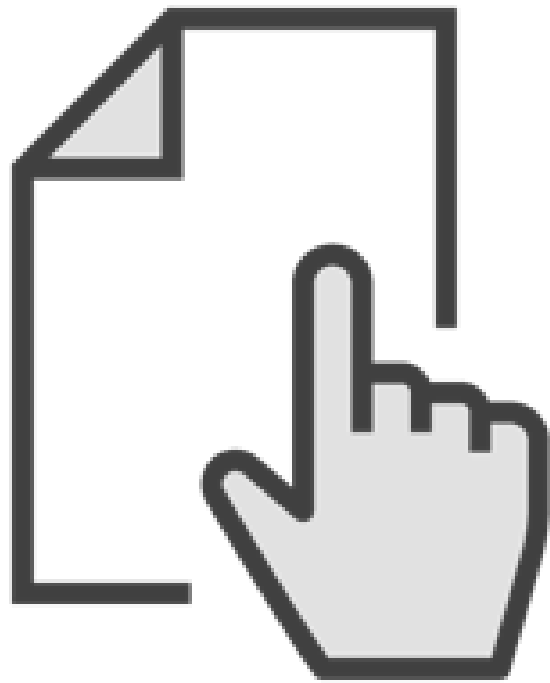
Sample project: REST API

- **Annotated controllers**
- **Functional endpoints**

API client with WebClient

Testing with WebTestClient

Prerequisites



Spring Framework

Spring MVC

Lambdas and streams



For Spring

Spring Framework: Spring Fundamentals

Bryan Hansen



For Spring MVC

Spring Framework: Spring MVC Fundamentals

Bryan Hansen



For lambda expressions and streams

Using Lambda Expressions in Java Code

Jose Paumard

Platform and Software



Java 8+



IntelliJ IDEA

Different Programming Model




Mind-shifting Paradigm



Ask Questions

 Resume Course

 Bookmark

 Add to Channel

 Download Course

 Schedule Reminder

Table of contents

Description

Transcript

Exercise files


Discussion

Related Courses

87 Comments

Pluralsight Course Discussions

 Disqus' Privacy Policy

 **Esteban Herrera** ▾

 Recommend

 Tweet

 Share

Sort by Newest ▾



Join the discussion...

What Is Reactive Programming?

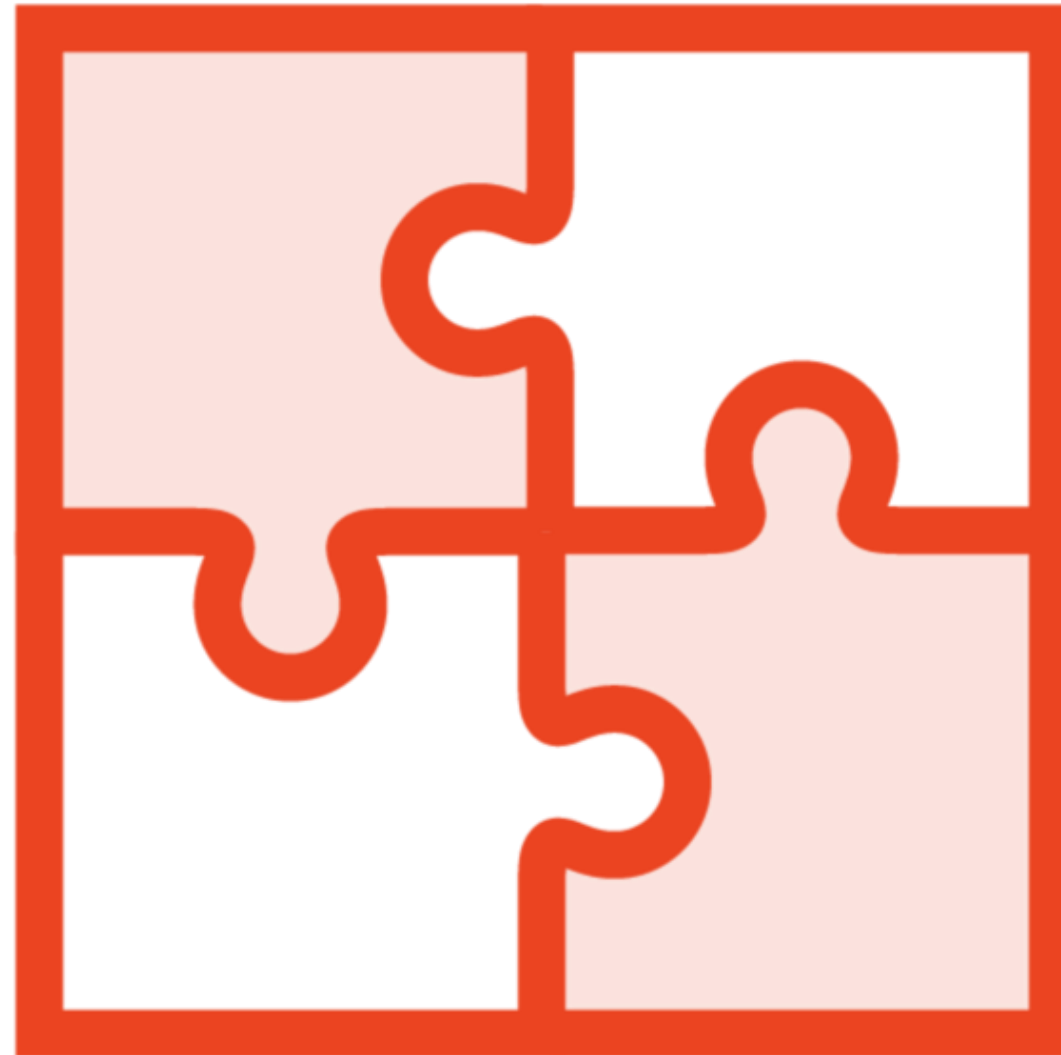
Reactive in Software Development



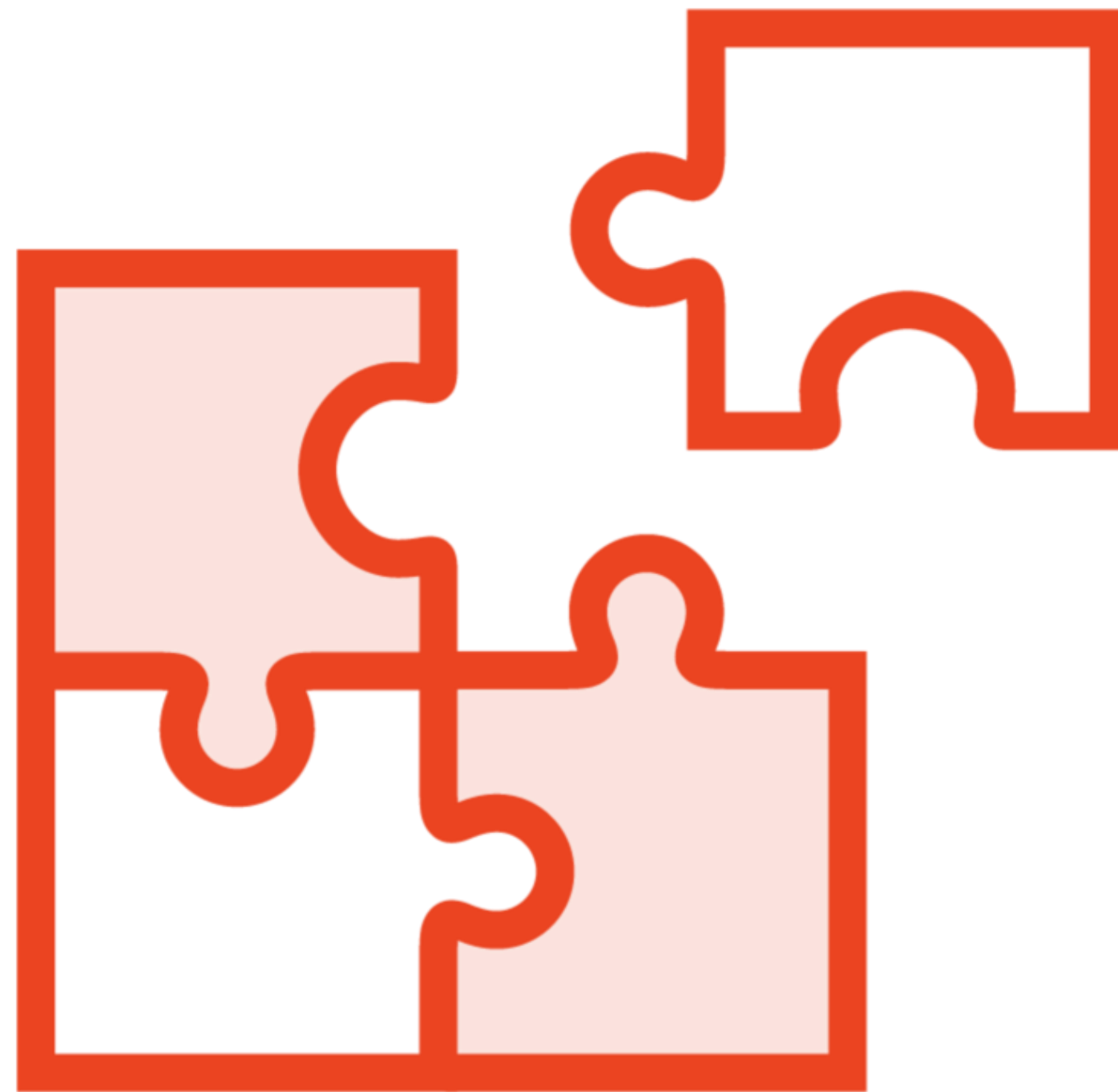
Reactive systems

Reactive programming

Reactive System



Reactive System



Reactive
Programming

Reactive System

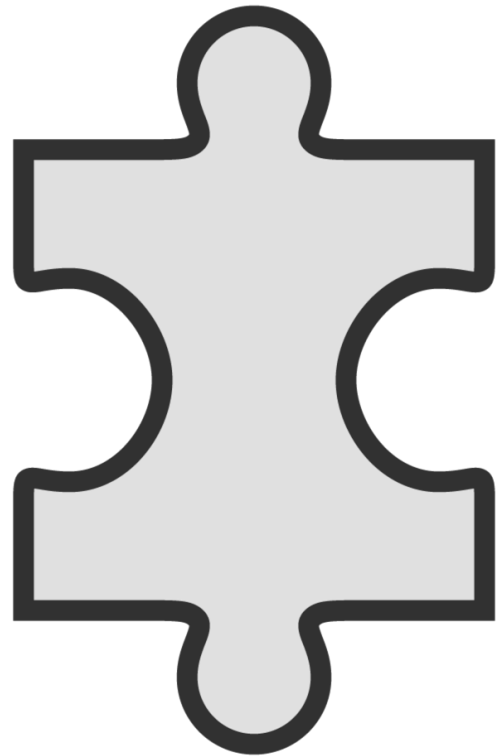
Responsive

Resilient

Scalable

Message

Reactive Programming



Event-driven

Data flow

Traditional (Imperative) Model

```
int a = 2;
```

```
int b = a * 10;
```

```
System.out.println(b);
```

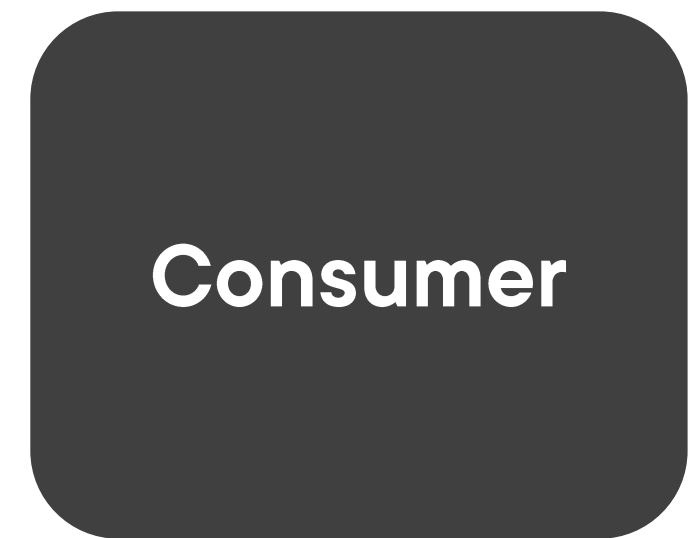
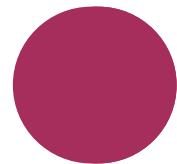
```
a = 3;
```

```
System.out.println(b);
```

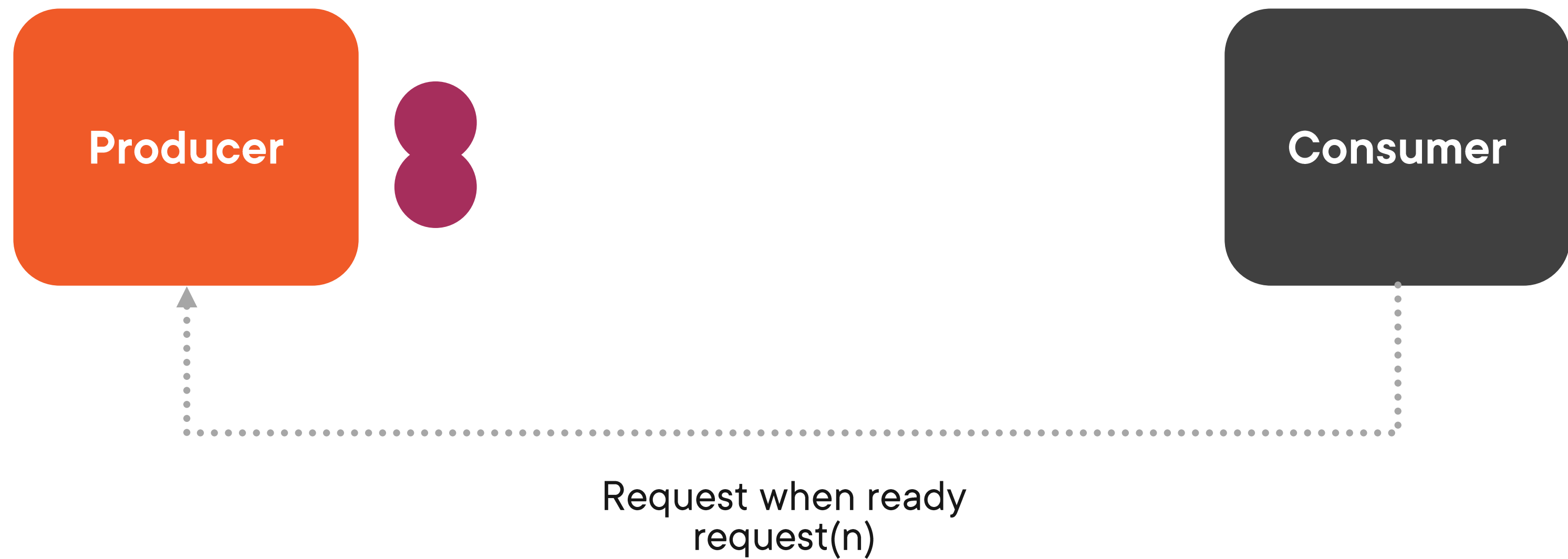
Observer Pattern



Observer Pattern



Backpressure



Reactive Programming Model



Non-blocking



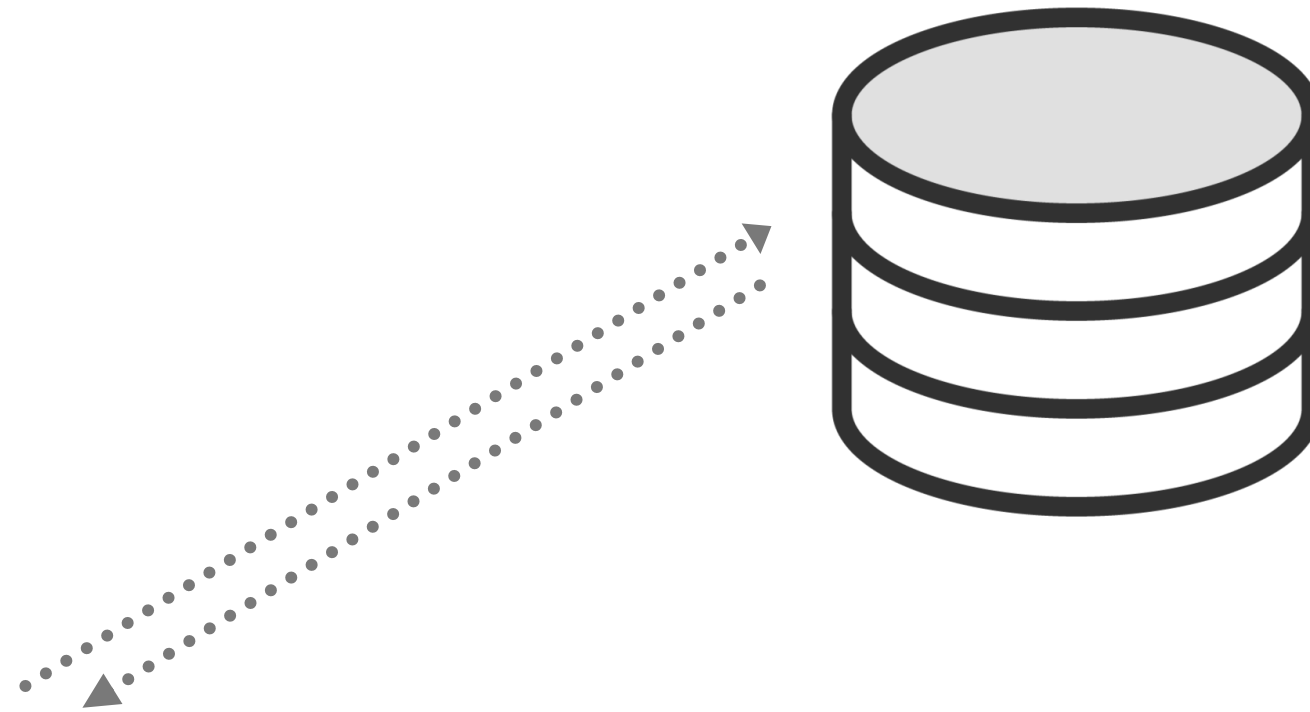
Asynchronous



Functional/Declarative

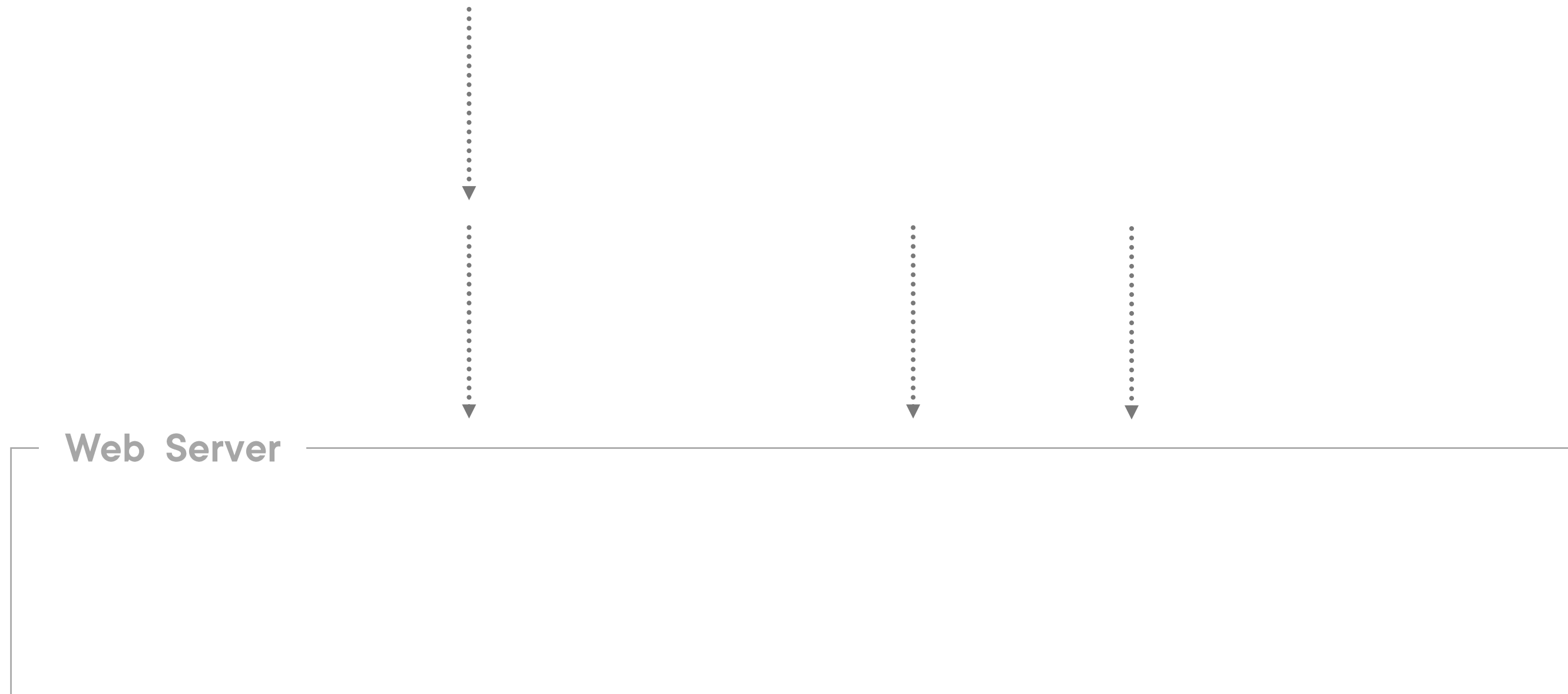
Non-blocking Programming

Blocking Call

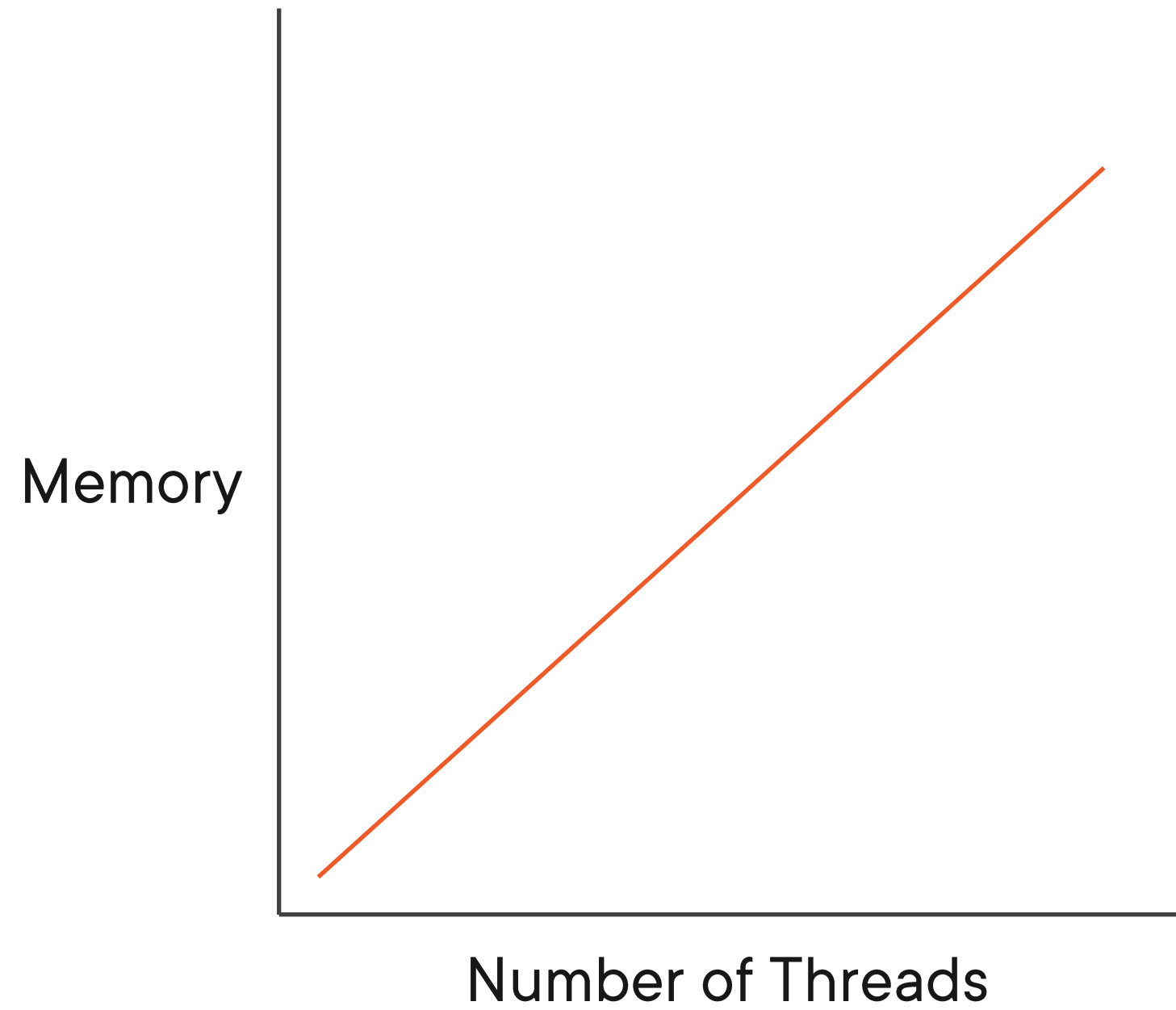


```
Product p = db.getProduct(id);  
show(p);
```

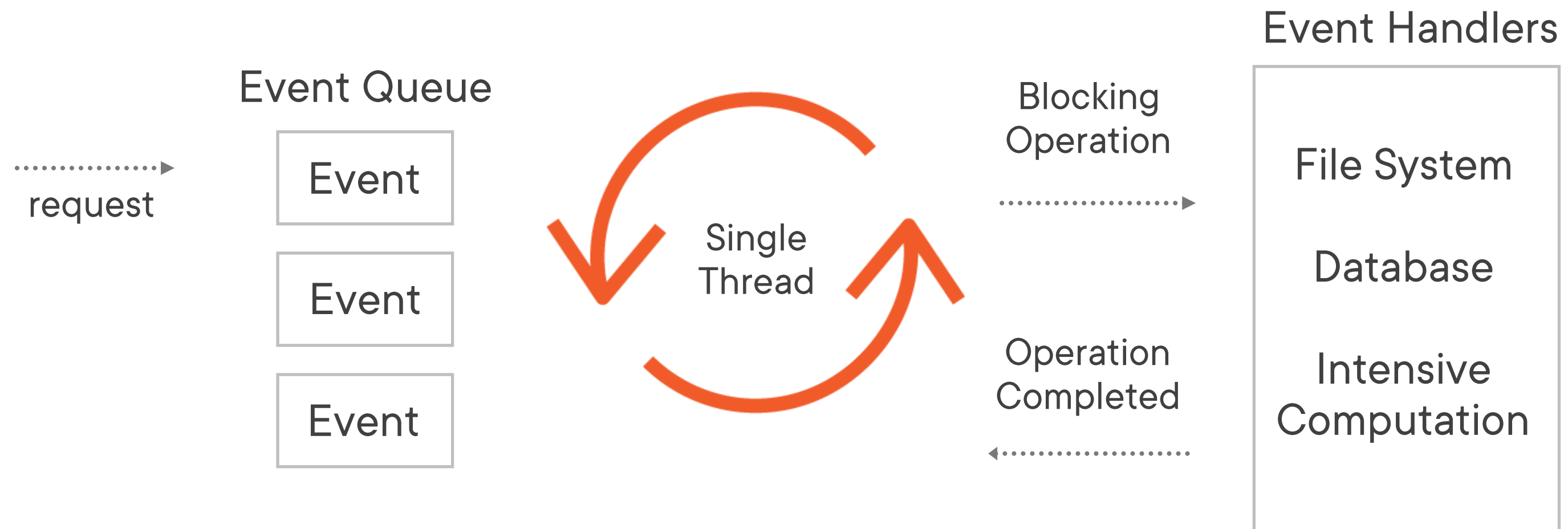
Web Server



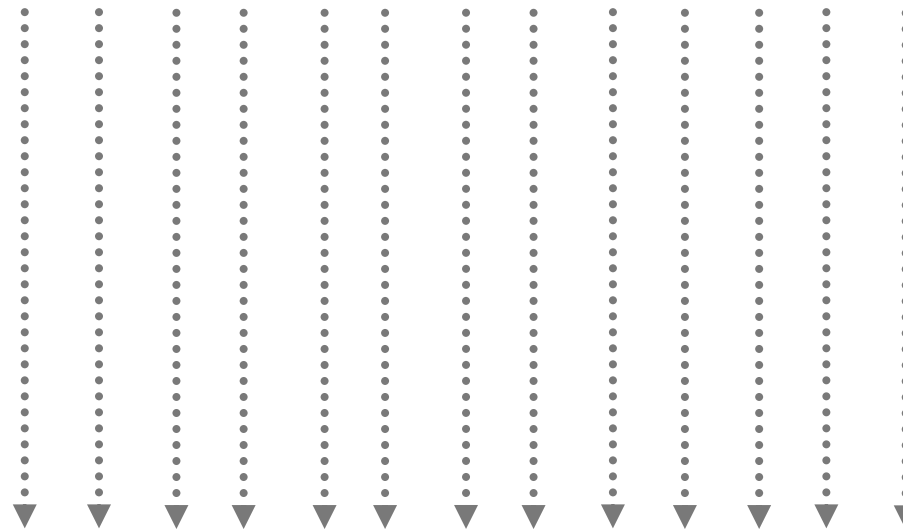
Memory Consumption



Event Loop



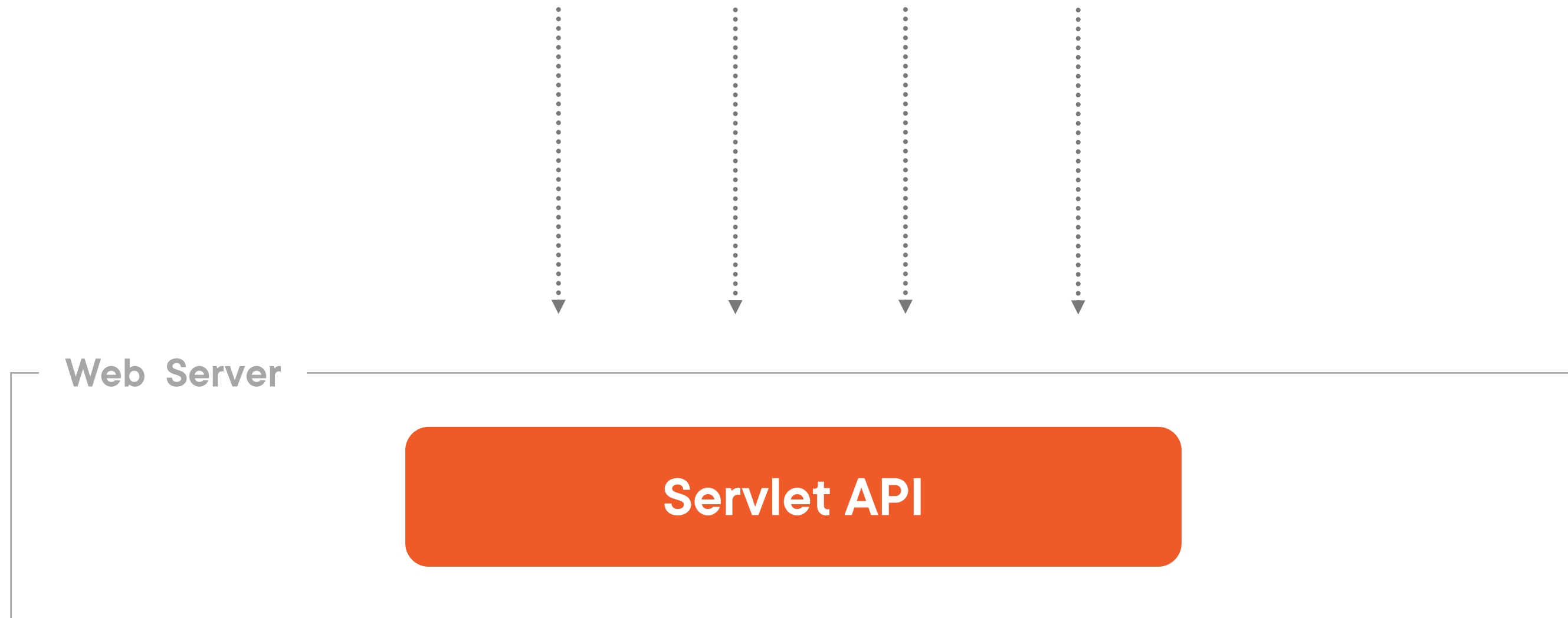
Web Server



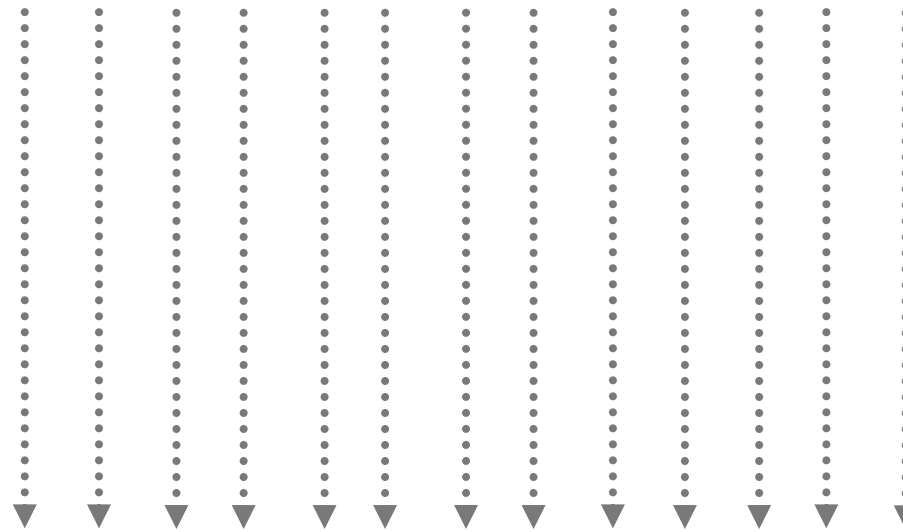
Web Server



Web Server (Blocking)



Web Server



Web Server

**Servlet API 3.1+
(non-blocking)**

Asynchronous Programming

Blocking Code

```
Product p = db.getProduct(id);  
show(p);
```

Callbacks

```
db.getProduct(id, (p, error) -> {  
    show(p);  
});
```

Callback Hell

```
db.getProduct(id, (p, error) -> {  
    if(error) {  
        // ...  
    } else {  
        show(p);  
        db.getProductDetails(p, (d, error) -> {  
            if(error) {  
                // ...  
            } else {  
                // ...  
            }  
        });  
    }  
});
```

Async in Java



Thread pools

Fork/join framework

Parallel streams

CompletableFuture

CompletableFuture

```
CompletableFuture.supplyAsync(this::processOperation)  
    .thenApply(this::sendEmail)  
    .thenAccept(this::completeOperation)
```

JavaScript's Async/Await

```
async function getProductDetail(id) {  
    const p = await db.getProduct(id);  
    const d = await db.getProductDetail(p);  
    return d;  
}
```

Blocking Code

```
Product p = db.getProduct(id);
```


Publisher/Subscriber

```
Publisher<Product> p = db.getProduct(id);
```

Publisher/Subscriber



Functional and Declarative Programming

Functional Programming

Pure Functions

Lambda Expressions

Immutability

Testable

Declarative

Maintainable

Imperative Code

```
List prices = productService.getHistoricalPrices(productId);
Iterator iterator = prices.iterator();
while (iterator.hasNext()) {
    ProductPrice price = (ProductPrice) iterator.hasNext();
    List details = productService.getDetails(price);
    if(details == null) {
        details = historyService.getDetails(productId);
        // ...
    }
    // ...
}
```

Declarative API

```
productService.getHistoricalPrices(productId)
    .flatMap(productService::getDetails)
    .switchIfEmpty(historyService.getDetails(productId))
    .take(2)
    .timeout(Duration.ofMillis(200))
    .onErrorResume(this::getDummyPrices)
    .publishOn(Schedulers.parallel())
    .subscribe(this::graph);
```

Readable and Composable

```
productService.getHistoricalPrices(productId)
```

Readable and Composable

```
productService.getHistoricalPrices(productId)  
    .flatMap(productService::getDetails)
```


Readable and Composable

```
productService.getHistoricalPrices(productId)  
    .flatMap(productService::getDetails)  
    .switchIfEmpty(historyService.getProduct(productId))
```

Readable and Composable

```
productService.getHistoricalPrices(productId)  
    .flatMap(productService::getDetails)  
    .switchIfEmpty(historyService.getProduct(productId))  
    .take(2)
```

Readable and Composable

```
productService.getHistoricalPrices(productId)  
    .flatMap(productService::getDetails)  
    .switchIfEmpty(historyService.getProduct(productId))  
    .take(2)  
    .timeout(Duration.ofMillis(200))
```

Readable and Composable

```
productService.getHistoricalPrices(productId)  
    .flatMap(productService::getDetails)  
    .switchIfEmpty(historyService.getProduct(productId))  
    .take(2)  
    .timeout(Duration.ofMillis(200))  
    .onErrorResume(this::getDummyPrices)
```

Data as Flow

```
productService.getHistoricalPrices(productId)
    .flatMap(productService::getDetails)
    .switchIfEmpty(historyService.getProduct(productId))
    .take(2)
    .timeout(Duration.ofMillis(200))
    .onErrorResume(this::getDummyPrices)
```



Data as Flow

```
productService.getHistoricalPrices(productId)
    .flatMap(productService::getDetails)
    .switchIfEmpty(historyService.getProduct(productId))
    .take(2)
    .timeout(Duration.ofMillis(200))
    .onErrorResume(this::getDummyPrices)
    .subscribe(this::graph);
```



High-level Abstraction

```
productService.getHistoricalPrices(productId)  
    .flatMap(productService::getDetails)  
    .switchIfEmpty(historyService.getProduct(productId))  
    .take(2)  
    .timeout(Duration.ofMillis(200))  
    .onErrorResume(this::getDummyPrices)  
    .publishOn(Schedulers.parallel())  
    .subscribe(this::graph);
```

High-level Abstraction

```
Publisher<ProductPrice> productPrices =  
    productService.getHistoricalPrices(productId)  
        .flatMap(productService::getDetails)  
        .switchIfEmpty(historyService.getProduct(productId))  
        .take(2)  
        .timeout(Duration.ofMillis(200))  
        .onErrorResume(this::getDummyPrices)  
        .publishOn(Schedulers.parallel())  
        .subscribe(this::graph);
```


Reactive Streams

Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols.

JDK9 `java.util.concurrent.Flow`

The interfaces available in JDK9's [java.util.concurrent.Flow](#), are 1:1 semantically equivalent to their respective Reactive Streams counterparts. This means that there will be a migratory period, while libraries move to adopt the new types in the JDK, however this period is expected to be short - due to the full semantic equivalence of the libraries, as well as the Reactive Streams <-> Flow adapter library as well as a TCK compatible directly with the JDK Flow types.

Read [this](#) if you are interested in learning more about `Reactive Streams` for the JVM.

The Problem

Handling streams of data—especially “live” data whose volume is not predetermined—requires special care in an asynchronous system. The most prominent issue is that resource consumption needs to be controlled such that a fast data source does not overwhelm the stream destination. Asynchrony is needed

Stream API

[1, 2, 3, 4, 5]



```
numbers.stream()  
    .map(this::convert)  
    .forEach(System.out::println)
```

Reactive Streams

[]

```
numberService.getNumbers()  
    .map(this::convert)  
    .subscribe(System.out::println)
```

Reactive Streams

$$[1, \quad]$$

```
numberService.getNumbers()  
    .map(this::convert)  
    .subscribe(System.out::println)
```



Reactive Streams

[1, 2,]

```
numberService.getNumbers()  
    .map(this::convert)  
    .subscribe(System.out::println)
```



Reactive Streams

[1, 2, 3,]

```
numberService.getNumbers()  
    .map(this::convert)  
    .subscribe(System.out::println)
```



Reactive Streams

[1, 2, 3, ...]

```
numberService.getNumbers()  
    .map(this::convert)  
    .subscribe(System.out::println)
```

Spring WebFlux

Spring 5 Web Stack

Spring MVC

Servlet API

Blocking API

Synchronous

One request per thread

Spring WebFlux

Reactive Streams

Non-blocking API (Servlet 3.1+)

Asynchronous

Concurrent connections with few threads

Spring 5 Web Stack

Annotations

**Functional
Endpoints**

spring-web-mvc

spring-web-reactive

Servlet API

HTTP / Reactive Streams

Servlet Container

Netty, Tomcat, Jetty, Undertow

Annotation Style

```
@RestController
@RequestMapping(value = "/products")
class ProductController {
    private final ProductRepository repository;

    ...

    @GetMapping(value = "/")
    public Flux<Product> listProducts() {
        return repository.findAll();
    }

    ...
}
```

Functional Style

```
RouterFunction<ServerResponse> productRoute =  
    route(GET("/product").and(  
        accept(APPLICATION_JSON),  
        handler::listProducts  
    ));
```

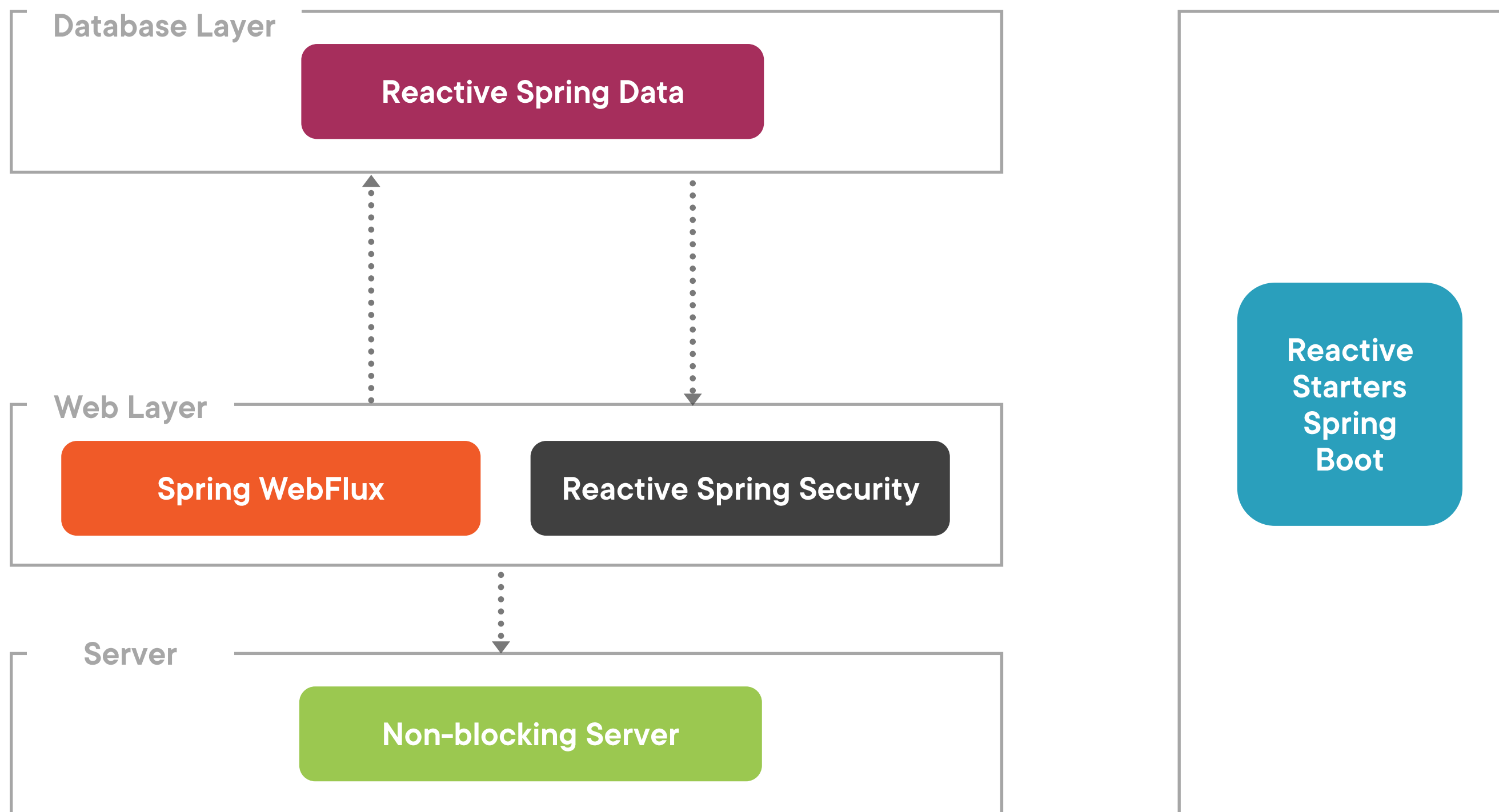
...

```
public Mono<ServerResponse> listProducts(ServerRequest request) {  
    Flux<Product> products = repository.findAll();  
    return ServerResponse.ok().contentType(APPLICATION_JSON).body(  
        products, Product.class);  
}
```

Kotlin DSL

```
router {  
    "/product".nest {  
        accept(APPLICATION_JSON).nest {  
            GET("/users", handler::listProducts)  
        }  
    }  
    resources("/**", ClassPathResource("public/"))  
}
```

Reactive All the Way



Reactive NoSQL Spring Data Access

MongoDB

Cassandra

Redis

Couchbase

Reactive data access needs an
async database driver

R2DBC

Reactive Relational Database Connectivity

**Reactive, non-blocking API for relational
databases**

Reactive Programming Model



Non-blocking

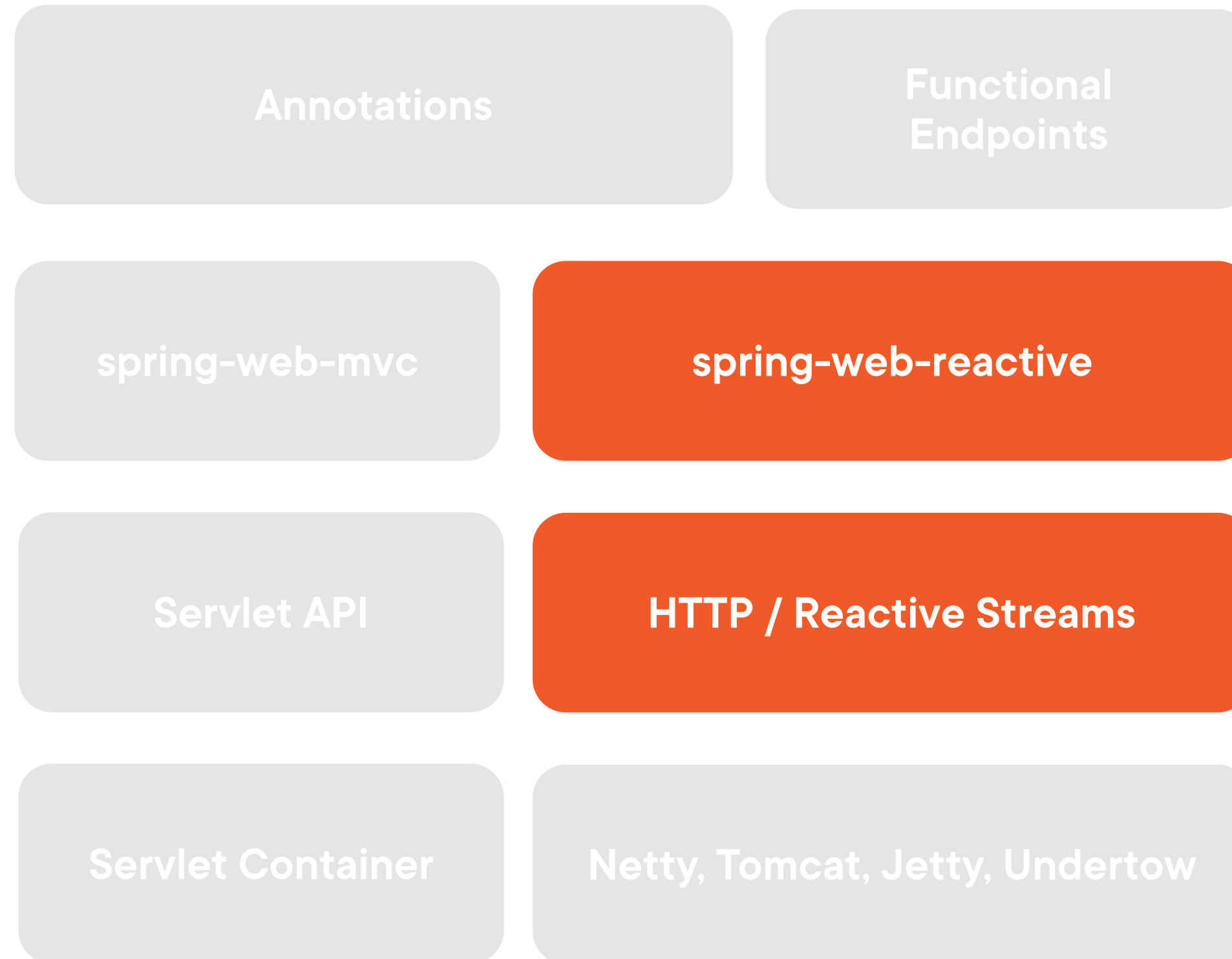


Asynchronous



Functional/Declarative

Spring 5 Web Stack



Reactive Streams

Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols.

JDK9 `java.util.concurrent.Flow`

The interfaces available in JDK9's [java.util.concurrent.Flow](#), are 1:1 semantically equivalent to their respective Reactive Streams counterparts. This means that there will be a migratory period, while libraries move to adopt the new types in the JDK, however this period is expected to be short - due to the full semantic equivalence of the libraries, as well as the Reactive Streams <-> Flow adapter library as well as a TCK compatible directly with the JDK Flow types.

Read [this](#) if you are interested in learning more about `Reactive Streams` for the JVM.

The Problem

Handling streams of data—especially “live” data whose volume is not predetermined—requires special care in an asynchronous system. The most prominent issue is that resource consumption needs to be controlled such that a fast data source does not overwhelm the stream destination. Asynchrony is needed

org.springframework.core

Class ReactiveAdapterRegistry

java.lang.Object
org.springframework.core.ReactiveAdapterRegistry

```
public class ReactiveAdapterRegistry
extends Object
```

A registry of adapters to adapt Reactive Streams `Publisher` to/from various async/reactive types such as `CompletableFuture`, `RxJava Observable`, and others.

By default, depending on classpath availability, adapters are registered for Reactor, RxJava 2/3, or RxJava 1 (+ RxJava Reactive Streams bridge), `CompletableFuture`, Java 9+ `Flow.Publisher`, and Kotlin Coroutines' `Deferred` and `Flow`.

Note: As of Spring Framework 5.3, support for RxJava 1.x is deprecated in favor of RxJava 2 and 3.

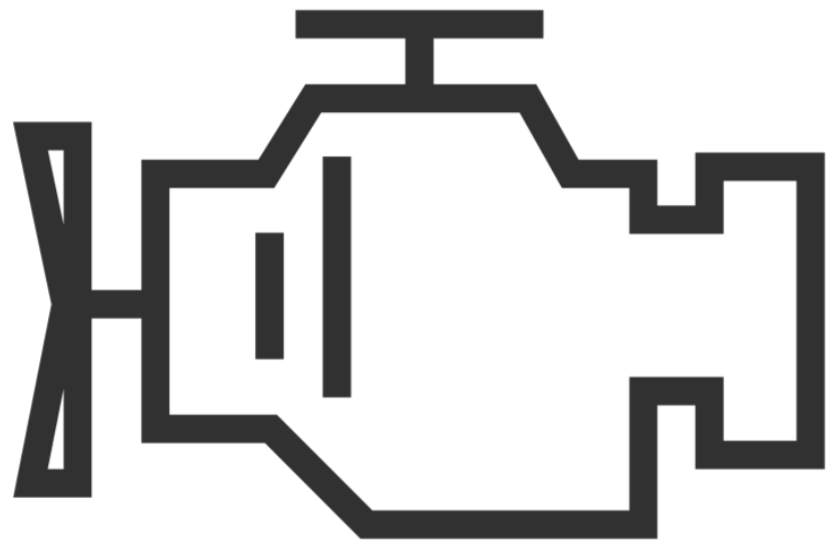
Since:
5.0

Author:
Rossen Stoyanchev, Sebastien Deleuze

Nested Class Summary

Nested Classes	
Modifier and Type	Class and Description
static class	<code>ReactiveAdapterRegistry.SpringCoreBlockHoundIntegration</code> BlockHoundIntegration for spring-core classes.

Reactive Support



Reactor

RxJava 2/3

CompletableFuture

Java 9 Flow API

Things to Remember



Reactive programming

- **Different from reactive systems**
- **Non-blocking**
- **Asynchronous**
- **Functional/declarative**

Spring WebFlux

- **Alternative to Spring MVC**
- **Annotation and functional model**
- **Reactive stack**
- **Project Rector**