Spring WebFlux: Getting Started

INTRODUCING SPRING WEBFLUX

Esteban Herrera
JAVA ARCHITECT

@eh3rrera   www.eherrera.net

What does “reactive” mean?

We Are Reactive

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

Overview

Introducing Spring WebFlux

Reactive programming with Reactor

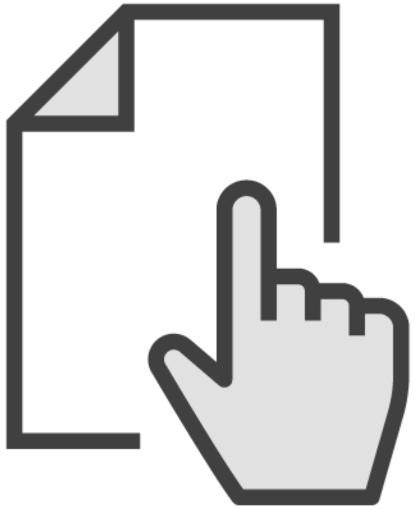
Sample project: REST API

- Annotated controllers
- Functional endpoints

API client with WebClient

Testing with WebTestClient

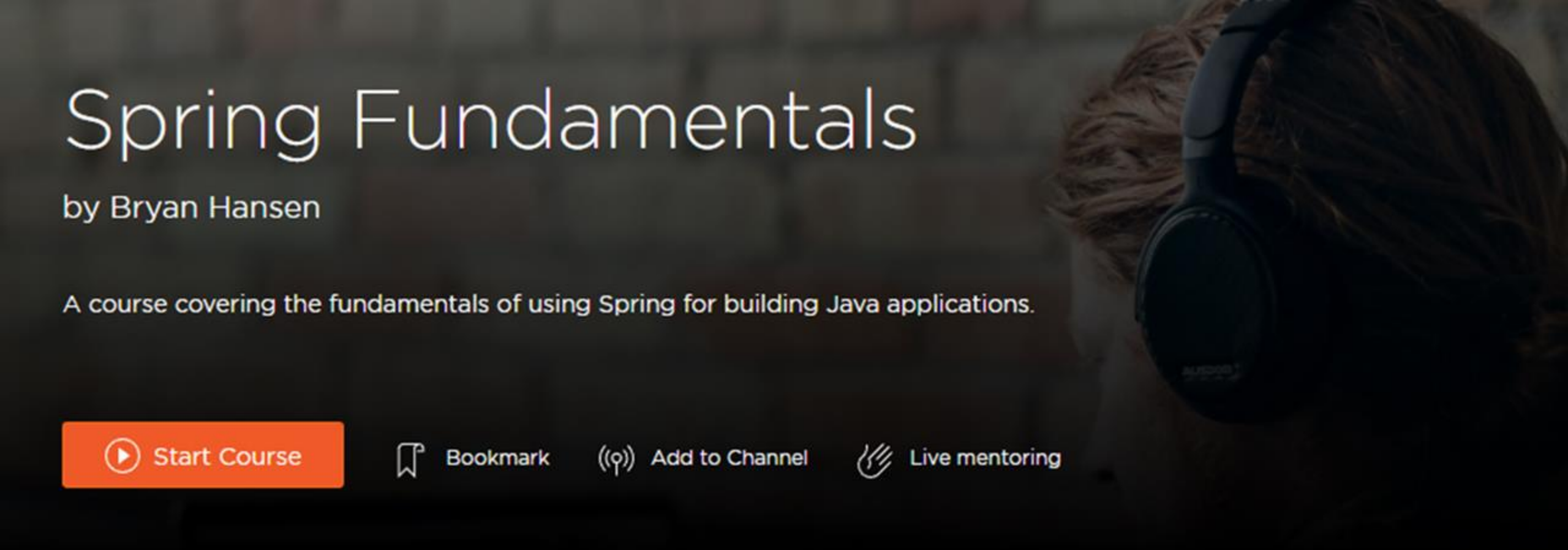
Prerequisites



Spring Framework

Spring MVC

Lambdas and streams



Spring Fundamentals

by Bryan Hansen

A course covering the fundamentals of using Spring for building Java applications.

http://bit.ly/spring-fundamentals

Introduction to Spring MVC 4

by Bryan Hansen

This module provides an introduction to building applications using Spring MVC4 and the Java configuration approach.

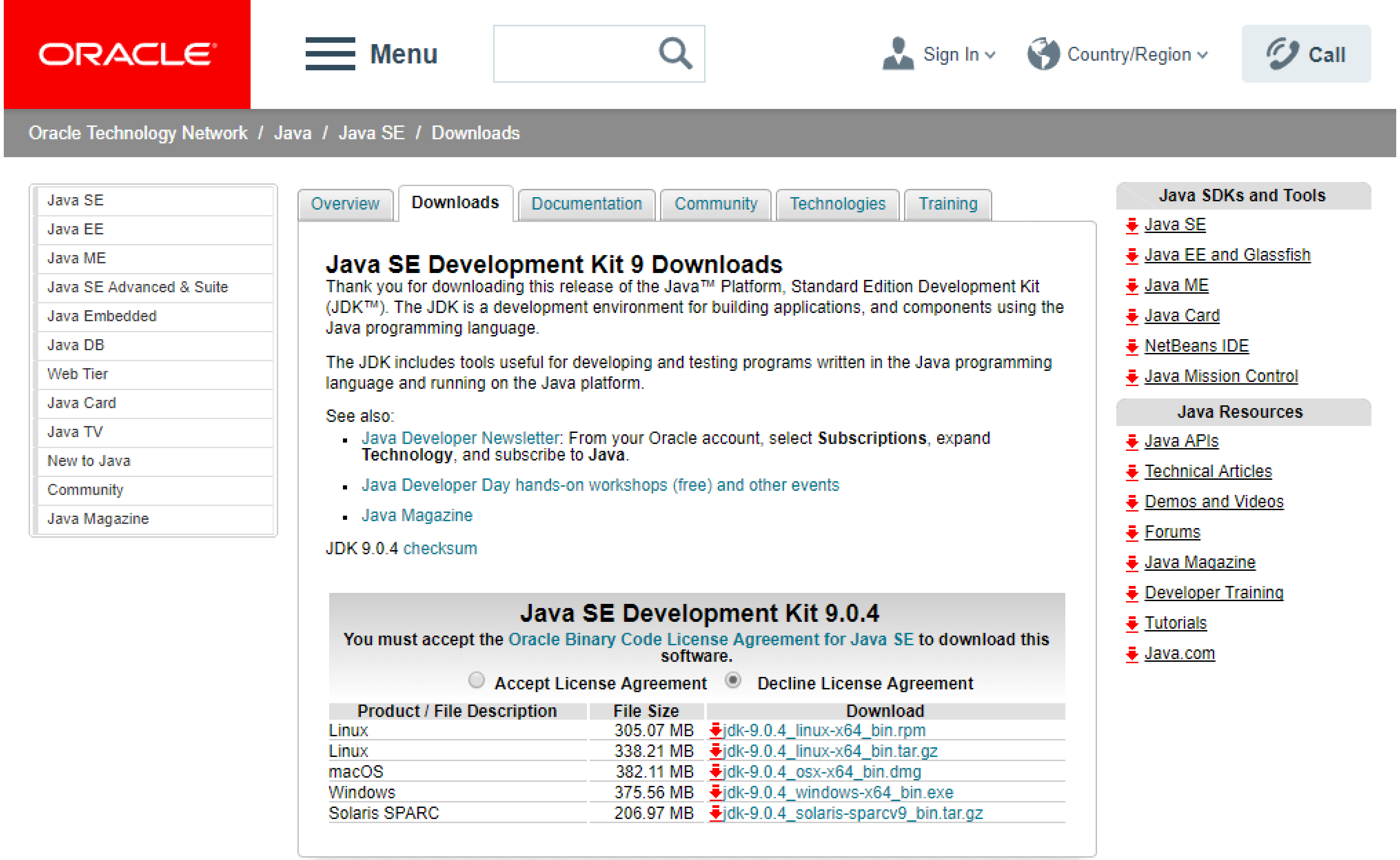
http://bit.ly/intro-spring-mvc-4

From Collections to Streams in Java 8 Using Lambda Expressions

by Jose Paumard

This course shows the new patterns introduced in Java 8, based on lambda expressions, the functional interfaces, the Collection Framework and the Stream API.

http://bit.ly/collections-to-streams



Java SE Development Kit 9 Downloads

Thank you for downloading this release of the Java™ Platform, Standard Edition Development Kit (JDK™). The JDK is a development environment for building applications, and components using the Java programming language.

The JDK includes tools useful for developing and testing programs written in the Java programming language and running on the Java platform.

See also:

- Java Developer Newsletter: From your Oracle account, select **Subscriptions**, expand **Technology**, and subscribe to **Java**.
- Java Developer Day hands-on workshops (free) and other events
- Java Magazine

JDK 9.0.4 checksum

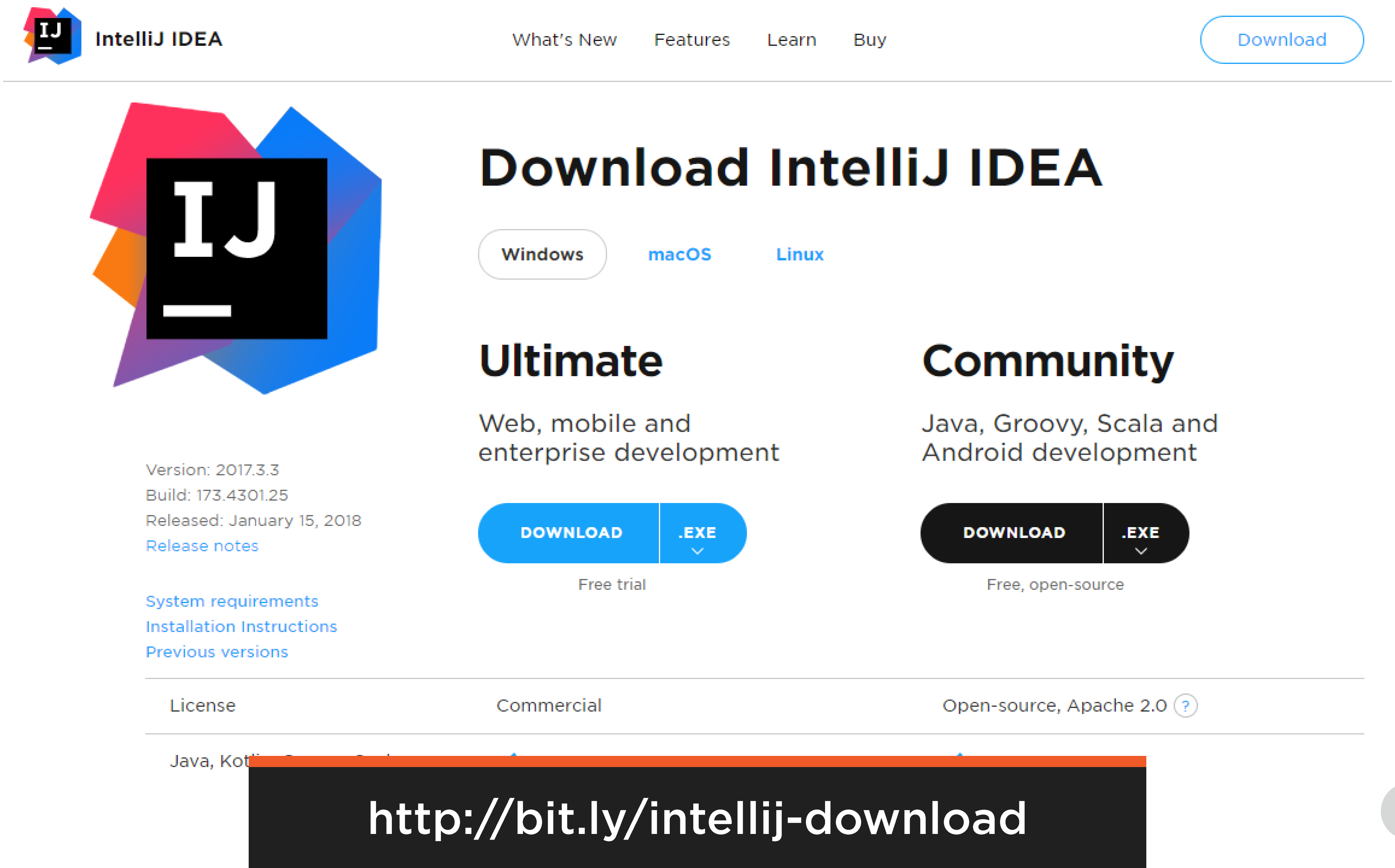
Java SE Development Kit 9.0.4

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Accept License Agreement [✓ Decline License Agreement]

Product / File Description	File Size	Download
Linux	305.07 MB	jdk-9.0.4_linux-x64_bin.rpm
Linux	338.21 MB	jdk-9.0.4_linux-x64_bin.tar.gz
macOS	382.11 MB	jdk-9.0.4_osx-x64_bin.dmg
Windows	375.56 MB	jdk-9.0.4_windows-x64_bin.exe
Solaris SPARC	206.97 MB	jdk-9.0.4_solaris-sparcv9_bin.tar.gz

http://bit.ly/jdk9-download



IntelliJ IDEA

What's New
Features
Learn
Buy
Download

Download IntelliJ IDEA

Windows
macOS
Linux

Version: 2017.3.3
Build: 173.4301.25
Released: January 15, 2018
Release notes

System requirements
Installation Instructions
Previous versions

Ultimate

Web, mobile and enterprise development

DOWNLOAD .EXE

Free trial

Community

Java, Groovy, Scala and Android development

DOWNLOAD .EXE

Free, open-source

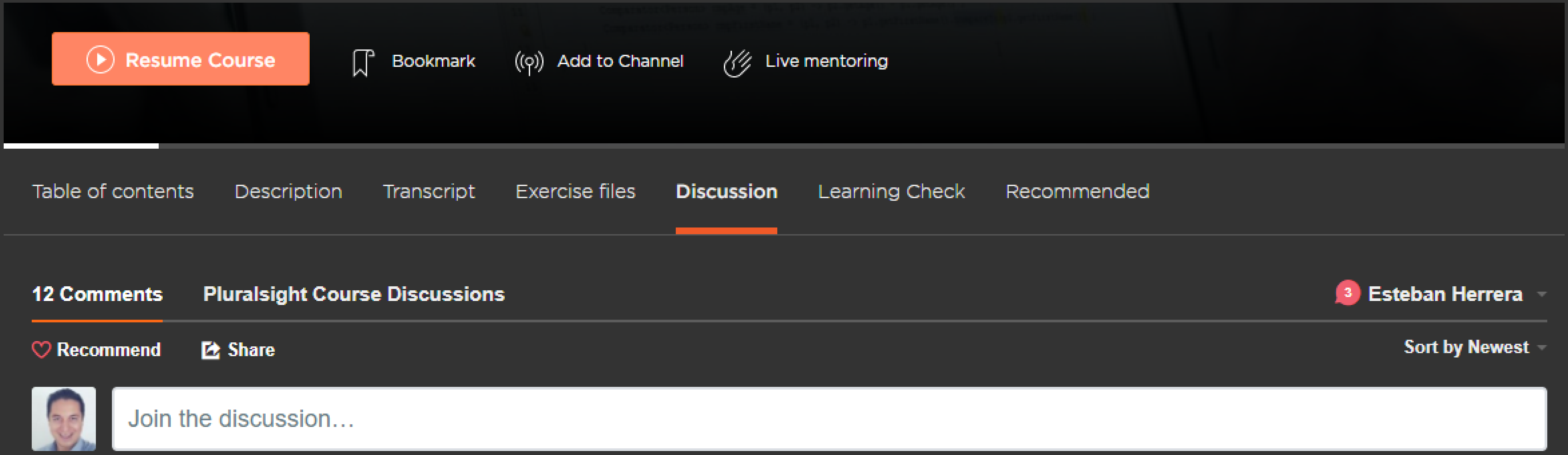
License	Commercial	Open-source, Apache 2.0 ?
Java, Kot		

http://bit.ly/intellij-download

Different Programming Model

Mind-shifting Paradigm

Ask Questions



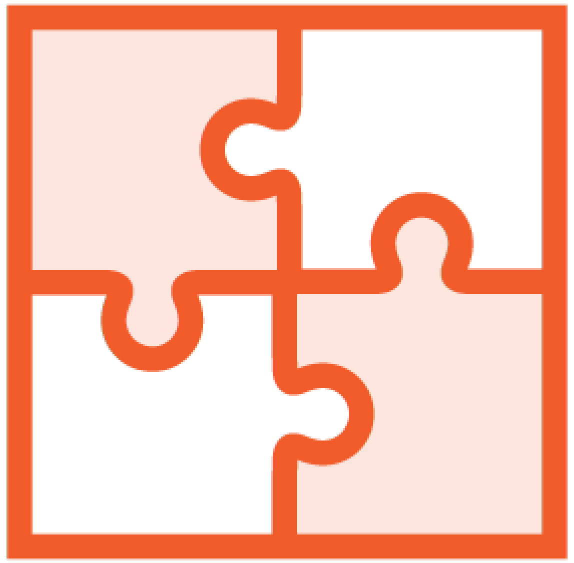
What Is Reactive Programming?

Reactive in Software Development

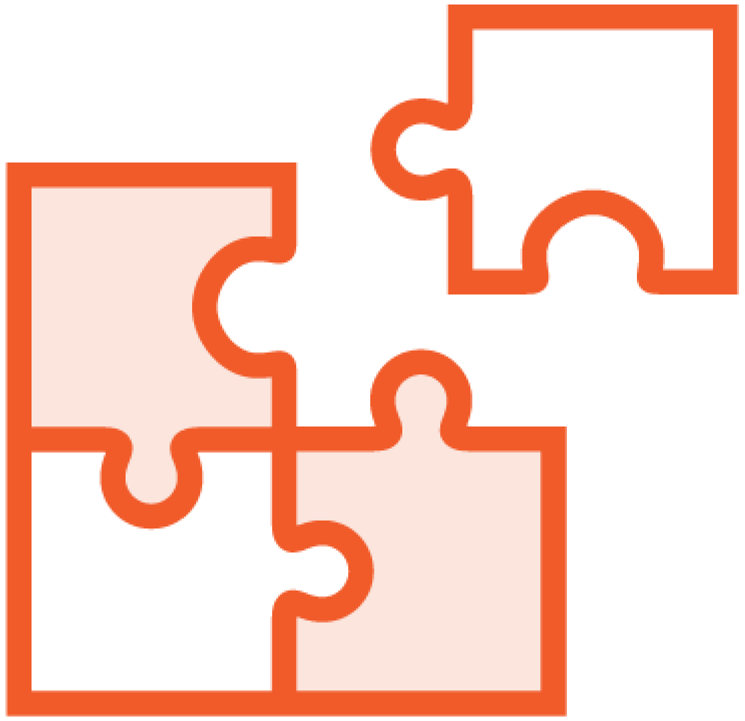
Reactive systems

Reactive programming

Reactive System

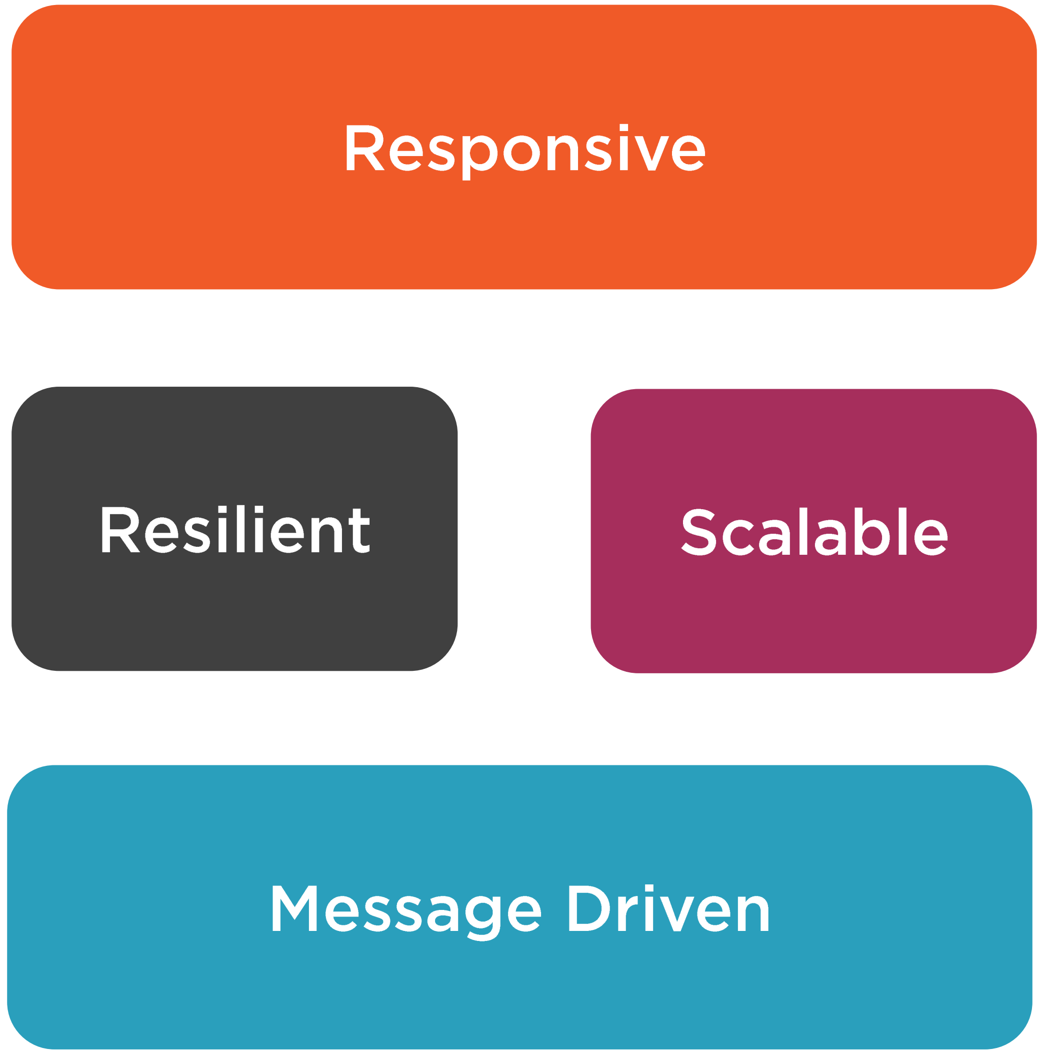


Reactive System

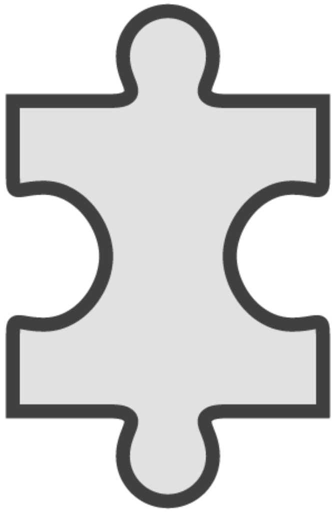


Reactive Programming

Reactive System



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);
```

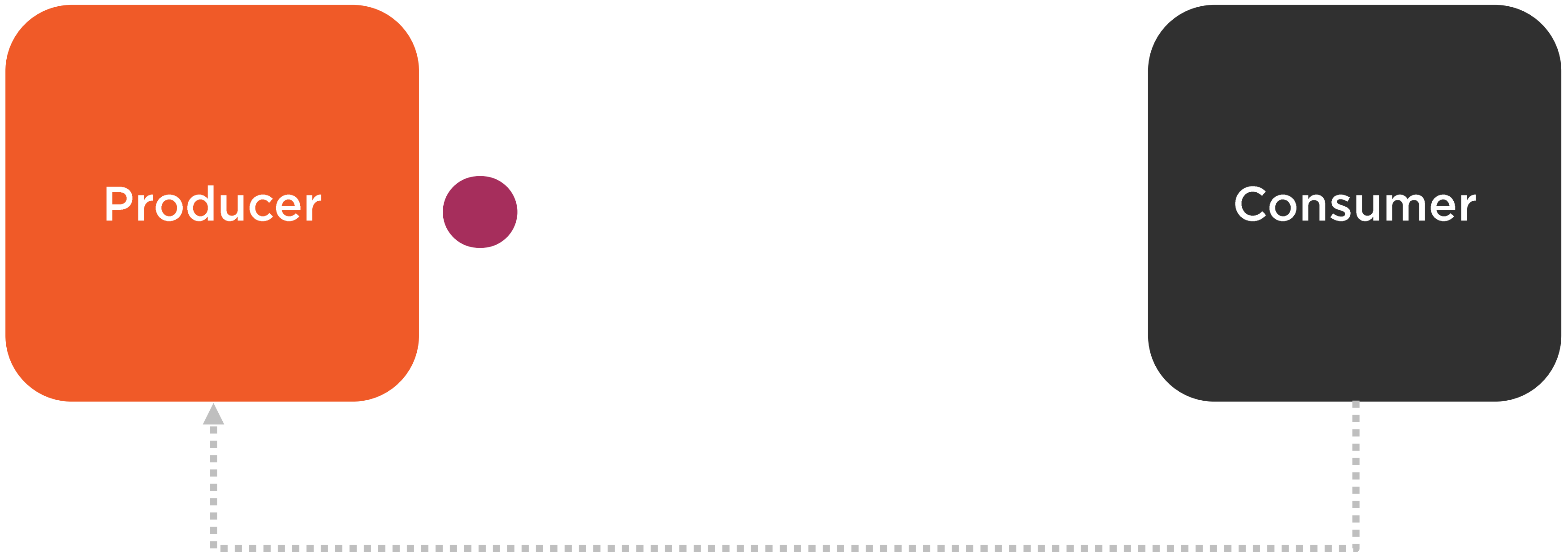
Observer Pattern

Producer
Consumer

Observer Pattern

Producer
Consumer

Backpressure



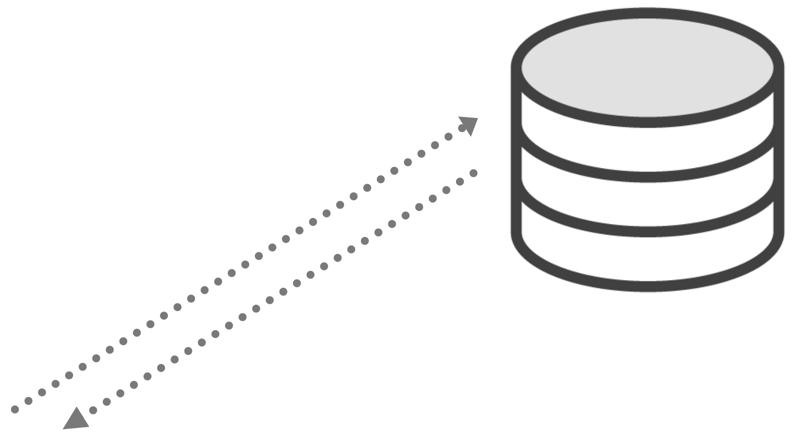
Request when ready
request(n)

Reactive Programming Model

- Non-blocking
- Asynchronous
- Functional/Declarative

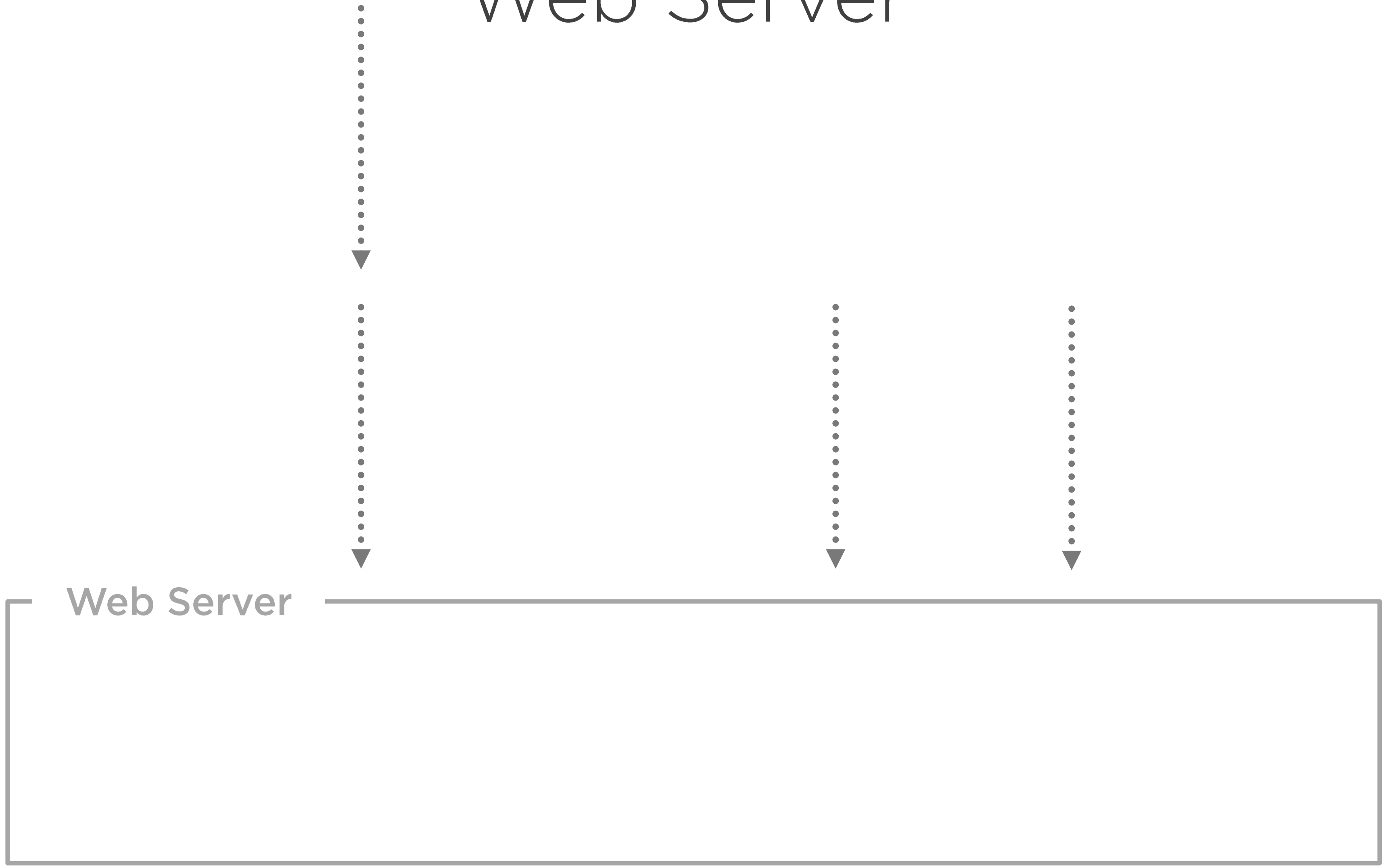
Non-blocking Programming

Blocking Call

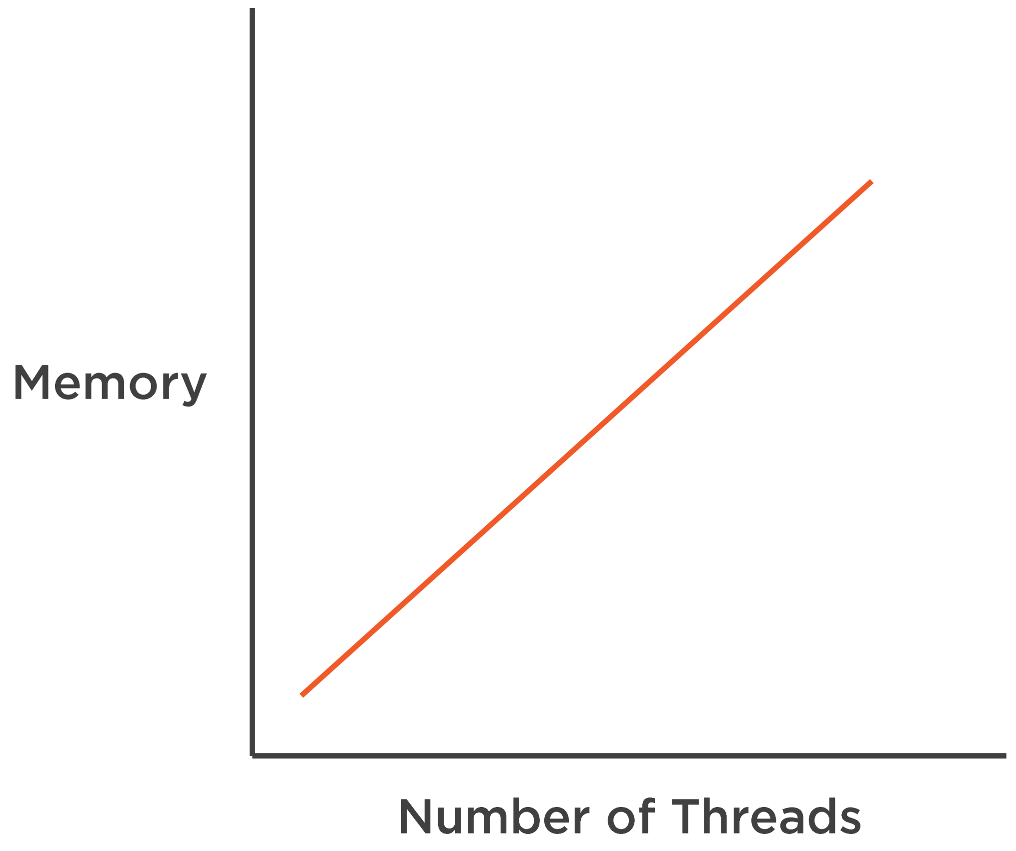


```
Product p = db.getProduct(id);
show(p);
```

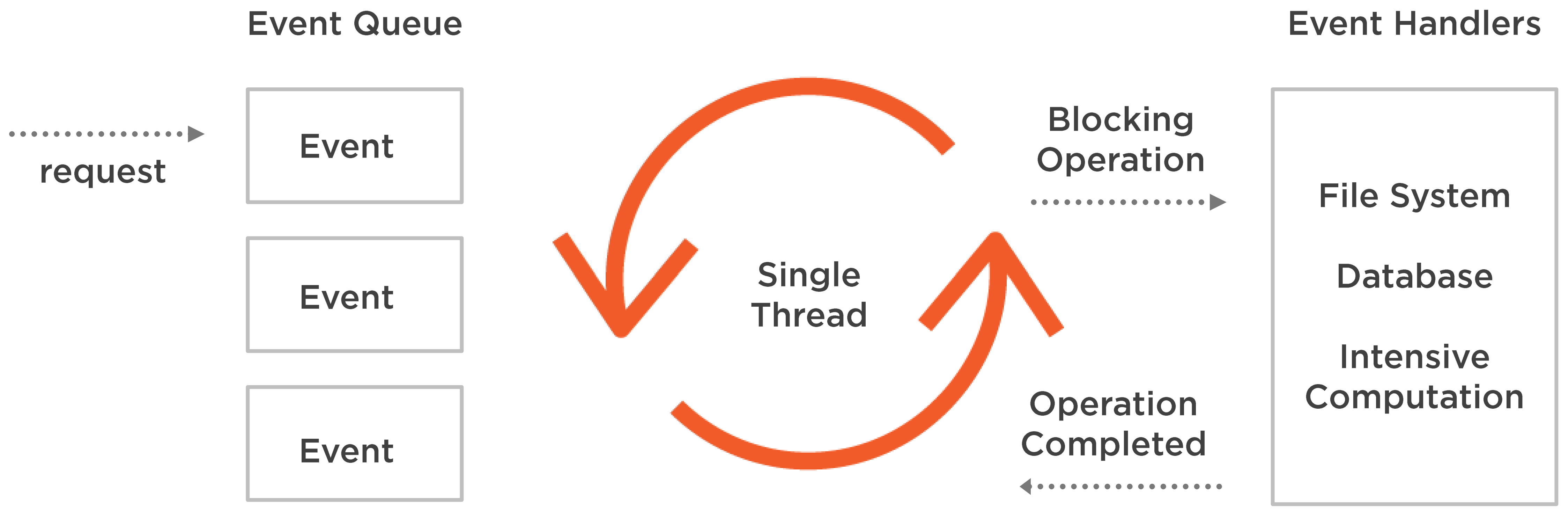
Web Server



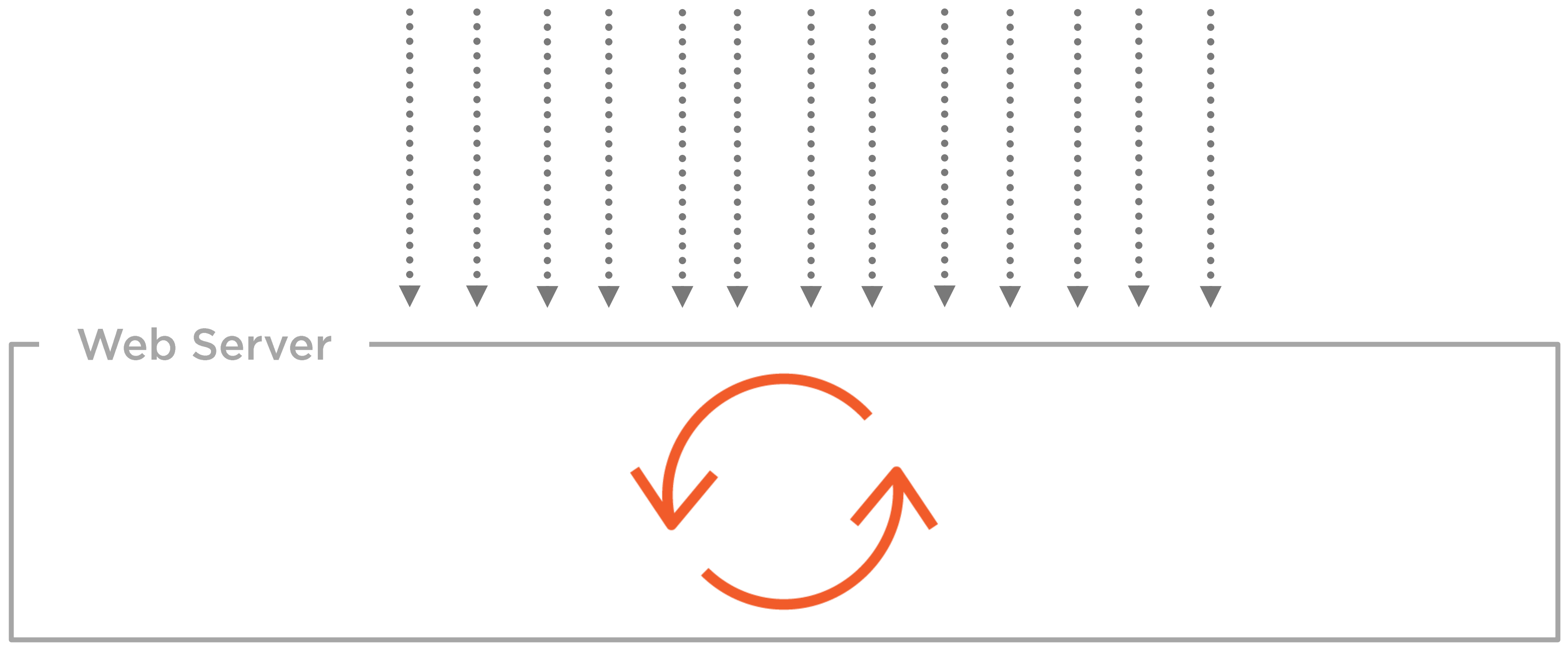
Memory Consumption



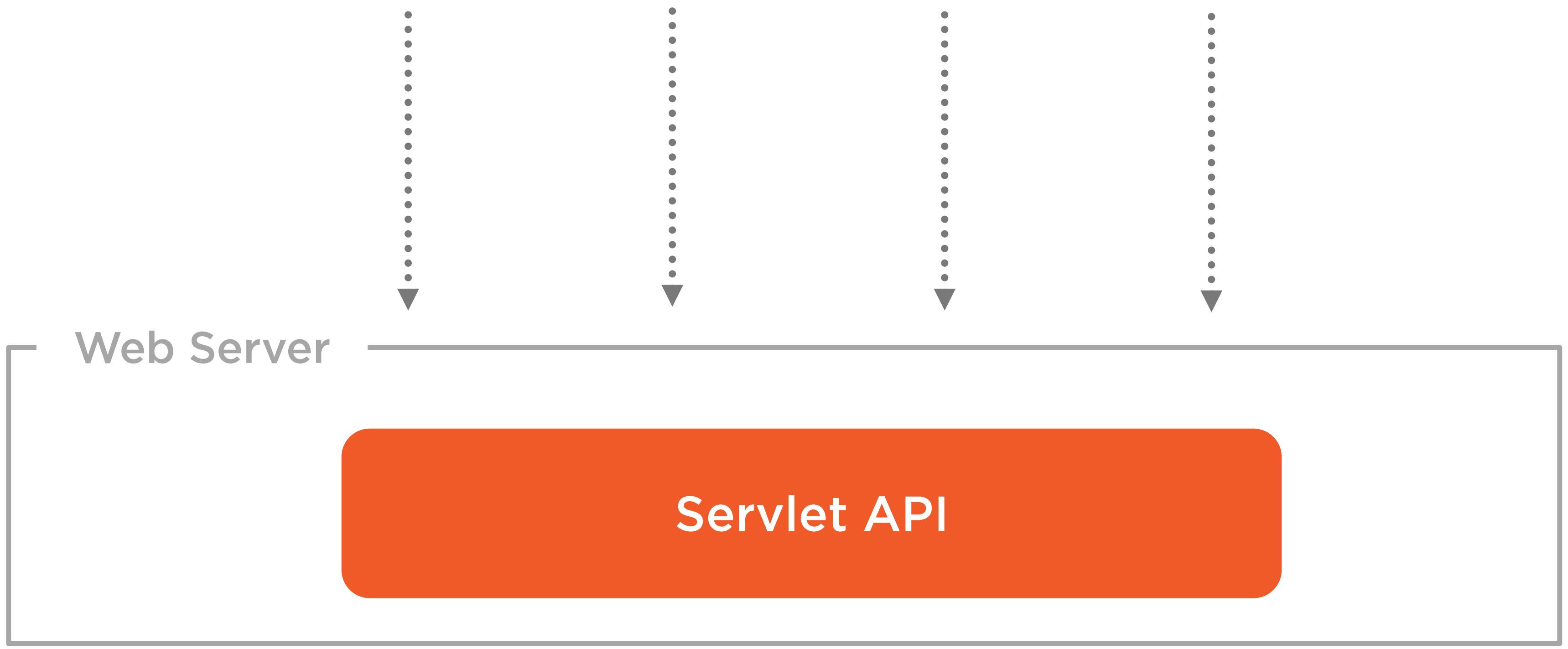
Event Loop


Event Queue
request
Event
Event
Event
Single Thread
Blocking Operation
Operation Completed
Event Handlers
File System
Database
Intensive Computation

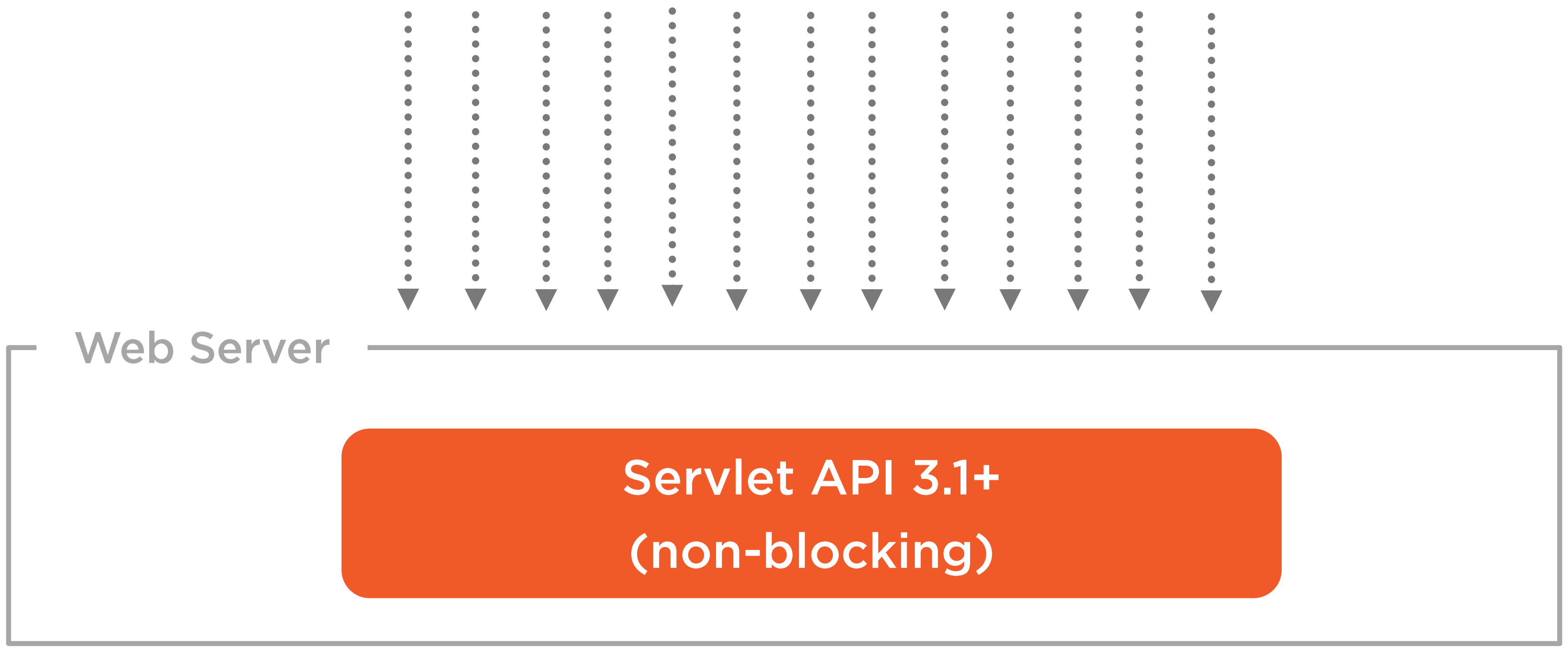
Web Server



Web Server (Blocking)



Web Server (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;
}
```

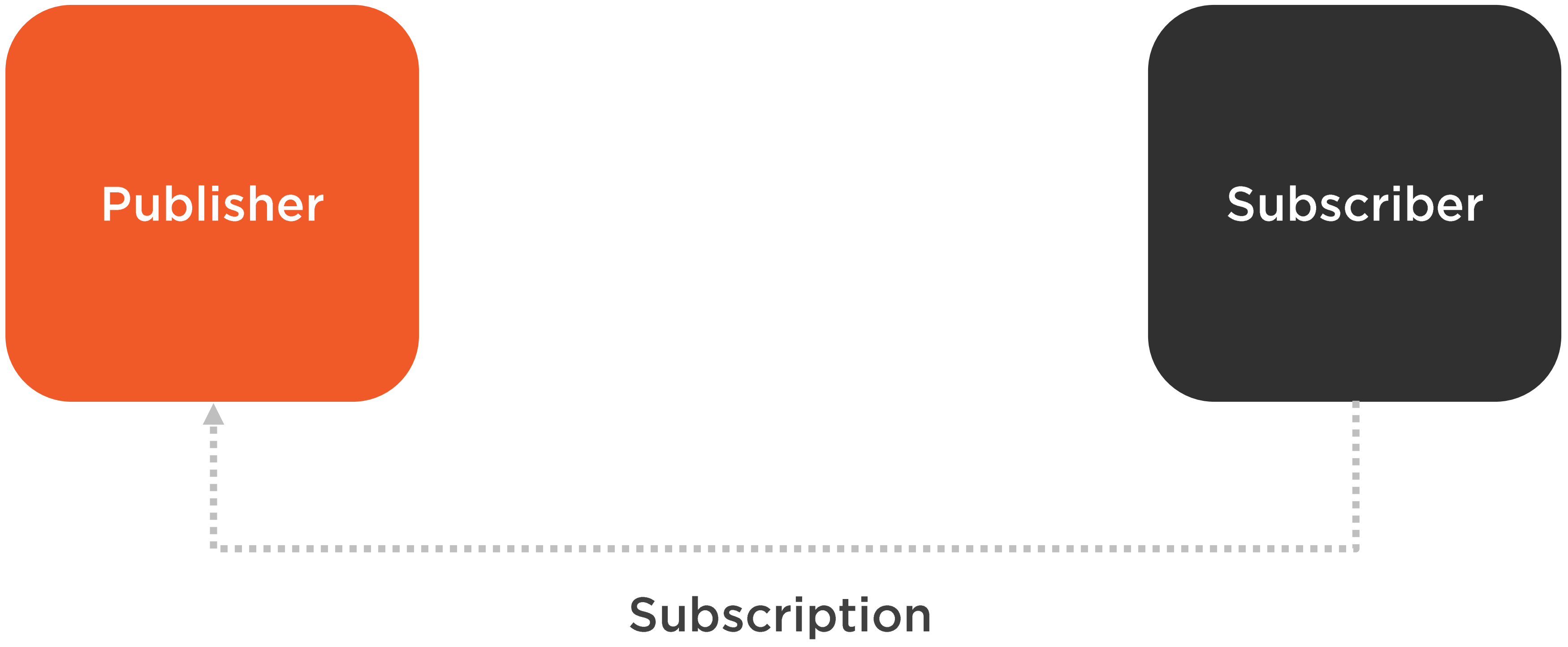
Blocking Code

```
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.

NEWS: 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,   ]

```
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, ...]

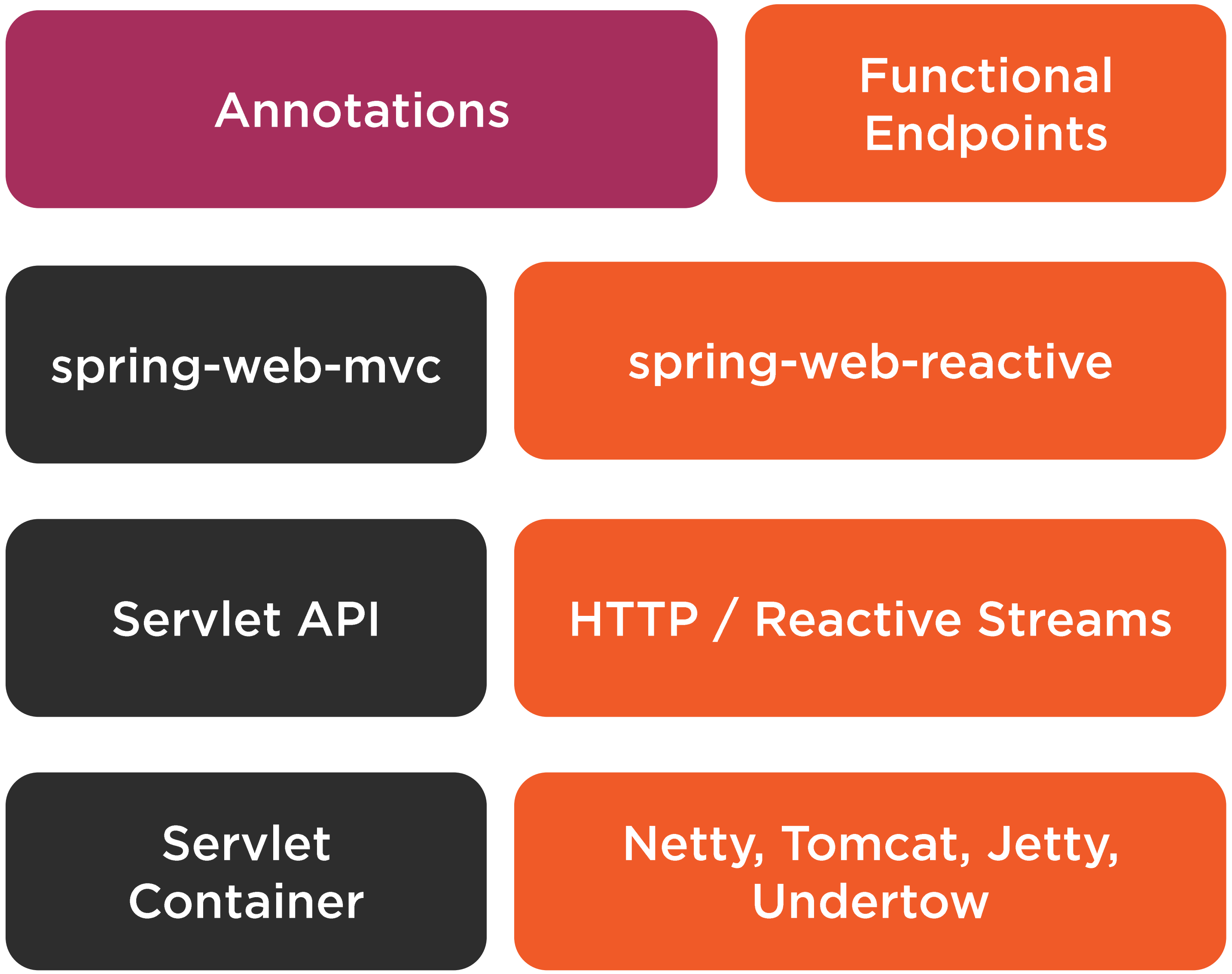
```
numberService.getNumbers()
  .map(this::convert)
  .subscribe(System.out::println)
```

Spring WebFlux

Spring 5 Web Stack

Spring MVC	Spring WebFlux
Servlet API	Reactive Streams
Blocking API	Non-blocking API (Servlet 3.1+)
Synchronous	Asynchronous
One request per thread	Concurrent connections with few threads

Spring 5 Web Stack



Annotation Style

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

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

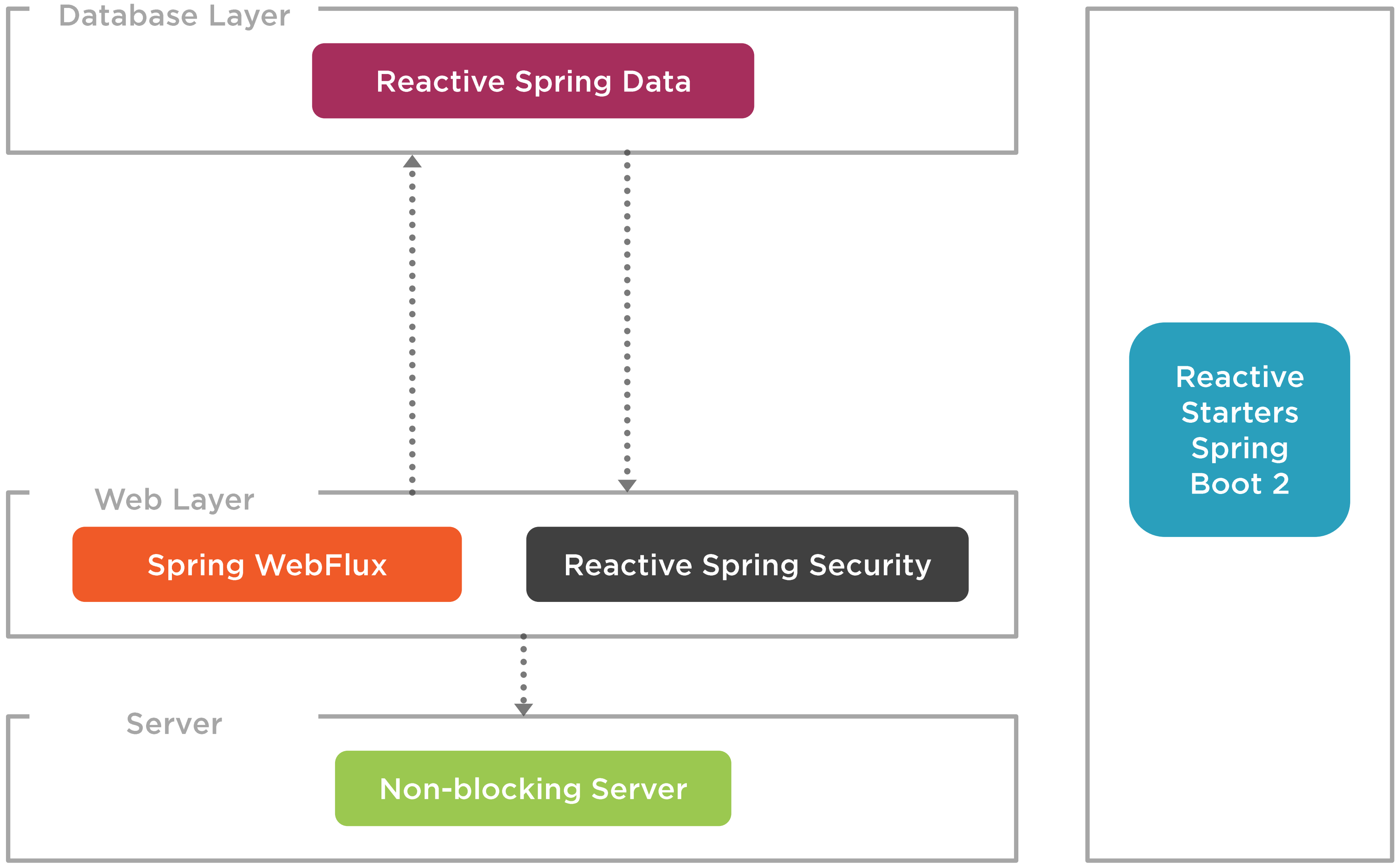
Functional Style

```
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 Spring Data Access

MongoDB

Cassandra

Redis

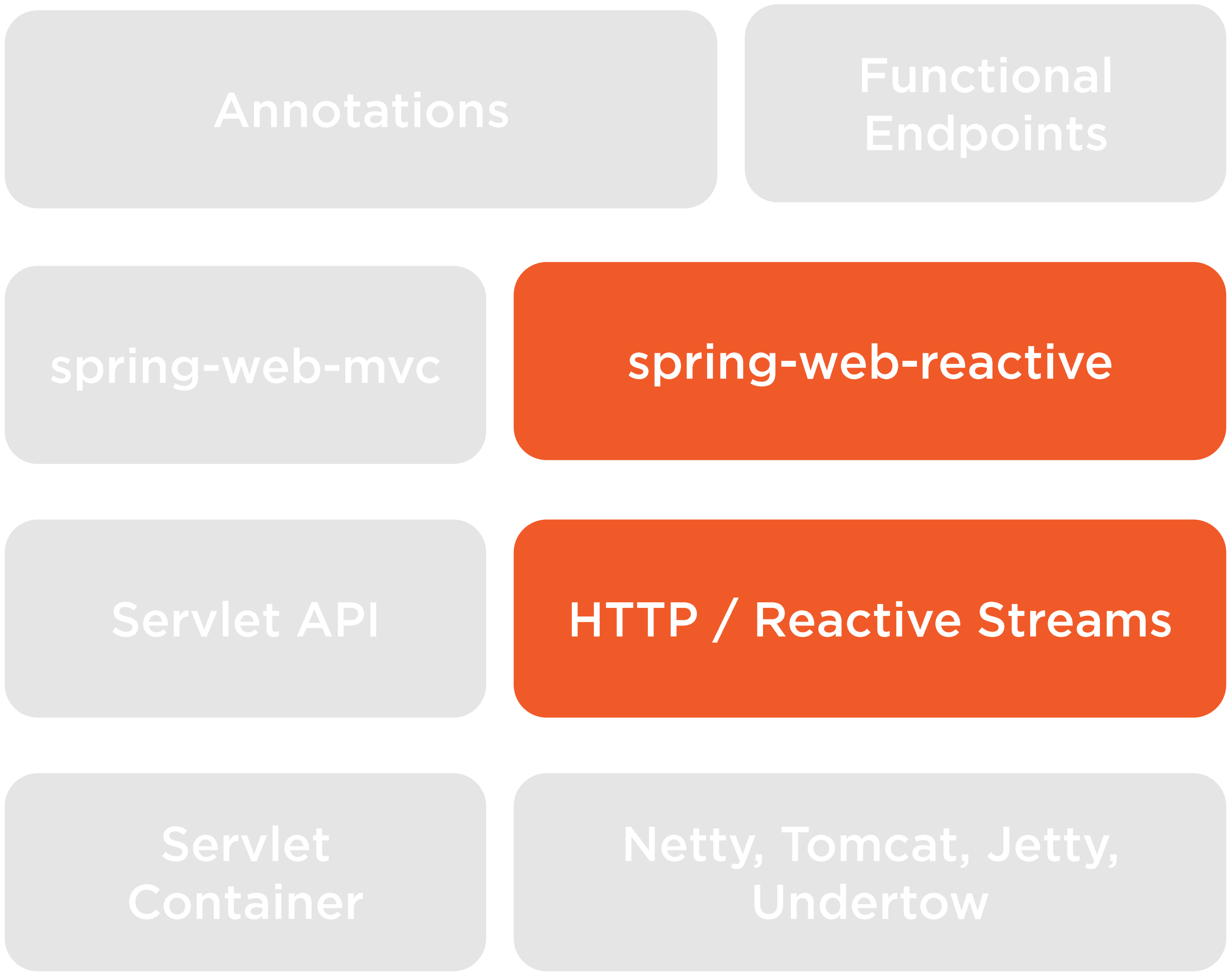
Couchbase

Reactive data access needs an async database driver

Reactive Programming Model

- Asynchronous
- Non-blocking
- 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.

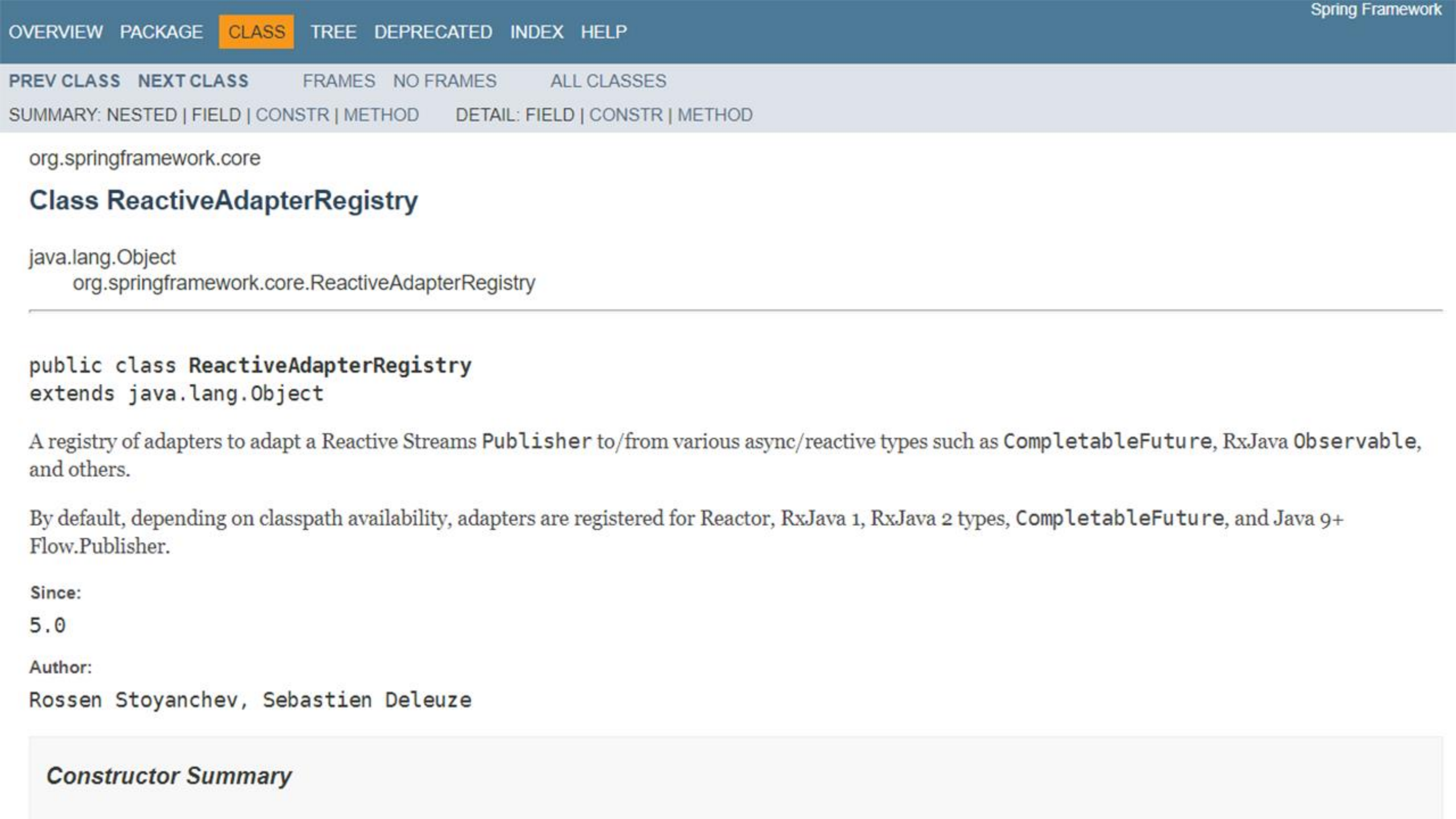
NEWS: JDK9 `java.util.concurrent.Flow`

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 java.lang.Object
```

A registry of adapters to adapt a 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 1, RxJava 2 types, `CompletableFuture`, and Java 9+ Flow.Publisher.

Since:

5.0

Author:

Rossen Stoyanchev, Sebastien Deleuze

Constructor Summary

Reactive Support

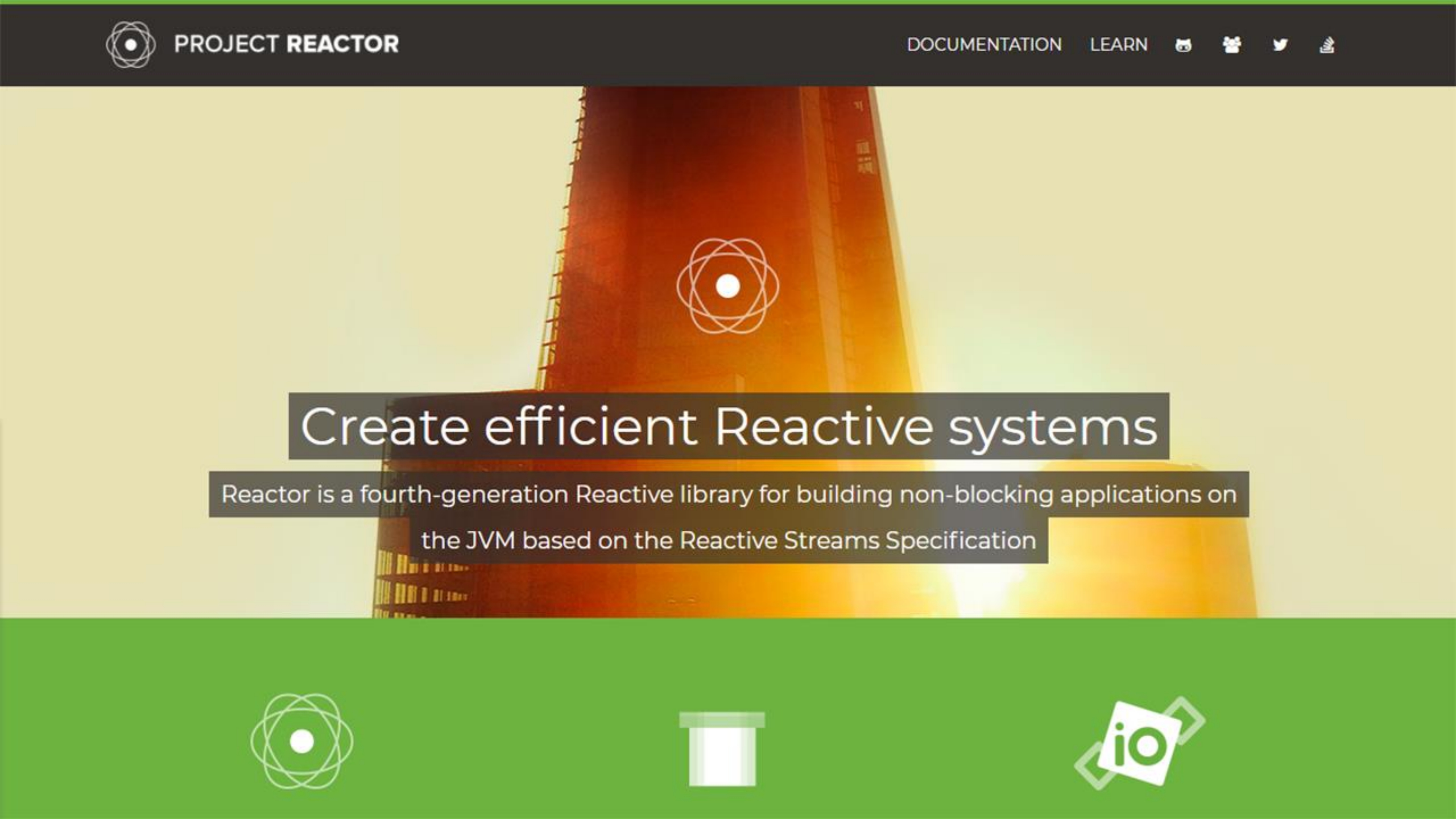
Reactor

RxJava

RxJava2

CompletableFuture

Java 9 Flow API


PROJECT **REACTOR**

DOCUMENTATION LEARN

Create efficient Reactive systems

Reactor is a fourth-generation Reactive library for building non-blocking applications on the JVM based on the Reactive Streams Specification

Things to Remember

Reactive programming

- Different than Reactive systems
- Non-blocking
- Asynchronous
- Functional/Declarative

Spring WebFlux

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