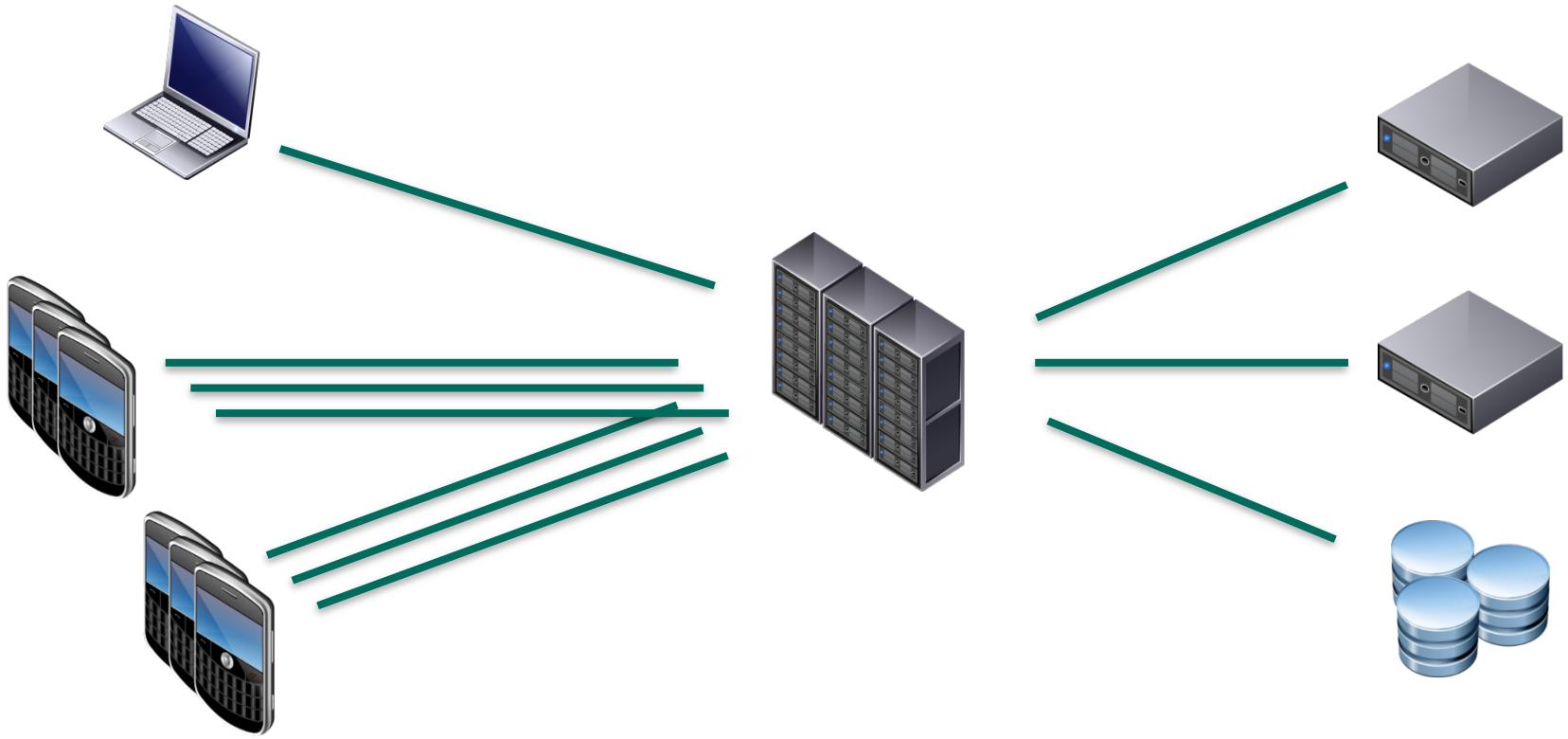
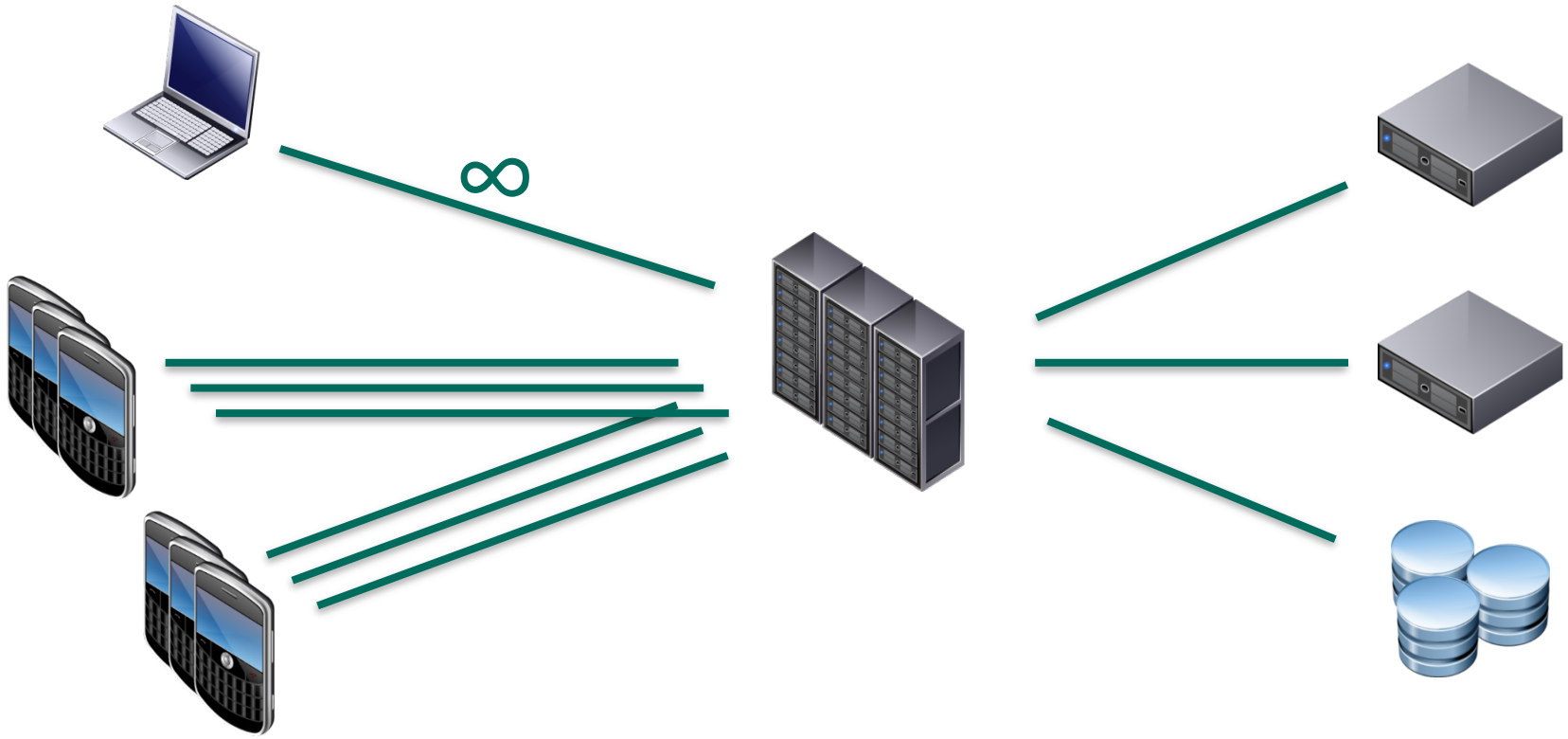
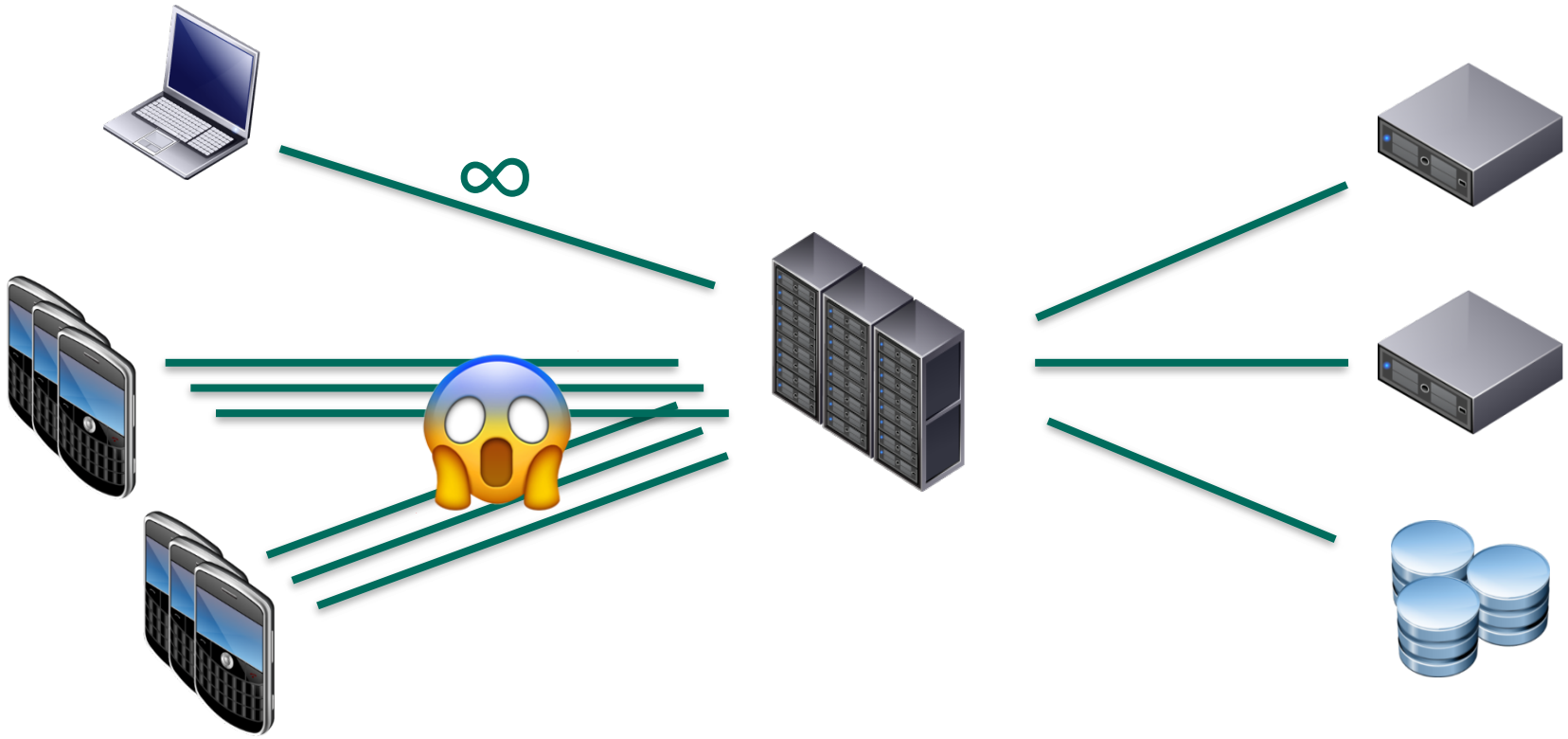
Reactive Spring

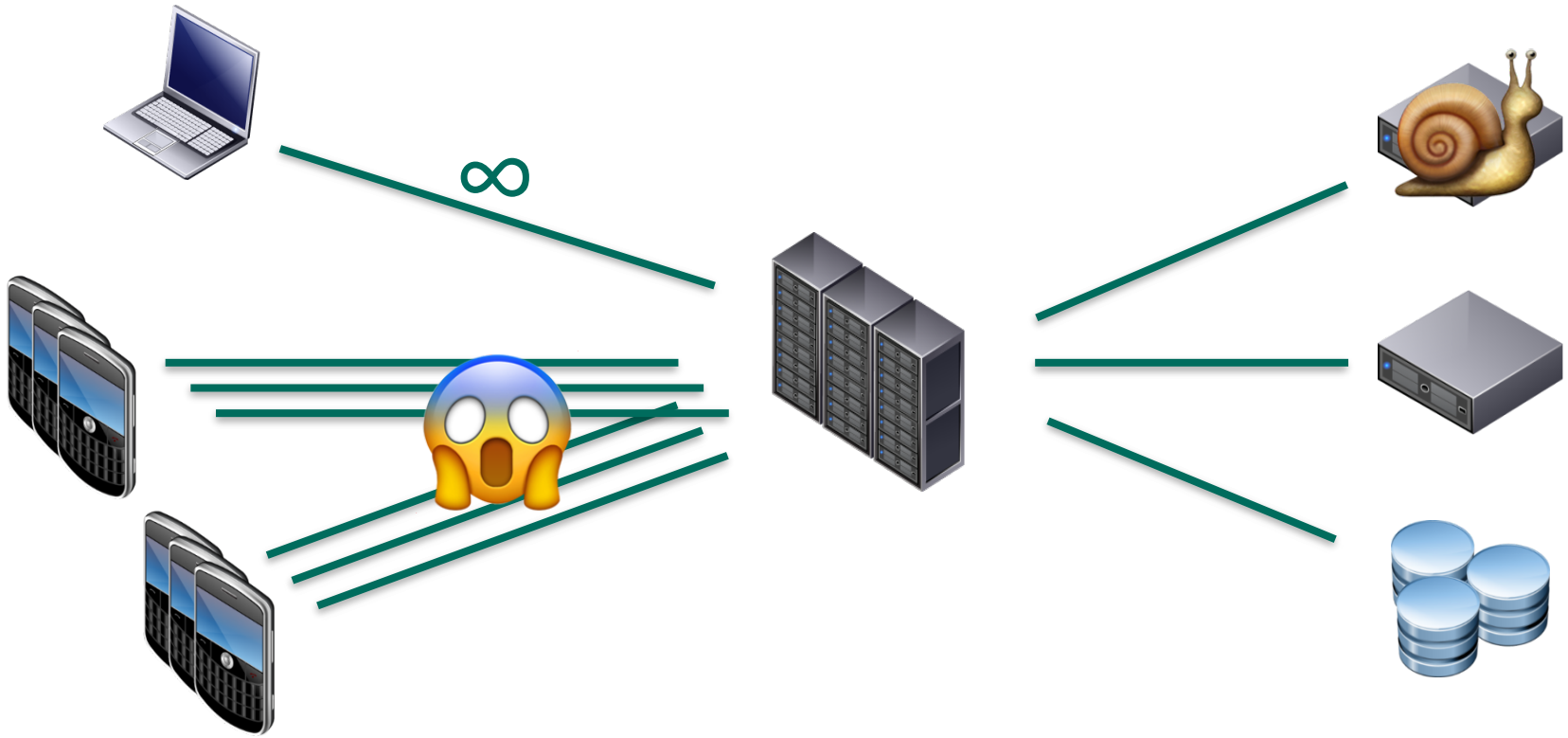
Sébastien Deleuze - @sdeleuze
Brian Clozel - @bclozel

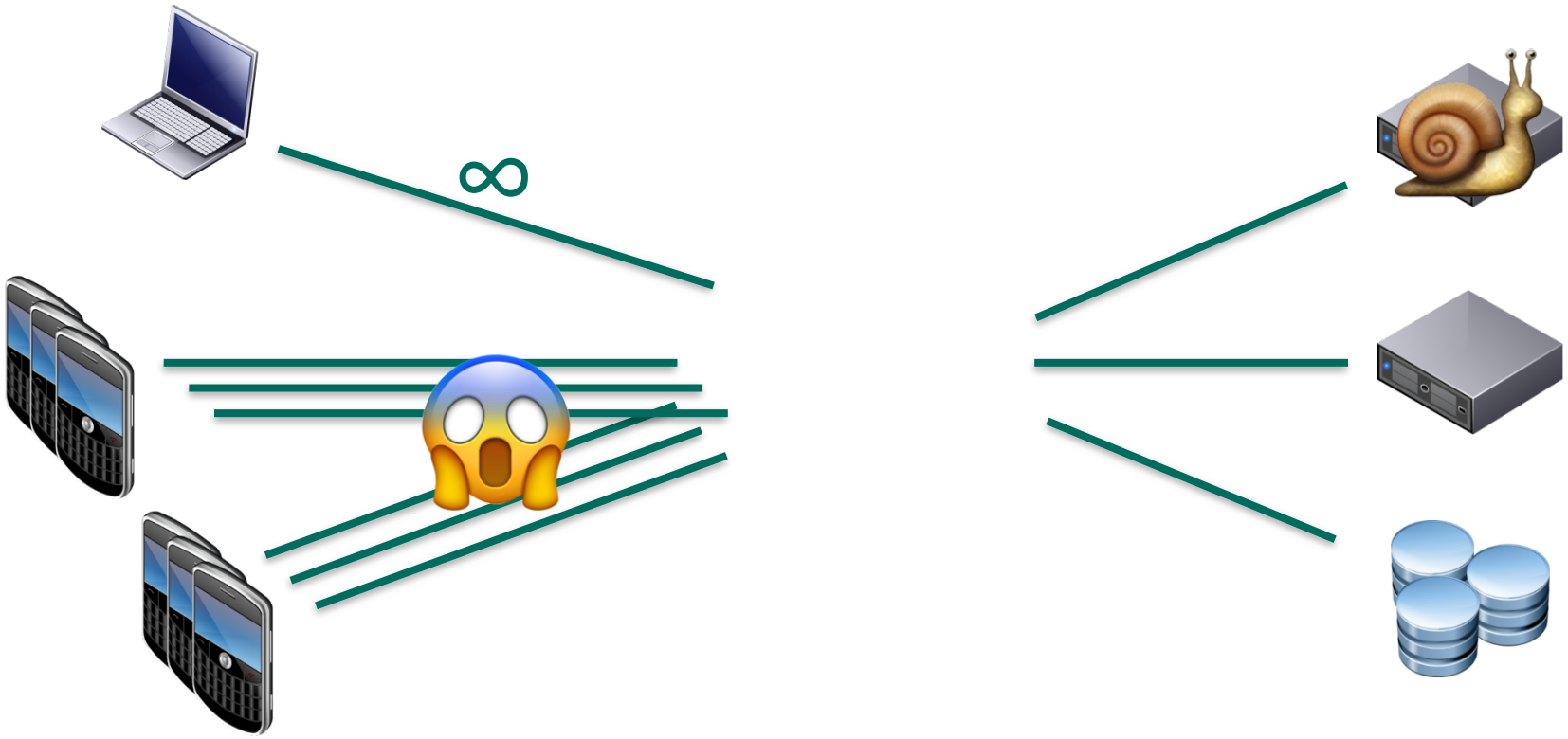
Pivotal



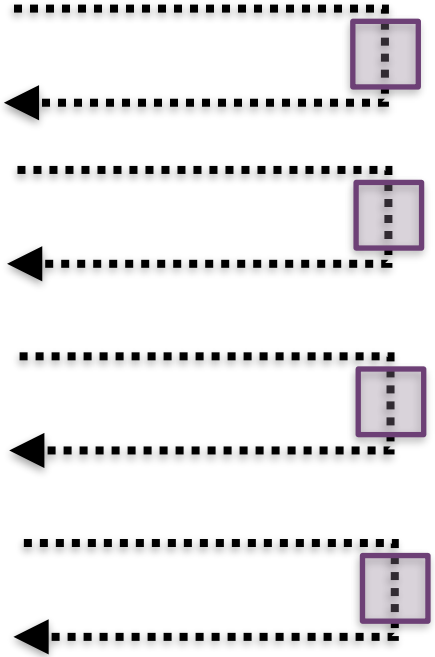


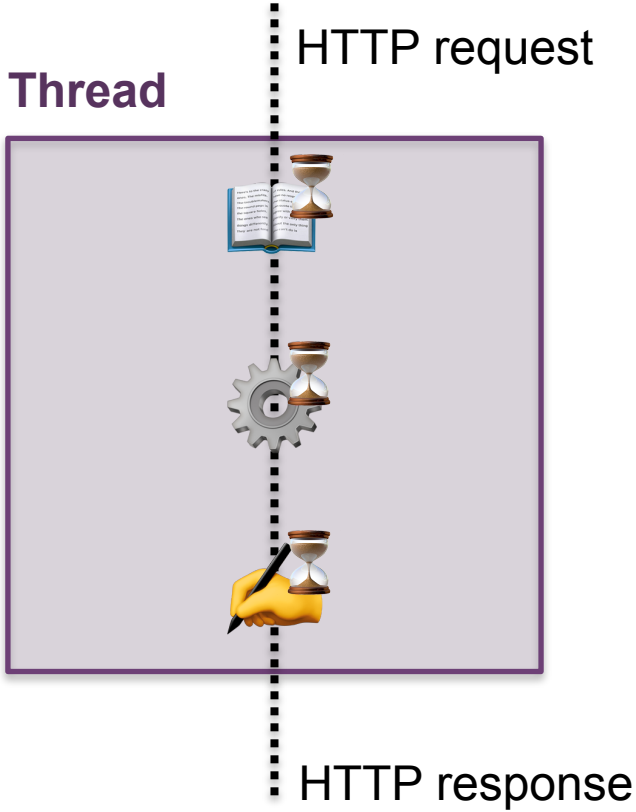




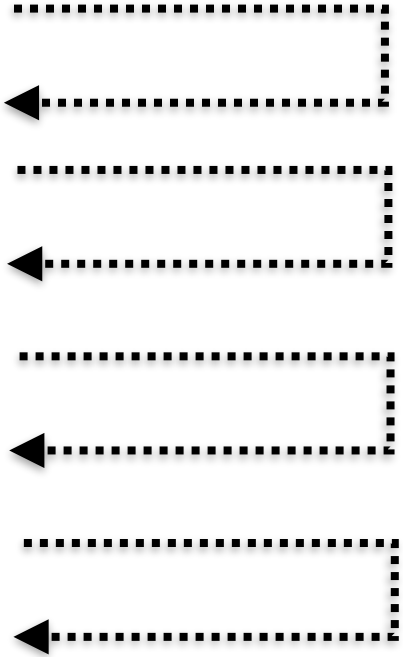


Blocking + Thread pools



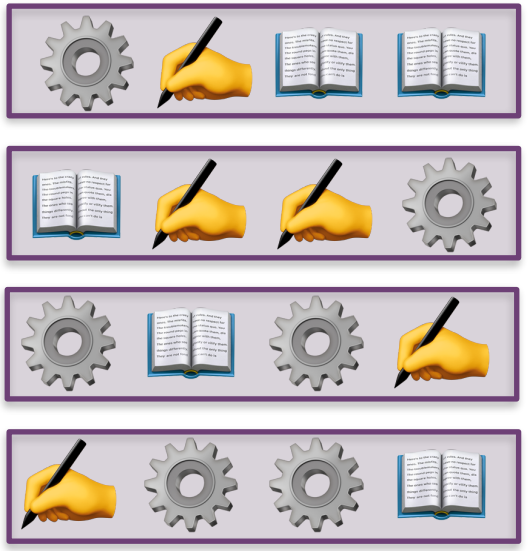


Non-blocking and event-loop



IO Selector Thread

Worker Threads



From blocking to event-based

Neutral to latency

Reactive Foundations

Going Reactive

More for **scalability** and **stability** than for speed

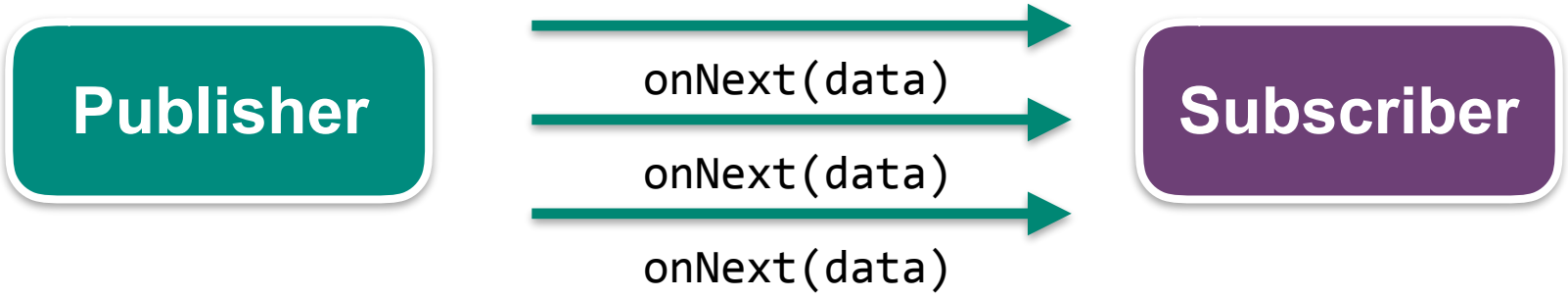
Reactive Streams

Reactive Streams

Reactive Streams

Backpressure

Reactive Streams



Backpressure

Reactive Streams

Backpressure

Reactive Streams API

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

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

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

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

Reactive Streams based libraries:

Reactive Streams based libraries:


RxJava


Reactor

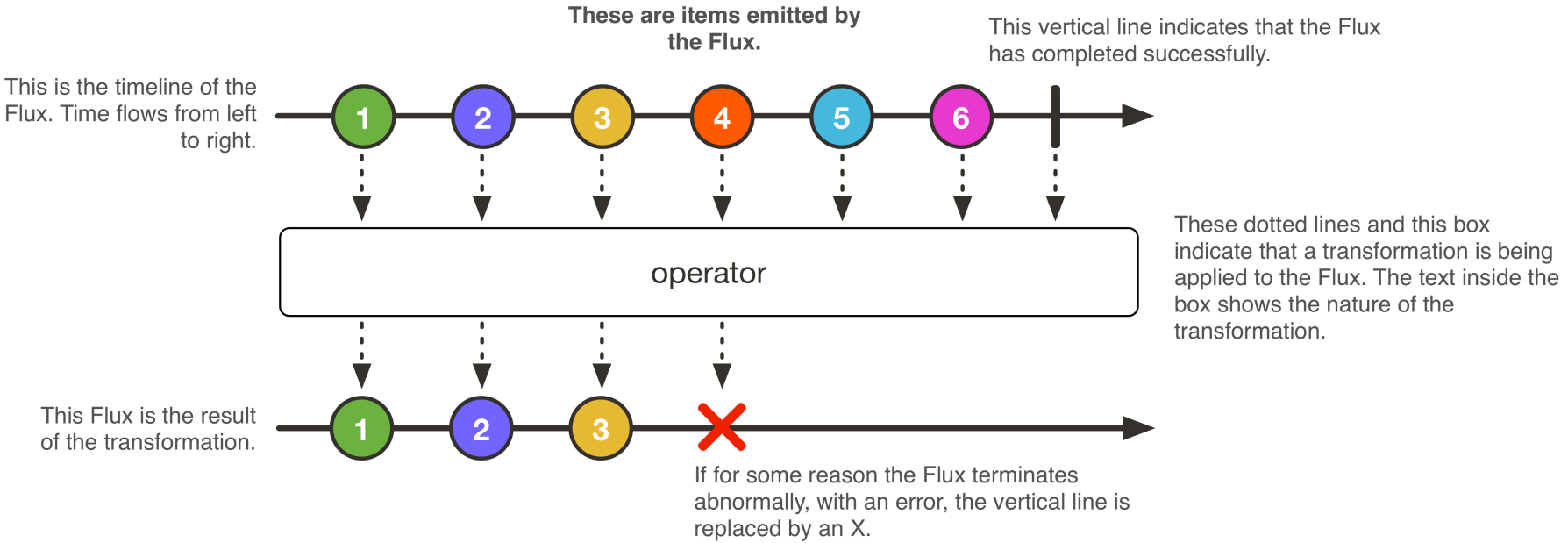

Akka Streams

Reactor is a reactive API
based on reactive streams and Java 8

`Flux<T>`

`Mono<T>`

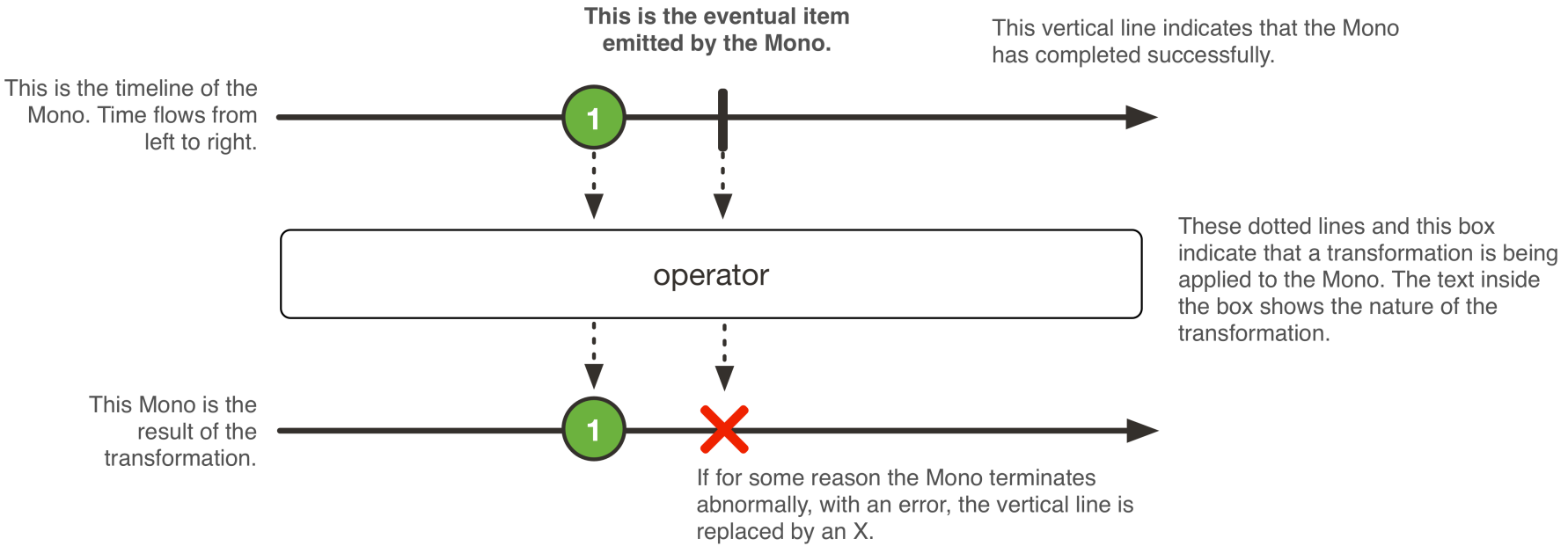
`Flux<T>` is a `Publisher<T>` for 0..n elements



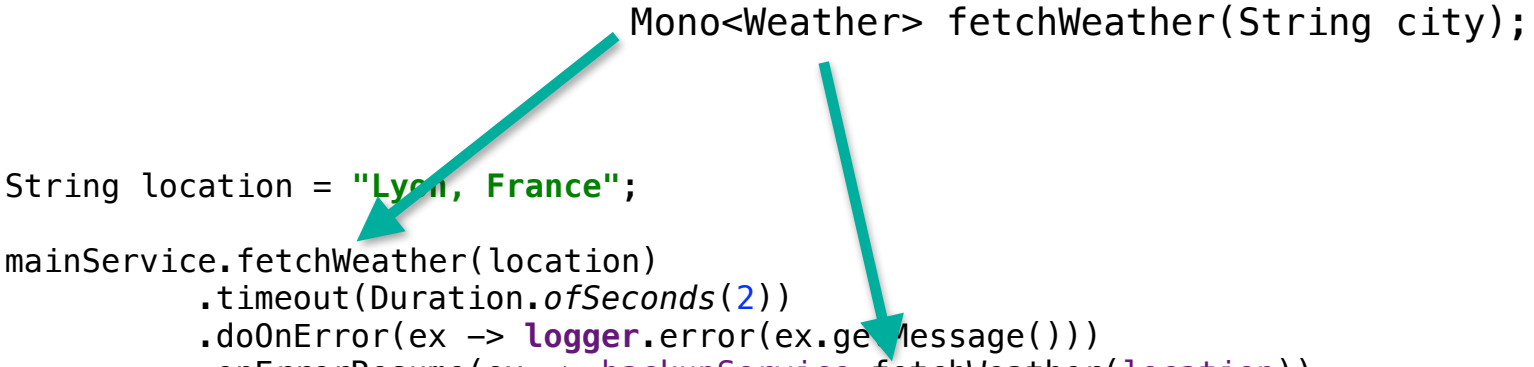
`Flux<T>`

`Mono<T>`

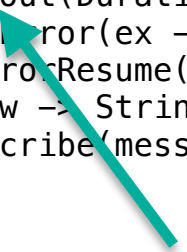
Mono<T> is a Publisher<T> for 0..1 element



```
String location = "Lyon, France";

mainService.fetchWeather(location)
          .timeout(Duration.ofSeconds(2))
          .doOnError(ex -> logger.error(ex.getMessage()))
          .onErrorResume(ex -> backupService.fetchWeather(location))
          .map(w -> String.format("Weather in %s is %s", w.getLocation(), w.getDescription()))
          .subscribe(message -> logger.info(message));
```

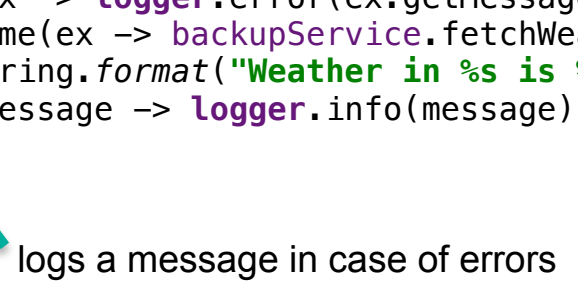

Mono<Weather> fetchWeather(String city);
String location = "Lyon, France";
mainService.fetchWeather(location)
.timeout(Duration.ofSeconds(2))
.doOnError(ex -> logger.error(ex.getMessage()))
.onErrorResume(ex -> backupService.fetchWeather(location))
.map(w -> String.format("Weather in %s is %s", w.getLocation(), w.getDescription()))
.subscribe(message -> logger.info(message));

```
String location = "Lyon, France";

mainService.fetchWeather(location)
           .timeout(Duration.ofSeconds(2))
           .doOnError(ex -> logger.error(ex.getMessage()))
           .onErrorResume(ex -> backupService.fetchWeather(location))
           .map(w -> String.format("Weather in %s is %s", w.getLocation(), w.getDescription()))
           .subscribe(message -> logger.info(message));
```

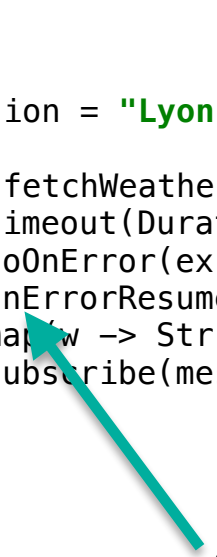


times out and emits an error after 2 sec

```
String location = "Lyon, France";

mainService.fetchWeather(location)
           .timeout(Duration.ofSeconds(2))
           .doOnError(ex -> logger.error(ex.getMessage()))
           .onErrorResume(ex -> backupService.fetchWeather(location))
           .map(w -> String.format("Weather in %s is %s", w.getLocation(), w.getDescription()))
           .subscribe(message -> logger.info(message));
```

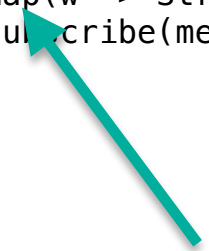


```
String location = "Lyon, France";

mainService.fetchWeather(location)
           .timeout(Duration.ofSeconds(2))
           .doOnError(ex -> logger.error(ex.getMessage()))
           .onErrorResume(ex -> backupService.fetchWeather(location))
           .map(w -> String.format("Weather in %s is %s", w.getLocation(), w.getDescription()))
           .subscribe(message -> logger.info(message));
```



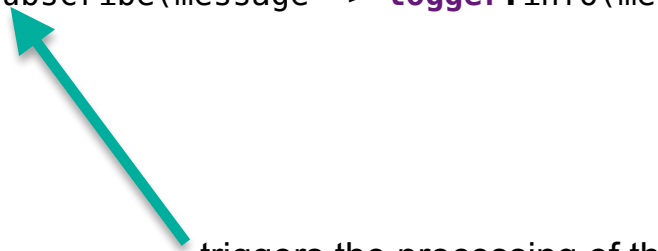
switches to a different service in case of error

```
String location = "Lyon, France";

mainService.fetchWeather(location)
           .timeout(Duration.ofSeconds(2))
           .doOnError(ex -> logger.error(ex.getMessage()))
           .onErrorResume(ex -> backupService.fetchWeather(location))
           .map(w -> String.format("Weather in %s is %s", w.getLocation(), w.getDescription()))
           .subscribe(message -> logger.info(message));
```



transforms a weather instance into a String message

```
String location = "Lyon, France";

mainService.fetchWeather(location)
           .timeout(Duration.ofSeconds(2))
           .doOnError(ex -> logger.error(ex.getMessage()))
           .onErrorResume(ex -> backupService.fetchWeather(location))
           .map(w -> String.format("Weather in %s is %s", w.getLocation(), w.getDescription()))
           .subscribe(message -> logger.info(message));
```



triggers the processing of the chain

Building Reactive applications

Blocking API

```
interface UserRepository {
    User findOne(String id);
    List<User> findAll();
    void save(User user);
}
```

Reactive API

```
interface ReactiveUserRepository {
    Mono<User> findOne(String id);
    Flux<User> findAll();
    Mono<Void> save(Mono<User> user);
}
```

Blocking API

```
interface UserRepository {
    User findOne(String id);
    List<User> findAll();
    void save(User user);
}
```

Blocking API

```
interface UserRepository {
    User findOne(String id);
    List<User> findAll();
    void save(User user);
}
```

Method returns when whole list is received

Blocking API

```
interface UserRepository {
    User findOne(String id);
    List<User> findAll();
    void save(User user);
}
```

Throws exception if an error happens

Blocking API

```
interface UserRepository {
    User findOne(String id);
    List<User> findAll();
    void save(User user);
}
```

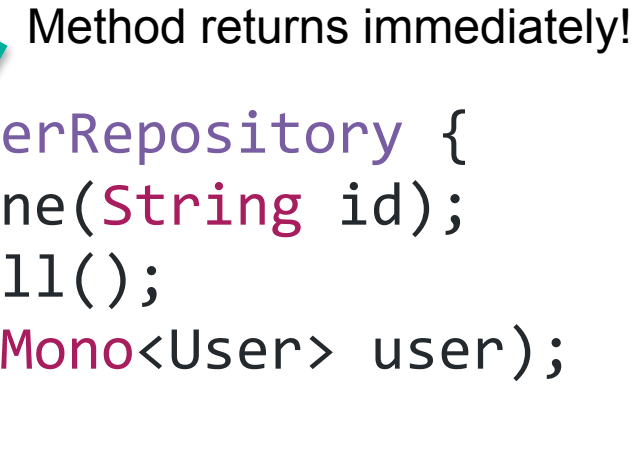
Reactive API

```
interface ReactiveUserRepository {
    Mono<User> findOne(String id);
    Flux<User> findAll();
    Mono<Void> save(Mono<User> user);
}
```

Reactive API

```
interface ReactiveUserRepository {
    Mono<User> findOne(String id);
    Flux<User> findAll();
    Mono<Void> save(Mono<User> user);
}
```

Reactive API



Reactive API

```
interface ReactiveUserRepository {
    Mono<User> findOne(String id);
    Flux<User> findAll();
    Mono<Void> save(Mono<User> user);
}
```

Elements are received as soon as they are available, ending with a success "onComplete" event.

Reactive API

```
interface ReactiveUserRepository {
    Mono<User> findOne(String id);
    Flux<User> findAll();
    Mono<Void> save(Mono<User> user);
}
```

Mono<Void> means error or success,
no data will be received

Flux<T>:

Async collection vs. Stream of data

```
Flux<User>
```

`Flux<User>` as a JSON array

Flux<User> as a JSON array

```
GET /users
…
200 OK
Content-Type: application/json

 [
   {"a": "foo", "b": "bar"},
   {"a": "baz", "b": "boo"}
 ]
```

Flux<User> as Server Sent Events

Flux<User> as Server Sent Events

```
GET /users
…
200 OK
Content-Type: text/event-stream

data: {"a": "foo", "b": "bar"}
data: {"a": "baz", "b": "boo"}
data: {"a": "dev", "b": "oxx"}
data: {"a": "lov", "b": "es"}
data: {"a": "spr", "b": "ing"}
data: {"a": "and", "b": "more"}
data: {"a": "data", "b": "coming"}
```

`Flux<User>` as a JSON stream

Flux<User> as a JSON stream

```
GET /users
…
200 OK
Content-Type: application/stream+json

{"a": "foo", "b": "bar"}
{"a": "baz", "b": "boo"}
{"a": "dev", "b": "oxx"}
{"a": "lov", "b": "es"}
{"a": "spr", "b": "ing"}
{"a": "and", "b": "more"}
{"a": "data", "b": "coming"}
```

Reactive Spring

Spring Framework 5 uses

Reactor for its reactive foundations

Spring
Framework

Spring
Data

Spring
Security

Spring
Framework

Spring
Cloud

Spring
Boot

With Reactive Spring, you can use your reactive library of choice for your application:

With Reactive Spring, you can use your reactive library of choice for your application:

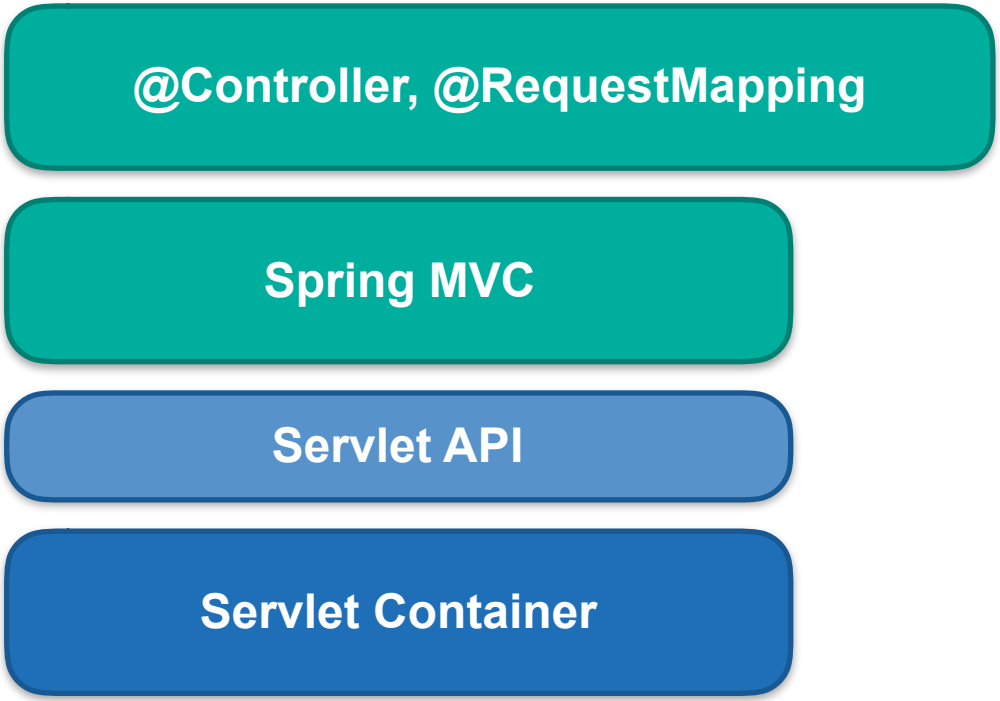
 RxJava

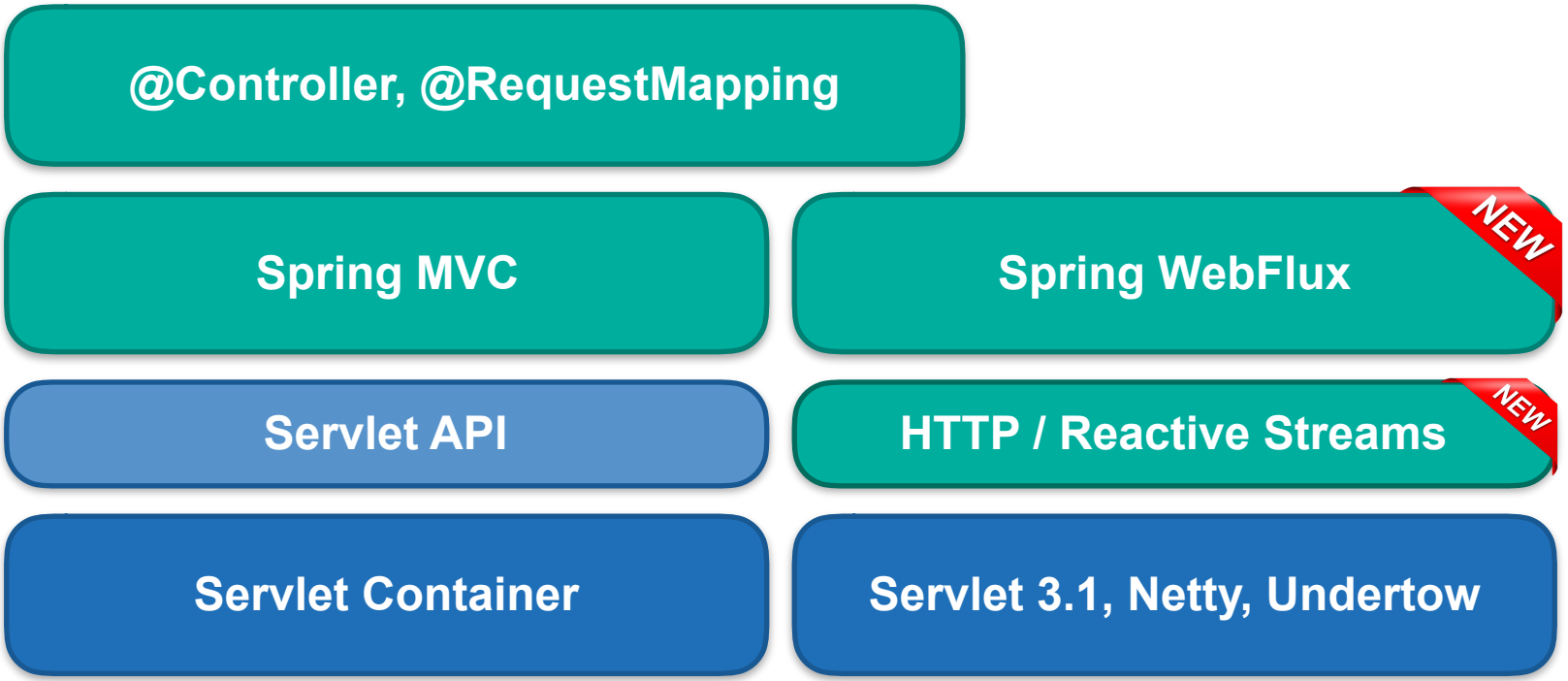
 Reactor

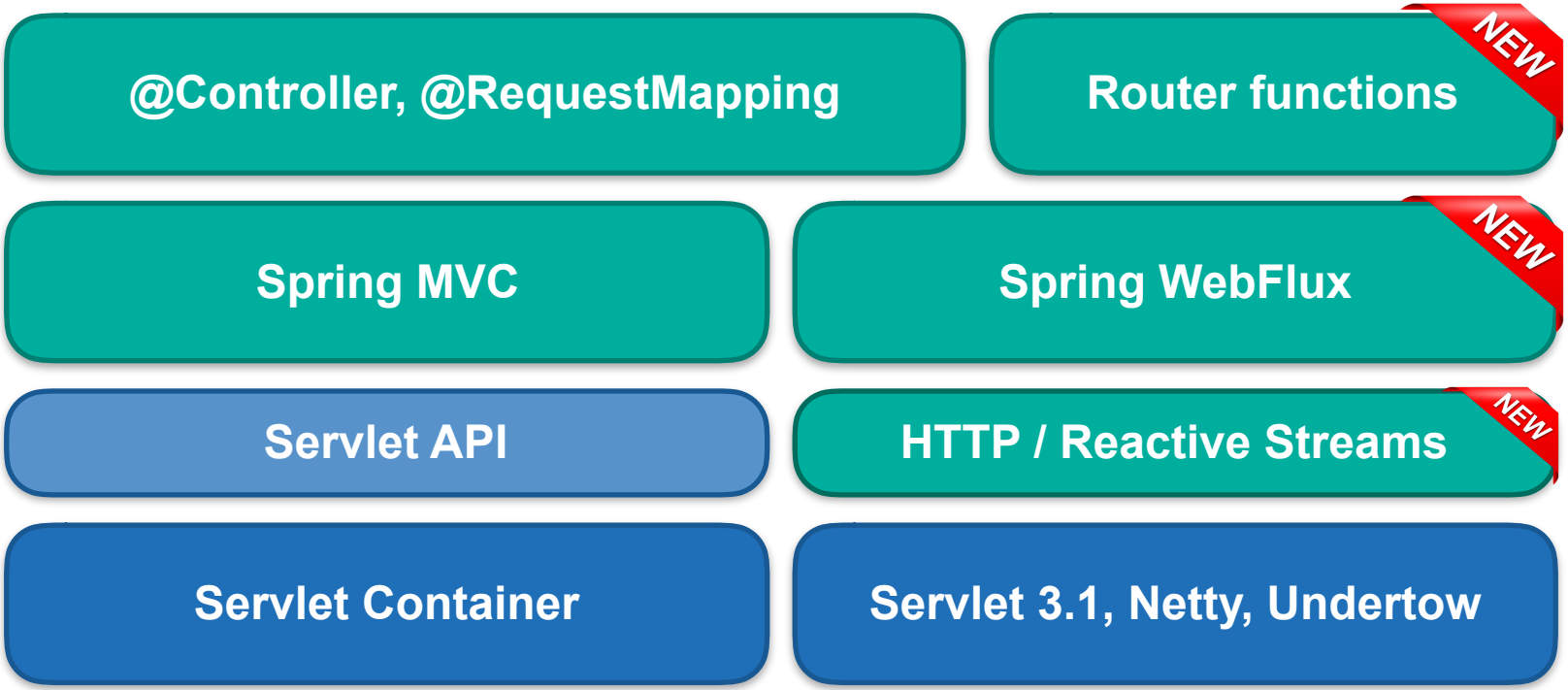
 Akka Streams


Reactor

Reactor

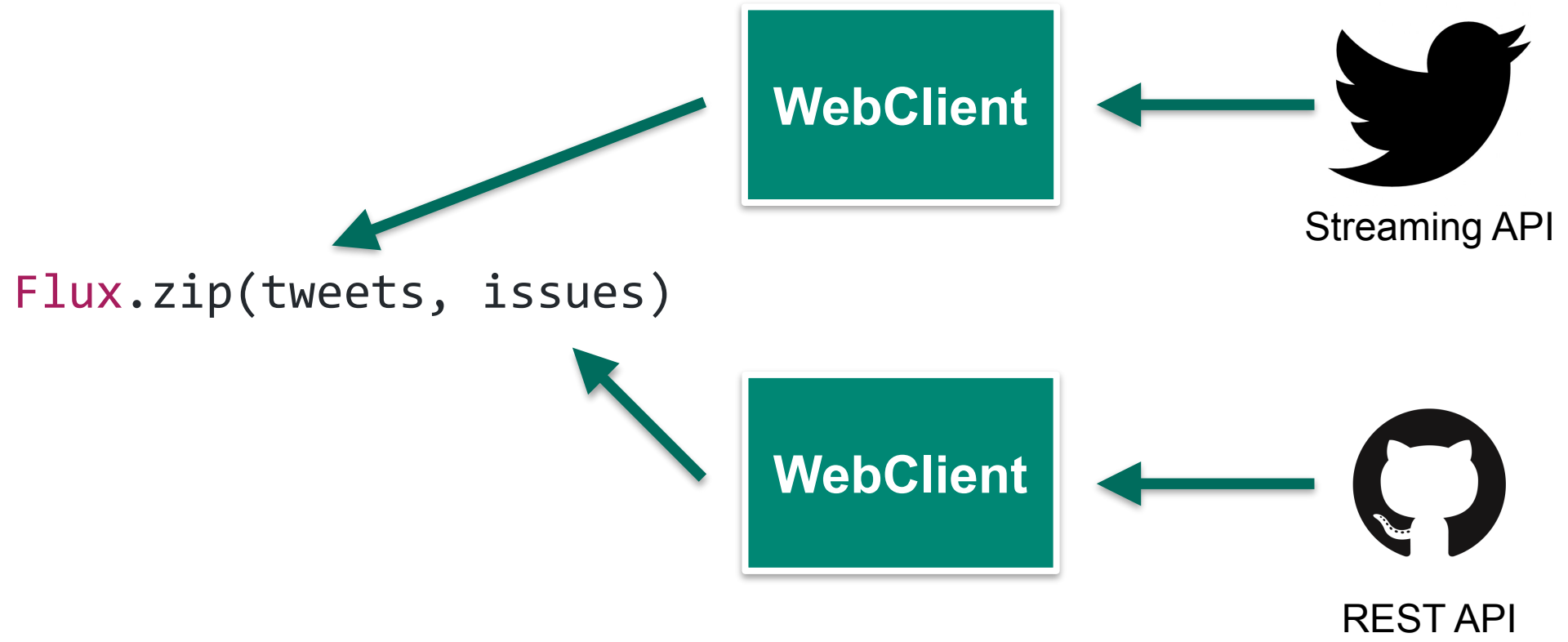

@Controller, @RequestMapping
Spring MVC
Servlet API
Servlet Container

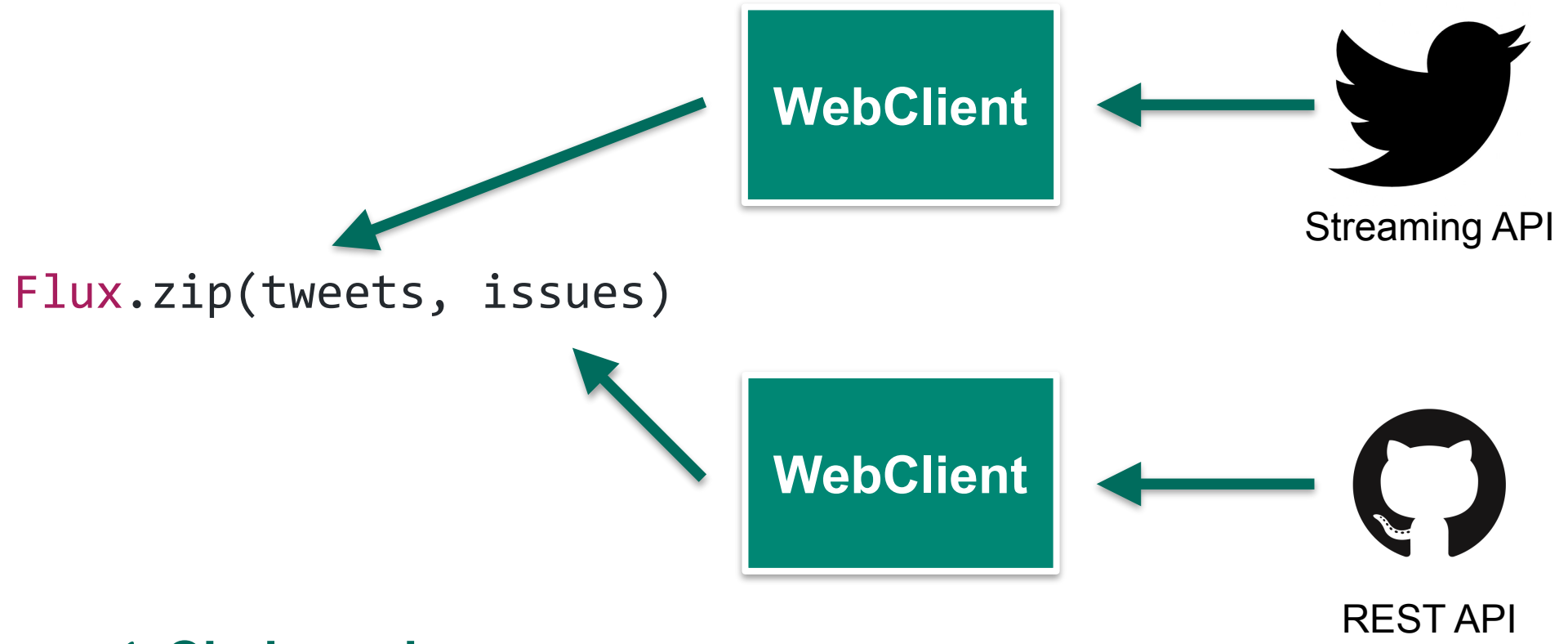

@Controller, @RequestMapping
Spring MVC
Spring WebFlux
NEW
Servlet API
HTTP / Reactive Streams
NEW
Servlet Container
Servlet 3.1, Netty, Undertow


@Controller, @RequestMapping
Router functions
NEW
Spring MVC
Spring WebFlux
NEW
Servlet API
HTTP / Reactive Streams
NEW
Servlet Container
Servlet 3.1, Netty, Undertow


Flux<Tweet>
WebClient
Streaming API

- ✔ **Reactive API**
- ✔ **Can consume (infinite) streams**


WebClient
Streaming API
Flux.zip(tweets, issues)
WebClient
REST API



✔ **Chain and compose**

WebFlux

WebFlux

WebFlux

- ✔ Shared resources (event loop, buffers)
- ✔ Built-in mocking capabilities

WebClient,
a nice RestTemplate alternative

WebClient,
a nice RestTemplate alternative

- ✔ Fluent and flexible API
- ✔ Easy configuration

WebFlux demo! 😉

https://github.com/bclozel/webflux-streaming-showcase

WebFlux,
the functional way 😍🙃

- `RouterFunction`
- `RequestPredicate`
- `HandlerFunction`
- `HandlerFilterFunction`

- **RouterFunction**

```
Mono<HandlerFunction<ServerResponse>> route(ServerRequest req)
```

- **RequestPredicate**
- **HandlerFunction**
- **HandlerFilterFunction**

- **RouterFunction**

- **RequestPredicate**

```
boolean test(ServerRequest req)
```

- **HandlerFunction**

- **HandlerFilterFunction**

- **RouterFunction**
- **RequestPredicate**
- **HandlerFunction**

```
Mono<ServerResponse> handle(ServerRequest req)
```

- **HandlerFilterFunction**

- **RouterFunction**
- **RequestPredicate**
- **HandlerFunction**
- **HandlerFilterFunction**

```
Mono<ServerResponse> filter(ServerRequest req,
                            HandlerFunction<ServerResponse> next)
```

- `RouterFunction`
- `RequestPredicate`
- `HandlerFunction`
- `HandlerFilterFunction`

```java
// Annotation-based Java
@RequestMapping("/quotes/feed", produces = TEXT_EVENT_STREAM_VALUE)
public Flux<Quote> fetchQuotesStream() { ... }
```

```java
// Annotation-based Java
@RequestMapping("/quotes/feed", produces = TEXT_EVENT_STREAM_VALUE)
public Flux<Quote> fetchQuotesStream() { ... }
```

```java
// Functional Java without static imports
RouterFunctions.route(
        RequestPredicates.path("/quotes/feed")
        .and(RequestPredicates.accept(TEXT_EVENT_STREAM)),
        { ... }
)
```

```java
// Annotation-based Java
@RequestMapping("/quotes/feed", produces = TEXT_EVENT_STREAM_VALUE)
public Flux<Quote> fetchQuotesStream() { ... }
```

```java
// Functional Java with static imports
route(
      path("/quotes/feed").and(accept(TEXT_EVENT_STREAM)),
      { ... }
)
```

```java
// Annotation-based Java
@RequestMapping("/quotes/feed", produces = TEXT_EVENT_STREAM_VALUE)
public Flux<Quote> fetchQuotesStream() { ... }
```

```kotlin
// Functional Kotlin
router {
    "/quotes/feed" and accept(TEXT_EVENT_STREAM) { ... }
}
```

```java
@RestController
public class UserController {

    @GetMapping("/users")
    public Flux<User> fetchAll() {
        // ...
    }

    @GetMapping("/users/{id}")
    public Mono<User> fetch(@PathVariable String userId) {
        // ...
    }
}
```

```
@Configuration
class ApplicationRoutes(val userHandler: UserHandler,
                        val blogHandler: BlogHandler,
                        val shopRepository: ShopRepository) {
    @Bean
    fun appRouter() = router {
        GET("/users", userHandler::fetchAll)
        GET("/users/{id}", userHandler::fetch)
    }

    @Bean
    fun nestedRouter() = router { ... }

    @Bean
    fun dynamicRouter() = router { ... }
}
```

```
@Bean
fun nestedRouter() = router {
   ("/blog" and accept(TEXT_HTML)).nest {
      GET("/", blogHandler::findAllView)
      GET("/{slug}", blogHandler::findOneView)
    }
    ("/api/blog" and accept(APPLICATION_JSON)).nest {
      GET("/", blogHandler::findAll)
      GET("/{id}", blogHandler::findOne)
      POST("/", blogHandler::create)
    }
}
@Bean
fun dynamicRouter() = router { ... }
```

```
@Bean
fun dynamicRouter() = router {
    shopRepository.findAll()
                  .toIterable()
                  .forEach { shop ->
                      GET("/${shop.id}") {
                          req -> shopHandler.homepage(shop, req)
                      }
                  }
}
```

Spring ❤️ Kotlin

Introducing Kotlin support in Spring Framework 5.0

Following the Kotlin support on start.spring.io we introduced a few months ago, we have continued to work to ensure that Spring and Kotlin play well together. One of the key strengths ...

spring.io

RETWEETS 203 J'AIME 227

15:06 - 4 janv. 2017

3 203 227

ServerResponse extensions provided in spring-webflux.jar

```
inline fun <reified T : Any> ServerResponse.BodyBuilder.body(publisher: Publisher<T>) =
    body(publisher, T::class.java)
```

ServerResponse extensions provided in spring-webflux.jar

```
inline fun <reified T : Any> ServerResponse.BodyBuilder.body(publisher: Publisher<T>) =
    body(publisher, T::class.java)
```

Extend existing Spring Java type

ServerResponse extensions provided in spring-webflux.jar

```
inline fun <reified T : Any> ServerResponse.BodyBuilder.body(publisher: Publisher<T>) =
    body(publisher, T::class.java)
```

Extend existing Spring Java type

Need to specify the type because of type erasure

```
Mono<ServerResponse> findAll(ServerRequest request) {
  return ServerResponse.ok().body(repository.findAll(), User.class);
}
```

ServerResponse extensions provided in spring-webflux.jar

```
inline fun <reified T : Any> ServerResponse.BodyBuilder.body(publisher: Publisher<T>) =
    body(publisher, T::class.java)
```

Extend existing Spring Java type

No need to specify explicitly types thanks to Kotlin reified type parameters

```
fun findAll(req: ServerRequest) = ok().body(repository.findAll())
```

Kotlin extensions bundled with Spring 5

Kotlin extensions bundled with Spring 5

WebFlux server

Spring MVC

Reactor (via reactor-kotlin-extensions)

RestTemplate

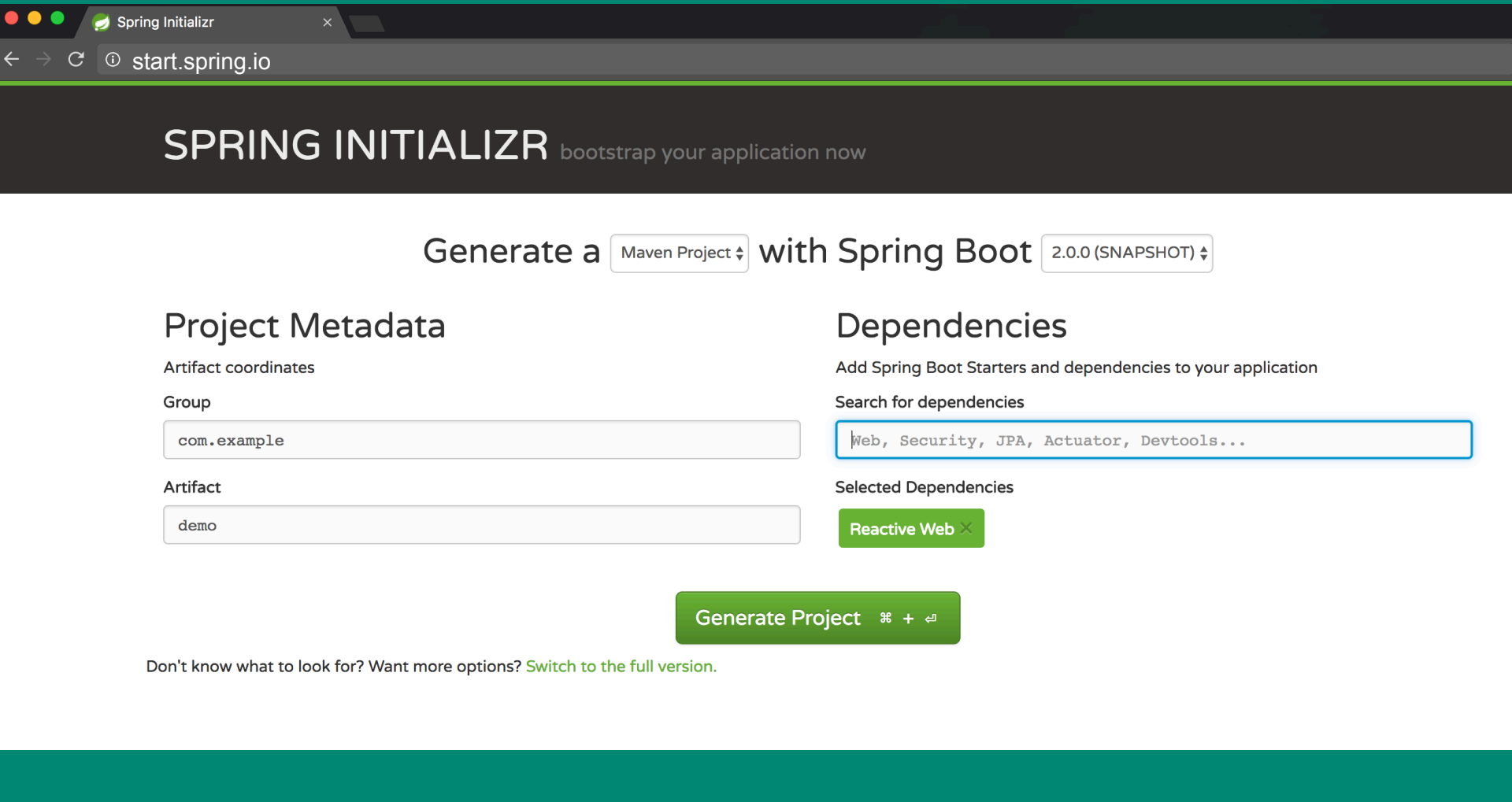
JDBC

WebFlux client

ApplicationContext

Spring Data extensions are coming

start.spring.io


Spring Initializr
start.spring.io
SPRING INITIALIZR bootstrap your application now
Generate a Maven Project with Spring Boot 2.0.0 (SNAPSHOT)
Project Metadata
Artifact coordinates
Group
com.example
Artifact
demo
Dependencies
Add Spring Boot Starters and dependencies to your application
Search for dependencies
Web, Security, JPA, Actuator, Devtools...
Selected Dependencies
Reactive Web
Generate Project ⌘ + ⏎
Don't know what to look for? Want more options? Switch to the full version.

Thank You!