Spring Reactive

Sou Alberto :)

Trabalho no grupo Caelum

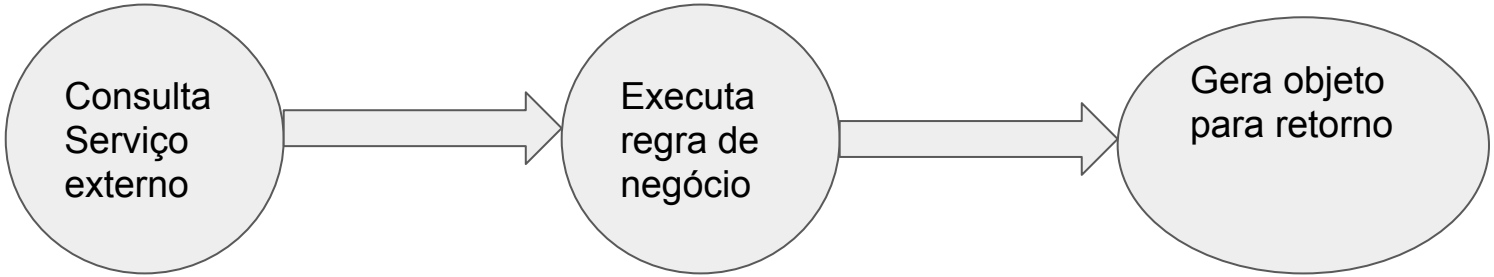
Reativo?

Muitas opiniões

Abstração do Observer

Visão mais rebuscada

Abstração para composição de eventos


Consulta
Serviço
externo
Executa
regra de
negócio
Gera objeto
para retorno

Quem implementa isso para você?

RX Java e Project Reactor

Reactive Streams

Para que serve isso?

```java
@PostMapping("/api/client/like/property/{propertyId}")
public LikedPropertyDTO like(@PathVariable("propertyId") Integer propertyId
        ,@AuthenticationPrincipal SystemUser requester){
    return propertyDao.findById(propertyId)
        .map(requester :: like)
        .map(likedPropertyDao :: save)
        .map(LikedPropertyDTO :: new)
        .orElseThrow(() -> new RuntimeException("não deu like por algum motivo"));

}
```

Onde esse código é executado?

Thread por request

E se pudesse rodar em outras threads?

Ali já tem 4 funções

```java
@PostMapping("/api/client/like/property/{propertyId}")
public LikedPropertyDTO like(@PathVariable("propertyId") Integer propertyId
        ,@AuthenticationPrincipal SystemUser requester){
    return propertyDao.findById(propertyId)
        .map(requester :: like)
        .map(likedPropertyDao :: save)
        .map(LikedPropertyDTO :: new)
        .orElseThrow(() -> new RuntimeException("não deu like por algum motivo"));

}
```

```
@PostMapping("/api/client/like/property/{propertyId}")
public Mono<LikedPropertyDTO> like(@PathVariable("propertyId") Integer propertyId,
        @AuthenticationPrincipal SystemUser requester) {
    return Mono.fromSupplier(
            () -> propertyDao.findById(propertyId)
            .orElseThrow(() -> new NotFoundException()))
           .map(requester::like)
           .map(likedPropertyDao::save)
           .map(LikedPropertyDTO::new);

}
```

Mono?

Publicador de apenas uma informação

```
@PostMapping("/api/client/like/property/{propertyId}")
public Mono<LikedPropertyDTO> like(@PathVariable("propertyId") Integer propertyId,
        @AuthenticationPrincipal SystemUser requester) {
    Mono<Property> publicador = Mono.fromSupplier(
                    () -> propertyDao.findById(propertyId)
                    .orElseThrow(() -> new NotFoundException()));
    return publicador
            .map(requester::like)
            .map(likedPropertyDao::save)
            .map(LikedPropertyDTO::new);

}
```

Publicador de várias informações

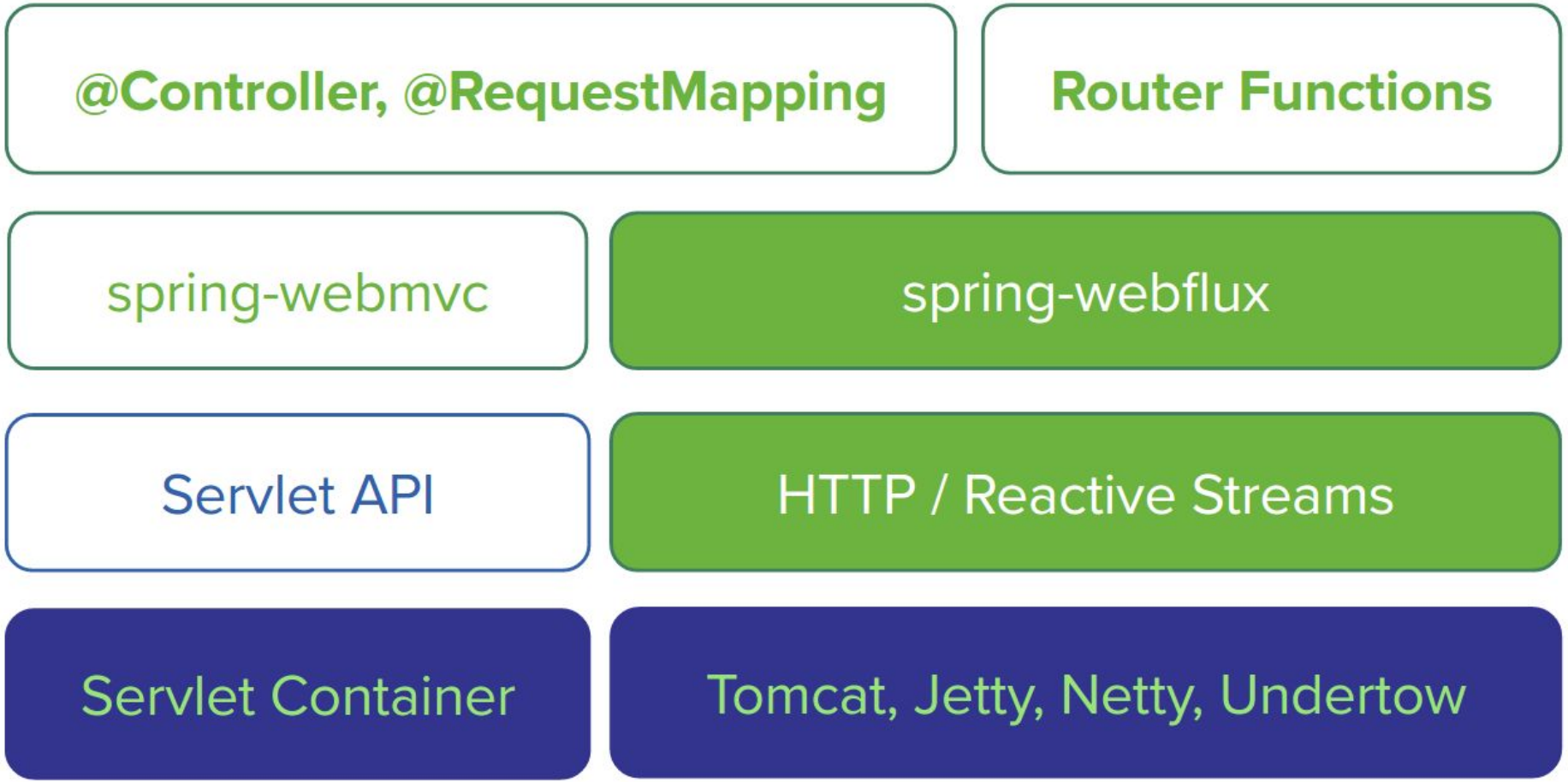
```
@GetMapping("/api/client/likes")
public Flux<LikedPropertyDTO> listOfLikes(@AuthenticationPrincipal SystemUser requester) {
    Flux<LikedProperty> publicador = Flux
            .fromStream(() -> likedPropertyDao.findByLikerId(requester.getId()));

    return publicador.map(LikedPropertyDTO::new);
}
```

Quem executa o fluxo de funções?

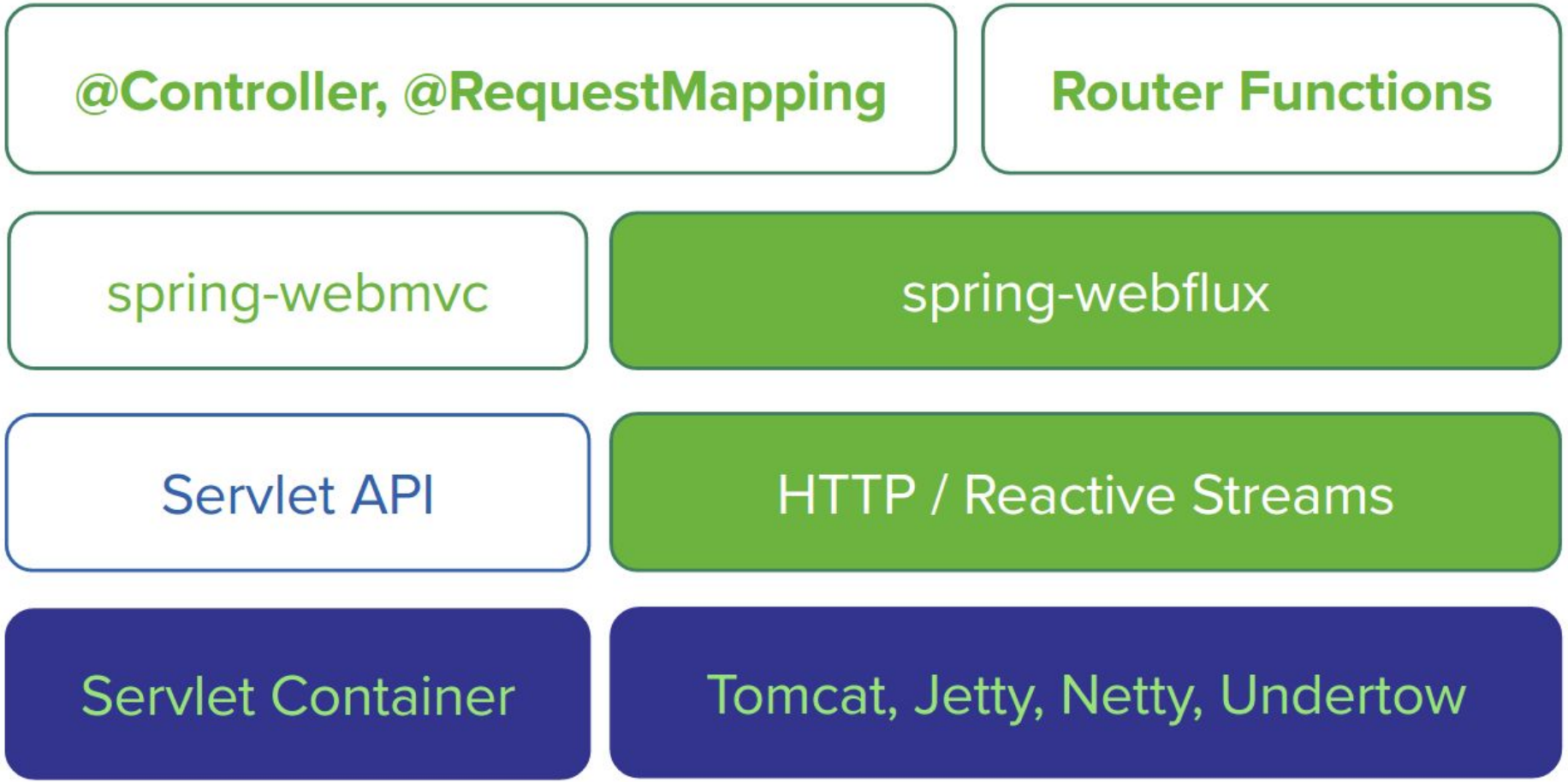
Spring Reactive!

Camada reativa do Spring


@Controller, @RequestMapping
Router Functions
spring-webmvc
spring-webflux
Servlet API
HTTP / Reactive Streams
Servlet Container
Tomcat, Jetty, Netty, Undertow

Como eu ativo isso?

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```


@Controller, @RequestMapping
Router Functions
spring-webmvc
spring-webflux
Servlet API
HTTP / Reactive Streams
Servlet Container
Tomcat, Jetty, Netty, Undertow

Netty?

Não depende de Servlets

Mas se integra :)

Benefícios

Delegação total da execução de código

Pools de threads separados para vários momentos

Pools especializados para códigos blocantes

```
return Flux.fromStream(() -> visitRequestDao
        .findByRequestedPropertyOwnerIdAndApprovalInstantIsNull(owner.getId()))
        .subscribeOn(Schedulers.elastic())
        .map(PendingVisitRequest::new);
```

Maior escalabilidade com menor custo

Bom encaixe para API

Muito bom para drivers de bancos não relacionais

Complexidades adicionais

Modelo de programação diferente

Talvez seja mais complicado de manter

Drivers JDBC são blocantes

Debug mais complicado

Teste mais complicado

Exceptions para indicar tipos de retorno

```
@PostMapping("/api/client/like/property/{propertyId}")
public Mono<LikedPropertyDTO> like(@PathVariable("propertyId") Integer propertyId,
        @AuthenticationPrincipal SystemUser requester) {
    Mono<Property> publicador = Mono.fromSupplier(
                    () -> propertyDao.findById(propertyId)
                    .orElseThrow(() -> new NotFoundException()));
    return publicador
            .map(requester::like)
            .map(likedPropertyDao::save)
            .map(LikedPropertyDTO::new);

}
```

Múltiplos pools não casam bem com ThreadLocal

```
return Mono.fromSupplier(() -> visitRequestDao.findById(requestId).get())
        .map(request ->  request.approve())
        .map(request -> ResponseEntity.created(URI.create("/api/request/visit/approve/pe
```

```
return Mono.fromSupplier(() -> visitRequestDao.findById(requestId).get())
        .map(request -> transactionalContext.execute(() -> request.approve()))
        .map(request -> ResponseEntity.created(URI.create("/api/request/visit/approve/pe
```

```
@Component
@Transactional
public class TransactionalContext {

    public <T> T execute(Supplier<T> supplier) {
        return supplier.get();
    }
}
```

Dúvidas?

github.com/asouza

twitter.com/alberto_souza

Cliente reativo

```
WebClient client = WebClient.create("http://localhost:8080");

Mono<ClientResponse> exchange = client.get()
        .uri("/api/client/likes")
        .header("X-AUTH-TOKEN", "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJh
        .accept(MediaType.APPLICATION_JSON)
        .exchange();
```

Como consumimos as informações?

```
Flux<LikedPropertyDTO> likes = new ClientController()
        .listOfLikes(new SystemUser());

likes.subscribe(dto -> {
    //logica aqui
});
```

Backpressure

Não mande mais do que eu posso consumir

Longe da realidade do povo