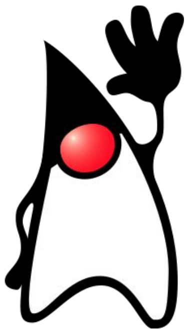
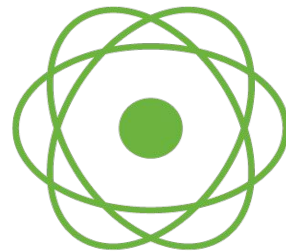
Servlet vs Reactive stacks in 5 use cases

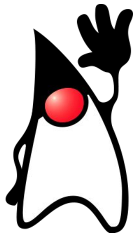




Rossen Stoyanchev
Pivotal

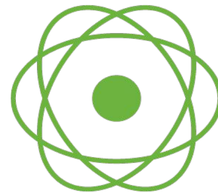
Servlet Stack

- Servlet container
- Servlet API
- Spring MVC



Reactive Stack

- Netty, Servlet 3.1+, Undertow
- Reactive Streams
- Spring **WebFlux**



Reactive Spring

Reactive starters in **Spring Boot 2.0**

Generate Project alt + ⏎

Spring Framework 5 WebFlux endpoints + reactive WebClient

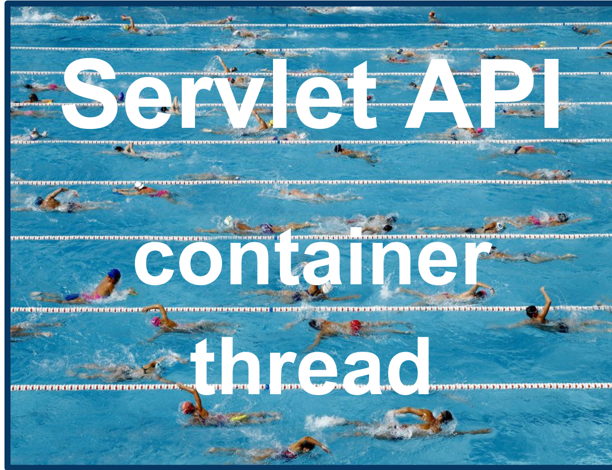
Reactive **Spring Data Kay** repositories

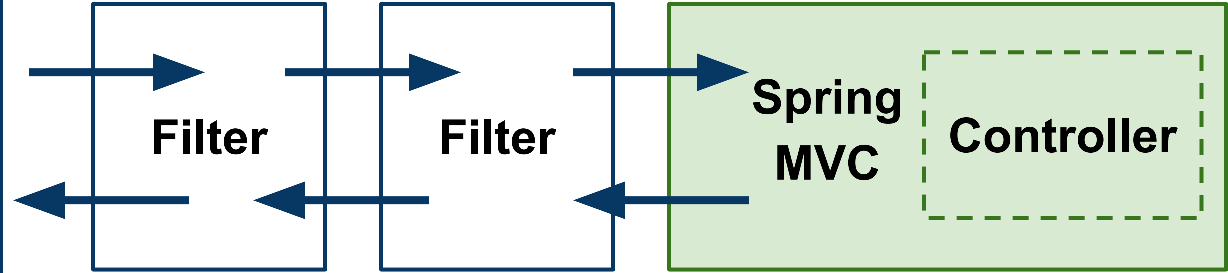
Spring Security

and more…

Servlet Stack

SERVLET STACK





SERVLET STACK

Synchronous API

`Filter`, `Servlet` … **`void`**

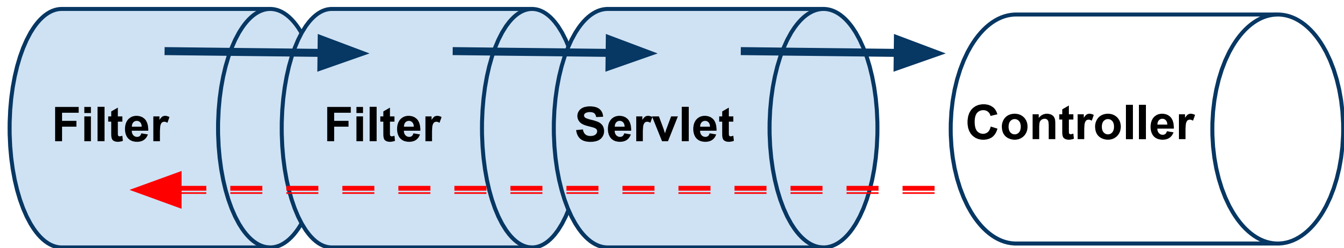
Blocking I/O

`InputStream`, `OutputStream`

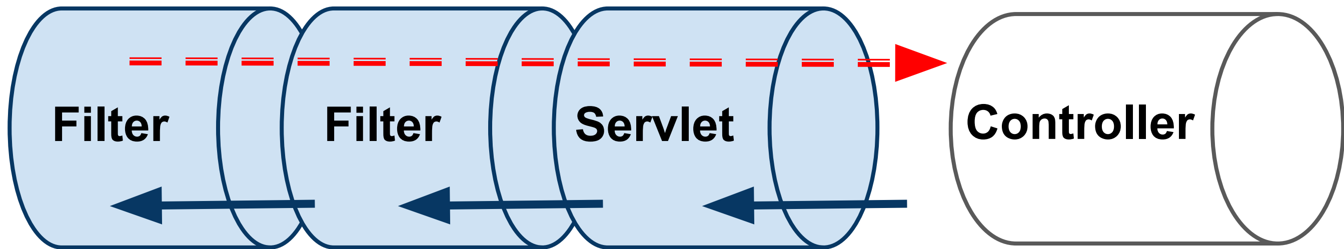
SERVLET STACK

```
servletRequest.startAsync()
```

SERVLET STACK



… do work or receive event + `dispatch()` *…*



SERVLET STACK

`startAsync()`

`Input` & `OutputStream`

Controller

can use reactive clients

Concurrency
models

Synchronous APIs

100s, 1000s
waiting blocked threads

Non-blocking code

~ per CPU core
busy worker threads

What does it take to not block ?

event loop at the core

event driven architecture

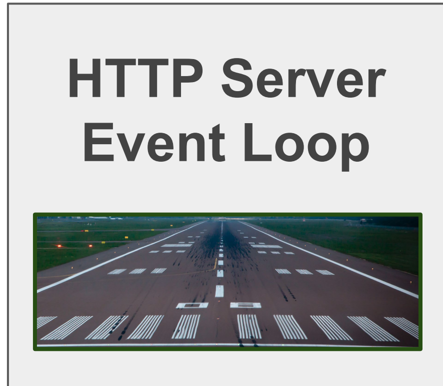
message passing

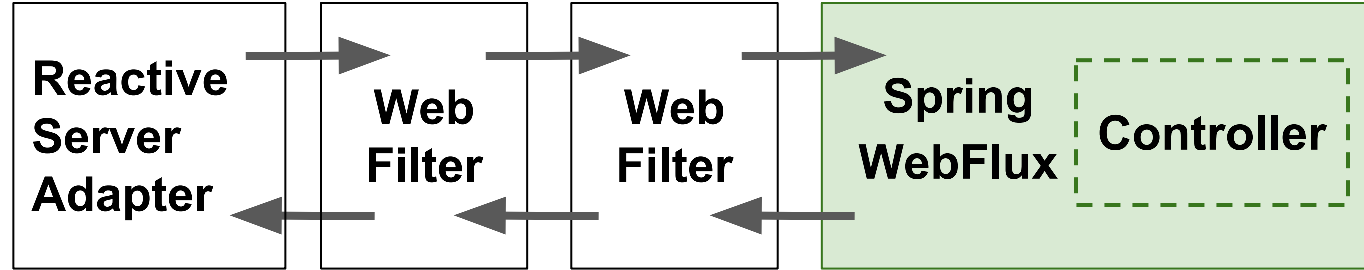
means to compose async logic

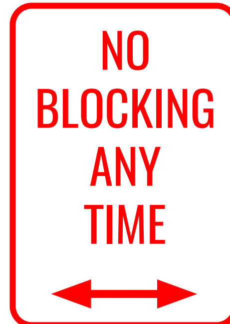
bonus:
back pressure (a.k.a flow control)

Reactive Stack

REACTIVE STACK







REACTIVE STACK

Asynchronous API

`WebFilter`, `WebHandler`...

`Mono<Void>`

REACTIVE STACK

Reactor **`Mono`**

[**Reactive Streams**]() `Publisher`

0..1 elements

REACTIVE STACK

Non-blocking **read**:

```
Flux<DataBuffer> getBody()
```

REACTIVE STACK

Non-blocking **write**:

```
writeWith(Flux<DataBuffer>)
```

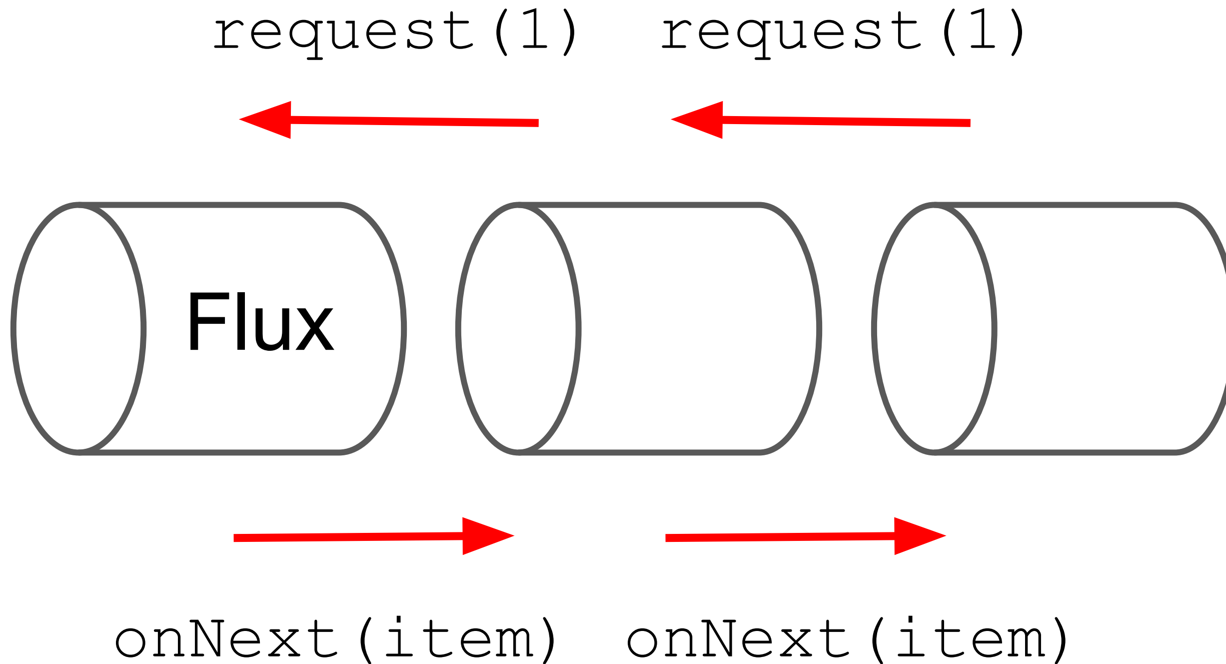
REACTIVE STACK

Reactor **`Flux`**

Reactive Streams `Publisher`

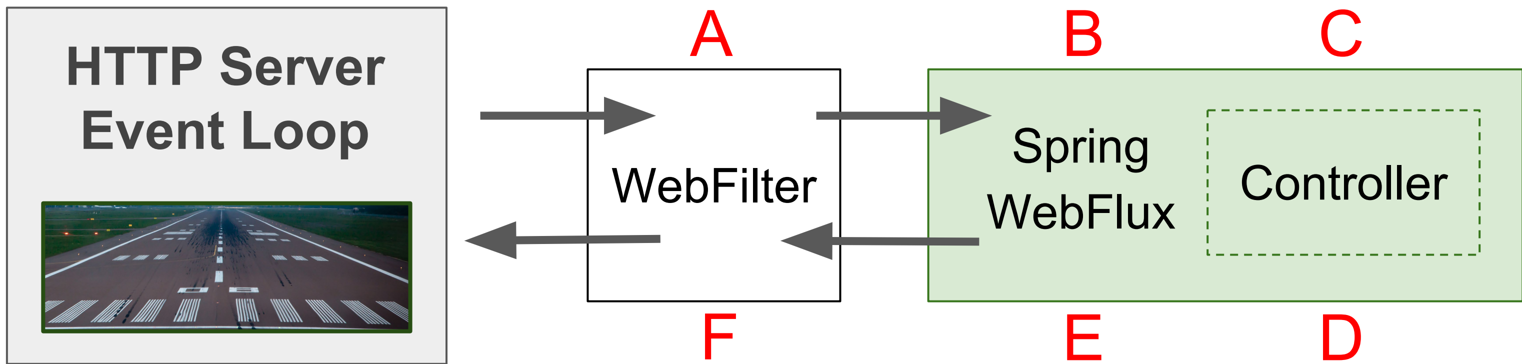
0..N elements

Reactive Streams back pressure



REACTIVE STACK

Composition of async logic



Actual processing

request(n)

A Mono B Mono C Mono D Flux E Mono F

onNext(item)

Use Case #1

Reactive data repository

Demo

HTTP GET with reactive data repository

Designed to work on both Spring MVC **and** Spring WebFlux

Simply return reactive type (**`Flux`**, **`Observable`**) from `@Controller`

```
@GetMapping("/cars")
@ResponseBody
public Flux<Car> getCars() {
    return this.repository.findAll();
}
```

`Flux<T>:`

finite collection or *infinite* stream?

Use media type to decide

`“application/json”`

finite collection (JSON array)

No back pressure:

`Flux#collectToList`

(request all + buffer)

Use Case #2

Response stream with back pressure

“text/event-stream”,

“application/stream+json”

infinite stream

Back pressure:

request(n),
write, flush,
repeat

HTTP GET with streaming response

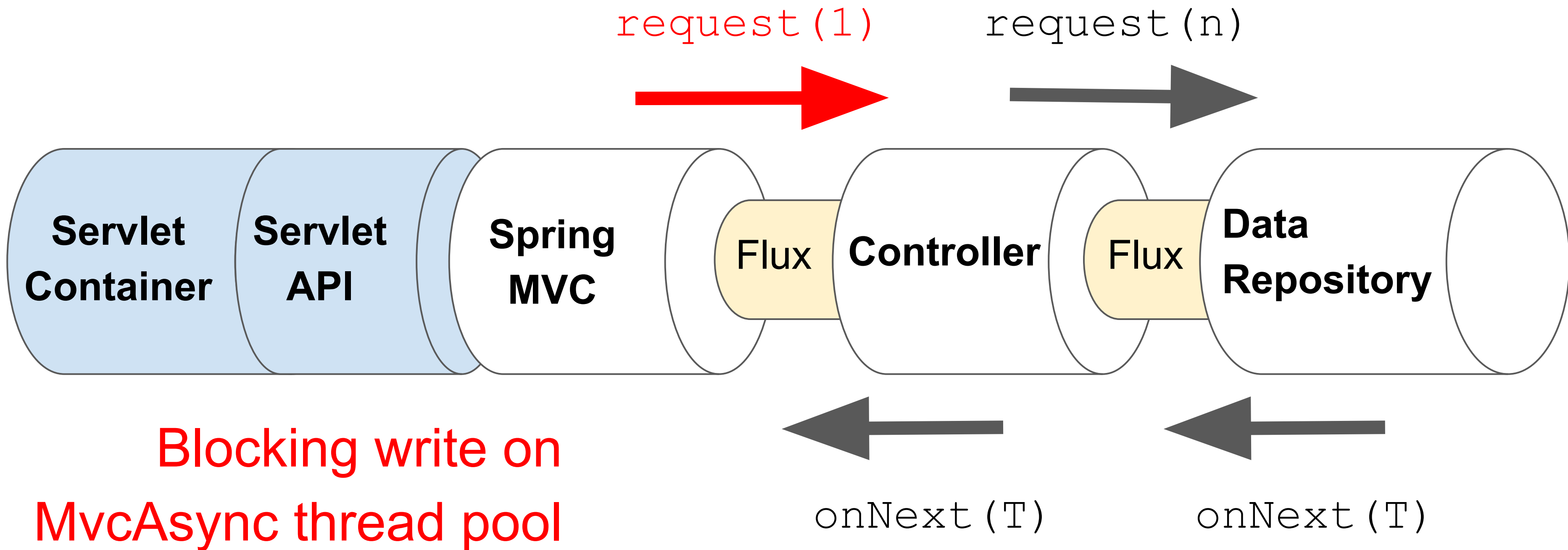
Simply return reactive type (**`Flux`**, **`Observable`**) from `@Controller`

Back pressure on Spring MVC and WebFlux

```
@GetMapping(path="/cars", produces="text/event-stream")
public Flux<Car> getCars() {
    return this.repository.findCarsBy();
}
```

SERVLET STACK ...

Back pressure against blocking `OutputStream`



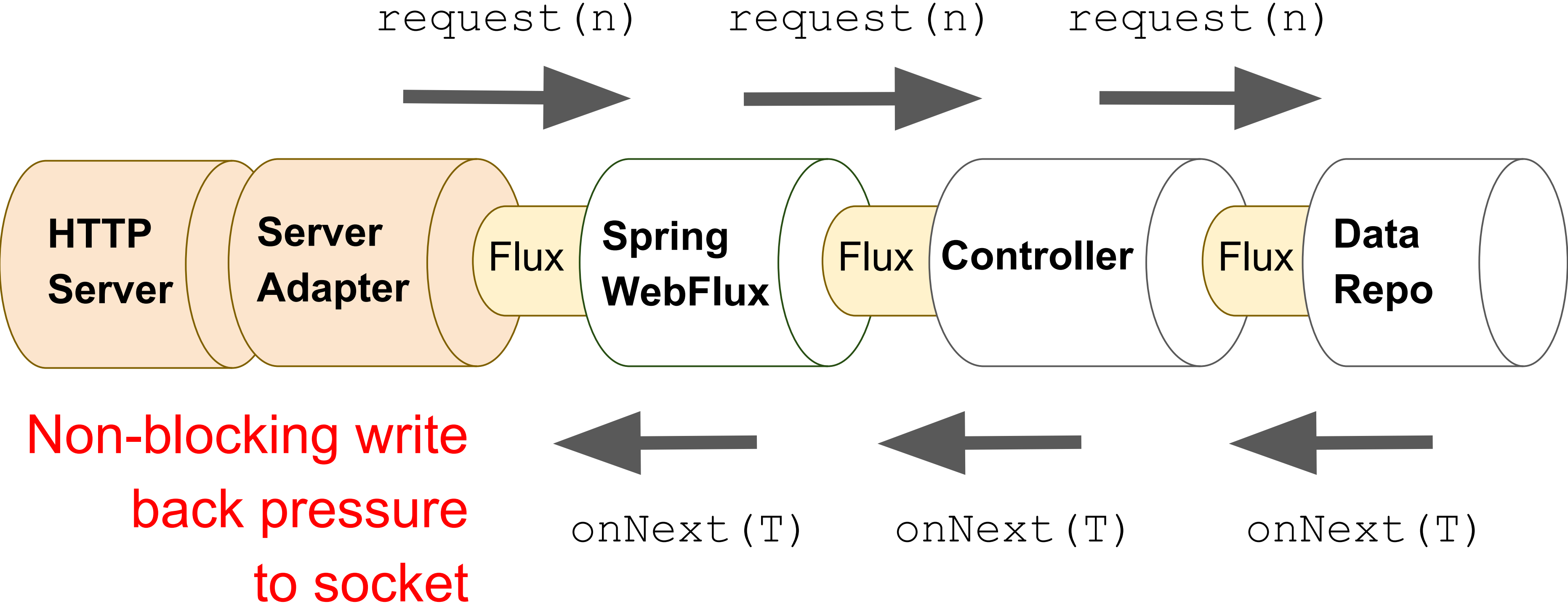
Servlet 3.1 non-blocking I/O ?

Unlike Servlet 3.0 async, Servlet 3.1 non-blocking is hard to retrofit

Requires deeper change

Mutually exclusive with rest of the Servlet API

Response streaming on **reactive** stack



Demo

Use Case #3

Reactive remote service orchestration

Demo

Reactive `WebClient`

Orchestrate non-blocking, nested remote service calls with ease

Similar to reactive data access

Spring MVC and Spring WebFlux

```java
@PostMapping("/booking")
public Mono<ResponseEntity<Void>> book() {

    return locationClient.get()
            .uri("/cars")
            .retrieve()
            .bodyToFlux(Car.class)
            .take(5)
            .flatMap(car -> bookingClient.post()
                    .uri("/cars/{id}/booking", car.getId())
                    .exchange()
                    .map(this::toBookingResponseEntity))
            .next();
}
```

Use Case #4

Reactive request input

Back pressure to socket

No reading until **reactive demand** signalled from upstream

Non-blocking

Reactive stack only territory !

HTTP POST with data

`@RequestBody` argument with reactive type (`Mono`, `Single`)

Reactive type is not required

```
@PostMapping("/cars")
@ResponseStatus(HttpStatus.CREATED)
public Mono<Void> loadCars(@RequestBody Mono<Car> car) {
    return this.repository.insert(car).then();
}
```

Use Case #5

Data Ingestion with back pressure

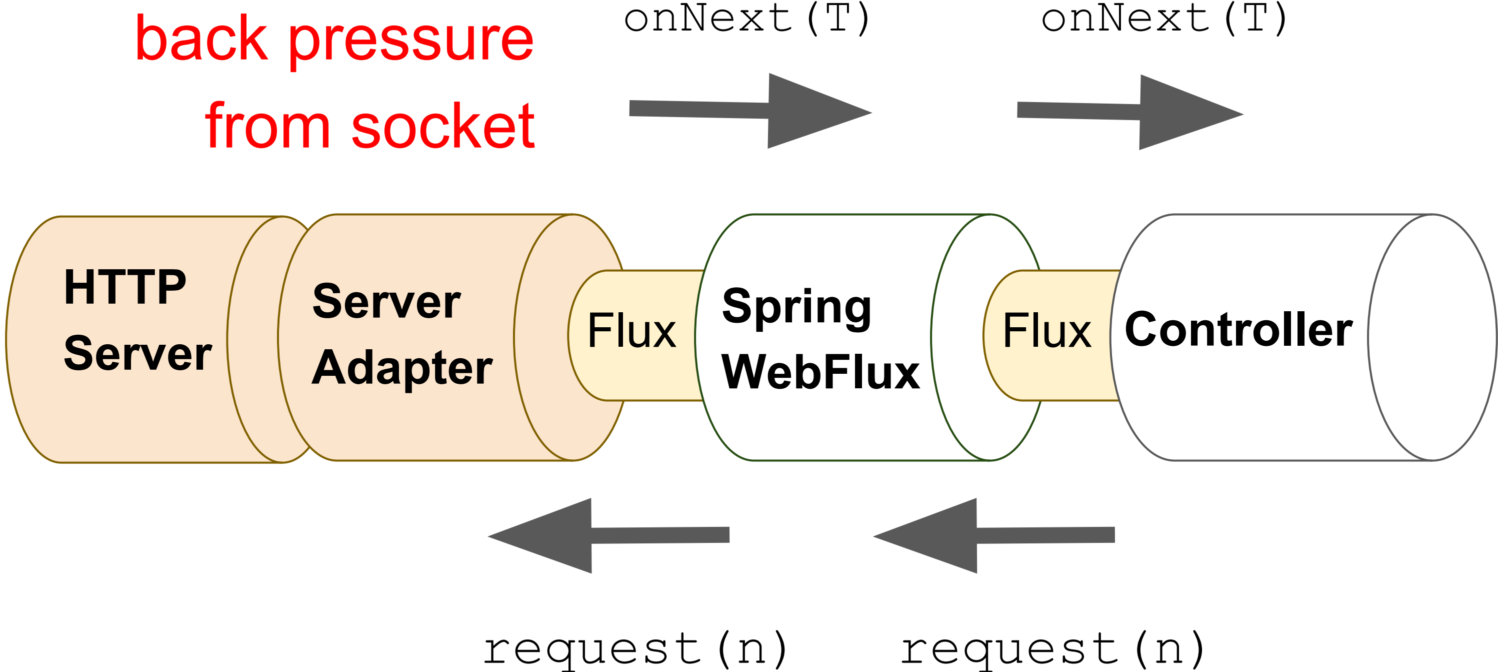
HTTP POST with stream of data

Media type indicates **infinite** stream is expected

Non-blocking streaming + back pressure

```
@PostMapping(path="/cars", consumes="application/stream+json")
public Mono<Void> loadCars(@RequestBody Flux<Car> cars) {
    return this.repository.insert(cars).then();
}
```

Data ingestion on reactive stack



Servlet stack summary

- ✓ Reactive data repository
- ✓ Streaming to the response with back pressure
- ✓ Reactive orchestration of remote services
- ✗ Reactive request input
- ✗ Data ingestion with back pressure

Reactive stack summary

- ✓ Reactive data repository
- ✓ Streaming to the response with back pressure
- ✓ Reactive orchestration of remote services
- ✓ Reactive request input
- ✓ Data ingestion with back pressure

Q & A

@rstoya05