**MICROSERVICES WITH SPRING WEBFLUX**

--> With traditional programming if you hace 400 concurrent requests you will need 400 threads and 1MB memory per thread.

You send a request to the server and the thread will be idle waiting for the response to come back (blocking!!)

--> With webflux you use resources in a more efficient way, the threads won't keep idle waiting for the response to come back -> **It’s ASYNC AND NON-BLOCKING**

Throughput is **the measure of how many requests a server processes in a given unit of time….Spring webflux has a better server throughput than spring web.**

\*\*\*\*\*Spring boot's embeded tomcat has 200 threads by default --> If you use spring web

\*\*\*\*\*If you use spring web flux --> Just uses 4 threads or it depends on the number of cores you have

**\*\*\*REACTIVE MANIFESTO**

- Responsive --> Responding in a timely manner -> Keeping things lazily --> It helps to the user experience, facebook doesn't load everything when you open it, just when you click something or you scroll down.

Yo work when necessary and when the user is still there

- Resilient --> Stay responsive in case of failure

- Elastic --> Staying responsive under varying load --> Our app doesn't have any bottleneck -> You use resources efficiently and scale out just when necessary.

- Message Driven --> Loosely coupled and rely on asynchronous message passing --> Asynchronous non-blocking way - backpressure -> The publisher emits just the items the subscriber can process (something like that)

**SPRING WEB FLUX**

- Spring framework from versión 5 started to allow reactive streams.

-Netty by default 🡪 Event loop group

There are 2 thread groups or pools in netty

One is called the boss thread pool – Listening on the socket for requests

The other is called worker thread group 🡪 Read and process -> Event loop -> They not stay idle and keep processing requests -> They’re not blocked

**In the controller you won’t be subscribing, you Will be always returning a Publisher of some type.**

**REACTIVE CONTROLLER THAT STREAMS VALUES TO THE FRONTEND**

@GetMapping(value = "table/{input}/stream", produces = MediaType.*TEXT\_EVENT\_STREAM\_VALUE*)  
public Flux<Response> multiplicationTableStream(@PathVariable int input){  
 return this.mathService.multiplicationTable(input);  
}

**IN A TRADICIONAL CONTROLLER WHEN THE SUBSCRIBER CANCELS, THE BACKEND CONTINUES PROCESSING….. THIS DOESN’T HAPPEN WITH THE REACTIVE CONTROLLER**

**BAD PRACTICE: Return Flux.fromIterable();**

**FUNCTIONAL ENDPOINTS**

- There are request predicates, handlers, etc…

// @Bean  
 private RouterFunction<ServerResponse> serverResponseRouterFunction(){  
 return RouterFunctions.*route*()  
 .GET("square/{input}", RequestPredicates.*path*("\*/1?"), requestHandler::squareHandler)  
 .GET("square/{input}", req -> ServerResponse.*badRequest*().bodyValue("only 10-19 allowed"))  
 .GET("table/{input}", requestHandler::tableHandler)  
 .GET("table/{input}/stream", requestHandler::tableStreamHandler)  
 .POST("multiply", requestHandler::multiplyHandler)  
 .GET("square/{input}/validation", requestHandler::squareHandlerWithValidation)  
 .onError(InputValidationException.class, exceptionHandler())  
 .build();  
 }

**WEB CLIENT**

- Reactor based fluent API por making HTTP requests.

- I want to have non-blocking and asynchronous communication 🡪 Not keeping the thread idle waiting for the response.

- It´s thread safe

- You can use the builder pattern

**MAP VS FLATMAP**

Use map for synchronous calls and for transformation operations that don´t return publishers. It returns a transformed flux with the result of the operations.

Use flatmap for asynchronous calls that returns another Publisher. What it does is that if the result is another flux it flattens or extracts the elements in the returned Publisher. It is that you won´t have a flux inside another flux. It helps you when you want to chain some different calls to a database for example.

**R2DBC**

Reactive Relational Database Connectivity IS DIFFERENT to JPA

For example, you don’t have to add the @Entity