1. Goals

The goal of this lesson is to show you the first implementation steps towards implementing a reactive API with Spring WebFlux.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is : m11-lesson2-start

If you want to skip and see the complete implementation, feel free to jump ahead and import: m11-lesson2

2.1. Migrate Maven and Application Configuration

Let's start the migration with some simple **Maven changes**:

- spring-webmvc => spring-webflux
- spring-boot-starter-web => spring-boot-starter-webflux
- remove tomcat
- remove the war packaging

Then, **in the config, let's:**

- WebMvcConfigurer -> WebFluxConfigurer
- @EnableWebMvc -> @EnableWebFlux
- remove SpringBootServletInitializer
- remove server.servlet.context-path (not supported)

2.2. Migrate the Web Layer

Let’s now focus on the API - starting with the controller layer.

We'll now need to start returning publishers:

- *List -> Flux*

- *T -> Mono<T>*

Naturally, we need to do a lot more.

We'll replace the servlet-specific support and replace it with WebFlux support:

- HttpServletRequest -> ServerHttpRequest (reactive)

We'll then bridge the controller and the service layers:

- Flux.fromStream(list.stream());
- Mono.just(obj);

And, finally, the new API of the HTTP request; instead of:

request.getParameterNames().hasMoreElements()

Let's do:

!request.getQueryParams().isEmpty()

2.3. Running the App and Caveats

Finally, let's now run the application and use PostMan to send a request and retrieve all existing roles in the system.

The name of the request in PostMan, is: *PostMan - M12-L2 - Retrieve All Roles*

And let's define a simple breakpoint in *DispatcherHandler.handle* - the core routing logic - corresponds to the *DispatcherServlet* in the traditional Servlet model.

And, with the app open in debugging mode, let's make sure the breakpoint gets hit.

Now, that the migration is done, there are a few critical notes and caveats we need to keep in mind.

First, keep in mind this is a a surface-level reactive application. Once we go past the controller layer, nothing is really reactive.

No, in order to achieve real reactive semantics, and actually benefit from it, we need to be reactive all the way through (including persistence).

And, there are, of course, multiple persistence solutions - that do support these semantics.

But, for now, let's simply simulate these semantics.

2.4. New Operation - Implementation

We're going to implement the *findAll()* operation for Privileges - using Server Sent Events (SSE)

We're going to emit one privilege every 5 seconds:

```
Flux<Long> interval = Flux.interval(Duration.ofMillis(5000));

Flux<Privilege> privilege = findAllInternal(request);

return Flux.zip(interval, privilege).map(Tuple2::getT2);
```

Note how this data is regenerated from scratch, for each subscriber?

That makes this **a cold publisher**. Basically - nothing happens until you subscribe.

In contrast, **a hot publisher** would not care about subscribers - and would publish data with or without them. That means a subscriber would only see the data published after they subscribed.

3. Resources

- Web on Reactive Stack (reference)

- Spring 5 Reactive (on Baeldung)