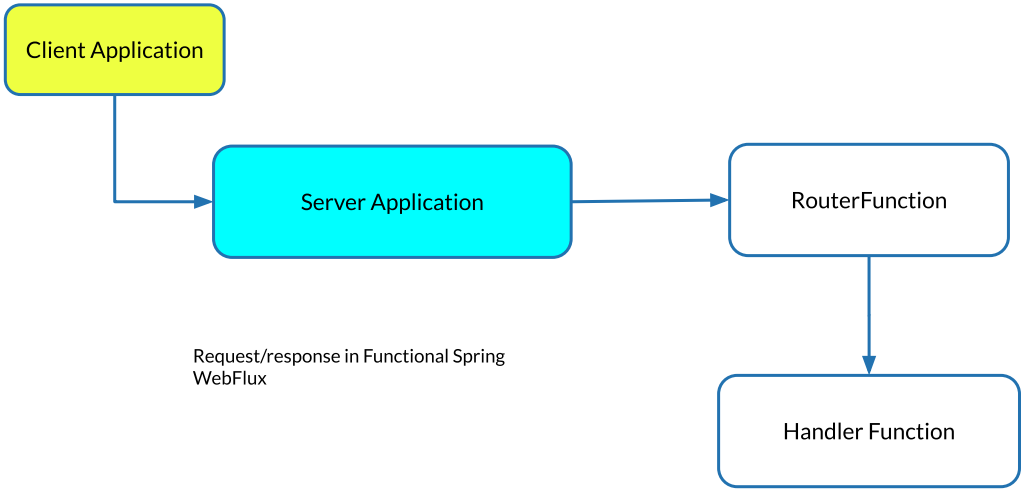
A functional endpoint in Spring WebFlux

In this article, you will learn to create a Functional endpoint in Spring WebFlux. In the previous article, I have introduced you to create an endpoint using the Annotations, now we will look into the pure functional programming way to achieve this.

1. Overview of Functional endpoint



An Http request initiated by a client app arrives at the Server (Netty/Undertow etc.). This request is forwarded to the **RouterFunction** to assign an appropriate **HandlerFunction** to serve the Http request. The RouterFunction is similar to `@RequestMapping` and the HandlerFunction is similar to the body of `@RequestMapping` method in the annotation-driven programming model.

2. Functional reactive endpoints

We will use the standard start.spring.io to generate the starter Spring-Boot-WebFlux project to implement functional endpoints. The most important dependency that you need is spring-boot-

Table of Contents

Spring WebFlux Tutorial

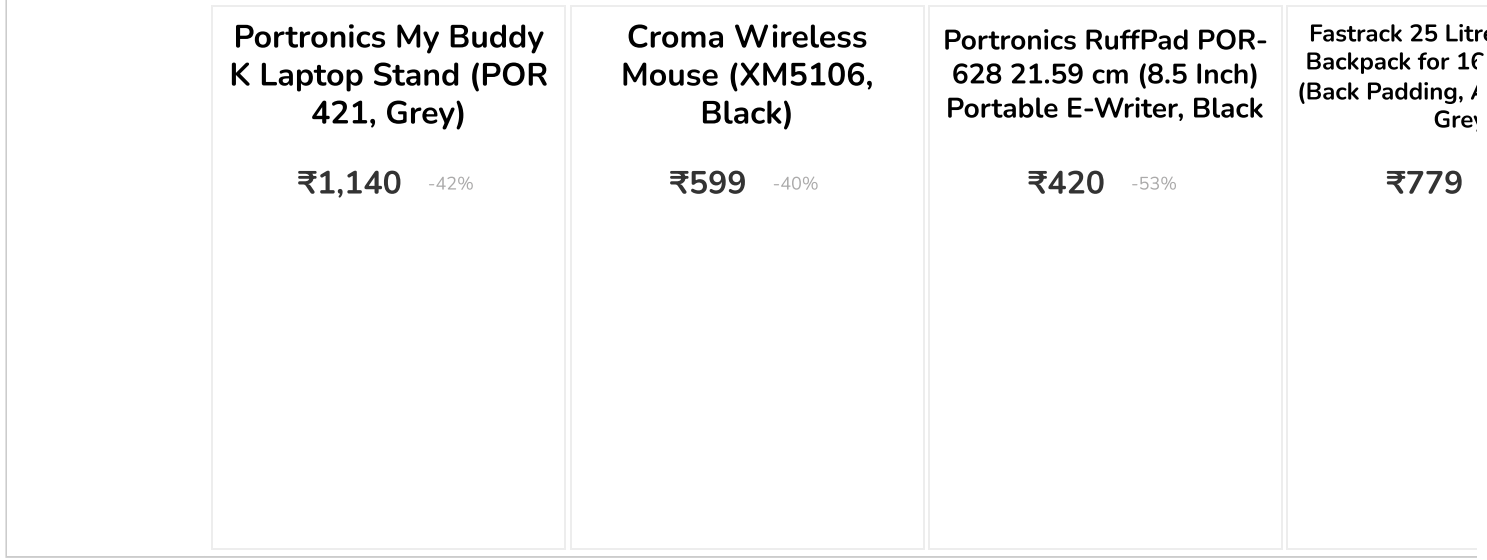
Step-1: Generate the project from start.spring.io

We will generate our Spring WebFlux project from **start.spring.io**, with the dependency `spring-boot-starter-webflux` . Other dependencies are for testing and ease of development. Use the link to generate the project, unzip it, and import it to your IDE.

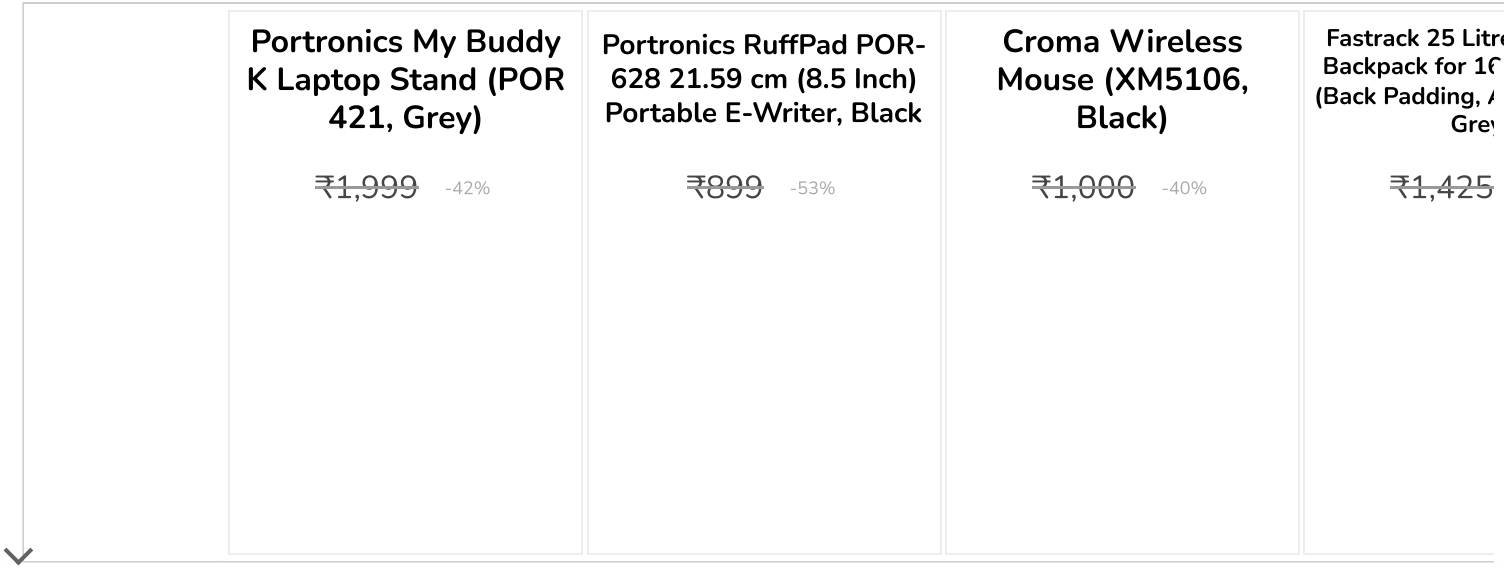
```
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.0.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
....
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-webflux</artifactId>
    </dependency>
  <dependencies>
.....
```

Table of Contents

Spring WebFlux Tutorial



Step-2: Create the handler functions in HelloHandler



as `HelloHandler` with two methods. One will serve us the `TEXT_PLAIN` and other one to serve `APPLICATION_STREAM_JSON` content.

HelloHandler.java FunctionalAppConfig.java

```java
package c.jbd.webflux;

import org.springframework.http.MediaType;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.time.Duration;

@Component
public class HelloHandler {
  /**
   * Serves a plain_text
   */
  public Mono<ServerResponse> monoMessage(ServerRequest request) {
    return ServerResponse.ok()
      .contentType(MediaType.TEXT_PLAIN)
      .body(
        Mono.just("Welcome to JstoBigdata.com"), String.class
      );
  }
```

Table of Contents

Spring WebFlux Tutorial

```
28.   public Mono<ServerResponse> fluxMessage(ServerRequest request) {
29.     return ServerResponse.ok()
30.       .contentType(MediaType.APPLICATION_STREAM_JSON)
31.       .body(
32.         Flux.just("Welcome ", "to ", "JstoBigdata.com")
33.           .delayElements(Duration.ofSeconds(1)).log(), String.class
34.       );
35.   }
36. }
```


Step-3: Create the Router functions in HelloRouter

We have the logic in the HelloHandler to serve the response. Now, we will write our router function in HelloRouter class to bind the specific endpoints. Basically, every endpoint will be linked to the specific router function

```java
package c.jbd.webflux;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.server.RequestPredicates;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.RouterFunctions;
import org.springframework.web.reactive.function.server.ServerResponse;

@Configuration
public class HelloRouter {
  @Bean
  public RouterFunction<ServerResponse> functionalRoutes(HelloHandler helloHandler) {
    return RouterFunctions
      .route(RequestPredicates.GET("/functional/mono")
        , helloHandler::monoMessage)
      .andRoute(RequestPredicates.GET("functional/flux")
        , helloHandler::fluxMessage);
  }
}
```

> It is recommended to declare the **Router Function class** using `@Configurations` annotations, and the router functions as beans.

Step-4: Start the App and Test the endpoints

To test the **Mono endpoint**, open the below url in the browser.

```
http://localhost:8080/functional/mono
```

You will be able to get the message, Welcome to JstoBigdata.com on the browser. There is nothing special here.

Similarly, to test the **Flux endpoint**, open the below url in browser and observe the output.

```
http://localhost:8080/functional/flux
```

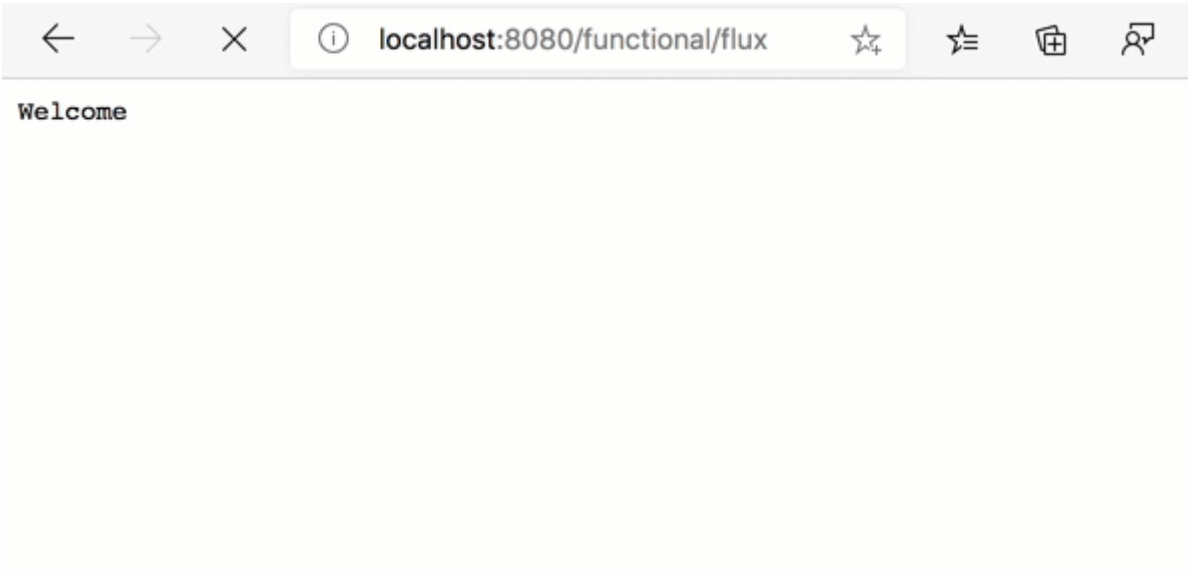


Table of Contents

Spring WebFlux Tutorial

3. Test the endpoints using WebTestClient

The spring-test module provides mock implementations of ServerHttpRequest , ServerHttpResponse , and ServerWebExchange . The **WebTestClient** is built on these mock request and response objects to provide support for testing WebFlux applications without an HTTP server. You can use the WebTestClient for end-to-end integration tests, too.

```
1. package c.jbd.webflux;
2.
3. import org.junit.jupiter.api.Test;
4. import org.springframework.beans.factory.annotation.Autowired;
5. import
   org.springframework.boot.test.autoconfigure.web.reactive.AutoConfigureWebTestClient;
6. import org.springframework.boot.test.context.SpringBootTest;
7. import org.springframework.http.MediaType;
8. import org.springframework.test.web.reactive.server.WebTestClient;
9. import reactor.core.publisher.Flux;
10. import reactor.test.StepVerifier;
11.
12. @SpringBootTest
13. @AutoConfigureWebTestClient //Important
14. public class FunctionalAppTest {
15.   @Autowired
16.   private WebTestClient webTestClient;
```

```
21.   public void testMonoEndpoint() {
22.     Flux<String> msg$ = webTestClient.get()
23.       .uri("/functional/mono")
24.       .accept(MediaType.TEXT_PLAIN)
25.       .exchange()
26.       .expectStatus().isOk()
27.       .returnResult(String.class).getResponseBody()
28.       .log();
29.
30.     StepVerifier.create(msg$)
31.       .expectNext(TEST_MESSAGE)
32.       .verifyComplete();
33.   }
34.
35.   @Test
36.   public void testFluxEndpoint() {
37.     Flux<String> msg$ = webTestClient.get()
38.       .uri("/functional/flux")
39.       .accept(MediaType.APPLICATION_STREAM_JSON)
40.       .exchange()
41.       .expectStatus().isOk()
42.       .returnResult(String.class).getResponseBody()
43.       .log();
44.
45.     StepVerifier.create(msg$)
46.       .expectNext(TEST_MESSAGE)
47.       .verifyComplete();
48.   }
49. }
```

Table of Contents

Spring WebFlux Tutorial

Conclusion

I have given you a brief guide on creating functional endpoint in Spring WebFlux. We will explore much more in the upcoming articles. Reactive programming provides better performance with limited hardware as compared to the traditional Spring MVC with too many threads. I would suggest trying this framework to see if this fulfils your project needs. You can download the complete source code from Github.

CODE EXAMPLE

By Bikram Kundu | July 10th, 2020 | Categories: Spring Framework | Tags: Spring 5.x, Spring WebFlux

Share This Page, Choose Your Platform!

Table of Contents

Spring WebFlux Tutorial

Leave A Comment

Comment...

Name (required) Email (required) Website

Table of Contents

RECENT POSTS

- The API Gateway Pattern in Microservices
- Introduction to Microservices Architecture
- Getting Started with JUnit 5
- Spring @Import and @ImportResource annotations
- Handle Resources in Spring
- JUnit 5 – Software Testing Framework
- Spring WebFlux REST Api with MongoDB and Spring Data
- A functional endpoint in Spring WebFlux
- Use of @Order annotation in Spring

CATEGORIES

Select Category

ABOUT JSTOBIGDATA

- About us
- Contact us
- Privacy Policy
- Terms & Conditions
- Sitemap

Spring WebFlux Tutorial