Germán Caballero Rodríguez
gcaballero@gmail.com

INDICE

Núcleo reactivo

- El módulo spring-web contiene el siguiente soporte básico para aplicaciones web reactivas:
  - Para el procesamiento de solicitudes del servidor hay dos niveles de soporte.
    - **HttpHandler:** Contrato básico para el manejo de solicitudes HTTP con E / S sin bloqueo y contrapresión de contrapresión de flujos, junto con adaptadores para Reactor Netty, Undertow, Tomcat, Jetty y cualquier contenedor Servlet 3.1+.
    - **WebHandler API:** API web de uso general de nivel ligeramente superior para el manejo de solicitudes, además de la cual se construyen modelos de programación concretos, como controladores anotados y puntos finales funcionales.

Núcleo reactivo

- El módulo spring-web contiene el siguiente soporte básico para aplicaciones web reactivas:
  - Para el lado del cliente, hay un contrato ClientHttpConnector básico para realizar solicitudes HTTP con E / S sin bloqueo y contrapresión de contrapresión, junto con adaptadores para Reactor Netty y para el reactivo Jetty HttpClient.
    - El WebClient de nivel superior utilizado en las aplicaciones se basa en este contrato básico.

Núcleo reactivo

- El módulo spring-web contiene el siguiente soporte básico para aplicaciones web reactivas:
  - Para el cliente y el servidor, los códecs se utilizan para serializar y deserializar el contenido de las solicitudes y respuestas HTTP.

Núcleo reactivo

- Filtros:
  - En la API de WebHandler, puede usar un filtro web para aplicar la lógica de estilo de intercepción antes y después del resto de la cadena de procesamiento de filtros y el administrador web de destino.
  - Cuando se utiliza la configuración de WebFlux, registrar un filtro web es tan simple como declararlo como un bean Spring y (opcionalmente) expresar precedencia utilizando @Order en la declaración del bean o implementando Ordered.

Núcleo reactivo

- Excepciones:
  - En la API de WebHandler, puede usar un WebExceptionHandler para manejar las excepciones de la cadena de instancias de WebFilter y el WebHandler de destino.
  - Cuando se utiliza la configuración de WebFlux, registrar un WebExceptionHandler es tan simple como declararlo como un bean Spring y (opcionalmente) expresar precedencia utilizando @Order en la declaración del bean o implementando Ordered.

DispatcherHandler

- Spring WebFlux, al igual que Spring MVC, está diseñado alrededor del patrón del controlador frontal, donde un WebHandler central, el DispatcherHandler, proporciona un algoritmo compartido para el procesamiento de solicitudes, mientras que el trabajo real se realiza mediante componentes configurables, delegados.
- Este modelo es flexible y soporta diversos flujos de trabajo.

DispatcherHandler

- DispatcherHandler descubre los componentes delegados que necesita de la configuración de Spring.
- También está diseñado para ser un bean Spring e implementa ApplicationContextAware para acceder al contexto en el que se ejecuta.
- Si DispatcherHandler se declara con un nombre de bean de webHandler, es, a su vez, descubierto por WebHttpHandlerBuilder, que reúne una cadena de procesamiento de solicitudes, como se describe en la API de WebHandler.

Controladores anotados

- Spring WebFlux proporciona un modelo de programación basado en anotaciones, donde los componentes @Controller y @RestController usan anotaciones para expresar las asignaciones de solicitudes, solicitar entradas, manejar excepciones y más.
- Los controladores anotados tienen firmas de métodos flexibles y no tienen que extender las clases base ni implementar interfaces específicas.

Controladores anotados

- Spring WebFlux proporciona un modelo de programación basado en anotaciones:

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    public String handle() {
        return "Hello WebFlux";
    }
}
```

Controladores anotados

- También hay variantes de acceso directo específicas del método HTTP de @RequestMapping:
  - @GetMapping
  - @PostMapping
  - @PutMapping
  - @DeleteMapping
  - @PatchMapping

Controladores anotados

- Las anotaciones anteriores son anotaciones personalizadas que se proporcionan porque, posiblemente, la mayoría de los métodos de controlador deben asignarse a un método HTTP específico en lugar de usar @RequestMapping, que, de forma predeterminada, coincide con todos los métodos HTTP.
- Al mismo tiempo, se necesita un @RequestMapping a nivel de clase para expresar asignaciones compartidas.

Controladores anotados

- El siguiente ejemplo usa mapeos de nivel de método y tipo:

```
@RestController
@RequestMapping("/persons")
class PersonController {

    @GetMapping("/{id}")
    public Person getPerson(@PathVariable Long id) {
        // ...
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public void add(@RequestBody Person person) {
        // ...
    }
}
```

Puntos finales funcionales

- Spring WebFlux incluye WebFlux.fn, un modelo de programación funcional liviano en el que las funciones se usan para enrutar y manejar solicitudes y los contratos están diseñados para ser inmutables.
- Es una alternativa al modelo de programación basado en anotaciones, pero, por lo demás, se ejecuta en la misma base de núcleo reactivo.

Puntos finales funcionales

- En WebFlux.fn, una solicitud HTTP se maneja con HandlerFunction: una función que toma ServerRequest y devuelve un ServerResponse retrasado (es decir, Mono <ServerResponse>).
- Tanto la solicitud como el objeto de respuesta tienen contratos inmutables que ofrecen acceso fácil para JDK 8 a la solicitud y respuesta HTTP.
- HandlerFunction es el equivalente del cuerpo de un método @RequestMapping en el modelo de programación basado en anotaciones.

Puntos finales funcionales

- Las solicitudes entrantes se enrutan a una función de controlador con un RouterFunction: una función que toma ServerRequest y devuelve un HandlerFunction retrasado (es decir, Mono <HandlerFunction>).
- Cuando la función del enrutador coincide, se devuelve una función de controlador; De lo contrario un Mono vacío.

Puntos finales funcionales

- RouterFunction es el equivalente a una anotación @RequestMapping, pero con la gran diferencia de que las funciones del enrutador no solo proporcionan datos, sino también comportamiento.
- RouterFunctions.route () proporciona un generador de enrutadores que facilita la creación de enrutadores, como muestra el siguiente ejemplo:

Puntos finales funcionales

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.server.RequestPredicates.*;
import static org.springframework.web.reactive.function.server.RouterFunctions.route;

PersonRepository repository = ...
PersonHandler handler = new PersonHandler(repository);

RouterFunction<ServerResponse> route = route()
    .GET("/person/{id}", accept(APPLICATION_JSON), handler::getPerson)
    .GET("/person", accept(APPLICATION_JSON), handler::listPeople)
    .POST("/person", handler::createPerson)
    .build();

public class PersonHandler {

    public Mono<ServerResponse> listPeople(ServerRequest request) {
        // ...
    }
    public Mono<ServerResponse> createPerson(ServerRequest request) {
        // ...
    }
    public Mono<ServerResponse> getPerson(ServerRequest request) {
        // ...
    }
}
```

WebClient

- Spring WebFlux incluye un WebClient reactivo y no bloqueante para solicitudes HTTP.
- El cliente tiene una API funcional y fluida con tipos reactivos para la composición declarativa, consulte Bibliotecas reactivas.
- El cliente y el servidor de WebFlux se basan en los mismos códecs no bloqueantes para codificar y decodificar el contenido de las solicitudes y respuestas.

WebClient

- Internamente, el cliente web delega a una biblioteca de cliente HTTP.
- De forma predeterminada, utiliza Reactor Netty, hay un soporte integrado para el HttpClient reactivo de Jetty, y otros se pueden conectar a través de un ClientHttpConnector.

WebClient

- La forma más sencilla de crear un WebClient es a través de uno de los métodos de fábrica estática:
  - WebClient.create()
  - WebClient.create(String baseUrl)
- Los métodos anteriores usan el Reactor Netty HttpClient con la configuración predeterminada y esperan que io.projectreactor.netty: reactor-netty esté en la ruta de clase.

WebClient

- También puede usar WebClient.builder () con más opciones:
  - **uriBuilderFactory**: UriBuilderFactory personalizado para usar como una URL base.
  - **defaultHeader**: Encabezados para cada solicitud.
  - **defaultCookie**: Cookies para cada solicitud.
  - **defaultRequest**: Consumer para personalizar cada solicitud.
  - **filtro**: filtro de cliente para cada solicitud.
  - **exchangeStrategies**: personalizaciones del lector / escritor de mensajes HTTP.
  - **clientConnector**: configuración de la biblioteca del cliente HTTP.

WebClient

```
ExchangeStrategies strategies = ExchangeStrategies.builder()
        .codecs(configurer -> {
            // ...
        })
        .build();

WebClient client = WebClient.builder()
        .exchangeStrategies(strategies)
        .build();
```

WebClient

- El método retrieve () es la forma más fácil de obtener un cuerpo de respuesta y decodificarlo.
- El siguiente ejemplo muestra cómo hacerlo:

```
WebClient client = WebClient.create("http://example.org");

Mono<Person> result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .retrieve()
        .bodyToMono(Person.class);
```

WebClient

- También puede obtener una secuencia de objetos decodificados a partir de la respuesta, como muestra el siguiente ejemplo:

```
Flux<Quote> result = client.get()
        .uri("/quotes").accept(MediaType.TEXT_EVENT_STREAM)
        .retrieve()
        .bodyToFlux(Quote.class);
```

Otras capacidades

- URI's:
  - UriComponentsBuilder ayuda a crear URI a partir de plantillas de URI con variables, como muestra el siguiente ejemplo:

```
UriComponents uriComponents = UriComponentsBuilder
        .fromUriString("http://example.com/hotels/{hotel}")  1
        .queryParam("q", "{q}")  2
        .encode() 3
        .build(); 4

URI uri = uriComponents.expand("Westin", "123").toUri();  5
```

1) Método de fábrica estático con una plantilla URI.
2) Añadir o reemplazar componentes URI.
3) Solicitud para tener la plantilla URI y las variables URI codificadas.
4) Construir un UriComponents.
5) Expandir variables y obtener el URI.

Otras capacidades

- CORS:
  - Spring WebFlux te permite manejar CORS

    (Cross-Origin Resource Sharing

    ó Intercambio de recursos de origen cruzado)
- Seguridad web:
  - El proyecto Spring Security proporciona soporte para proteger aplicaciones web de ataques maliciosos.
  - Consulte la ***documentación de referencia de Spring Security***, que incluye:
    - Seguridad de WebFlux
    - Soporte de pruebas de WebFlux
    - Protección CSRF
    - Cabeceras de respuesta de seguridad

Otras capacidades

- WebSockets
- Testing