Spring WebFlux

Anotações de melhores partes do artigo: **Guide to Spring 5 WebFlux**

O Spring 5 inclui o Spring WebFlux, que fornece suporte de programação reativa para aplicativos da web.

Neste tutorial, criaremos um pequeno aplicativo REST reativo usando os componentes Web reativos *RestController* e *WebClient.*

Também veremos como proteger nossos endpoints reativos usando o Spring Security.

Estrutura Spring WebFlux:

O Spring WebFlux usa internamente o Project Reactor e suas implementações de publicador, *Flux* e *Mono* .

A nova estrutura suporta dois modelos de programação:

- Componentes reativos baseados em anotação
- Roteamento funcional e manipulação

Vamos nos concentrar nos componentes reativos baseados em anotação, pois já exploramos o estilo funcional – roteamento e manipulação em outro artigo.

Uma coisa que muda entre o Spring MVC e o Spring WebFlux é o **servlet container**, se temos uma aplicação Spring MVC precisamos do **Tomcat** para subir a aplicação, já se estivermos usando programação reativa com Spring WebFlux precisamos de um servidor que trabalha com tempo de execução assíncrona como é o servidor **netty**, então se formos utilizar programação reativa com Spring iremos subir a nossa aplicação no servidor **netty**.

Se estivermos usando o **Spring Boot** quando baixamos a dependência do Spring MVC ele trás um Tomcat imbutido para subir nossa aplicação, no caso se utilizarmos o Spring Webflux e baixar essa dependência ele não trás mais o **Tomcat** e sim o servidor **netty** que é o que trabalha com tempo de execução assíncrona, então ele é o mais apropríado para trabalhar com programação reativa.

Dependências:

Vamos começar com a dependência *spring-boot-starter-webflux* , que extrai todas as outras dependências necessárias:

- *spring-boot* and *spring-boot-starter* for basic Spring Boot application setup
- *spring-webflux* framework
- *reactor-core* that we need for reactive streams and also *reactor-netty*

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
    <version>2.6.4</version>
</dependency>
```

O mais recente spring-boot-starter-webflux pode ser baixado do Maven Central.

RestController reativo:

O Spring WebFlux suporta configurações baseadas em anotação da mesma forma que o framework Spring Web MVC.

Para começar, **no servidor, criamos um controlador anotado que publica um fluxo reativo do recurso *Employee* .**

Vamos criar nosso *EmployeeController* anotado:

```
@RestController
@RequestMapping("/employees")
public class EmployeeController {

    private final EmployeeRepository employeeRepository;

    // constructor...
}
```

EmployeeRepository pode ser qualquer repositório de dados que suporte fluxos reativos sem bloqueio.

Single Resource:

Em seguida, vamos criar um endpoint em nosso controlador que publique um único *recurso Employee:*

```
@GetMapping("/{id}")
public Mono<Employee> getEmployeeById(@PathVariable String id) {
    return employeeRepository.findEmployeeById(id);
}
```

Envolvemos um único recurso *Employee* em um *Mono* porque retornamos no máximo um funcionário.

Collection Resource:

Também adicionamos um endpoint que publica o recurso de coleta de todos os *funcionários*:

```
@GetMapping
public Flux<Employee> getAllEmployees() {
    return employeeRepository.findAllEmployees();
}
```

Para o recurso de coleção, usamos um *Flux* do tipo *Employee*, pois é o publicador de *0.* .n elementos.

Recuperando um único recurso:

Para recuperar um único recurso do tipo *Mono* do endpoint */employee/{id}*:

```
// Import the necessary classes
import reactor.core.publisher.Mono;

// Create a Mono object that retrieves an Employee object from a client
Mono<Employee> employeeMono = client.get()
  .uri("/employees/{id}", "1")
  .retrieve()
  .bodyToMono(Employee.class);

// Subscribe to the Mono object and print the Employee object to the console
employeeMono.subscribe(System.out::println);
```

Recuperando um recurso de coleção:

Da mesma forma, para recuperar um recurso de coleta do tipo *Flux* do endpoint */employees*:

```
Flux<Employee> employeeFlux = client.get()
  .uri("/employees")
  .retrieve()
  .bodyToFlux(Employee.class);

employeeFlux.subscribe(System.out::println);
```

Também tem um artigo detalhado sobre como configurar e trabalhar com o WebClient.

Segurança do Spring WebFlux:

Podemos usar o Spring Security para proteger nossos endpoints reativos.

Vamos supor que temos um novo endpoint em nosso *EmployeeController.* Este ponto de extremidade atualiza os detalhes *do funcionário* e envia de volta o *funcionário atualizado.*

Como isso permite que os usuários alterem os funcionários existentes, queremos restringir esse ponto de extremidade apenas aos usuários com função *ADMIN .*

Como resultado, vamos adicionar um novo método ao nosso *EmployeeController*:

```
@PostMapping("/update")
public Mono<Employee> updateEmployee(@RequestBody Employee employee) {
    return employeeRepository.updateEmployee(employee);
}
```

Agora, para restringir o acesso a este método, vamos criar *o SecurityConfig* e definir algumas regras baseadas em caminhos para permitir apenas usuários ADMIN:

```
import org.springframework.security.config.annotation.web.reactive.EnableWebFluxSecurity;
import org.springframework.security.config.web.server.ServerHttpSecurity;
import org.springframework.security.web.server.SecurityWebFilterChain;

@EnableWebFluxSecurity
public class EmployeeWebSecurityConfig {

    // ...

    /**
     * Configures the security filter chain for the application.
     *
     * @param http The ServerHttpSecurity object that is used to configure the security filter chain.
     * @return A SecurityWebFilterChain object that represents the configured security filter chain.
     */
    @Bean
    public SecurityWebFilterChain springSecurityFilterChain(
      ServerHttpSecurity http) {
        // Disable CSRF protection
        http.csrf().disable()
          // Authorize access to certain endpoints based on user roles
          .authorizeExchange()
          .pathMatchers(HttpMethod.POST, "/employees/update").hasRole("ADMIN")
          .pathMatchers("/**").permitAll()
          // Use HTTP basic authentication
          .and()
          .httpBasic();
        // Build and return the configured security filter chain
        return http.build();
    }
}
```

Essa configuração restringirá o acesso ao endpoint */employees/update*. Portanto, apenas usuários com função *ADMIN* poderão acessar esse terminal e atualizar um *funcionário existente.*

Por fim, a anotação *@EnableWebFluxSecurity* adiciona suporte ao Spring Security WebFlux com algumas configurações padrão.

Para obter mais informações, também tem um artigo detalhado sobre como configurar e trabalhar com a segurança do Spring WebFlux.

Handler & Router:

Também podemos criar end-points de outra forma sem ser pelo **Controller**, uma meneira funcional, que utiliza uma classe **Handler** que iremos nela determinar quais os métodos e também como ele vai responder a regras de negócio, já a classe **Router** ela que vai ser responsável por rotear todas as requisições e solicitações **HTTP** para a classe **Handler**, então ela determina que para determinada requisição qual método **Handler** que vai responder e montar a reposta para o cliente.

EX de uma Classe **Handler:**

```
@Component
public class PlaylistHandler {

    @Autowired
    PlaylistService playlistService;

    public Mono<ServerResponse> findAll(ServerRequest request) {
        return ok()
                .contentType(MediaType.APPLICATION_JSON)
                .body(playlistService.findAll(), Playlist.class /* O Playlist.class serve para informamos qual tipo de retorno que
    }

    public Mono<ServerResponse> findById(ServerRequest request) {
        String id = request.pathVariable("id");
        return ok()
                .contentType(MediaType.APPLICATION_JSON)
                .body(playlistService.findById(id), Playlist.class);
    }

    public Mono<ServerResponse> save(ServerRequest request) {
        final Mono<Playlist> playlistMono = request.bodyToMono(Playlist.class);  // Pegando a Playlist que está vindo no corpo da r
        return ok()
                .contentType(MediaType.APPLICATION_JSON)
                .body(fromPublisher(playlistMono.flatMap(playlistService::save), Playlist.class));  // Ficou diferente, pois temos
    }

    public Mono<ServerResponse> deleteById(ServerRequest request) {
        String id = request.pathVariable("id");
```

```
        return ok()
                .contentType(MediaType.APPLICATION_JSON)
                .body(playlistService.delete(id), Playlist.class);
    }

}
```

EX de uma Classe **Router**:

```
//Uma maneira de criar end-points usando programação com estilo funcional quando estamos trabalhando com Spring WebFlux.
@Configuration
public class PlaylistRouter {

    @Bean
    public RouterFunction<ServerResponse> route(PlaylistHandler playlistHandler) {
        return RouterFunctions
                .route(GET("/playlist").and(accept(MediaType.APPLICATION_JSON)), playlistHandler::findAll)
                .andRoute(GET("/playlist/{id}").and(accept(MediaType.APPLICATION_JSON)), playlistHandler::findById)
                .andRoute(POST("/playlist").and(accept(MediaType.APPLICATION_JSON)), playlistHandler::save)
                .andRoute(DELETE("/playlist/{id}").and(accept(MediaType.APPLICATION_JSON)), playlistHandler::deleteById);
    }

}
```

Conclusão

Neste artigo, exploramos como criar e trabalhar com componentes da Web reativos, conforme suportado pela estrutura Spring WebFlux. Como exemplo, construímos uma pequena aplicação Reactive REST.

Em seguida, aprendemos como usar *RestController* e *WebClient* para publicar e consumir fluxos reativos.

Também analisamos como criar um endpoint reativo seguro com a ajuda do Spring Security.

Além de Reactive *RestController* e *WebClient* , a estrutura *WebFlux também oferece suporte a WebSocket* reativo e o *WebSocketClient* correspondente para streaming de estilo de soquete de Reactive Streams.

Para mais informações, também temos um artigo detalhado focado em trabalhar com WebSocket Reativo com Spring 5.

Por fim, o código-fonte completo usado neste artigo está disponível no Github.