whitehorses

member of the human network

WHITEBOOK

Non-blocking Reactive Streams met Spring Webflux

Juni 2020

Auteur:
Mike Heeren
JAVA- EN INTEGRATIE SPECIALIST

Introductie

Bij het ontwikkelen van nieuwe applicaties is de trend steeds meer om zaken asynchroon en op de achtergrond te verwerken. Wanneer een applicatie grote hoeveelheden data moet verwerken, wil je niet dat de gebruiker niets anders kan doen tijdens het verwerken van de data. Denk hierbij bijvoorbeeld aan een applicatie die (grote) bestanden moet uploaden, controleren op virussen en ten slotte naar een ander formaat moet converteren. Dit zijn doorgaans operaties die lang duren. Het is daarom wenselijk om dit soort operaties “op de achtergrond” af te kunnen handelen.

Om dit voor elkaar te krijgen, is *Spring Webflux* sinds Spring 5 toegevoegd aan het Spring framework. *Spring Webflux* kan worden gebruikt voor het ontwikkelen van de services, die volgens het *Reactive Streams* principe werken. Hiermee wordt het mogelijk om operaties asynchroon en *non-blocking* uit te voeren. Voor de gebruiker lijkt het hierdoor dat de operaties op de achtergrond worden uitgevoerd.

In dit Whitebook nemen we een duik in *Spring Webflux* en hoe het toegepast kan worden.

Wat is Spring Webflux?

Voordat *Spring Webflux* aan het Spring portfolio werd toegevoegd, was *Spring Web MVC* het originele web framework voor het ontwikkelen van web services. *Spring Web MVC* bood ondersteuning voor de “klassieke” synchrone en *blocking* web services. Dat wil zeggen dat wanneer een gebruiker een operatie van de web service aanroept, deze moet wachten tot de server de aanvraag volledig verwerkt heeft. In de tussentijd kan de gebruiker de applicatie dan ook niet voor iets anders gebruiken.

In Java EE 7 werd *Servlet* implementatie 3.1 geïntroduceerd. Eén van de grote wijzigingen in deze versie was de introductie van *non-blocking* I/O. Dit betekent dat een client niet langer hoeft te wachten tot de server een response teruggeeft nadat de aanvraag volledig is afgewerkt, maar dat er een *Reactive Stream* wordt teruggegeven. Middels deze *Reactive Stream* kan de server op asynchrone, *non-blocking* wijze, meerdere antwoorden naar de applicatie sturen. Tijdens het wachten op antwoord(en) kan de gebruiker de applicatie ook voor andere zaken blijven gebruiken. Voor de gebruiker lijkt het dus alsof de operatie “op de achtergrond” wordt uitgevoerd.

We kunnen dit vergelijken met de *publish-subscribe* of *message queue* (topic) mechanismen. De server zal berichten op een *Reactive Stream* plaatsen, waar applicaties zich op kunnen “abonneren”. Zolang de applicaties op de stream geabonneerd zijn, zullen alle berichten die door de server aan de stream worden toegevoegd, worden ontvangen.

Project Reactor

Voor de ondersteuning van *Reactive Streams* maakt *Spring Webflux* standaard gebruik van *Project Reactor*. Project Reactor is een library die is gebaseerd op de *Reactive Streams* specificatie. Deze kan dus gebruikt worden om *non-blocking* Java applicaties te ontwikkelen.

Project Reactor biedt de *Mono* en *Flux* API’s aan. *Mono* kan worden gebruikt om 0 of 1 responses te retourneren aan de client. Met de *Flux* API daarentegen, kunnen 0 tot N responses worden teruggegeven.

Nieuwe Spring Webflux applicatie

We beginnen met het aanmaken van een nieuwe applicatie met Spring Boot. Hierin voegen we de volgende *dependencies* toe:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-webflux</artifactId>
    </dependency>
</dependencies>
```

Toepassen van de Mono API

We zijn nu klaar om een eerste web service operatie te ontwikkelen. Natuurlijk beginnen we hierbij met een eenvoudige '*Hello World*' methode. Hiervoor maken we een nieuwe klasse aan die we annoteren met *@RestController*. Hierin voegen we een *@GetMapping* toe, waarbij we als resultaat een implementatie van de *Mono* API teruggeven. In deze *Mono* wordt vervolgens de bekende groet geretourneerd.

```
@RestController
@RequestMapping("/hello")
public class HelloWorldController {

    @GetMapping
    private Mono<String> helloWorld(@RequestParam(required = false) String name) {
        return Mono.just(String.format("Hello, %s!", name));
    }

}
```

Zoals eerder vermeld, is het ook mogelijk om een *Mono* geen enkel resultaat terug te laten geven aan de client. Dit zouden we bijvoorbeeld kunnen doen wanneer de *name* parameter niet mee is gestuurd aan de operatie:

```
@GetMapping
private Mono<String> helloWorld(@RequestParam(required = false) String name) {
    if(name == null) {
        return Mono.never();
    }
    return Mono.just(String.format("Hello, %s!", name));
}
```

Wanneer we nu naar http://localhost:8080/hello gaan in de browser, is te zien dat de browser blijft "laden" (wachten op een response). Het heeft natuurlijk niet veel zin om een client "eindeloos" te laten wachten op een response waarbij we op voorhand al weten dat het nooit gaat komen (wanneer de *name* parameter leeg was, zal die in datzelfde request namelijk nooit meer gevuld gaan worden). In dat geval kunnen we *Mono.never()* beter vervangen door *Mono.empty()*. Hiermee wordt direct een leeg response teruggegeven.

Meer invloed op responses middels ResponseEntity

In de "klassieke" *Spring Web MVC* applicaties, was het mogelijk om de REST operatiemethoden een instantie van de *ResponseEntity* terug te laten geven. Hierdoor kan vanuit de methode niet enkel de inhoud van de response worden bepaald, maar ook metadata zoals *HTTP headers* en de *HTTP response code*. Ook met *Spring Webflux* is het mogelijk om de *Mono* en *Flux* API's te combineren met de *ResponseEntity* klasse. We kunnen de *helloWorld* methode bijvoorbeeld uitbreiden door een 404 code terug te geven wanneer er geen *name* is ingevuld.

```
@GetMapping
private ResponseEntity<Mono<String>> helloWorld(@RequestParam(required = false)
String name) {
    if (name == null) {
        return new ResponseEntity<>(Mono.empty(), HttpStatus.NOT_FOUND);
    }
    return new ResponseEntity<>(Mono.just(String.format("Hello, %s!", name)),
HttpStatus.OK);
}
```

Toepassen van de Flux API

Nu we de *Mono* API bekeken hebben, is het tijd om dieper op de *Flux* API in te gaan. Hiervoor maken we een nieuwe *GET* methode. Deze methode geeft een implementatie van de *Flux* klasse terug. Daarnaast geven we aan dat het resultaat van de methode een "*text/event-stream*" is. Dit is het officiële media type voor *Server Sent Events (SSE)*.

```
@GetMapping(path = "/multiple", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
private Flux<String> greetMultiple(@RequestParam String names) {
    List<String> greetings = Arrays.stream(names.split(","))
            .map(name -> String.format("Hello, %s!", name.trim()))
            .collect(Collectors.toList());
    return Flux.just(greetings.toArray(new String[0]));
}
```

Als we nu naar http://localhost:8080/hello/multiple?names=Mike,Whitehorses,Spring,Webflux gaan, zien we vrijwel meteen de vier begroetingen als *SSE verschijnen:*

```
data:Hello, Mike!

data:Hello, Whitehorses!

data:Hello, Spring!

data:Hello, Webflux!
```

Een interessantere use case voor de *Flux* API is echter dat de events niet direct verschijnen, maar dat deze met enige vertraging naar de client worden gestuurd. In het onderstaande voorbeeld is de methode aangepast zodat er elke seconde één groet wordt gestuurd.

Hiervoor gebruiken we de *DirectProcessor* klasse, wat een implementatie van de *Flux* API is. Belangrijk is dat na het sturen van het laatste event, de *onComplete()* methode wordt aangeroepen. Hierdoor weet de client dat er geen verdere responses meer zullen komen.

```
@GetMapping(path = "/multiple", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
private Flux<String> greetMultiple(@RequestParam String names) {
    List<String> greetings = Arrays.stream(names.split(","))
            .map(name -> String.format("Hello, %s!", name.trim()))
             .collect(Collectors.toList());

    DirectProcessor<String> response = DirectProcessor.create();
    new Thread(() -> {
        greetings.forEach(greeting -> {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                // Do nothing
            }
            response.onNext(greeting);
        });
        response.onComplete();
    }).start();
    return response;
}
```

Wanneer de *onComplete()* methode in het bovenstaande voorbeeld niet aangeroepen zou worden, blijft de client “oneindig” wachten op verdere responses, vergelijkbaar met het eerdere *Mono.never()* voorbeeld. Dit zou je eventueel voor een situatie kunnen gebruiken, waarbij er een “continue” stroom aan data van de server naar de client is.

In het onderstaande voorbeeld creëren we zo’n “continue” stroom aan data. Hiervoor registreren we de *DirectProcessor* klasse als instantie variabele. Ook voegen we twee HTTP operaties toe. Met de GET methode kunnen applicaties zich abonneren op nieuwe berichten. Deze berichten kunnen via de POST methode worden toegevoegd. Wanneer deze berichten worden toegevoegd, worden deze dus naar alle applicaties die zich via de *GET* methode hebben geabonneerd gestuurd.

Tevens is in dit voorbeeld te zien dat de klassieke *Spring Web MVC REST* methodes zonder problemen in combinatie met de *Webflux* operaties gebruikt kunnen worden. De *GET* operatie is namelijk een *Webflux* implementatie, terwijl de *POST* operatie volledig gebouwd is als *Spring Web MVC* operatie.

```
private final DirectProcessor<String> notifier = DirectProcessor.create();

@GetMapping(path = "/notifier", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
private Flux<String> greetNotifierConsumer() {
    return notifier;
}

@PostMapping(path = "/notifier")
private ResponseEntity<Void> greetNotifierProducer(@RequestParam(required = false)
String name) {
    notifier.onNext(String.format("Hello, %s!", name));
    return new ResponseEntity<>(HttpStatus.CREATED);
}
```

Ontsluiten met Angular

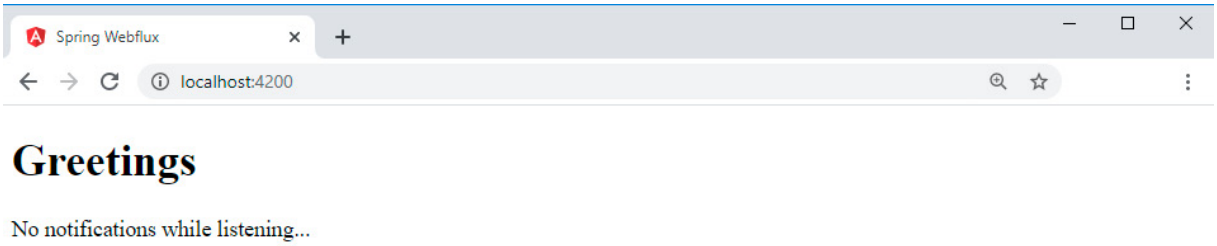
De laatste stap in het opzetten van de applicatie, is het ontsluiten van de *Server Sent Events* via een front-end. Hiervoor maken we gebruik van *Angular*. Voor de ondersteuning van *SSE* in *Angular* kan het *EventSource* object gebruikt worden. Deze ontsluiten we vervolgens via een *Observable* implementatie om de responses in de *notifications* variabele op te slaan.

```
import {ChangeDetectorRef, Component, OnInit} from '@angular/core';
import {Observable} from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent implements OnInit {

  notifications: string[] = [];

  constructor(private _changeDetector: ChangeDetectorRef) {}

  ngOnInit() {
    this.getGreetingNotifierObservable().subscribe((next: string) => {
      this.notifications.push(next);
      this._changeDetector.detectChanges();
    }, (error: any) => {
      console.log(error);
    }, () => {
      // Will never occur because we don't invoke the onComplete() in the back end
    });
  }

  private getGreetingNotifierObservable(): Observable<string> {
    return Observable.create(observer => {
      const eventSource = new EventSource('http://localhost:8080/hello/notifier');
      eventSource.onmessage = (event: MessageEvent) => observer.next(event.data);
      eventSource.onerror = (error: MessageEvent) => {
        // readyState === 0 (closed) means the remote source closed the
connection,
        // so we can safely treat it as a normal situation. Another way of
detecting the end of the stream
        // is to insert a special element in the stream of events, which the
client can identify as the last one.
        if (eventSource.readyState === 0) {
          eventSource.close();
          observer.complete();
        } else {
          observer.error('EventSource error: ' + error);
        }
      };
      return () => eventSource.close();
    });
  }

}
```

Nu de data in de *notifications* variabele wordt toegevoegd op het moment dat de server een nieuwe response stuurt, is het eenvoudig om deze data ook op het scherm te tonen:

```
<h1>Greetings</h1>
<p *ngIf="notifications.length === 0">No notifications while listening...</p>
<ul>
  <li *ngFor="let notification of notifications">{{notification}}</li>
</ul>
```

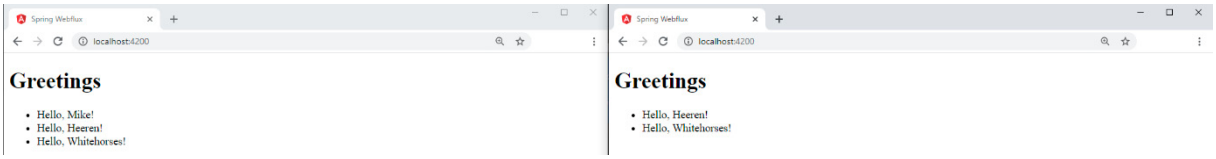
Wanneer de Angular applicatie wordt geopend in de browser, ziet deze er initieel als volgt uit:



Hierna voeren we opvolgend de volgende stappen uit:

- Het *POST* request: http://localhost:8080/hello/notifier?name=Mike
- Openen van een 2e browser scherm met http://localhost:4200
- Het *POST* request: http://localhost:8080/hello/notifier?name=Heeren
- Het *POST* request: http://localhost:8080/hello/notifier?name=Whitehorses

Het resultaat is dan dat de begroetingen in beide browsers getoond worden. Hierbij is het belangrijk dat het browser scherm dat ná het eerste *POST* request geopend werd, alleen de begroetingen toont die na het openen van de browser (dus na het abonneren op de *GET* operatie) gestuurd zijn.



Conclusie

Bij het ontwikkelen van (nieuwe) applicaties is de trend steeds meer om zaken zoveel mogelijk asynchroon en *non-blocking*, “op de achtergrond” af te handelen. Als je een groot bestand aan het uploaden bent naar de cloud, wil je tegelijkertijd ook de inhoud van een andere folder kunnen bekijken. Of wanneer je met internet bankieren een IBAN-nummer en naam invult, wil je dat je (tijdens het valideren van de combinatie van deze gegevens) direct verder kunt gaan met het invullen van de andere velden.

Spring Webflux lijkt uitstekend geschikt om mee te gaan in deze trend. Zeker in situaties waar clients asynchroon (tussen)statussen van de server willen ontvangen. Hiervoor pakken we het voorbeeld uit de inleiding er weer even bij. Via *Spring Webflux* zou de gebruiker bijvoorbeeld een tussenstatus kunnen ontvangen na het voltooien van de upload. Na het afronden van de virusscan kan de tussenstatus vervolgens weer bijgewerkt worden. Ten slotte kan de server terugkoppelen dat de aanvraag volledig verwerkt is wanneer het bestand ook succesvol geconverteerd is.

Tevens is het eenvoudig om te beginnen met *Spring Webflux* voor ontwikkelaars die al enige kennis van Spring (Boot) hebben. Dit komt mede doordat veel principes van het klassieke *Spring Web MVC* ook toepasbaar zijn in *Spring Webflux*.

Bronnen

Project Reactor
Reactive Streams
Spring - Web on Reactive Stack