Spring Cloud Stream

Message-Driven Microservices

In Java Magazine 04-2016 hebben we kennis gemaakt met Spring Cloud en hebben we kunnen lezen hoe je de verschillende componenten kunt inzetten om REST gebaseerde microservices te maken. In dit artikel maken we de stap naar Spring Cloud Stream. Spring Cloud Stream laat zich het beste omschrijven als een framework voor het creëren van message-driven microservices.

Waar REST gebaseerde microservices reageren op http-aanroepen, reageren message-driven microservices op berichten die afgeleverd worden. Daarvoor maakt Spring Cloud Stream gebruik van de message channels uit Spring Integration.

Deze message channels zijn binnen Spring Cloud Stream geïmplementeerd volgens het publish-subscribe model, wat het eenvoudig maakt om een bericht door meerdere services tegelijk te laten verwerken. Alle subscribers krijgen het bericht immers afgeleverd. Zonder aanpassingen van bestaande microservices kun je een nieuwe microservice toevoegen en bestaande berichten laten ontvangen en verwerken. Als je hetzelfde wilt bereiken bij REST gebaseerde microservices, dan zal de nieuwe microservice die het bericht moet ontvangen, aangeroepen moeten worden door de al bestaande microservices.

Bij de REST gebaseerde microservices wordt een discovery service gebruikt om de verschillende services te vinden en daarna te kunnen gebruiken. Bij message-driven microservices wordt gebruik gemaakt van één of meerdere message brokers voor het versturen en ontvangen van berichten. De ingezette message brokers hoeven niet allemaal hetzelfde te zijn. Je kunt bijvoorbeeld prima zowel RabbitMQ als ActiveMQ actief hebben binnen hetzelfde landschap van Spring Cloud Stream componenten. De details van de message brokers zijn binnen Spring Cloud Stream verstopt achter een abstractie die 'binder' wordt genoemd. Dit zorgt ervoor dat het voor de developer niet uitmaakt welke message broker gebruikt wordt. Het programmeermodel van Spring Cloud Stream blijft gelijk. Voor de populaire message brokers zijn standaard binder implementaties aanwezig. Voor message brokers waar geen implementatie is, kan deze zelf toegevoegd worden door gebruik te maken van de Binder SPI.

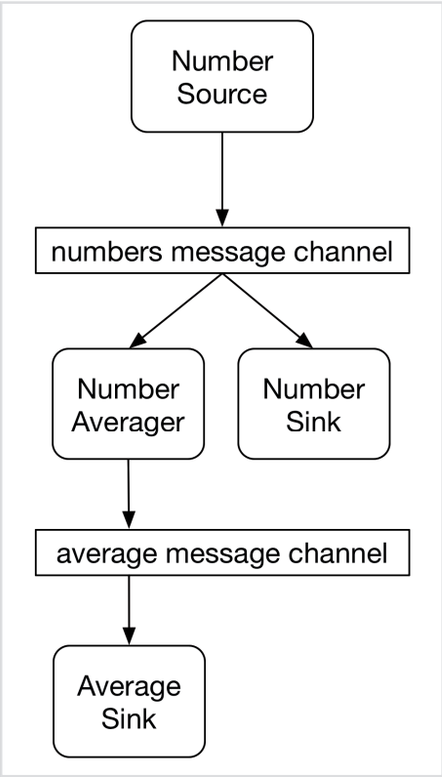
Spring Cloud Stream Applicaties

Zoals bij alle componenten in de Spring Cloud familie is ook een Spring Cloud Stream applicatie een Spring Boot applicatie. Spring Boot licht ik hier niet verder toe, bekijk daarvoor de Spring Boot referentie documentatie [1]. Om van een Spring Boot applicatie een Spring Cloud Stream applicatie te maken, maken we gebruik van de @EnableBinding annotatie. Deze accepteert één of meerdere interfaces die bindable componenten leveren. Bindable componenten zijn message channels die zijn gedefinieerd in deze interfaces en geannoteerd met ofwel @Input of @Output.

Er zijn een paar standaard bindable interfaces al gespecificeerd die we zo kunnen gebruiken. @Source voor het produceren van berichten, @Sink voor het ontvangen ervan en @Processor voor het ontvangen en weer doorsturen van berichten. Naast deze default interfaces kan elke interface die met @Input en @Output message channels definieert, gebruikt worden voor binding.

Afbeelding 1 geeft een voorbeeld aaneenschakeling van Spring Cloud Stream applicaties weer. Een viertal componenten zijn met elkaar verbonden via twee message channels. Er is een Number Source die berichten stuurt naar een Number Averager en een Number Sink. De Number Averager stuurt vervolgens weer een bericht naar de Averager Sink.

Bas Passon is werkzaam als software architect en Java consultant bij First8 BV. Op dit moment is hij vooral bezig met microservices in public en private cloud omgevingen.



Afbeelding 1: Spring Cloud Stream aaneenschakeling

De Praktijk

Dit klinkt natuurlijk allemaal erg mooi, maar hoe werkt dit dan allemaal? Dat zal ik laten zien aan de hand van een voorbeeldopstelling, een vereenvoudiging van **afbeelding 1**. We concentreren ons op de Number Source, verantwoordelijk voor het genereren van getallen en de Number Sink, verantwoordelijk voor het afdrukken van getallen.

Stream Source Applicatie

Zoals de naam al doet vermoeden, is dit de bron van berichten in onze voorbeeldopstelling. Zoals te zien is in **listing 1**, binden we de Source interface. Om te bereiken dat er berichten op het message channel terechtkomen kan de @InboundChannelAdapter annotatie geplaatst worden op een methode die een resultaat produceert. Als message channel voor de inbound adapter gebruiken we het output message channel van Source. Als we de applicatie starten, zal elke seconde een getal op het message channel afgeleverd worden.

Stream Sink Applicatie

De sink applicatie zal de gegenereerde getallen ontvangen en het ontvangen getal op de console afdrukken. De applicatie verschilt niet heel veel van de source applicatie. Met dien verstande dat we nu niet binden naar de Source interface uit **listing 1**, maar naar de Sink interface zoals we kunnen zien in **listing 2**. Hiermee geven we aan dat we berichten willen ontvangen via het input channel van de sink. De @ServiceActivator annotatie kan gebruikt worden om de methode te markeren die aangeroepen moet worden wanneer een bericht binnenkomt.

```
@EnableBinding(Source.class)
public class NumberSource {

    private static final Random rng = new Random();

    @InboundChannelAdapter(channel = Source.OUTPUT)
    public String generateNumber() {
        return Integer.toString(rng.nextInt(100));
    }
}
```

Listing 1: Number source

```
@EnableBinding(Sink.class)
@Slf4j
public class NumberSink {

    @ServiceActivator(inputChannel=Sink.INPUT)
    public void printNumber(String number) {
        log.info("Received number: {}", number);
    }
}
```

Listing 2: Number sink

Configuratie

Net als ik de eerste keer, kun je nu met de vraag zitten: 'Hoe komen de berichten van het output message channel naar het input message channel?'. Zoals zo vaak is het antwoord eenvoudiger dan je in eerste instantie zou vermoeden: configuratie. Via de application.yml file kun je aangeven hoe de mapping gemaakt moet worden. Zie hiervoor **listing 3** en **4**. Dit omvat de relevante stukken van de application.yml voor beide applicaties. Hier is aangegeven dat voor het output en input message channel dezelfde bestemming gebruikt moet worden. Omdat we gebruik maken van RabbitMQ zal in dit voorbeeld een topic exchange met de naam numbers gebruikt worden.

```
spring:
  cloud:
    stream:
      bindings:
        output:
          destination: numbers
```

Listing 3: Number source application.yml

```
spring:
  cloud:
    stream:
      bindings:
        input:
          destination: numbers
```

Listing 4: Number sink application.yml

STANDAARD IS ELKE SPRING CLOUD STREAM APPLICATIE ZIJN EIGEN CONSUMER GROUP, WAARBIJ DE NAAM RANDOM GEGENEREERD WORDT

Message Payload Conversie

Bij gebruik van de @ServiceActivator annotatie ontvang je de payload van het bericht. De *SimpleTypeConverter* wordt gebruikt voor conversie van de payload. Als deze niet toegepast kan worden, krijg je de ruwe content van het bericht en zal je zelf conversie moeten doen naar iets wat betekenis heeft voor het component. Niet iets waar je als ontwikkelaar op zit te wachten en dat hoeft gelukkig ook niet. Je kunt gebruik maken van de annotatie @StreamListener om Spring Cloud Stream, op basis van de content type van het bericht, automatisch een conversie te laten doen.

In tegenstelling tot @ServiceActivator kan bij @SteamListener geen output channel gedefinieerd worden, waardoor deze feature op het eerste gezicht niet heel handig lijkt. Gelukkig voor ons is hier al over nagedacht. Mocht je wel een return waarde voor de methode met de @SteamListener annotatie willen gebruiken, dan kan gebruik worden gemaakt van de @SendTo annotatie. De return waarde wordt dan ook meteen geconverteerd naar het gewenste content type van het output message channel. Het content type kan geconfigureerd worden per channel. Zie **listing 5** voor een message channel configuratie die JSON conversie zal toepassen, mits het bericht daar geschikt voor is natuurlijk.

Aggregatie

Onze voorbeeldapplicatie bestaat uit twee microservices die los van elkaar opgestart kunnen worden en via RabbitMQ als message broker met elkaar verbonden zijn via message channels. Om performance redenen wil je soms wel de decompositie in verschillende services, maar wil je liever geen berichten uitwisselen via een message broker. Je kunt dan onder voorwaarden, die je kunt terug vinden in de referentie documentatie [2], de applicaties aggregeren. Spring Cloud Stream biedt hiervoor de AggregateApplicationBuilder class. In **listing 6** is weergegeven hoe dit werkt als we aggregatie toepassen op het voorbeeld.

Horizontaal Schalen

In het artikel uit Java Magazine 04-2016 hebben we kunnen lezen dat voor de REST gebaseerde microservices Eureka (discovery) samen met Ribbon (loadbalancing) zorgen voor het schaalbare aspect van de microservices. Op het moment dat een tweede instantie van een specifieke service opgestart wordt, zal Ribbon op basis van de informatie uit Eureka de load verdelen over beide services.

```
spring:
  cloud:
    stream:
      bindings:
        helloworld:
          destination: helloworld
          content-type: application/json
```

Listing 5: JSON content type voor helloworld message channel

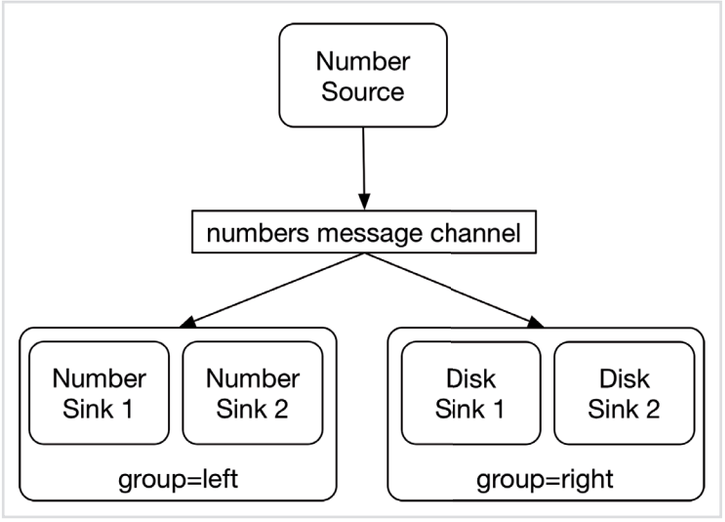
```
@SpringBootApplication
public class AggregateApplication {

   public static void main(String[] args) {
      new AggregateApplicationBuilder()
            .from(NumberSourceApplication.class)
            .to(NumberSinkApplication.class).run(args);
   }
}
```

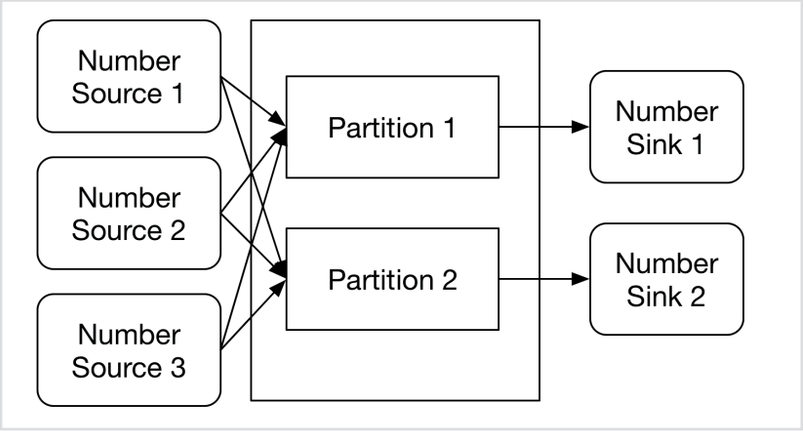
Listing 6: Voorbeeld aggregatie

Message gebaseerde microservices maken geen gebruik van een discovery service. Ze binden zich alleen aan een message channel. Kunnen we deze microservices dan wel horizontaal schalen? Ja, dat kan. In het begin hebben we even heel kort aangestipt dat message channels geïmplementeerd zijn volgens het publish-subscribe model met als bijkomstigheid dat een tweede microservice die het bericht ook wil ontvangen alleen maar naar hetzelfde message channel hoeft te luisteren. Dit is prima in het geval het een andere microservice betreft die een andere verwerking van het bericht doet.

Wat nu als we meerdere dezelfde microservices willen starten en dus horizontaal schalen? Deze extra microservices zullen dan gaan luisteren naar hetzelfde message channel en allemaal het bericht ontvangen en verwerken. In veel gevallen is dit niet het



Afbeelding 2: Consumer Groups



Afbeelding 3: Partitionering

gewenste gedrag. Berichten mogen vaak maar één keer verwerkt worden om te voorkomen dat rare effecten optreden. Om dit te bereiken, heb je binnen Spring Cloud Stream het concept van consumer groups. Alle leden van dezelfde consumer group worden gezamenlijk gezien als één afnemer van berichten en via een simpel loadbalancing algoritme zal één microservice uit de groep gekozen worden die het bericht gaat verwerken. **Afbeelding 2** geeft schematisch weer hoe dit zou kunnen werken voor onze voorbeeldapplicatie als we die uitbreiden met een Disk Sink en deze onder brengen in twee consumer groups, left en right, met elk twee leden. Beide consumer groups zullen elk bericht ontvangen, maar hierbinnen zal steeds maar één lid het bericht verwerken.

Standaard is elke Spring Cloud Stream applicatie zijn eigen consumer group, waarbij de naam random gegenereerd wordt. Je kan de consumer group zelf aanpassen door gebruik te maken van de *spring.cloud.stream.bindings.[channel].group* configuratie instelling. Hier kun je de naam van de consumer group opgeven waar de microservice bij gaat horen. Consumer groups zijn de manier om het verwerken van berichten te verspreiden over meerdere dezelfde microservices.

Tip: Bepaal altijd zelf de naam van de consumer group, daarmee houd je volledige controle over wie welke berichten verwerkt en kom je niet voor verrassingen te staan.

Deze manier van schalen is eenvoudig en werkt in veel gevallen uitstekend. Als je echter te maken hebt met het verwerken van informatie die aan elkaar gerelateerd is, waarbij het ook nog eens belangrijk is dat deze informatie door dezelfde microservice verwerkt wordt, dan werken consumer groups niet. Je hebt immers geen invloed op welke microservice welk bericht gaat verwerken. Om deze informatiestromen te verwerken, kun je gebruik maken van partitionering. **Afbeelding 3** geeft schematisch weer hoe dit eruitziet voor onze voorbeeldapplicatie.

Deze manier van schalen is wel iets complexer dan het gebruik van consumer groups. Deze manier vereist dat zowel de source als de sink informatie hebben over de partities die in gebruik zijn. Voor de details over hoe partitionering geconfigureerd moet worden, verwijs ik naar de referentie documentatie [2].

Testen

Om het testen zo gemakkelijk mogelijk te maken, is test support beschikbaar waarmee je elke microservice afzonderlijk kunt testen zonder gebruik te hoeven maken van een feitelijke message broker. Hiermee kun je JUnit tests schrijven die alles afdekken inclusief de in- en uitgaande berichten. Voor het opvangen van de berichten kun je gebruik maken van de MessageCollector. Zie **listing 7** voor een voorbeeld.

Conclusie

Met Spring Cloud Stream is het gemakkelijk om een schaalbaar (en testbaar!) systeem te maken. Door deze uitbreiding op Spring Cloud is het triviaal om stream-based programming te doen en zo een schaalbare decompositie te maken van je applicatie. ■

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class StreamSourceApplicationTests {

   @Autowired
   private Source source;

   @Autowired
   private MessageCollector messageCollector;

   @Test
   @SuppressWarnings("unchecked")
   public void shouldHaveMessage() throws Exception {
      Message<?> message = messageCollector.forChannel(source.output())
            .poll(3, TimeUnit.SECONDS);
      System.out.println(message + " " + message.getPayload().getClass());
      Assertions.assertThat(message).isNotNull();
   }
}
```

Listing 7: Message collector voorbeeld.

REFERENTIES

[1] **http://docs.spring.io/spring-boot/docs/1.4.1.RELEASE/reference/html/**
[2] **http://docs.spring.io/spring-cloud-stream/docs/1.0.3.RELEASE/reference/html**