whitehorses

member of the human network

WHITEBOOK

Enterprise Integratie Patronen met Spring

December 2020

Auteur:
Mike Heeren
INTEGRATIESPECIALIST

Introductie

Een tijdje terug heb ik een Whitebook geschreven over Full Code integratie met Apache Camel. In dit Whitebook nam ik een duik in Apache Camel, een framework dat ondersteuning biedt voor het ontwikkelen van *Enterprise Integration Patterns*. In dit Whitebook nemen we een duik in een ander framework wat hiervoor gebruikt kan worden: Spring Integration.

Enterprise Integration Patterns

Enterprise Integration Patterns (EIP) kunnen worden gebruik voor het ontwerpen van integratie oplossingen tussen verschillende applicaties. EIP is niet verbonden aan een specifieke technologie, maar een “abstracte” set patronen die door veel integratie frameworks omarmd wordt.

In veel producten waar de integratie specialisten van Whitehorses mee werken, komen deze patronen dan ook terug. Denk hierbij aan Messaging oplossingen als ActiveMQ of *Enterprise Service Bus* (ESB) oplossingen als Oracle Service Bus, MuleSoft, of Apache Camel. Ook Spring heeft een ESB framework waarmee EIP-oplossingen kunnen worden gerealiseerd. Dit is Spring Integration.

Wat is Spring Integration

Spring Integration is het Spring project dat gebruikt kan worden voor het ontwikkelen van EIP-integraties. Het ondersteunt de ontwikkeling van lichtgewicht *messaging* binnen (bestaande) Spring applicaties. Via zogenaamde *adapters* en *gateways* wordt functionaliteit ontsloten zoals het verbinden/integreren met externe systemen, messaging oplossingen en *scheduling*:

- *Adapters* gebruiken het *one-way*-principe. Inkomend ontvangen ze dan berichten van een *endpoint*, of uitgaand verzenden ze een bericht naar een *endpoint*, maar nooit beide.
- *Gateways* gebruiken het *request-reply*-principe. Inkomend ontvangen ze dan ook berichten, echter zullen ze ook een antwoord naar de aanroepende partij sturen. Bij uitgaande berichten, wachten ze na het verzenden van het uitgaande bericht ook op een antwoord van een *endpoint*.

Spring Integration omschrijft zichzelf als een project waarbij het primaire doel is, om op eenvoudige wijze, Enterprise integratie oplossingen te kunnen realiseren waarbij *Separation of Concerns* (SoC) hoog in het vaandel staat. Bij SoC wordt er bij het ontwerpen van de software rekening gehouden dat de software in verschillende onderdelen (een zogenaamd "*concern*") wordt opgeknipt. Wanneer je een project hebt waarbij 4 systemen moeten worden geïntegreerd, zou je kunnen stellen dat "integratie 1" van systeem A met systeem B een *concern* is. Daarnaast is "integratie 2" van systeem X naar systeem Y is een ander *concern*. Als deze onderdelen goed gescheiden zijn in de code, kunnen er bijvoorbeeld probleemloos wijzigingen in "integratie 1" worden doorgevoerd, zonder dat er wijzigingen nodig zijn voor "integratie 2". Dit moet eraan bijdragen om tot goed onderhoud- en testbare code te komen.

Met Spring Integration is het mogelijk om de integraties te realiseren via XML bestanden. Via de *Java DSL* is het echter ook mogelijk om de integraties volledig in Java te ontwikkelen. Een *domain-specific language* (DSL) is een programmeertaal gespecialiseerd in een specifiek (applicatie)domein. In dit geval is dit dus een DSL die het op eenvoudige wijze mogelijk maakt om integraties te realiseren in het Spring Integration domein middels Java.

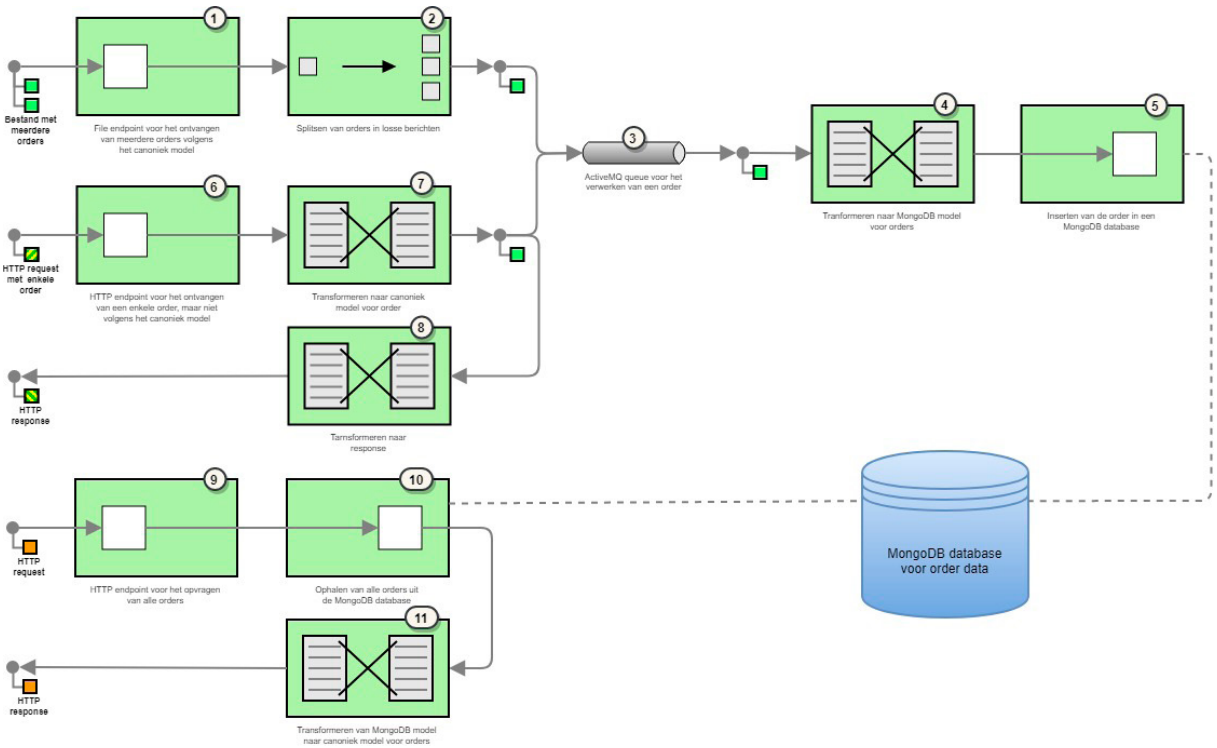
Omdat dit ook een project uit de Spring portefeuille is, kan het naadloos gecombineerd worden met andere Spring projecten, zoals Spring Boot. Hiermee kunnen dus nog sneller en makkelijker integratieprojecten gerealiseerd worden.

Realisatie van de demo applicatie

Tijdens het schrijven van het Whitebook over Apache Camel, heb ik een demo applicatie opgezet, welke diverse zaken bevat die vaak voorkomen bij het opzetten van integratie tussen systemen. Denk hierbij aan zaken als:

- Het oppakken en verwerken van bestanden.
- HTTP/REST *endpoints* voor het registreren en inzien van order(s).
- Transformeren van data (JSON, XML, POJO).
- Gebruik van (ActiveMQ) *messaging queues*.
- Opslaan van data in een (MongoDB) database.

Bij het schrijven van dit Whitebook zullen we een vergelijkbare applicatie bouwen, maar nu met Spring Integration in plaats van Apache Camel. Onderstaand is de EIP-diagram terug te vinden, met het ontwerp van de demo applicatie. In dit hoofdstuk zal ik op verschillende plaatsen verwijzen naar componenten uit het diagram door “(x)” verwijzingen toe te voegen.



Afbeelding 1: EIP-diagram opzet demo applicatie.

Kickstart met Spring Boot

Ook bij de realisatie van deze demo applicatie zullen we gebruik maken van Spring Boot. Dit is een project van Spring, die het mogelijk maakt om op eenvoudige wijze met minimale configuratie een Spring applicatie op te zetten.

Hiervoor voegen we de onderstaande parent toe aan het Maven project:

GROUP ID	ARTIFACT ID	VERSION
org.springframework.boot	spring-boot-starter-parent	2.4.0

Tevens voegen we de onderstaande *dependencies* toe. Hierbij wordt de *spring-boot-starter-integration* vanzelfsprekend als “startpunt” voor Spring Integration projecten gebruikt. Daarnaast voegen we de *spring-boot-starter-web* toe voor het openstellen van de HTTP *endpoints* (6 / 9). Ook voor ActiveMQ (3) en MongoDB (5 / 10) gebruiken we starter projecten. We hoeven van deze *dependencies* geen versies te specificeren. Deze worden namelijk overgenomen uit het hierboven genoemde *parent* project.

GROUP ID	ARTIFACT ID
org.springframework.boot	spring-boot-starter-integration
org.springframework.boot	spring-boot-starter-web
org.springframework.boot	spring-boot-starter-activemq
org.springframework.boot	spring-boot-starter-data-mongodb

Hierna hoeven we enkel een eenvoudige Java klasse toe te voegen, deze met *@SpringBootApplication* te annoteren en te voorzien van een eenvoudige *main* methode om de applicatie te kunnen starten.

```java
package nl.mikeheeren.springintegration;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringIntegrationApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringIntegrationApplication.class, args);
    }

}
```

Opzetten van de eerste integratie flow

Nu de basis van de applicatie staat, kunnen we onze eerste *IntegrationFlow* opzetten. Dit is de *Java DSL* abstractie van een integratie. Deze klasse registreren we als *@Bean* in de Spring context.

De eerste *IntegrationFlow* is de implementatie van het *file endpoint* (1) in het EIP-diagram. Omdat we gebruik willen maken van zowel een *file* als een ActiveMQ (JMS) *endpoint*, voegen we eerst de volgende *dependencies* toe:

GROUP ID	ARTIFACT ID
org.springframework.integration	spring-integration-file
org.springframework.integration	spring-integration-jms

Nu kunnen we een *IntegrationFlow* op gaan bouwen die een *inbound file* adapter als startpunt heeft. Omdat we het bestand enkel van het *file system* op willen pakken, zonder een antwoord naar dit systeem te sturen, gebruiken we geen *gateway* maar een *adapter*. Deze adapter stellen we zo in dat we de directory waar we bestanden oppakken, via de *order.file.location* property in het *application.properties* bestand kunnen configureren.

In de *poller* configureren we dat er elke 5(000 milli)seconden wordt gecheckt of er bestanden klaarstaan in de *directory*. Vervolgens kijken we in de *filter* of de bestandsnaam *orders.json* is. Ook specificeren we in het filter wat er moet gebeuren als het bericht niet aan het filter voldoet. De rest van de flow wordt dan ook niet meer uitgevoerd. Als er een ander bestand in de directory wordt geplaatst, zal dus enkel gelogd worden dat het een ongeldig bestand was.

Wanneer het bestand aan het filter voldoet, wordt het (canonieke) JSON bericht vertaald naar een array van de *Order* POJO. Deze wordt vervolgens opgeknipt (2) in losse berichten via de *split* methode. Nu roepen we per (losse) *Order* een *outbound JMS-adapter endpoint* (3) aan. Ook hier zijn we niet geïnteresseerd in een response van ActiveMQ, dus wederom gebruiken we een *adapter*. Deze configureren we middels de *JmsTemplate bean*. Deze *bean* is door de *spring-boot-starter-activemq dependency* automatisch toegevoegd op basis van gegevens uit het *application.properties* bestand.

Ten slotte loggen we een bericht nadat alle opgeknipte berichten verwerkt zijn. Dit is het geval als de berichten weer samengevoegd zijn door de aggregate methode.

```
@Bean
public IntegrationFlow fromFile(@Value("${order.file.location}") String
orderFileLocation,
                                JmsTemplate jmsTemplate) {
    return IntegrationFlows.from(Files.inboundAdapter(new
File(orderFileLocation)),
            e -> e.poller(Pollers.fixedDelay(5000)))
            .filter("headers.file_name == 'orders.json'",
                    filterEndpointSpec -> filterEndpointSpec.discardFlow(df
-> df
                            .log(LoggingHandler.Level.INFO, null,
"'Invalid file received: ' + headers.file_name")
                            .nullChannel()))
            .transform(new JsonToObjectTransformer(Order[].class))
            .split()
                .publishSubscribeChannel(s -> s
                    .subscribe(flow -> flow.handle(Jms.
outboundAdapter(jmsTemplate).destination("queue:order"))))
            .aggregate()
            .log(LoggingHandler.Level.INFO, null, "'Finished dispatching
orders to queue'")
            .get();
}
```

Opslaan van data in MongoDB

Nu we de berichten op de ActiveMQ queue toe kunnen voegen, is het tijd voor een *IntegrationFlow* die de berichten ook weer van deze queue afleest (4) en opslaat in MongoDB (5).

Hiervoor voegen we de volgende dependency toe:

GROUP ID	ARTIFACT ID
org.springframework.integration	spring-integration-mongodb

Na de *inbound adapter* gebruiken we nu een *publishSubscribeChannel*. Dit is een *channel* dat berichten naar elke *subscriber* doorstuurt. We hebben deze constructie nodig omdat we na het aanroepen van de *MongoDbStoringMessageHandler* willen loggen of het opslaan van de data succesvol was. Als we deze *handle* echter niet in een *channel* zouden *wrappen*, kunnen we deze log actie niet meer toevoegen. Omdat we een *outbound adapter* gebruiken, kunnen er namelijk geen stappen na de *handle* worden uitgevoerd (in tegenstelling tot na een *outbound gateway*).

De *MongoTemplate bean*, die we gebruiken voor het MongoDB *endpoint*, is beschikbaar gemaakt door de *spring-boot-starter-data-mongodb* starter.

```
@Bean
public IntegrationFlow fromAmqOrderQueue(JmsTemplate jmsTemplate,
                                         MongoTemplate mongoTemplate) {
    MongoDbStoringMessageHandler mongoDbStoringMessageHandler = new
MongoDbStoringMessageHandler(mongoTemplate);
    mongoDbStoringMessageHandler.setCollectionNameExpression(new
LiteralExpression("orders"));

    return IntegrationFlows.from(Jms.inboundAdapter(jmsTemplate).
destination("queue:order"),
            e -> e.poller(Pollers.fixedDelay(5000)))
            .publishSubscribeChannel(s -> s
                    .subscribe(flow -> flow.
handle(mongoDbStoringMessageHandler))
                    .subscribe(flow -> flow.log(LoggingHandler.Level.INFO,
null, "'Successfully inserted into collection: ' + payload.id"))
            )
            .get();
}
```

REST endpoint voor opslaan orders

Hierna voegen we ook het HTTP endpoint (6) toe. Dit is een endpoint wat berichten in XML-formaat kan ontvangen. Hiervoor moeten we de volgende dependencies toevoegen:

GROUP ID	ARTIFACT ID
org.springframework.integration	spring-integration-http
org.springframework.integration	spring-integration-xml

Omdat we deze keer wel een antwoord terug willen sturen naar het aanroepende systeem, gebruiken we nu een *inbound gateway*. Verder specificeren we dat het een HTTP *endpoint* is dat POST *requests* verwacht. Ook geven we aan wat de *content types* zijn voor het bericht dat we consumeren (*request*) en produceren (*response*).

We voegen een *errorChannel* toe. Deze wijst naar de (losse) *httpPostOrderErrorChannel IntegrationFlow*. In deze *flow* construeren we het fout-response bericht (8) en zorgen we dat de HTTP-statuscode voor een *Internal Server Error* (500) wordt teruggegeven.

Nadat het bericht wordt ontvangen, transformeren (7) we het eerst van XML naar POJO. Vervolgens transformeren we deze POJO naar het canonieke model. Dit doen we door een eigen implementatie van de *AbstractPayloadTransformer* klasse.

Hierna plaatsen we het bericht weer op de *JMS-queue*, op dezelfde wijze als we ook doen voor het *file endpoint*. Wanneer het bericht op de *queue* geplaatst is, wordt het *response* bericht opgebouwd (8).

```
@Bean
public IntegrationFlow httpPostOrder(JmsTemplate jmsTemplate) {
    Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
    marshaller.setClassesToBeBound(nl.mikeheeren.springintegration.model.
xml.Order.class);

    return IntegrationFlows.from(Http.inboundGateway("/orders")
            .requestMapping(m -> m.methods(HttpMethod.POST)
                    .consumes("application/xml")
                    .produces("application/json")
            )
            .errorChannel("httpPostOrderErrorChannel.input"))
            .transform(new UnmarshallingTransformer(marshaller))
            .transform(new TransformOrderXmlToCanonical())
            .publishSubscribeChannel(s -> s
                    .subscribe(flow -> flow.handle(Jms.
outboundAdapter(jmsTemplate).destination("queue:order")))
                    .subscribe(flow -> flow.transform(source -> {
                        HashMap<String, String> response = new
HashMap<>();
                        response.put("message", "Successfully created
order");
                        return response;
                    })))
            .get();
}

@Bean
public IntegrationFlow httpPostOrderErrorChannel() {
    return f -> f
            .enrichHeaders(c -> c.header(HttpHeaders.STATUS_CODE,
HttpStatus.INTERNAL_SERVER_ERROR))
            .transform(source -> {
                HashMap<String, String> response = new HashMap<>();
                response.put("message", "Failed to insert order");
                return response;
            });
}
```

REST endpoint voor ophalen orders

De laatste flow is het HTTP endpoint (9) voor het ophalen van alle orders uit MongoDB (10). In deze flow maken we zowel voor het inbound- als het outbound endpoint gebruik van een gateway. Dit is nodig omdat we nu ook daadwerkelijk een antwoord van MongoDB verwachten.

In de MongoDbOutboundGateway specificeren we al dat we het canonieke Order object verwachten terug te krijgen. We hoeven dit object niet meer expliciet naar JSON te casten (11). Dit wordt door de HTTP-gateway afgehandeld.

```
@Bean
public IntegrationFlow httpGetOrders(MongoTemplate mongoTemplate) {
    MongoDbOutboundGateway mongoDbOutboundGateway = new
MongoDbOutboundGateway(mongoTemplate);
    mongoDbOutboundGateway.setEntityClass(Order.class);
    mongoDbOutboundGateway.setQueryExpression(new
LiteralExpression("{}"));
    mongoDbOutboundGateway.setCollectionNameExpression(new
LiteralExpression("orders"));

    return IntegrationFlows.from(Http.inboundGateway("/orders")
            .requestMapping(m -> m.methods(HttpMethod.GET)
                    .produces("application/json")))
            .handle(mongoDbOutboundGateway)
            .get();
}
```

Demo applicatie

De volledige code van de demo applicatie is terug te vinden in Bitbucket. In de src/test/resources folder van de repository zijn tevens een voorbeeld bestand voor het file endpoint en een Postman collectie met voorbeeld requests voor de HTTP endpoints terug te vinden:

https://bitbucket.org/whitehorsesbv/spring-integration-demo

Conclusie

Met Spring Integration kunnen op een snelle manier integraties worden gebouwd. Dit kan middels XML worden gedaan, maar ook volledig in Java via de Java DSL. Omdat het een project uit de Spring portefeuille is, sluit het natuurlijk ook naadloos aan op andere Spring projecten zoals Spring Boot.

Dit maakt Spring Integration een erg logische keuze wanneer een organisatie al gebruik maakt van andere Spring projecten, waarbij vervolgens geïntegreerd moet worden met andere systemen.

Ook biedt Spring Integration ondersteuning voor een aantal standaard protocollen als File, FTP(S), SFTP, HTTP en MongoDB. Het volledige overzicht aan ondersteunde *endpoint* types kan in de Spring Integration Reference Documentation worden teruggevonden. Echter, wanneer we de (standaard ondersteunde) *endpoints* afzetten tegen Apache Camel (Zie de Apache Camel Components voor het volledige overzicht), moeten we toch concluderen dat Apache Camel een veel uitgebreidere ondersteuning biedt voor veel meer protocollen.

Tevens biedt Apache Camel meer mogelijkheden in het modelleren van de integratie *flows* zelf. Een voorbeeld hiervan is de ondersteuning van *choice-when/if-else* strucuren in de route. Hiervoor zit geen directe ondersteuning in Spring Integration, waardoor je dit via een *filter* moet implementeren:

```
IntegrationFlows.from(Files.inboundAdapter(new File(orderFileLocation)),
        e -> e.poller(Pollers.fixedDelay(5000)))
        .filter("headers.file_name == 'orders.json'",
                filterEndpointSpec -> filterEndpointSpec.discardFlow(df ->
df
                        // Do something else
                        .nullChannel()))
        // Do something
        .get();
```

In Apache Camel kan zoiets wel gewoon direct binnen een *flow* worden gedaan. In mijn ogen houdt dit de code een stuk overzichtelijker en daardoor dus beter onderhoudbaar.

```
from("file:{{order.file.location}}")
    .choice()
        .when(header("CamelFileName").isEqualTo("orders.json"))
            // Do something
        .endChoice()
        .otherwise()
            // Do something else
        .endChoice()
    .end();
```

Samenvattend vind ik Spring Integration een mooi framework, en zeker in organisaties waar ook andere projecten van Spring gebruikt worden, zal dit project goed en snel toegepast kunnen worden.

Echter, wanneer er wat "complexere" (bijv. if-else) constructies, of integraties met "exotischere" endpoints zoals HDFS, AWS of Azure geïmplementeerd moeten worden, zou mijn persoonlijke voorkeur op dit moment naar Apache Camel uitgaan.

Bronnen

Enterprise Integration Patterns
Spring Integration
Spring Integration – Reference Documentation