whitehorses

member of the human network

WHITEBOOK

Full Code integratie met Apache Camel

Augustus 2019

Auteur:
Mike Heeren
INTEGRATIESPECIALIST

Inleiding

Voor het ontwikkelen van applicaties zijn tegenwoordig steeds meer *No Code en Low Code* frameworks beschikbaar. Dit zijn frameworks waarmee applicaties gemaakt kunnen worden, terwijl daar weinig tot geen code voor geschreven hoeft te worden. Denk hierbij bijvoorbeeld aan APEX of OutSystems. Voor het ontwikkelen van integraties zijn daarnaast vele frameworks beschikbaar waar het weliswaar mogelijk is om zelf (onderdelen) te programmeren, maar waar zaken als de route die een bericht door de integratie neemt voornamelijk in het formaat van het betreffende framework (bijvoorbeeld XML) moeten worden vastgelegd. Voorbeelden hiervan zijn de Oracle Service Bus en MuleSoft.

In dit Whitebook nemen we een duik in het tegenovergestelde van de *No- en Low* Code stroming. Hiervoor gaan we kijken naar een framework waarmee volledige integratie trajecten zijn te programmeren in Java: Apache Camel.

Wat is Apache Camel?

Apache Camel is een open-source framework voor het ontwikkelen van enterprise integratie patronen. Voor het sturen, verwerken en routeren van het bericht door de integratie kan gebruik worden gemaakt van de Java *Domain Specific Language* (DSL).

Onderstaand is een code voorbeeld te zien van deze Java DSL syntax. In het voorbeeld is te zien hoe een bestand van een locatie kan worden gelezen. Aan de hand van de bestandsnaam wordt vervolgens een fout gelogd, of de (interne) route *direct:processFile* aangeroepen. Deze routeert dit bericht op zijn beurt naar een ActiveMQ queue. Het resultaat hiervan wordt ten slotte gebruikt om te loggen of een bericht succesvol of foutief is verwerkt.

```
1. from("file:/file/location")
2.     .choice()
3.         .when(header("CamelFileName").isEqualTo("test.txt"))
4.             .to("direct:processFile")
5.         .endChoice()
6.         .otherwise()
7.             .log("Invalid file received: ${headers.CamelFileName}")
8.         .endChoice()
9.     .end();
10. from("direct:processFile")
11.      .to("activemq:queue:fileQueue")
12.      .choice()
13.          .when(simple("${body.status} == '200'"))
14.              .transform(simple("Successfully processed file: ${body.filename}"))
15.          .endChoice()
16.          .otherwise()
17.              .transform(simple("Failed to process file: ${body.status} -
    ${body.message}"))
18.     .end();
```

Het is ook mogelijk om, in plaats van de Java DSL, XML configuratiebestanden te gebruiken voor het routeren van berichten door de integratie. In dit Whitebook ligt de focus echter op de Java DSL. We willen immers een full code integratie ontwikkelen.

Apache Camel is eenvoudig uit te bereiden met verschillende beschikbare componenten om bijvoorbeeld protocollen als HTTP, JMS, FTP en JDBC te ondersteunen. De volledige lijst met ondersteunde componenten is hier te vinden.

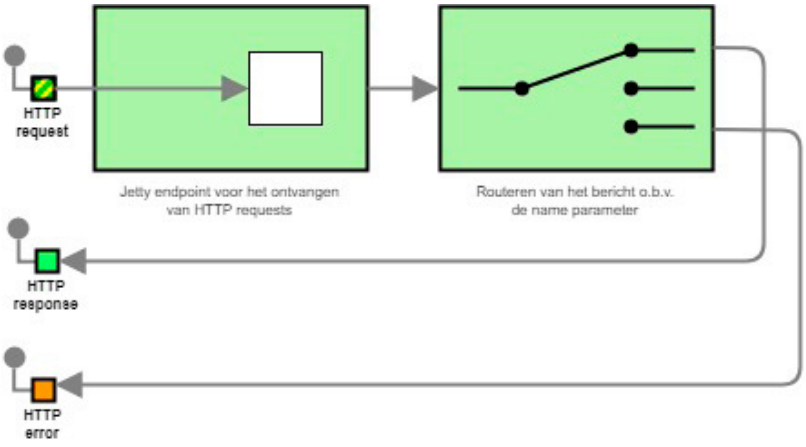
Eerste Apache Camel applicatie

Om meer gevoel te krijgen bij (de werking van) Apache Camel beginnen we met het opzetten van een eenvoudige HTTP integratie.

Belangrijk: Aan de Camel distributions is een breed scala aan voorbeeldprojecten toegevoegd. Om wat dieper in Apache Camel te duiken, is het dan ook een aanrader om deze voorbeeldprojecten eens te bekijken.

Opzet van de applicatie

We beginnen met de implementatie van een eenvoudige HTTP integratie. Op basis van de naam die met dit HTTP request wordt meegestuurd, zal worden bepaald of er een succesvol - of error resultaat terug wordt gegeven. In het onderstaande *Enterprise Integration Patterns* (EIP) diagram is deze verwerking schematisch weergegeven.



Afbeelding 1: EIP diagram van de eerste applicatie.

Dependency management via Maven

Voor de implementatie van de applicatie maken we gebruik van Maven voor het beheren van de dependencies. Hieraan voegen we de onderstaande dependencies toe.

GROUP ID	ARTIFACT ID	VERSION
org.apache.camel	camel-core	2.24.1
org.apache.camel	camel-jetty	2.24.1

De *camel-core* dependency is de basis voor een Apache Camel applicatie. Voor het openstellen van het HTTP endpoint gebruiken we Jetty, een HTTP (web) server. Om Jetty te kunnen gebruiken in Apache Camel, is ook de *camel-jetty* dependency toegevoegd.

De versie van beide dependencies is 2.24.1. Dit is momenteel de laatste release van Apache Camel.

CamelContexts, RouteBuilders en Exchanges

Nadat de dependencies zijn toegevoegd aan het Maven project, is het relatief eenvoudig om te beginnen met het ontwikkelen van Apache Camel. Hiervoor dient een *CamelContext* instantie te worden aangemaakt. De *CamelContext* kan worden gezien als de ruggengraat van de Camel applicatie. Dit is de verbindende factor tussen concepten als routes, componenten en endpoints.

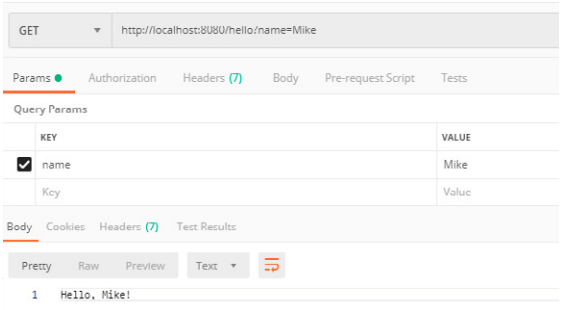
Aan de *CamelContext* kan vervolgens een *RouteBuilder* worden toegevoegd. Dit is een klasse waarin de routes geïmplementeerd worden. Beide uiteinden van een route (dus zowel het begin- als het eindpunt, of de *from()* en *to()* binnen de routes) worden endpoints genoemd.

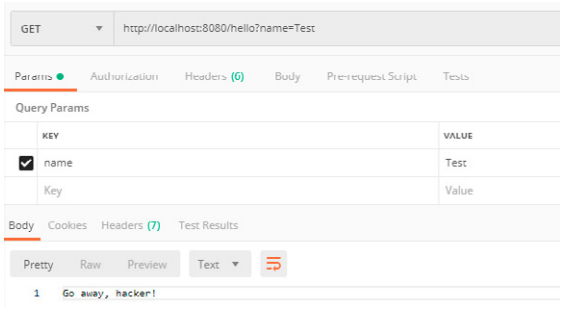
Een ander belangrijk concept in Apache Camel is de *Exchange* implementatie. Dit is een container voor alle data met betrekking tot een bericht gedurende een route. Zowel de huidige waarde van het bericht, als het opgebouwde response bericht zijn hier beschikbaar. Daarnaast is ook metadata voorhanden. Afhankelijk van het protocol van het inkomende endpoint zijn bijvoorbeeld bestandsnamen (voor endpoints die bestanden lezen) of de HTTP methode (bij HTTP endpoints) op te vragen. Een *Exchange* bericht ondersteunt een aantal patronen. Zo kan het *InOut* patroon gebruikt worden wanneer er een response bericht verwacht wordt. Het *InOnly* patroon kan gebruikt worden voor fire-and-forget operaties.

Nadat de RouteBuilder implementatie aan de CamelContext is toegevoegd, hoeven we de context enkel nog te starten.

```
1. package nl.whitehorses.camel;
2.
3. import org.apache.camel.CamelContext;
4. import org.apache.camel.Exchange;
5. import org.apache.camel.builder.RouteBuilder;
6. import org.apache.camel.impl.DefaultCamelContext;
7.
8. public class MyFirstCamelApplication {
9.   public static void main(final String[] args) throws Exception {
10.      final CamelContext context = new DefaultCamelContext();
11.      context.addRoutes(new RouteBuilder() {
12.        public void configure() {
13.          from("jetty:http://0.0.0.0:8080/hello")
14.              .choice()
15.                .when(header("name").isEqualTo("Mike"))
16.                  .setBody(simple("Hello, ${in.header.name}!"))
17.                  .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(200))
18.                .otherwise()
19.                  .setBody(simple("Go away, hacker!"))
20.                  .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(500));
21.        }
22.      });
23.      context.start();
24.   }
25. }
```

Na het starten van de applicatie, kunnen we HTTP requests sturen naar http://localhost:8080/hello. Op basis van de waarde van de header *name* wordt vervolgens een goed - of fout bericht teruggegeven.





Apache Camel in combinatie met Spring

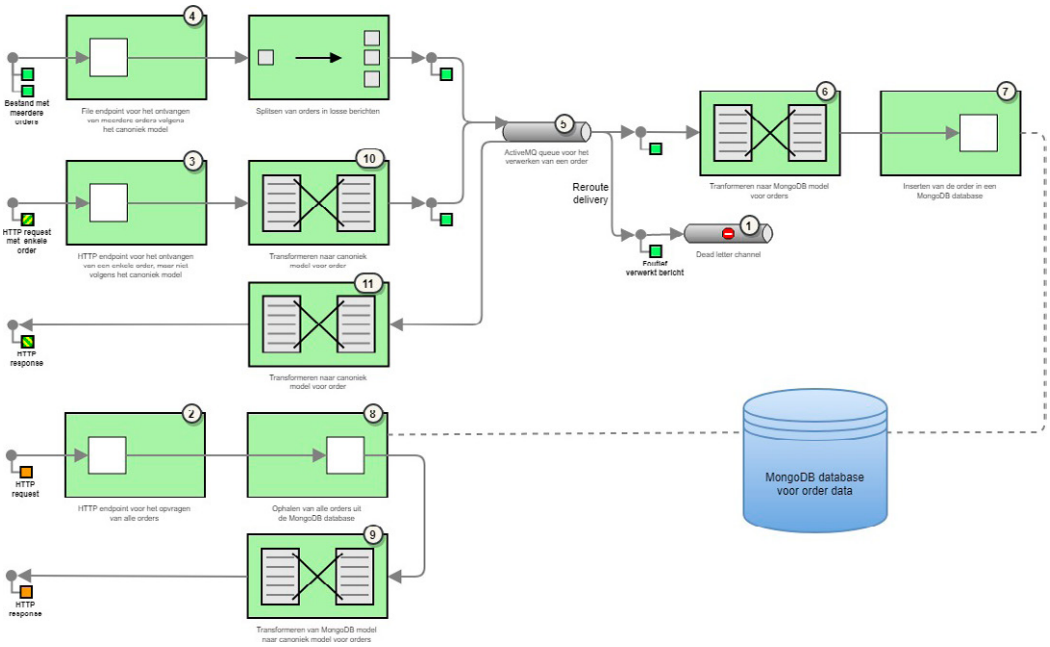
Nu we een eenvoudig voorbeeld van een Apache Camel applicatie hebben gezien, is het tijd voor complexere integraties. Hiervoor zullen we Apache Camel combineren met het Java framework Spring. Deze frameworks werken namelijk naadloos samen. Zo kunnen de (Spring) *Beans* bijvoorbeeld direct vanuit de Apache Camel router worden benaderd. Daarnaast kunnen de Apache Camel routers automatisch worden gedetecteerd en gestart met de Spring applicatie. Hier hoeven we dus, in tegenstelling tot in het eerdere voorbeeld, geen rekening mee te houden.

Opzet demo applicatie met Apache Camel, Spring Boot, ActiveMQ en MongoDB

Als demo applicatie maken we een integratie voor *order* data. Deze orders kunnen op 2 manieren aan de integratie worden aangeboden. Met de getallen in de onderstaande tekst wordt gerefereerd aan componenten van het bijbehorende EIP diagram terug te vinden zijn.

- Via een *file* endpoint [4] kunnen meerdere orders tegelijk worden aangeboden.
- Via een HTTP endpoint [3] kunnen losse orders worden aangeboden.

Voor het verwerken van de orders (via beide ingangen) wordt gebruik gemaakt van *message queuing*. Hiervoor gebruiken we een ander product van Apache, namelijk ActiveMQ [5]. Berichten die niet verwerkt konden worden, zullen uitvallen naar een ActiveMQ d*ead letter channel* [1]. Via dit *message queueing* principe kunnen berichten die een fout veroorzaken (bijvoorbeeld wanneer een systeem tijdelijk niet bereikbaar is), meerdere keren opnieuw geprobeerd worden zonder tussenkomst van het aanroepende systeem.
De opslag van de orders zal worden gedaan in een MongoDB [7] [8] database. Ten slotte zijn de opgeslagen orders ook op te halen via een HTTP endpoint [2].



Afbeelding 4: EIP diagram opzet Apache Camel, Spring Boot, ActiveMQ en MongoDB.

De kickstart met Spring Boot

We starten de applicatie met het opzetten van een Spring applicatie. Hiervoor zullen we Spring Boot gebruiken. Dit is een project bovenop het Spring framework, wat het mogelijk maakt om op eenvoudige wijze, met minimale configuratie, een applicatie op te zetten.

Net als het eerdere voorbeeld, gebruiken we Maven voor het beheren van de dependencies. We baseren het project op de onderstaande *parent*:

GROUP ID	ARTIFACT ID	VERSION
org.springframework.boot	spring-boot-starter-parent	2.1.6.RELEASE

Ook voegen we de volgende dependency toe:

GROUP ID	ARTIFACT ID	VERSION
org.springframework.boot	spring-boot-starter-web	Geen, overerven van parent

Ten slotte maken we de Java klasse *Application* aan, die als startpunt van de applicatie gaat fungeren. Deze hoeven we nu enkel te voorzien van de *@SpringBootApplication* annotatie en een eenvoudige implementatie van de *main* methode om de applicatie te kunnen starten.

```
1. package nl.whitehorses.camel;
2.
3. import org.springframework.boot.autoconfigure.SpringBootApplication;
4. import org.springframework.boot.web.servlet.ServletRegistrationBean;
5.
6. @SpringBootApplication
7. public class Application {
8.   public static void main(final String[] args) {
9.     SpringApplication.run(Application.class, args);
10.  }
11.}
```

Een RouteBuilder, maar nu zonder een CamelContext?

Om Apache Camel te gebruiken binnen Spring Boot, kunnen we gebruikmaken van de *Spring Boot Starter*. Hiervoor voegen we de onderstaande Maven dependency aan het project toe.

GROUP ID	ARTIFACT ID	VERSION
org.apache.camel	camel-spring-boot-starter	2.24.1

Nu kunnen we beginnen met het implementeren van de Camel *RouteBuilder*. Echter hoeft deze, in tegenstelling tot het eerdere voorbeeld, niet meer “handmatig” te worden toegevoegd aan de *CamelContext*. Ook hoeft deze niet meer zelf gestopt en gestart te worden.

Wanneer de Spring annotatie *@Component* wordt toegevoegd aan de implementatie van de *RouteBuilder*, zal deze automatisch gedetecteerd en gestart worden met de applicatie.

```
1. package nl.whitehorses.camel;
2.
3. import org.apache.camel.builder.RouteBuilder;
4. import org.springframework.stereotype.Component;
5.
6. @Component
7. public class CamelRouter extends RouteBuilder {
8.     public void configure() {
9.        // Implementatie van de routes
10.    }
11.}
```

RESTful API's met Apache Camel

In het eerste voorbeeld werd gebruik gemaakt van een Jetty endpoint voor het ontvangen van HTTP requests. Omdat we nu gebruikmaken van Spring, kunnen we echter de standaard Spring *servlets* gebruiken voor het openstellen van HTTP endpoints.

Om HTTP (REST) endpoints open te stellen via een Spring *servlet*, moet de onderstaande dependency worden toevoegen:

GROUP ID	ARTIFACT ID	VERSION
org.apache.camel	camel-servlet	2.24.1

Vervolgens moeten we een *CamelHttpTransportServlet* aan de applicatie toevoegen. Dit doen we door de volgende *bean* aan de *Application* klasse toe te voegen.

```
1. @Bean
2. public ServletRegistrationBean camelServletBean() {
3.     final ServletRegistrationBean servlet = new ServletRegistrationBean(new CamelH
   ttpTransportServlet(), "/camel/*");
4.     servlet.setName("CamelServlet");
5.     servlet.setLoadOnStartup(1);
6.     return servlet;
7. }
```

In de *RouteBuilder* kunnen we nu REST endpoints definiëren op de volgende manier:

```
1. rest("/orders")
2.     .get()
3.         .produces("application/json")
4.         .to("direct:getOrders")
5.     .post().consumes("application/xml")
6.         .produces("application/json")
7.         .to("direct:postOrder");
```

Message queuing met ActiveMQ

Om gebruik te maken van ActiveMQ queues, moeten de volgende Maven dependencies worden toegevoegd aan het project:

GROUP ID	ARTIFACT ID	VERSION
org.apache.camel	activemq-broker	Geen, overerven van parent
org.apache.camel	activemq-camel	Geen, overerven van parent
org.apache.camel	camel-jms	2.24.1

Vervolgens moet een *ActiveMQConnectionFactory* instantie beschikbaar worden gemaakt. Hierin kunnen onder andere de *broker URL* van ActiveMQ en de trusted packages worden ingesteld. Objecten die in de trusted package staan, kunnen vervolgens (geserialiseerd) via ActiveMQ verzonden en ontvangen worden. Ook de *ActiveMQConnectionFactory* kan via een *Bean* in de *Application* klasse beschikbaar gesteld worden.

```
1. @Bean
2. public ActiveMQConnectionFactory activeMqConnectionFactoryBean() {
3.     final ActiveMQConnectionFactory activeMqConnectionFactory = new ActiveMQConnect
   ionFactory();
4.     activeMqConnectionFactory.setBrokerURL("tcp://localhost:61616");
5.     activeMqConnectionFactory.setTrustedPackages(Collections.singletonList("nl.white
   horses.camel"));
6.     return activeMqConnectionFactory;
7. }
```

Nu kunnen de ActiveMQ endpoints [5] (zowel inkomend als uitgaand) worden gespecificeerd in de *RouteBuilder*. Daarnaast zullen we ook gebruikmaken van een (dead letter) queue [1]. Wanneer er op de flow geen specifieke *errorHandler* is ingesteld, zullen berichten die niet verwerkt kunnen worden hierdoor automatisch uitvallen naar een dead letter queue:

```
1. errorHandler(deadLetterChannel("activemq:queue:nameOfDeadLetterQueue"));
2.
3. from("activemq:queue:nameOfInboundQueue")
4.     .to("activemq:queue:nameOfOutboundQueue");
```

MongoDB voor het opslaan van data

Een andere externe applicatie die we gebruiken is MongoDB. Dit gaat worden gebruikt voor het opslaan van de data. Ook voor de connectie met deze database kan een Maven dependency worden toegevoegd.

GROUP ID	ARTIFACT ID	VERSION
org.apache.camel	camel-mongodb	2.24.1

Net als voor het ontsluiten van REST endpoints en het gebruik van ActiveMQ, is voor de koppeling met MongoDB ook een *Bean* nodig. Echter, in tegenstelling tot de 2 eerder toegevoegde beans, is de naam van de *Bean* hier wél van belang. De naam van de *Bean* moet namelijk overeenkomen met de naam die we in de *RouteBuilder* URL gaan gebruiken.

```
1. @Bean
2. public MongoClient mongoClientBean() {
3.     return new MongoClient("localhost", 27017);
4. }
```

Omdat er geen expliciete naam aan de *Bean* is toegekend, wordt impliciet de naam van de functie gebruikt. In dit geval dus *mongoClientBean*. In de *RouteBuilder* kunnen verzoeken aan de MongoDB database [7] [8] nu als volgt worden geïmplementeerd.

```
1. from("direct:findAll")
2.     .to("mongodb:mongoClientBean?
   database=mongoDatabaseName&collection=mongoCollectionName&operation=findAll").
```

Verschillende manieren van transformeren

Er zijn verschillende manieren voor de transformatie van data in Apache Camel. In de demo applicatie worden verschillende wijzen van transformeren getoond.

(Un)marshallen van data

Om binnen Apache Camel te interacteren met data, is het van belang dat het presentatie formaat (bijvoorbeeld JSON of XML) eerst omgezet wordt naar een *Plain Old Java Object* (POJO). Dit wordt het *unmarshallen* van data genoemd. Het *marshallen* van data, doet logischerwijs het tegenovergestelde. Hiermee kan een POJO dus omgezet worden in het gewenste presentatie formaat.

Hiervoor kan gebruik worden gemaakt van diverse soorten *marshallers* [9] [11]. In de demo applicatie worden hiervoor o.a. XML en JSON gebruikt. Wanneer bij het *unmarshallen* géén Java klasse wordt gespecificeerd, wordt de data omgezet naar key-value data in de vorm van een *Map*.

```
1. from("direct:unmarshalToClassAndMarshal")
2.     .unmarshal().json(JsonLibrary.Jackson, Order[].class)
3.     .marshal().json(JsonLibrary.Jackson);
4. from("direct:unmarshalToMap")
5.     .unmarshal().jacksonxml();
```

Voor het *(un)marshallen* van XML [10] is wel een extra dependency nodig:

GROUP ID	ARTIFACT ID	VERSION
org.apache.camel	camel-jacksonxml	2.24.1

Transform, simple en constant

De "eenvoudigste" transformaties kunnen worden gedaan via de *transform()* en *simple()* of *constant()* transformatie [11]. Hierbij kan de *constant()* (logischerwijs) gebruikt worden om een statische waarde in te stellen. De *simple()* daarentegen is ooit gestart als "eenvoudig" component. In de loop der tijd zijn er echter flink wat functies aan toegevoegd. Zo kunnen er onder andere *bodies*, *headers*, *CamelContext* waarden en omgevingsvariabelen mee worden uitgelezen.

```
1. from("direct:getStaticValue")
2.     .transform(constant("Dit is een statische waarde"));
3.
4. from("direct:getIdFromBody")
5.     .setBody(() -> {
6.         HashMap<String, String> example = new HashMap<>();
7.         example.put("id", "test");
8.         return example;
9.     })
10.    .transform(simple("Het ID is: ${body[id]}"));
```

(Inline) processors

Naast de *setBody()*, die ook al te zien is in het voorbeeld van de *simple()* transformatie, is het ook mogelijk om complexere transformaties te doen via een *process()*. Aan deze methode moet een implementatie van een *Processor* klasse worden meegegeven.

Dit kan *inline* worden gedaan [11], dus dat de implementatie van de *Processor* klasse direct in de *RouteBuilder* wordt opgebouwd. Echter, wanneer de logica bijvoorbeeld complex wordt, of dat de logica op meerdere plekken hergebruikt moet kunnen worden, kan hiervoor natuurlijk ook een losse Java klasse voor worden gebruikt [6] [9].

```
1. from("direct:inlineProcessor")
2.     .process(exchange -> {
3.         final Order body = exchange.getIn().getBody(Order.class);
4.         exchange.getIn().setBody("Het order ID is: " + body.getId());
5.     });
6. from("direct:processorClass")
7.     .process(new MyCustomProcessor());
```

Belangrijk: Let op dat de nieuwe *body* niet via een *return* statement wordt ingesteld, maar dat dit via de Exchange parameter moet worden gedaan!

Beans

Ten slotte is het ook mogelijk om een *Bean* te gebruiken voor het transformeren van de data. Als voorbeeld nemen we de volgende *transformationBean*. Via de *@Body* annotatie wordt de huidige *body* geïnjecteerd in de parameters, waardoor deze uit te lezen is.

```
1. package nl.whitehorses.camel.beans;
2.
3. import nl.whitehorses.camel.model.Order;
4. import org.apache.camel.Body;
5. import java.util.String;
6.
7. public class TransformationBean {
8.     public String transformBody(@Body final Order body) {
9.         return "Het order ID is: " + body.getId();
10.    }
11.}
```

In de *RouteBuilder* kunnen we deze als volgt gebruiken voor het uitvoeren van de transformatie.

```
1. from("direct:transformViaBean")
2.     .bean(TransformationBean.class, "transformBody");
```

Deployen op Tomcat

Om de Spring Boot en Apache Camel applicatie te kunnen bouwen zodat hij op een Apache Tomcat applicatieserver gedraaid kan worden, moet de *packaging* naar *war* worden gewijzigd in het *pom.xml* bestand. Tevens moet de onderstaande dependency nog toegevoegd worden.

GROUP ID	ARTIFACT ID	VERSION
org.springframework.boot	spring-boot-starter-tomcat	Geen, overerven van parent

Ten slotte moet de *Application* klasse nu een extensie worden van de *SpringBootServletInitializer* klasse.

De demo applicatie

De volledige code van de demo applicatie is terug te vinden op Bitbucket. Tevens is er een *Dockerimage* te vinden, waarmee een Docker image met een werkende proefomgeving opgebouwd kan worden. Deze omgeving is dus voorzien Tomcat, ActiveMQ en MongoDB.

https://bitbucket.org/whitehorsesbv/spring-boot-camel

De *CamelRouter* implementatie bevat in het commentaar in de code tevens de verwijzingen naar de componenten uit het bijbehorende EIP diagram.

In de *src/test/resources* folder van de repository is een Postman collectie te vinden met voorbeelden van de REST requests. Hiernaast is in deze folder een voorbeeldbestand toegevoegd dat aan het *file* endpoint aangeboden kan worden.

Conclusie

Met Apache Camel kunnen op een snelle manier, volledig in Java code, integraties worden gebouwd. Zeker de combinatie met Spring (Boot) is erg sterk, maar ook externe applicaties als ActiveMQ en MongoDB zijn naadloos geïntegreerd in het framework.

Met behulp van de Java DSL zijn op een snelle, overzichtelijke manier routes van berichten door de integraties heen te ontwikkelen.

Voor complexere logica kan waar Apache Camel geen standaardoplossing biedt altijd uitgeweken worden naar het gebruik van een *Bean* of het maken van een eigen *Processor*. Hiermee is op eenvoudige wijze complexere logica te implementeren. Omdat je zelf de code hiervan implementeert, zijn de mogelijkheden “eindeloos”.

Vergeleken met *No* - en *Low code* platformen kan de learningcurve wel wat hoger zijn voor ontwikkelaars die nog niet veel ervaring hebben met Java-ontwikkeling. Wanneer je echter thuis bent in de Java-standaarden, vind je snel je weg met Apache Camel.

Bronnen:

Apache Camel
Enterprise Integration Patterns
Building an Application with Spring Boot
Introduction to Apache Camel