whitehorses
member of the human network

WHITEBOOK

Camel K: Gebundelde krachten van Camel en Kubernetes

oktober 2021

Auteur:
Mike Heeren
INTEGRATIESPECIALIST

Introduction

Bij het starten van nieuwe projecten, zien we tegenwoordig steeds vaker dat er voor microservices, containerized en/of serverless applicaties wordt gekozen. In een eerder Whitebook nam ik al een kijkje hoe Apache Camel kan worden toegepast bij integratieprojecten. Met Apache Camel is het namelijk mogelijk om integraties volledig in (Java) code te ontwikkelen. Apache Camel heeft nu een nieuw onderdeel toegevoegd aan het portfolio, genaamd Camel K. Dit is een lichtgewicht integratie framework dat speciaal bedoeld is om Camel routes direct op Kubernetes of OpenShift te laten draaien.

Installatie

Bij het schrijven van dit Whitebook is gebruikgemaakt van een locale MiniKube (v1.23.2) installatie. MiniKube is een lokale installatie van Kubernetes, waarbij de focus ligt om op snelle en eenvoudige wijze aan het werk te gaan met (software)ontwikkeling op Kubernetes.

Om te beginnen met het ontwikkelen met Camel K hoeft alleen de "kamel" *Command Line Interface* (CLI) geïnstalleerd te worden. Deze kan zowel gebruikt worden om Kubernetes/ OpenShift clusters te configureren, maar ook voor het starten en stoppen van routes.

Omdat we gebruikmaken van MiniKube, is er nog een benodigde (tussen)stap noodzakelijk. Camel K maakt gebruik van *registries*. Dit is een intern register waar (Docker) images naartoe gepusht kunnen worden.

Kortgezegd zijn Docker images nog geen actieve containers. Een image is alleen de "blauwdruk" die beschrijft hoe een container opgebouwd dient te worden. Wanneer het commando wordt gegeven om een container daadwerkelijk op te starten (op basis van een image), zal de container worden ingericht (volgens de blauwdruk) en opgestart. Bij het starten van een container, zal Kubernetes de images dus weer ophalen uit het interne register.

Om de *registries* feature in te schakelen, dient het onderstaande commando te worden uitgevoerd:

```
$ minikube addons enable registry
```

Nu het MiniKube cluster klaar is, voeren we het volgende commando uit om het cluster te prepareren en de Camel K operator te installeren.

```
$ kamel install
```

Opzetten eerste route

Nu de (ontwikkel)omgeving geconfigureerd is, is het tijd om een eerste route op te zetten. Een route is hoe een integratie in Apache Camel wordt gedefinieerd. Zo beschrijft deze de *source* (from-endpoint), *destination(s)* (to-endpoints) en alle stappen die daartussen genomen moeten worden, zoals validatie en transformatie van de data.
Wat direct opvalt, is dat met weinig moeite snel een eenvoudige route opgezet kan worden met Camel K. Om een leeg skelet voor een Camel K integratie op te zetten, kan gebruik worden gemaakt van het “`init`” commando:

```
$ kamel init FirstRoute.java
```

In dit geval zal dus een Java route worden aangemaakt. Echter, Camel K biedt ondersteuning voor veel meer talen, zoals XML, YAML, Groovy, Kotlin, JShell en JavaScript.

Het gegenereerde bestand ziet er als volgt uit:

```
// camel-k: language=java

import org.apache.camel.builder.RouteBuilder;

public class FirstRoute extends RouteBuilder {
  @Override
  public void configure() throws Exception {

      // Write your routes here, for example:
      from(“timer:java?period=1000”)
        .routeId(“java”)
        .setBody()
          .simple(“Hello Camel K from ${routeId}”)
        .to(“log:info”);

  }
}
```

Deze route kunnen we direct starten door het onderstaande commando uit te voeren:

```
$ kamel run FirstRoute.java
```

Met het commando “`kamel get`” kunnen we een overzicht opvragen van alle Camel K integraties:

```
NAME              PHASE   KIT
first-route     Running default/kit-XXXXXXXXXXXXXXXXXXX
```

Logging

Omdat het een simpele route is, die elke seconde “Hello Camel K from java” zal loggen, kan alleen in de logging geverifieerd worden of de applicatie goed werkt. Ook dit kunnen we doen via de kamel CLI:

```
$ kamel log first-route
```

Hierdoor zal de logging van de applicatie getoond worden:

```
[1] ←[39m←[38;5;145m2021-11-08 19:56:44,293←[39m←[38;5;188m
←[39m←[38;5;107mINFO ←[39m←[38;5;188m
[←[39m←[38;5;69minfo←[39m←[38;5;188m] (←[39m←[38;5;71mCamel
(camel-1) thread #0 - timer://java←[39m←[38;5;188m)
←[39m←[38;5;151mExchange[ExchangePattern: InOnly, BodyType: String, Body:
Hello Camel K from java]←[39m←[38;5;203m←[39m←[38;5;227m
[1] ←[39m←[38;5;145m2021-11-08 19:56:45,272←[39m←[38;5;188m
←[39m←[38;5;107mINFO ←[39m←[38;5;188m
```

Het is ook mogelijk om *structured* logs te produceren vanuit een Camel K route. Dit kan ingesteld worden via zogenaamde *traits*. *Traits* zijn high-level configuraties om het uiteindelijke gedrag van een integratie te beïnvloeden. Traits kunnen worden toegevoegd door de “`--trait`” of “`-t`” parameter toe te voegen aan het run-commando.
Wanneer tools als Splunk of Kibana worden gebruikt voor het analyseren van logging, is het handig als logging in een standaard (structured) formaat wordt gelogd. Door de json optie van de logging trait te activeren, zal de logging als JSON-structuur worden gerepresenteerd:

```
$ kamel run FirstRoute.java -t logging.json=true
Integration “first-route” created
$ kamel log first-route
```
…

```
[1] {"timestamp":"2021-11-08T20:01:24.556Z","se
quence":126,"loggerClassName":"org.slf4j.impl.
Slf4jLogger","loggerName":"info","level":"INFO","message":"Exchange[ExchangePattern:
InOnly, BodyType: String, Body: Hello Camel K from java]","threadName":"Camel
(camel-1) thread #0 - timer://java","threadId":13,"mdc":{},"ndc":"","hostNam
e":"first-route-65697b647-45mlp","processName":"io.quarkus.bootstrap.runner.
QuarkusEntryPoint","processId":1}
```

Naast de logging van de integraties, kan ook de logging van de Camel K operator bekeken worden. Hiervoor moet eerst de exacte naam van de *camel-k-operator* pod (een pod is een (groep) actieve container(s) in Kubernetes) worden opgevraagd. Vervolgens kan met de naam van de pod ook de logging worden getoond:

```
$ kubectl get pods
NAME                                READY   STATUS    RESTARTS     AGE
camel-k-operator-XXXXXXXXXX-XXXXX   1/1     Running   1 (Xm ago)   XdXh
$ kubectl logs --follow camel-k-operator-XXXXXXXXXX-XXXXX
```

Dependency management

Bij het starten van een integratie zal Camel K automatisch proberen om de benodigde dependencies te laden. Wanneer statische URL's worden gebruikt voor de endpoints, kunnen de benodigde Camel dependencies automatisch geladen worden.

Echter, wanneer gebruik wordt gemaakt van dynamische URL's, of wanneer libraries gebruikt worden die geen onderdeel uitmaken van de Camel catalogus, zullen deze dependencies handmatig moeten worden toegevoegd. Dit kan op 2 verschillende manieren worden gedaan.

De eerste manier is om de "--dependency" of "-d" parameter toe te voegen aan het run- commando. Camel dependencies kunnen hier direct worden toegevoegd. Het is natuurlijk ook mogelijk om externe libraries toe te voegen, bijvoorbeeld via Maven. Maven dependencies kunnen middels het formaat "`mvn:<groupId>:<artifactId>:<version>`" worden toegevoegd. Wanneer bijvoorbeeld de "camel-ftp" en de (externe) "commons-lang3" dependencies toegevoegd moeten worden, kan het run-commando als volgt worden uitgevoerd:

```
$ kamel run <RouteFile> -d camel-ftp -d
mvn:org.apache.commons:commons-lang3:3.12.0
```

Het is echter ook mogelijk om de dependencies binnen een integratiebestand te specificeren. Hiervoor kunnen *modeline hooks* gebruikt worden. Alle opties die op het run-commando kunnen worden toegepast, kunnen ook via de *modeline* worden toegevoegd.

De modeline is de eerste regel (comment) van de integratie, beginnend met "`// camel-k:`". Vervolgens wordt in het onderstaande voorbeeld aangegeven dat het een Java integratie betreft, en worden de eerdergenoemde dependencies ingeladen. Naast de taal en dependencies, zouden nog veel meer zaken via de modeline kunnen worden geconfigureerd. Denk hierbij aan zaken als properties, (environment)variabelen, resources en de eerdergenoemde *traits*.

```
// camel-k: language=java dependency=camel-ftp dependency=mvn:org.apache.
commons:commons-lang3:3.12.0

import org.apache.camel.builder.RouteBuilder;
import org.apache.commons.lang3.StringUtils;

public class SftpRoute extends RouteBuilder {

    @Override
    public void configure() {

        String host = "host.minikube.internal";
        String path = "upload";
        String username = "foo";
        String password = "pass";

        // The below line requires the "camel-ftp" dependency, because it's
a dynamic URL.
        from(String.format("sftp://%s/%s?username=%s&password=%s", host,
path, username, password))
            .process(exchange -> {
                String body = exchange.getIn().getBody(String.class);

                //The below line requires the "mvn:mvn:org.apache.
commons:commons-lang3:X.X.X" dependency.
                String lowerCase = StringUtils.lowerCase(body);

                exchange.getIn().setBody(lowerCase);
            })
            .log("FILE CONTENT: [${body}]");
    }

}
```

whitehorses
member of the human network

Wanneer de route met het run-commando wordt gestart, zal ook gelogd worden dat er *modeline* opties zijn geladen uit het integratiebestand, en hoe het “volledige” commando eruitziet:

```
$ kamel run SftpRoute.java
Modeline options have been loaded from source files
Full command: kamel run SftpRoute.java --dependency=camel-ftp
--dependency=mvn:org.apache.commons:commons-lang3:3.12.0
```

Development mode

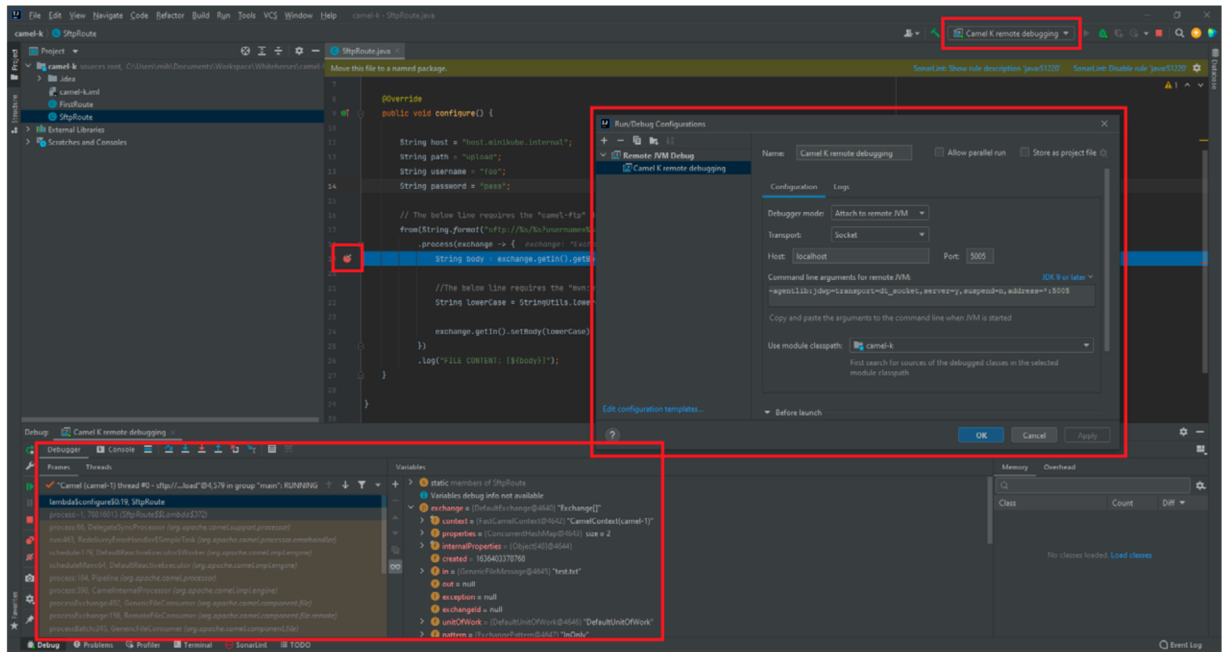
In Camel K is het tevens mogelijk om een applicatie in *development* mode te starten. Deze mode is bedoeld om ontwikkelaars snel terugkoppeling te geven over de geschreven code. Wanneer de “`--dev`” parameter wordt toegevoegd aan het run-commando, zal naast dat de integratie wordt gestart, ook direct de logging zichtbaar worden. Daarnaast worden wijzigingen, die in het integratie-bestand worden gemaakt, direct opgepakt en geactiveerd.

Debuggen van integraties

Voor het troubleshooten van actieve integraties, kunnen ook debug sessies worden opgestart. Wanneer de *sftp-route* integratie eerder is gestart middels het run-commando, kan op de volgende wijze een debug sessie gestart worden voor de actieve integratie:

```
$ kamel debug sftp-route
Enabling debug mode on integration “sftp-route”...
Forwarding from 127.0.0.1:5005 -> 5005
Forwarding from [::1]:5005 -> 5005
[1] Monitoring pod sftp-route-XXXXXXXXXX-XXXXX
[1] exec java -agentlib:jdwp=transport=dt_
socket,server=y,suspend=y,address=*:5005 -cp ...
[1] Listening for transport dt_socket at address: 5005
```

Via een IDE kan nu een (remote) debug sessie gestart worden op poort 5005. Nu kunnen breakpoints in de code gezet worden, zodat bijvoorbeeld problemen geanalyseerd kunnen worden:



Kamelets

Een ander nieuw concept in Camel K zijn Kamelets (Kamel route snippets). Dit zijn “templates” waarmee bepaalde implementatiedetails binnen de route verborgen kunnen worden. Een Kamelet kan worden gedefinieerd als *source* (from-endpoint), *sink* (destination/to-endpoint) of *action* (stappen “in het midden” van de route).

Camel K biedt out-of-the-box een brede set aan Kamelets aan in de Kamelet Catalog. Eén van de standaard Kamelets is de *sftp-source*. Hiermee kunnen we dus het SFTP-endpoint van de eerdere `SftpRoute` vervangen:

```
// camel-k: language=java dependency=mvn:org.apache.commons:commons-
lang3:3.12.0

import org.apache.camel.builder.RouteBuilder;
import org.apache.commons.lang3.StringUtils;

public class KameletRoute extends RouteBuilder {
```

```
    @Override
    public void configure() {

        from(“kamelet:sftp-source”)
            .process(exchange -> {
                String body = exchange.getIn().getBody(String.class);

                // The below line requires the “mvn:mvn:org.apache.
commons:commons-lang3:X.X.X” dependency.
                String lowerCase = StringUtils.lowerCase(body);

                exchange.getIn().setBody(lowerCase);
            })
            .log(“FILE CONTENT: [${body}]”);
    }

}
```

We zouden alle details via query parameters op het from-endpoint in kunnen stellen. Dan zouden we bijvoorbeeld “`?connectionHost=X&directoryName=X&username=X&password=X`” toe moeten voegen. In dit voorbeeld laden we deze configuratie echter in via een *properties* bestand:

```
camel.kamelet.sftp-source.connectionHost=host.minikube.internal
camel.kamelet.sftp-source.directoryName=upload
camel.kamelet.sftp-source.username=foo
camel.kamelet.sftp-source.password=pass
```

Om deze *properties* op te pakken bij het starten van de route, kan de “`--property`” parameter worden toegepast bij het starten van de route:

```
kamel run KameletRoute.java --property file:kamelet-route.properties
```

Wat direct opvalt is dat het gedrag van de *kamelet-route* iets is veranderd ten opzichte van de *sftp-route*. Dit komt door een aantal default waarden die door de Kamelet anders worden ingesteld vergeleken met de default waarden van het uiteindelijke SFTP-endpoint. Als we in de definitie van de sftp-source Kamelet kijken, zien we bijvoorbeeld dat de *idempotent* parameter standaard op *true* wordt gezet. Deze parameter zorgt ervoor dat bestanden niet dubbel worden opgepakt door een route.

Wat we in de route zien, is dat een bestand nu dus slechts één keer verwerkt wordt, in plaats van dat het bestand elke 500 milliseconden opnieuw opgepakt wordt.

Via Kamelets kan dus op een eenvoudige manier, standaard logica en gedrag worden geïmplementeerd in plaats van het op elke route zelf in te moeten stellen.

Eigen Kamelets gebruiken

Naast de standaard Kamelets die beschikbaar zijn, is het ook mogelijk om zelf Kamelets te specificeren en te gebruiken in routes. Kamelets worden gespecificeerd in een *.kamelet.yaml* bestand. In de YAML-structuur kan vervolgens worden gespecificeerd hoe de Kamelet zich moet gedragen wanneer deze wordt toegepast in een route.

De onderstaande Kamelet is een voorbeeld van een Kamelet die de *process* stap in de `KameletRoute` zou kunnen vervangen:

```
apiVersion: camel.apache.org/v1alpha1
kind: Kamelet
metadata:
  name: to-lower-case-action
  labels:
    camel.apache.org/kamelet.type: "action"
spec:
  dependencies:
    - mvn:org.apache.commons:commons-lang3:3.12.0
  definition:
    title: "To Lower Case Action"
    description: "Converts the Exchange body to lower case."
  flow:
    from:
      uri: kamelet:source
      steps:
        - convert-body-to:
            type: "java.lang.String"
        - set-body:
             groovy: "org.apache.commons.lang3.StringUtils.lowerCase(request.
body)"
```

In de *flow* zorgen we er eerst voor dat de inhoud van het request wordt geconverteerd naar een `String`. Vervolgens roepen we (via een *Groovy* language statement) de `toLowerCase` functie aan. Omdat dit een functie is uit een andere library, nemen we deze ook op in het *dependencies* blok. Tevens geven we aan dat het om een action Kamelet gaat. We willen deze namelijk "in het midden" van de route kunnen gebruiken.

Om de Kamelet te installeren op het Kubernetes cluster, kan het onderstaande commando worden gebruikt. **Let op:** De Kamelet wordt direct op het Kubernetes cluster geïnstalleerd. Hiervoor wordt dus niet de kamel CLI, maar het "kubectl" commando gebruikt!

```
kubectl apply -f to-lower-case-action.kamelet.yaml
```

Wanneer de Kamelet beschikbaar is, kunnen we de processtap in de `KameletRoute` vervangen door een referentie naar de Kamelet. Daarnaast kunnen we nu de dependency uit de modeline verwijderen. Deze wordt immers ingeladen door de Kamelet zelf:

```
// camel-k: language=java

import org.apache.camel.builder.RouteBuilder;

public class KameletRoute extends RouteBuilder {

    @Override
    public void configure() {

        from("kamelet:sftp-source")
            .to("kamelet:to-lower-case-action")
            .log("FILE CONTENT: [${body}]");
    }

}
```

Conclusie

Met Camel K worden de sterke kanten van Apache Camel en Kubernetes gebundeld. Op eenvoudige wijze kunnen snel Camel routes worden ontwikkeld. Daarnaast gaat het starten van deze routes op de Kubernetes omgeving erg snel en eenvoudig.

Door de development mode en de uitgebreide mogelijkheden om (actieve) routes de debuggen, is het erg makkelijk om problemen te traceren en op te lossen. Daarnaast kan door het gebruik van Kamelets logica, die door meerdere integraties wordt hergebruikt, op eenvoudige wijze geïmplementeerd worden.

Hoewel in dit Whitebook de nadruk op het "development" aspect lag, is ook overduidelijk aan het "operations" aspect gedacht. Bijvoorbeeld door zaken als het gebruik van property files en de out-of-the-box ondersteuning van *structured* logging. Daarnaast biedt Camel K natuurlijk alle voordelen van Kubernetes, zoals het eenvoudig op kunnen schalen of repliceren van applicaties.

Bronnen

Apache Camel K