Applause from Felipe Hoffa and 2 others

Deploying Apache Camel to Google Kubernetes Engine

This post originates from a number of discussions I have had in the last few months covering our cloud integration stack— Apache Camel deployed to Google Kubernetes Engine. So I decided to summarise the thinking in one place, so I can reference it when needed.

I will provide a brief overview as to *Why* and include the key aspects of the *How*.

The example referenced here is fully operational. The sample code available in my GitHub Repository and includes detailed walkthrough on getting the required toolstack, setting up Google Cloud infrastructure, building and deploying procedures.

Why Camel?

Apache Camel is a routing and mediation engine that can be considered to be a reference implementation for the majority of the Enterprise Integration Patterns defined by Gregor Hohpe and Bobby Woolf.

Being a top level Apache Project, it is well supported and comes with a generous Apache2.0 license, which is enterprise friendly.

It is easily extensible, provides a configuration DSL that conforms to the fluent interface principles described by Martin Fowler and includes a gazillion of components and adapters.

For a more detailed review please check a very nice summary from Jonathan Anstey Open Source Integration with Apache Camel.

Why Kubernetes Engine?

Why doing Containers and Kubernetes in the world where serverless computing is picking up steam?

As much as I like serverless architecture, the container based deployment is still perceived to be a bit more flexible. And this flexibility is quite important for the integration tasks, where requirements can be somewhat peculiar in terms of dependencies, latencies and execution times. As an example, the serverless deployments are not particularly friendly for the long running processes.

Docker containers, on the other hand, strike the balance between being stateless, being easily customisable and handling the unusual tasks rather well. And general adoption of Kubernetes as the emerging de-facto standard predefined the choice of the orchestrator. It is awesome—to see why just follow a typical DevOps team for a day and check the Kubernetes way of solving their challenges .

Why Google?

Google is where Kuberenetes originates from. It stems from Google experience with running containerised applications at scale. So, while there are a few Kubernetes PAAS available on the market, the new features are typically getting released first at Google Cloud Platform. Fantastic support and great pricing make it one the best offerings from my perspective.

But GCP is not just about the Kubernetes. It is the whole plethora of the additional capabilities—Logging, Monitoring, Persistence, Pub/Sub, Analytics, etc—all these have been made easy to use for the developers.

Google Cloud Platform allows design approach where the architect can just pick a set of the PAAS capabilities and compose a business solution. Yet the infrastructure is operated by someone else and it is one of the cheaper options for the comparable performance, features and quality of support.

Why Camel at Google Cloud Platform?

But is there a place for Apache Camel in the already rich Google Cloud Platform ecosystem? Is there a real need for its capabilities? Why not, for example, use Spark or Apache Beam? Both available as managed infrastructure—Dataproc and Dataflow respectively. Why not the latter one?

Indeed, Dataflow is a fantastic product. It is irreplaceable for the operations at scale where dynamic session calculations are a must and its pre integration with the rest of GCP technology is fantastic.

But let's consider simpler scenarios, where the focus is more on the message processing and orchestration, rather than the data handling, situations where there is only a few million exchanges a day to process, solutions where there is a number of the external systems need to be integrated. Dataflow would be coming as a tad too heavy and expensive.

The lowest deployment entity with Dataflow is a VM. Half a dozen of pure Dataflow solutions would end up in a considerable cost.

Plus there are not that many third party system adaptors available for Dataflow as yet, so integrating would require effort and some custom, low level code.

That's where Camel on Kubernetes comes in. The lowest deployment entity is a container—multiple deployments would fit into a single VM, even with a handful of adapters and components configured.

The number of the Apache Camel components available out of the box make integration and orchestration tasks simple, and Google ecosystem is enabled too—there are prepackaged adapters for Google PubSub and BigQuery.

From my perspective Apache Camel unlocks Google Cloud Platform for the legacy applications and complex integration solutions, complementing the existing Google capabilities such as Dataflow.

But enough of the *"Why?"*, let's get on with the *"How?"*.

How?

This section covers the key implementation aspects that unlock Apache Camel on Google Cloud Platform:

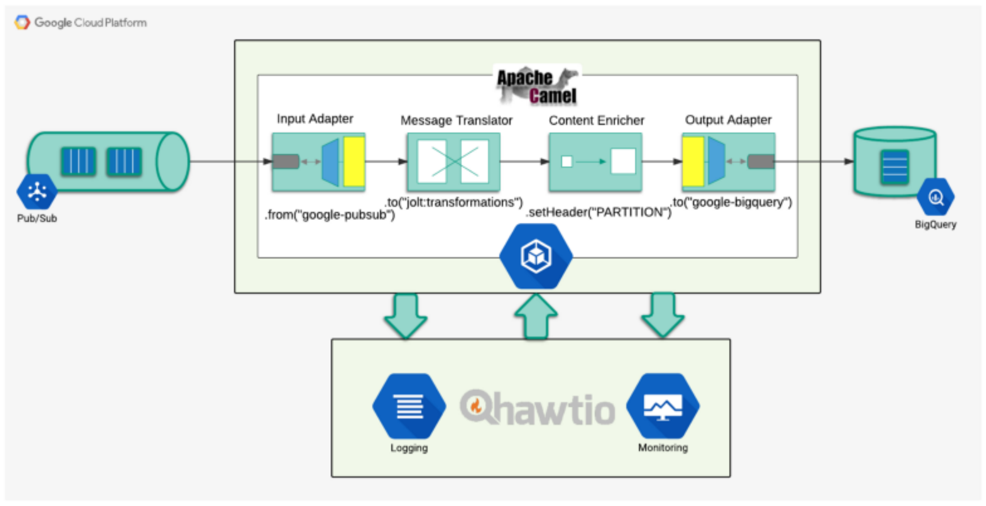
Configuration and Parameters

- Logging
- Metric Collection and Monitoring
- Live introspection with Hawtio Console

Sample scenario

The sample solution is based on a widely recognised Camel / Spring Boot combination that implements the following integration scenario:

- Camel consumes a message from Google PubSub subscription
- Transforms the data
- Updates the header to define the BigQuery table partition
- Writes to the BigQuery table



Every operation is recorded by Google Stackdriver Logging and operational metrics collected by Google Stackdriver Monitoring —both for JVM and Camel.

The solution Docker image is uploaded to the GCP Container Registry and deployed to the GCP Kubernetes Engine.

Parameters

Out of the box Kubernetes provides two mechanisms to deliver configuration settings into the container: a configuration map or a secret. Once those have been defined within the cluster, their values can be made available to the applications either as files or as environmental variables.

The demo relies on Spring Boot Externalised Configuration capability to ingest both types and that gives the flexibility to override them at any stage of the CI/CD lifecycle.

The blueprint provisions that there are four places where a Spring Boot property can be configured:

> Spring Boot `application.properties`. Values defined here can not be overridden. Generally used for Spring Boot configuration and unmodifiable settings.
>
> Custom `default.properties`. That's where usually the application specific defaults are defined by a developer.
>
> Optional file pointed by the `${external.config}` java property. The file is generally intended to be used by system administrators that would opt to use an external file instead of the environmental variables.
>
> Environment variables defined within a Docker image or through a Kubernetes deployment. These are captured and converted into Java properties by Spring Boot.

Please note—the relationship between the last three is important.

Options 2 and 3 are explicitly configured in the `ApplicationMain.java` :

and the their listing order defines the preference:

> Properties defined in the `default.properties` (option no 2) - will be **overriden** by the values defined in the external file - `${external.config}` (option 3) - if the same key is defined in both.
>
> Values defined through environmental variables—Option 4 —**supercede** the values from either property files.

This way the developer has an option of falling back onto some well defined defaults, provide their own configurations and yet to allow the administrator to override any of these through an external config file or the environmental variables.

This flexibility, though rarely required, came quite useful in a few unusual situations.

Logging

On the surface logging looks super easy—Kubernetes Engine sends the container standard output stream to Stackdriver

Logging automatically. No configuration required. Simple!

There is, however, a couple of gotchas. The first one—every line collected becomes a separate log entry. What's wrong with that? Well, java exceptions are usually multiline stack traces—reading those across multiple log entries can be a bit confusing.

The other interesting point is that these entries would be logged as INFO. Google Kubernetes logging agent—FluentD—does not know how to parse them to get the severity right. There is a number of different formats out there, so one size fits all automatic solution is indeed a hard problem.

However the application CAN provide the output in the format that FluentD can understand. The format is defined in `logback.xml` - a JSON string where the field names configured as required by the FluentD parser. And thanks to the Elastic.coteam, who provided an awesome Logstash component that encloses a multiline stack trace into a single entry!

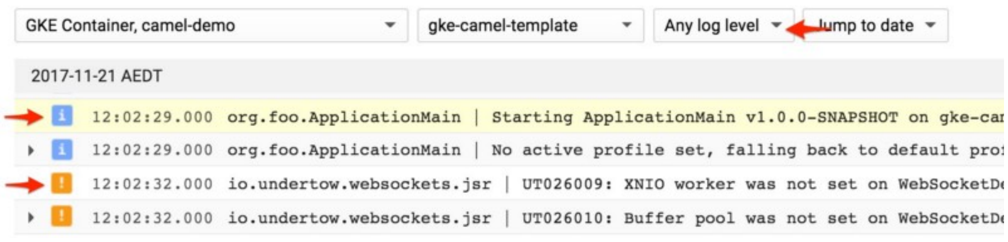
Checking output:

```
kubectl logs -f deploy/gke-camel-template
```

The output:

```
{"severity":"INFO","message":"org.foo.ApplicationMain
| Starting ApplicationMain v1.0.0-SNAPSHOT on gke-
camel-template-7b4bd9ccdb-dns4l with PID 1
(/u00/lib/gke-camel-template-1.0.0-SNAPSHOT.jar
started by root in /u00)"}
```

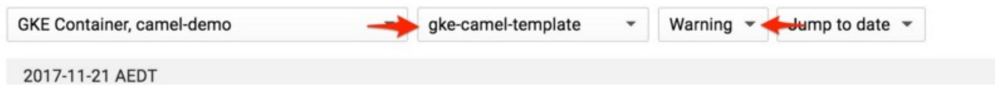
And as you can see—severity has been set explicitly. Let's check the Stackdriver Logging UI to confirm that it has been parsed correctly:

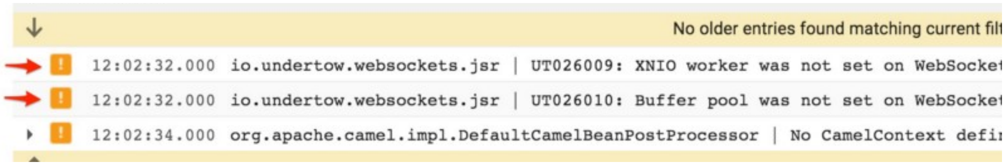


```
12:02:29.000 org.foo.ApplicationMain | Starting ApplicationMain v1.0.0-SNAPSHOT on gke-ca
12:02:29.000 org.foo.ApplicationMain | No active profile set, falling back to default pro
12:02:32.000 io.undertow.websockets.jsr | UT026009: XNIO worker was not set on WebSocketD
12:02:32.000 io.undertow.websockets.jsr | UT026010: Buffer pool was not set on WebSocketD
```

All good.

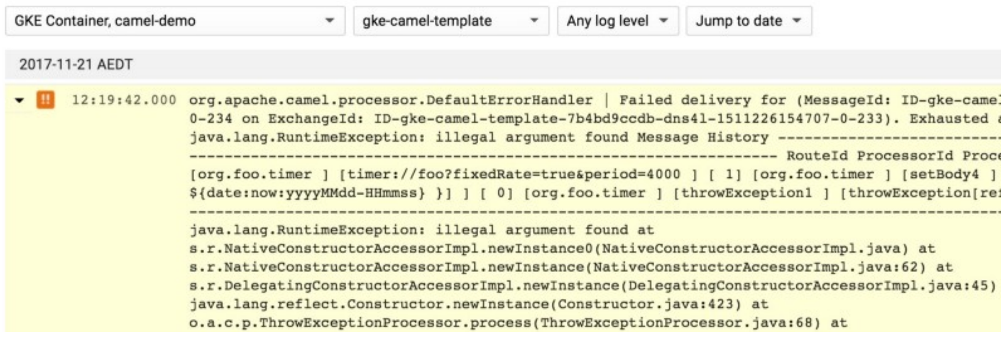
That also allows for a nice and easy filtering, which works even in the streaming mode, where the log entries are displayed as they come in the [real] time:





```
No older entries found matching current fil
12:02:32.000 io.undertow.websockets.jsr | UT026009: XNIO worker was not set on WebSocke
12:02:32.000 io.undertow.websockets.jsr | UT026010: Buffer pool was not set on WebSocke
12:02:34.000 org.apache.camel.impl.DefaultCamelBeanPostProcessor | No CamelContext defi
```

Notice how the exception is enclosed into a single entry:



```
2017-11-21 AEDT
12:19:42.000 org.apache.camel.processor.DefaultErrorHandler | Failed delivery for (MessageId: ID-gke-came
             0-234 on ExchangeId: ID-gke-camel-template-7b4bd9ccdb-dns41-1511226154707-0-233). Exhausted
             java.lang.RuntimeException: illegal argument found Message History ------------------------
             ------------------------------------------------------------------ RouteId ProcessorId Proce
             [org.foo.timer ] [timer://foo?fixedRate=true&period=4000 ] [ 1] [org.foo.timer ] [setBody4 ]
             ${date:now:yyyyMMdd-HHmmss} }] ] [ 0] [org.foo.timer ] [throwException1 ] [throwException[re
             --------------------------------------------------------------------------------------------
             java.lang.RuntimeException: illegal argument found at
             s.r.NativeConstructorAccessorImpl.newInstance0(NativeConstructorAccessorImpl.java) at
             s.r.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:62) at
             s.r.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)
             java.lang.reflect.Constructor.newInstance(Constructor.java:423) at
             o.a.c.p.ThrowExceptionProcessor.process(ThrowExceptionProcessor.java:68) at
```

Another interesting point is that Google Stackdriver Logging permits log based metrics, and having severity available as a selection condition makes the process somewhat faster.

Now lets have a look inside the running container.

Metric Collection and Monitoring

Quoting Lord Kelvin:

> If you can not measure it, you can not improve it.

Leaving aside the philosophical contention around the essence of the expression, I hope that there is a general agreement that tracking the application performance metrics is of paramount importance.

There have been two general approaches around capturing the metrics from a containerised application—either an external collection crawler (a sidecar container, node service, etc) or the application itself reporting in.

The option presented here can be considered to be the middle way between the two.

Even though it is the "reporting in" pattern, it is implemented by a JVM agent, isolated from the application.

The sample includes Jmxtrans Agent—a java agent that periodically inspects the application through JMX and reports the metrics to the collection point through a specific writer and Goggle Cloud Platform is supported by Google Stackdriver Writer .

The java agent is completely dependency free, and while it is loaded before the application it will not interfere with the app libraries. It is a generic java agent—can be used with any JVM based application as far one supports JMX.

While for the presentation purposes the metrics collector and the config file are included within the project itself, the preferred way would be to bake these into the base Docker image.

This way, while a reasonable default configuration is provided for collecting JVM and Apache Camel metrics it still can be superseded by a a different config file if a bit more granular introspection is required by the developer or the admin.

Dockerfile example:

And a couple of examples from the metrics.xml:

The Camel metric configuration above is a template that instructs the agent to capture the count of the completed Exchanges for all routes across all Camel contexts in the JVM.

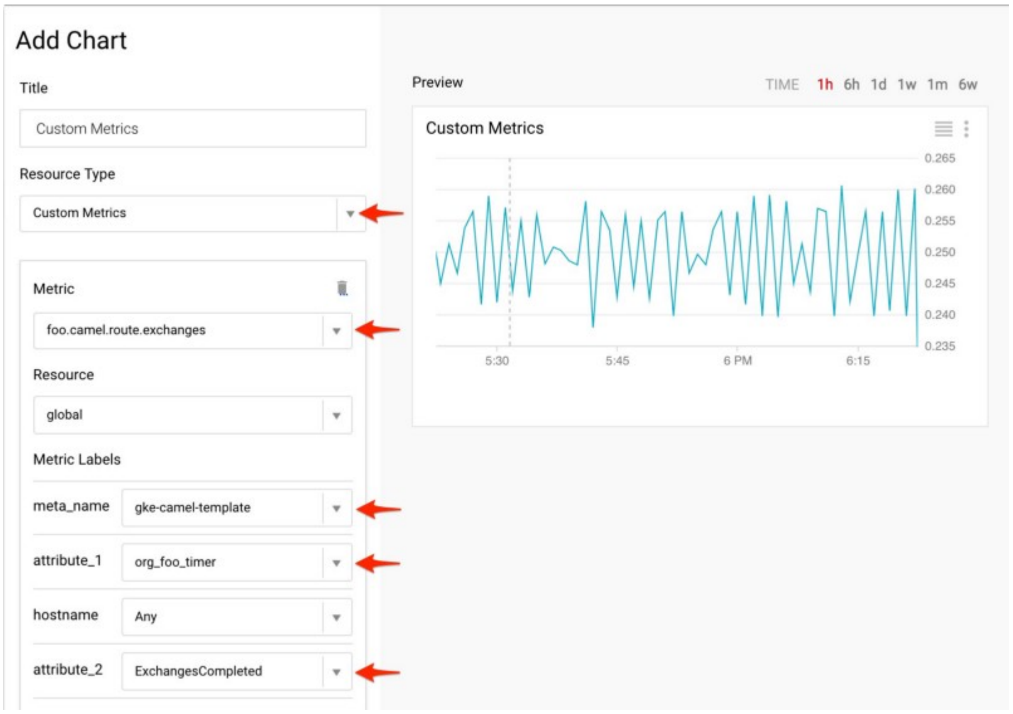
The metrics collected this way are classified as Custom Metrics and are available through Stackdriver Monitoring Dashboard.

Please check the Jmxtrans Agent documentation for the more information on the placeholders and the configuration in general. For the specifics around GAUGE vs CUMULATIVE types and configuring custom metrics please refer to Google Stackdriver Metrics API doco.

The only thing the application is required to provide explicitly is the APM_META_NAME environment variable—to group the metrics under umbrella of the originating deployment, similar to how logging does it. At present there is no way for a Kubernetes container to infer the entity that has created it—being it a Deployment, a Job or a CronJob—so it is a mandatory configuration option.

The parameter is usually included in the Kubernetes deployment YAML as an environment variable:

Please note how the attributes and the meta name permit granular metric selection:


Add Chart

Title: Custom Metrics

Resource Type: Custom Metrics

Metric: foo.camel.route.exchanges

Resource: global

Metric Labels

meta_name: gke-camel-template

attribute_1: org_foo_timer

hostname: Any

attribute_2: ExchangesCompleted

There is also a possibility to prefix the custom metric names. In that case it is “foo” and it has been defined explicitly in the metrics.xml configuration file:

```
<namePrefix>foo.</namePrefix>
```

Important!

GOOGLE_APPLICATION_CREDENTIALS is the environmental variable that usually defines location of the identity key for the Google Cloud Platform drivers.

Yet in this example, even though GOOGLE_APPLICATION_CREDENTIALS environment variable is explicitly configured, the Jmxtrans Agent Google Stackdriver Writer **will ignore** it when executed within the Google Kubernetes Engine. Kubernetes Engine cluster makes its service account available to the pods through an internal API. Jmxtrans agent Stackdriver Writer checks if the API is available when started. If so, the Kubernetes Cluster service account will be given preference.

This way both logging and monitoring operations are authorised by the same identity—cluster service account, leaving GOOGLE_APPLICATION_CREDENTIALS key to be used solely by the application. To illustrate this, the only access granted to the key referenced by the GOOGLE_APPLICATION_CREDENTIALS is to PubSub and BigQuery.

Yet when the agent runs outside the Kubernetes cluster (i.e. there is no internal API available) it will fall back onto the identity configured through GOOGLE_APPLICATION_CREDENTIALS.

Hawtio Console

Access to the Camel context in production can be the key to a quick issue resolution. Hawtio UI offers such possibility through its Camel plugin (and even allows route modifications!).

Yet it would be impractical to package Hawtio component with every deployment as it adds a considerable memory overhead, sometimes doubling the footprint. Even though the RAM is cheap, it should be avoided for the microservice style deployments.

Luckily, Hawtio can also operate as a standalone application— it just needs the access to Jolokia JMX endpoint. Which already has been enabled by Spring Boot!

build.grade excerpt:

```
// Logging and Monitoring
compile "org.jolokia:jolokia-core:1.3.7"
compile "net.logstash.logback:logstash-logback-
encoder:4.9"
```

Spring Boot application.properties:

```
server.port = 8091
camel.springboot.jmxEnabled=true

endpoints.jolokia.sensitive=false
endpoints.health.sensitive=false
```

For security reasons Jolokia port is not exposed outside the container and the only way to access it is through a tunnel.

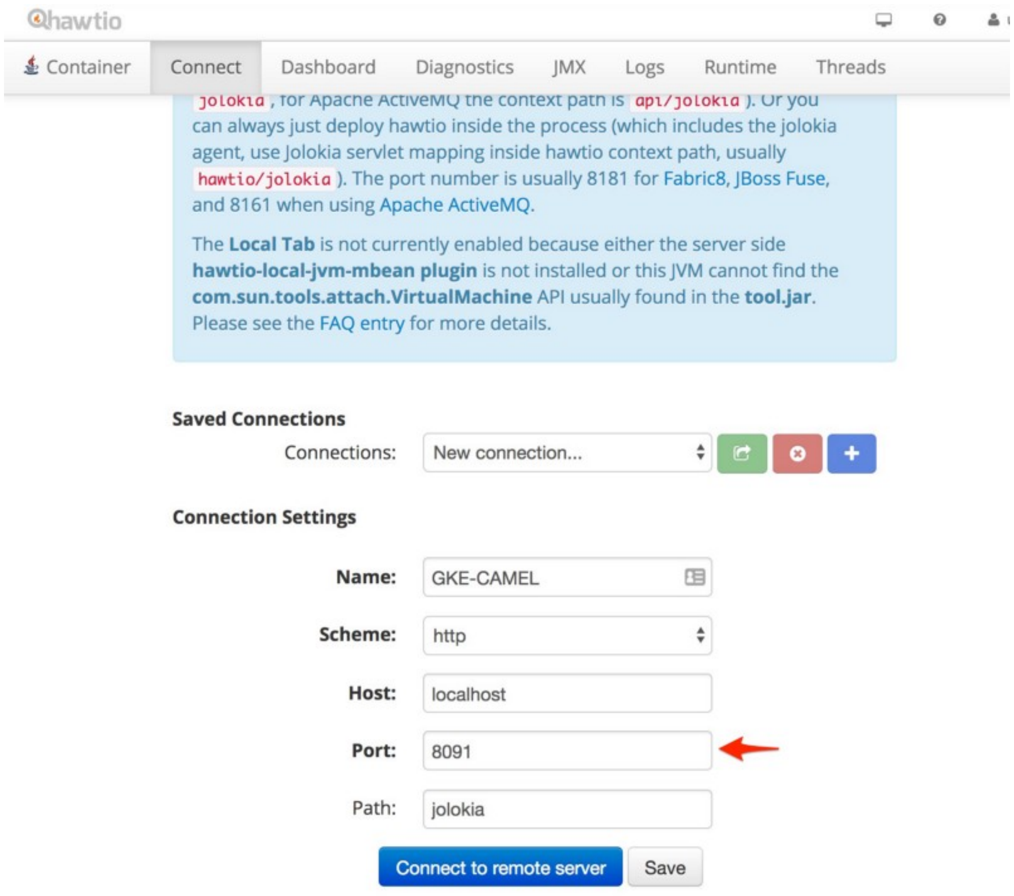
Kubernetes provides such tunnel with its `port-forward` command:

```
kubectl port-forward <pod_name> <port>

# Executable example
kubectl port-forward $(kubectl get po | grep gke-camel
| cut -d" " -f1) 8091
```

That would effectively map the port 8091 on the local machine to the one in the container.

Now start Hawtio in the standalone mode and select "Connect" tab. Change the port to **8091** and click Connect:



jolokia , for Apache ActiveMQ the context path is api/jolokia). Or you can always just deploy hawtio inside the process (which includes the jolokia agent, use Jolokia servlet mapping inside hawtio context path, usually hawtio/jolokia). The port number is usually 8181 for Fabric8, JBoss Fuse, and 8161 when using Apache ActiveMQ.

The **Local Tab** is not currently enabled because either the server side **hawtio-local-jvm-mbean plugin** is not installed or this JVM cannot find the **com.sun.tools.attach.VirtualMachine** API usually found in the **tool.jar**. Please see the FAQ entry for more details.

Saved Connections

Connections: New connection...

Connection Settings

Name: GKE-CAMEL

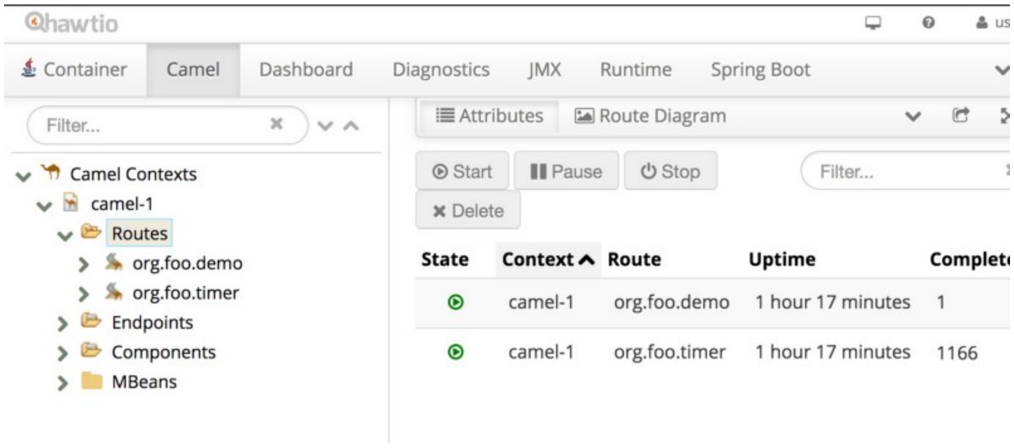
Scheme: http

Host: localhost

Port: 8091

Path: jolokia

That gets us Hawtio Console for the remote Camel context:



State	Context	Route	Uptime	Complet
●	camel-1	org.foo.demo	1 hour 17 minutes	1
●	camel-1	org.foo.timer	1 hour 17 minutes	1166

Look around—debugging, tracing, route updates—it is very powerful.

Conclusion

This demo presented how Camel application can be configured for the Google Kubernetes Engine, PubSub, BigQuery and Stackdriver.

The blueprint presents a flexible, resilient and yet scalable approach, battle tested across tens of integration deployments in production.

Yet the foundation of the blueprint is versatile. Any JVM based application can be deployed in this manner—ourselves we have been using it for both Apache Camel and Clojure based solutions.

With Kubernetes being the de-facto standard for the containerised deployments, such setup can be replicated anywhere—Azure, AWS, IBM, RedHat.

The code provides no warranty of any kind what so ever and any real use comes at on your own risk.

Support

Yet for the piece of mind, I would suggest considering Apache Camel support subscription through RedHat.

Having access to the awesome teams in Australia, UK and Denmark can be priceless. Their deep understanding and hands on experience has proven to be invaluable on a quite a few occasions.