Let's Learn about Camel K

Shawn Deggans

Let’s Learn about Camel K

There are multiple cloud and serverless solutions available for integration. Some of these are proprietary products like Microsoft BizTalk, Azure Logic Apps, or Dell’s Boomi.

There are hybrid options, like Saleforce’s Mulesoft, which has a lighter, open source version, but the real functionality is in the licensed version. Then there are a few truly open source solutions, like Apache ServiceMix or Apache Synapse (not to be confused with Azure Synapse).

We’re covering Camel K today, because it looks like an integration winner in the open source community. ServiceMix looked like an interesting competitor to Mulesoft, but it doesn’t seem to be as active as Camel K.

What is Camel?

Apache Camel is an enterprise integration framework based on the famous Enterprise Integration Patterns codified by Gregor Hohpe and Bobby Woolf in their book, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*

I’ve been waiting forever for a second edition of that book, but for the most part, that’s not necessary. I first learned the ins-and-outs of distributed systems from that book, and I would still recommend it as required reading today for cloud architects.

I won’t dive into the discipline and the craft needed to be a software integrator, but the fact that Camel focuses on these concepts and patterns makes a developer’s job much easier.

Let's Learn about Camel K

Camel is a small library designed to work with pluggable connectors. There are hundreds of pluggable connector choices for existing, known APIs. And like any good pluggable system, you can create your own plug-ins.

You can create routing and mediation rules in a variety of JVM languages, like Java, Groovy, and Kotlin. There's also the option to use XML and YAML if you're into that sort of thing (no judgement!).

Is there a difference between Camel and Camel K?

It's a native version of Apache Camel designed to run containerized on Kubernetes or Microservices. Basically, this is the serverless version of Camel.

Which is one of the reasons I believe it has thrived more than a product like ServiceMix.

The good news! If you are familiar with Camel, Camel K will just work. If you've written code in Camel DSL, that code can run on Kubernetes or OpenShift.

Why use Apache Camel?

I've been in IT long enough that my career is no longer carded at the bar when it wants to buy a beer. One practice that never seems to go away is integration. It's the glue that holds everything together.

As someone working in the technology field, having a minimum understanding and respect for the challenges of integrating disparate systems is necessary. No matter what platform you build on, at some point your platform will need to interface with some other systems.

It's best to understand the fundamentals of integration if you want successfully operating technology systems that bring multiple chains of value together for your customers.

Camel is happiest as the connection between two or more systems. It allows you to define a Camel Route. The Camel Route allows you to make decisions about what you do with data coming into the route, what type of processing that might need to be done to that data, and what that data needs to look like before it's sent to the other system.

And let me be clear, that data can be almost anything. It could be an event from a piece of manufacturing machinery, it could be a command from one system to another, or it could be the communication between two microservices.

The enterprise integration patterns were designed to help establish what actually happens to a message or data between two or more systems.

As you can probably imagine, having a system between two other systems that allows you to modify the data or take action on that data is a pretty powerful tool.

Why not just directly connect the two systems together?

There are times when this might not be a bad solution. Context is important when building technology solutions. Point-to-point connections aren't always evil.

However, when you get to the point where you have more than two systems that need to exchange data or messaging, you start to run into a problem. Keeping up with messages, source systems, and data transformations can get painful.

When things get painful, it's time to use a tool to help stop that pain.

Camel is excellent for this. It's exceptionally flexible, provides multiple patterns from manipulating XML or JSON files to working directly with cloud services.

Ready to learn!

Let's do this! Here's where we start.

Routes

Remember the integration patterns I discussed earlier? Well, this is where we start putting those to work.

Camel Routes are designed with a Domain Specific Language using a JVM programming language like Java, Kotlin, or Groovy. And you can also define these in YAML or XML.

If you are familiar with working on Mulesoft code under the hood, those ugly XML routes will be familiar.

```
<routes xmlns="<http://camel.apache.org/schema/spring>">
   <route>
      <from uri="timer:tick"/>
      <setBody>
         <constant>Hello Camel K!</constant>
       </setBody>
      <to uri="log:info"/>
   </route>
</routes>
```

That's not too bad. And ideally you want to keep these routes short and concise. If your routing logic starts to look scary, it's probably time to write code.

I don't want to dive too far into code here. This article's goal is to just give you a quick overview, but I do want to show you how easy this is using Java.

```
import org.apache.camel.builder.RouteBuilder;

public class Example extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("timer:tick")
            .setBody()
              .constant("Hello Camel K!")
            .to("log:info");
    }
}
```

Ok, so what if you come from the Mulesoft world or one of these other integration platforms that offer a visual interface to make this work? Let's be honest, if you've used Azure Logic Apps, Make, or Zapier, you probably want a similar experience.

Drag and Drop GUI

I don't want to jump too far ahead, but there is a solution for the low-code folks. And let's face it, seeing a visual representation of flows is much easier to work with than code.

Introducing Kaoto

There's a lot to Kaoto. I want to keep this brief, but I do want to assure those who are used to working with visual tools that you aren't losing that functionality. For the engineers in the room, Kaoto won't increase the footprint of the deployed code.

Why is using Kaoto a good idea?

- It's Cloud Native and works with Camel K
- We can extend it if needed
- It's not locked in No Code, we can switch back to code if we need to
- It's Open Source, so it will only cost you the time to learn it
- You can run it in Docker – I'm a big fan of doing all my development in containers, so this is always a plus for me

Routes and Integration Patterns

There are well over 300 connection points available for Camel. Many of these are common like JDBC, REST, SOAP. But there are also more specific connectors, like Slack, gRPC, Google Mail, WordPress, RabbitMQ, etc.

Many of the connectors you are used to seeing in commercial products are available in Camel. If you don't find something you need, you can create your own connector.

There are also integration patterns for almost any situation, and the patterns can build connected and built upon to create messaging pipelines.

I won't go into each pattern, but they fit within these categories:

- Messaging Systems, like a message, a message channel, or message endpoint
- Messaging Channel, like a point-to-point channel, dead letter channel, message bus
- Message Construction, like return address, message expiration, and event message
- Message Routing, like splitter, scatter-gather, and process manager
- Message Transformation, like content enricher, claim check, and validate
- Messaging Endpoints, like message dispatcher, idempotent consumer, and messaging gateway
- System Management, like detour, message history, and step

That is just a short collection of the patterns and their categories. It's well worth any developers time to read through these and understand the problems in integration that they address. As a bonus, familiarizing yourself with integration patterns will make you a better programmer and more adept at designing solutions for the cloud.

The Big K

Camel K allows us to run Camel in Kubernetes. Why is this important?

First, you'll want to understand that Camel K isn't just Camel, it's everything Camel can do, but it's written in Go. The original Camel is a Java product. Nothing necessarily wrong with Java and the JVM, but it does tend to have a bigger footprint than Go. Go eats less memory. Eating less memory is good for the cloud.

It also doesn't need the surrounding infrastructure that Camel requires. Camel can run almost anywhere the JVM is supported. Spring Boot is a good way of hosting Camel. And yes, you could containerize that and run it in Kubernetes.

However, Camel K was born for Kubernetes and containers. And there is a custom Kubernetes resource designed for Camel. This means that from a developer's standpoint, you just need to write your code locally, and then use the Kamel CLI to push your changes to the cloud.

Now, you might want a more defined DevOps process, but the idea is that there is far less friction between the code written and the code that runs in production.

The basic loop is as follows:

1. You change your integration code
2. You commit your changes to the cloud
3. The custom integration resource notifies a Camel K operator
4. The operator pushes the changes to the running pods
5. Go back to step 1

Camel K and Knative

What is Knative, and why do I want to use it with Camel k?

Knative is an extension of Kubernetes. It enables Serverless workloads to run on Kubernetes clusters, and provides tools to make building and managing containers easier.

Knative has three primary areas of responsibility:

1. Build
2. Serving
3. Eventing

The idea behind Serverless is that it should just work, and it should just work well in a variety of situations. For instance, it should scale up automatically when workloads increase, and it should scale down automatically when workloads decrease. This is what the serving portion of the solution does.

If you install Camel K on a Kubernetes cluster that already has Knative installed, the Camel K operator will automatically configure itself with a Knative profile. Pretty sweet!

When this in place, instead of just creating a standard Camel K deployment, Knative and the Camel Operator will create a Knative Service, which gives us the serverless experience.

KNative can also help with events. Event-driven architecture using Camel K is a bit too complex for this quick introduction, but I do want to touch on what possibilities this opens up for developers and architects.

Because Knative allows you to add subscription channels that are associated with your integration services, you can now build pipelines that work with events. This event-based system can be backed by Kafka. Managing evens within your Camel K integration service, allows you to also employ common integration patterns.

We can accept event sources from any event producing system. I typically use Mosquitto as my primary MQTT hub, and in this case I could pass all my incoming MQTT message to Camel K and allow it to manage the orchestration of event messages to its various subscribers.

Camel K and Quarkus

What is Quarkus? Think of Quarkus as the new Java Framework with a funny name. Quarkus is a Kubernetes-native Java framework made for GraalVM and HotSpot. It's also open source and an Apache project.

Why do we want to use it with our Camel K integrations?

Again, one of the things we want from our cloud native solutions is smaller library sizes. The Java Framework was conceived and built in the age of the monolith. Apps usually ran on powerful hardware in data centers, and they ran continuously. The concept of scaling up or down meant adding or removing hardware.

With Kubernetes and cloud solutions, we want small. The smaller, the better. Quarkus gives us that smaller size, so we can scale up or down as needed.

Basically, we’re designing our Java applications to compile much more like Go. It’s a binary now, and we don’t need the JVM.

Next Steps

Here are a few great resources for learning more about Camel K and how to use

https://developers.redhat.com/topics/camel-k

https://camel.apache.org/

https://github.com/apache/camel-k/tree/main/examples