Cloud Native Integration with Apache Camel

Building Agile and Scalable Integrations for Kubernetes Platforms

Guilherme Camposo

Apress®

Cloud Native Integration with Apache Camel

Building Agile and Scalable Integrations for Kubernetes Platforms

Guilherme Camposo

Apress®

Cloud Native Integration with Apache Camel: Building Agile and Scalable Integrations for Kubernetes Platforms

Guilherme Camposo
Rio De Janeiro, Brazil

ISBN-13 (pbk): 978-1-4842-7210-7 ISBN-13 (electronic): 978-1-4842-7211-4
https://doi.org/10.1007/978-1-4842-7211-4

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Divya Modi
Development Editor: Laura Berendson
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Pixabay

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-7210-7. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

I dedicate this work to my beloved wife, who supports me in every moment of my life. I also would like to dedicate this work to my deceased grandmother and grandfather, who always believed in me and encouraged me to follow my dreams. I really wish you were here to share this moment with me.

Table of Contents

About the Author ix

About the Technical Reviewer xi

Acknowledgments xiii

Introduction xv

Chapter 1: Welcome to Apache Camel 1

What Is System Integration? 2

Business or Integration Logic? 4

Cloud Native Applications 6

What is Apache Camel? 9

Integration Logic, Integration Routing 9

Exchanges and Messages 11

Expression Languages 13

Quarkus 14

Java Evolution 14

Microservices 16

Development Requirements 18

The MicroProfile Specification 22

Running the Code 24

Packing Applications 27

Fast Jar 28

Uber Jar 31
Container Images 33
Summary 37

Chapter 2: Developing REST Integrations 39

Camel DSLs 40
REST and OpenAPI 46
First Application: REST File Server 47
REST Interfaces and OpenAPI 48
Readability and Logic Reuse 54
Beans and Processors 60
Predicates 65
Data Formats 69
Type Converters 70
Summary 75

Chapter 3: Securing Web Services with Keycloak 77

Access Control 78
OAuth 2.0 79
OpenID Connect 85
Keycloak 86
Securing REST APIs with Keycloak 89
Exposing the Contact List API 90
Configuring Keycloak 94
Configuring The Resource Server 102
Consuming APIs with Camel 109
Summary 115

Chapter 4: Accessing Databases with Apache Camel 117

Relational Databases 118

Persistence with JPA.....118

Parameterized Queries with JPA 128

Transactions 135

Handling Exceptions 142

Try-Catch-Finally 142

Error Handlers 150

OnException Clause.....158

Summary.....164

Chapter 5: Messaging with Apache Kafka.....167

Message-Oriented Middleware.....168

Apache Kafka.....173

Concepts and Architecture 173

Installing and Running.....179

Testing the Installation 184

Camel and Kafka.....186

Setting Up the Application 186

First Test.....190

Scaling Consumers.....194

Offset Reset.....198

Unit Testing Applications 202

Summary.....209

Chapter 6: Deploying Applications to Kubernetes 211

Kubernetes.....212

Minikube.....215

First Application Deployment.....217

Quarkus-minikube.....227

App Configuration 234

Environment Variables.....234

ConfigMaps and Secrets.....242

Health Check and Probes 248

Request and Limits.....259

Summary.....267

Index.....269

About the Author

Guilherme Camposo is a solution architect. He started using open-source projects early in his career and completely fell in love with the Open Source philosophy and potential, leading him to work with an open source company in 2018. Throughout his more than 12-year career, starting as a Java developer, becoming a consultant, and then an architect, Guilherme has acquired vast experience in helping customers from a great variety of business sectors, giving him a broad view on how integration and good software practices can help businesses to grow.

About the Technical Reviewer

Rodrigo Ramalho is the Integration Leader for Latin America in the open-source company Red Hat.

In this role, he is responsible for spreading Red Hat's message of agile integration, leading customers on the API adoption journey through modern event-based architectures, microservices, and API management in multi-cloud container-based environments. He is an open-source enthusiast since a teenager and graduated in Computer Science in 2011. Also, he is a husband and father who likes to practice skydiving and surfing, and he is a brown belt in Brazilian Jiu-Jitsu.

Acknowledgments

I've always thought I should write a book. In my current profession, I often have to teach or explain a variety of technologies. I've also taught Java classes in the past but I never really tried to prepare myself to write a book. For that I would like to thank Divya Modi and Apress Media LLC for reaching out and giving me the amazing opportunity to write this book. I also would like to thank my friend Rodrigo Ramalho for doing a great job on this book's technical review, and all my colleagues and clients who influence me every day to keep studying and sharing knowledge.

Introduction

Building integration is a challenging task done by system architects and developers. Besides the complexity of intermediating the communication of at least two different systems, there is the necessity of adding complementary tools into the mix, like databases, message brokers, and access control tools, plus having to handle the integration tool itself. I want to support you in this challenge.

In *Cloud Native Integration With Apache Camel*, you will learn how to use an integration tool named Apache Camel. You'll learn how to integrate it with a database, a message broker, and how to deal with access control. We'll also discuss architecture and integration practices.

In the first chapter, you will learn the basis for your cloud native integration journey: how to build integration routes with Apache Camel and Quarkus. You will also understand the reason behind some decisions made for this approach to integration.

Chapter 2 is about the most popular cross-application communication architecture: REST APIs. You will see how to expose a route using REST and how to declare OAS definitions.

Chapter 3 describes how to secure and consume REST APIs using an open standard protocol, in this case OpenID Connect. You will learn a little bit about Keycloak, an identity and access management solution, and use it to work on access control.

In Chapter 4, I will show you how to access relational databases using Camel and Quarkus. You will also see how to work with transactions and how to properly handle exceptions in your integration routes.

Chapter 5 is related to asynchronous communication using message brokers. For the examples, you will use Apache Kafka, a very popular

open-source message broker streaming platform. Good practices on declaring routes and approaches in unit tests will also be explored.

The last chapter focuses on how to deploy your integration application to Kubernetes and the good practices you must be aware of before putting the application into production. You will see how easy it is to test locally and how using Quarkus will make your development process really fast.

There are a bunch of patterns, architectures, and technologies to explore in this book. The knowledge you will gain will enable you to face the most common challenges in integration, so you will know everything you need to get started. I hope you enjoy this content as much as I enjoyed writing it. See you in Chapter 1.

CHAPTER 1

Welcome to Apache Camel

Systems integration is one of the most interesting challenges I face in my job as a solution architect, so it is definitely something I'm passionate about discussing and writing. I feel that most books are just too technical, covering everything that a specific tool does, or are just to theoretical, having great discussions about patterns and standards but not showing you how to solve problems with any tool. My problem with these two approaches is that sometimes you read a book, learn a new tool but do not understand how to apply it to different uses cases, or you know the theory too well but not how to apply it in the real world. Although there is plenty of space for these kinds of reading, such as when you want a technical manual for reference or you just want to expand your knowledge on a subject, my objective is to create a material that goes from an introductory perspective to a real-world, hands-on experience. I want you to get to know Apache Camel well, to develop a deeper understanding of integration practices, and to learn other complementary tools that you can use in different use cases. Most importantly, I want you to feel confident in your choices as an architect or as a developer.

There is a lot going on in this book. The idea is to have a real-world approach where you deal with a lot of different technologies, as you normally do in the field. I'm assuming you know a little bit of Java, Maven, containers, and Kubernetes, but don't worry if you do not feel like an

G. Camposo, *Cloud Native Integration with Apache Camel*,
https://doi.org/10.1007/978-1-4842-7211-4_1

expert on these technologies. I will approach them in a way that will make sense to everyone, from Java beginners who need to deploy applications to Kubernetes, to people who already have a solid knowledge of Java but maybe don't know Camel or need to learn an approach to develop in Java for containers.

In this first chapter, I will set the basis for everything you are going to do in this book. You will learn the basic concepts of the selected tools and, as you progress, we'll discuss the patterns and standards behind them. We are going from theoretical content to running applications.

The three main topics of this chapter are system integration, Apache Camel, and Java applications with Quarkus. Let's get started!

What Is System Integration?

Although the name is very self-explanatory, I want to be very clear on what I mean by system integration. Let's see some examples and discuss aspects related to this concept.

First, let's take the following scenario as an example:

> *Company A has bought an ERP (enterprise resource planning) system that, besides many other things, is responsible for the company's financial records. This company also acquired a system that, based on financial information, can create complete and graphical reports on the company's financial status, how efficient their investments are, how their products are selling, and so on. The problem is that the ERP system does not have a native way to input its information in the BI (business intelligence) system, and the BI system does not have a native way to consume information from the ERP system.*

The scenario above is a very common situation where two proprietary software programs need to "talk" to each other, but they are not built for this particular integration. This is what I meant when I said "native way",

which means something already developed in the product. We need to create an integration layer between these two systems in order to make this work. Luckily for us, both systems are web API-oriented (application programming interface), allowing us to extract and input data using REST APIs. This way we can create an integration layer that can consume information from the ERP system, transform its data in a format accepted by the BI system, and then send this information to the BI system. You can see this illustrated in Figure 1-1.

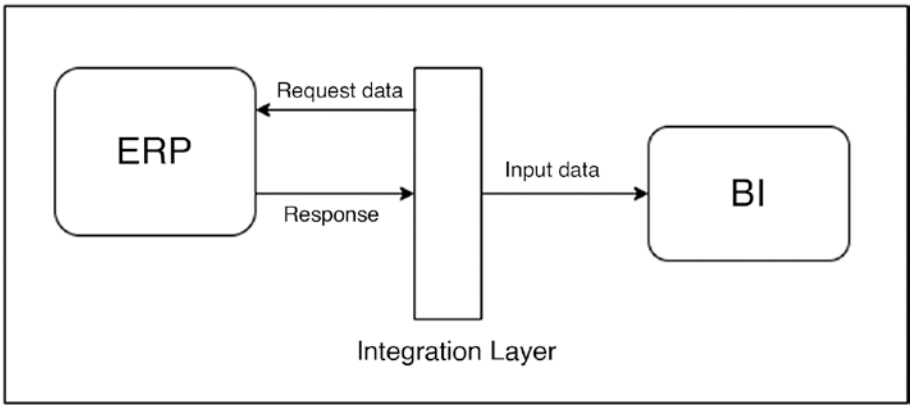

Figure 1-1. *Integration layer between two systems*

Despite being a very simple example, where I don't show you how this layer is built, it illustrates very well what this book means when I talk about system integration. In this sense, system integration is not just one application accessing another application. It is a system layer that can be composed of many applications, between two or more applications whose sole purpose is to integrate systems, not being directly responsible for business logic.

Let's see what separates those concepts.

Business or Integration Logic?

Business logic and integration logic are two different concepts. Although it may not be clear how to separate them, it is of great importance to know how to do so. Nobody wants to rewrite applications or integrations because you created a coupling situation, am I right? Let's define them and analyze some examples.

I finished the last section by saying that the integration layer shouldn't contain business logic, but "what does that mean?". Well, let me elaborate.

Take our first example. There are some things that the integration layer must know, like

- Which ERP endpoints to consume from and how to do it
- How to transform the data from the ERP in a way that the BI will be able to accept
- Which BI endpoints to produce data to and how to do it

This information is not related to dealing with financial records or providing business insights, which are capabilities expected to be handled by the respective systems being integrated. This information is only related to making the integration between the two systems work. Let's call this *integration logic*. Let's see another example to clarify what I mean by integration logic:

> *Imagine that System A is responsible for identifying customers who are in debt with our imaginary company. This company has a separate communication service that sends email, text messages, or even calls customers when they are in debt, but if a customer is in debt for more than two months, the Legal Service must be notified.*

If we consider that this situation is handled by an integration layer, it may seem that we have business logic inside our integration layer. That is

why this is a good example to show the differences between business logic and integration logic.

Although the result of the analysis of how long this customer is in debt will ultimately impact on a process or business decision, this logic is inserted here with the sole purpose of dictating how the integration between these three services will happen. We could call this *routing*, because what is being done is determining where to send this notification. Take a look at Figure 1-2.

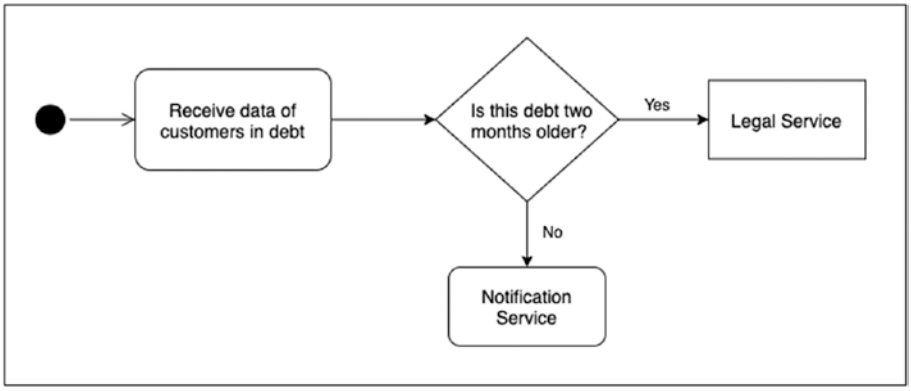

Figure 1-2. *Integration logic based on received data*

Removing the integration layer wouldn't mean that business information or data would get lost; it would only impact the integration of those services. If we had in this layer logic to determine how to calculate fees or how to negotiate this debt, it wouldn't be just an integration layer; it would be an actual service, where we would be inputting information from System A.

These are very short and simple examples just to get clear on what we are going to be approaching in this book. I will offer more complex and interesting cases to analyze as we go further. The idea is just to illustrate the concept, as I am going to do next for a cloud native application.

Cloud Native Applications

Now that I have clarified what I mean by saying *integration*, there is another term that you must know in order to fully understand this book's approach: *cloud native*.

One of the main objectives of this book is to give a modern approach on how to design and develop integrations, and at this point it's impossible to not talk about containers and Kubernetes. These technologies are so disruptive that they are completely changing the landscape of how people design enterprise systems, making tech vendors all over the world invest massive amounts of money to create solutions that run in this kind of environment or that support these platforms.

It is beyond this book's objective to fully explain how containers and Kubernetes work or to dive really deeply into their architectures, specific configurations, or usage. I hope you already have some knowledge of these technologies, but if you don't, don't worry. I will approach these technologies in a way that anybody can understand what we are doing and why we are doing it.

To set everyone on the same page, let's define these technologies.

Container: *"A way of packing and distributing applications and their dependencies, from libraries to runtimes. From an execution standpoint, it is also a way to isolate OS (operating system) processes, creating a sandbox for each container, similar to a virtual machine idea."*

A good way to understand containers is to compare them to a more commonly found technology, virtualization. Take a look at Figure 1-3.

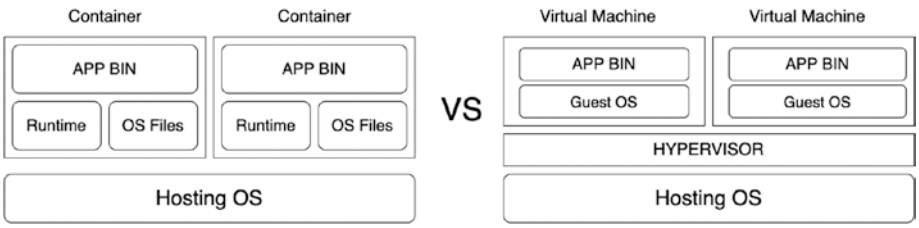

Figure 1-3. *Container representation*

Virtualization is a way to segregate physical machine resources to emulate a real machine. The Hypervisor is the software that manages the virtual machines and creates the hardware abstraction on top of one hosting operating system.

We virtualize for a number of different reasons: to isolate applications so they won't impact each other, to create different environments for applications that have different OS requirements or just different runtimes, to segregate physical resources per application, and so on. For some of these reasons, containerization may represent a much lighter way to achieve the same purpose, because it doesn't need a Hypervisor layer or hardware abstraction. It just reuses the hosting Linux kernel and allocates its resources per containers.

What about Kubernetes?

Kubernetes *"is an open source project focused on container orchestration at scale. It provides mechanisms and interfaces to allow container communication and management."*

Since we need software to manage a great number of virtual machines, or just to create high availability mechanisms, containers are no different. If we want to run containers at scale, we need complementary software to provide the required level of automation to do so. This is the importance of Kubernetes. It allows us to create clusters to manage and orchestrate containers at high scale.

This was a very high-level description of containers and Kubernetes. These descriptions give the idea of why we need these technologies, but

in order to understand the term *cloud native* you need to know a little bit about the history of those projects.

In 2014, Google launched the Kubernetes project. One year later, Google partnered with Linux Foundation to create the Cloud Native Computing Foundation (CNCF). The CNCF's objective was to maintain the Kubernetes project and also to serve as an umbrella for other projects that Kubernetes is based on or that would compose the ecosystem. In this context, *cloud native* means "made for the Kubernetes ecosystem."

Besides the origin of CNCF, there are other reasons why the name "cloud" fits perfectly. Nowadays Kubernetes can be easily considered an industry standard. This is particularly true when thinking about the big public cloud providers (e.g. AWS, Azure and GCP). All of them have Kubernetes services or container-based solutions, and all of them are contributors to the Kubernetes project. The project is also present in the solutions of players providing private cloud solutions such as IBM, Oracle, or VMWare. Even niche players that create solutions for specific uses such as logging, monitoring, and NoSQL databases have their products ready for containers or are creating solutions specifically for containers and Kubernetes. This shows how important Kubernetes and containers have become.

For the most part of this book I will be focusing on integration cases and the technologies to solve those cases, but all the decisions made will take into consideration cloud native application best practices. After you have solid understanding of integration technologies and patterns, in the last chapter you will dive into how to deploy and configure developed applications in Kubernetes.

So let's talk about our main integration tool.

What is Apache Camel?

First and foremost, you must understand what Apache Camel is and what Apache Camel is not, before starting to code and dive in integration cases.

Apache Camel is a framework written in Java that allows developers to create integrations in an easy and standardized way, using concepts of well-established integration patterns. Camel has a super interesting structure called *components*, where each component encapsulates the logic necessary to access different endpoints, such as databases, message brokers, HTTP applications, file systems, and so on. It also has components for integration with specific services, such as Twitter, Azure, and AWS, totaling over 300 components, making it a perfect Swiss knife for integration.

There a few low-code/no-code solutions to create integration. Some of these tools are even written using Camel, such as the open-source project Syndesis. Here you are going to learn how to write integration with Java using Camel as an integration specialized framework.

Let's learn the basics.

Integration Logic, Integration Routing

You are going to start by analyzing the following "Hello World" example shown in Listing 1-1.

Listing 1-1. HelloWorldRoute.java File

```
package com.appress.integration;
import org.apache.camel.builder.RouteBuilder;
public class HelloWorldRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("timer:example?period=2000")
```

```
        .setBody(constant("Hello World"))
        .to("log:" + HelloWorldRoute.class.getName() );
    }
}
```

This class creates a timer application that prints `Hello World` in the console every 2 seconds. Although this is not a real integration case, it can help you understand Camel more easily, because is better to start small, one bite at a time.

There are just a few lines of code here, but there is a lot going on.

The first thing to notice is that the `HelloWorldRoute` class extends a Camel class called `RouteBuilder`. Every integration built with Camel uses a concept called a **route**. The idea is that an integration always starts `from` an endpoint and then goes `to` one or multiples endpoints. That is exactly what is happening with this `Hello World` example.

The **route** starts with a timer **component** (`from`) and eventually hits the final destination, which is the log **component** (`to`). Another thing worth mentioning is that you have a single line of code to create your route, although it is indented to make it more readable. This is because Camel utilizes a fluent way to write routes where you can append definitions on how your route should behave or simply set attributes to your route.

Route builders, such as the class `HelloWorldRoute`, are just blueprints. This means that this type of class is only executed when the application is starting. By executing the `configure()` method, the outcome of these stacked calls is a **route definition** and it is used to instantiated many objects in memory. These objects in memory will react to events being triggered to or by the `from` (consumer) endpoint. In this particular case, the component will auto-generate its events that will go through the route logic until it reaches its final endpoint. This execution of an incoming event is called an **Exchange**.

Exchanges and Messages

You saw in the last section how the integration logic is created and executed, but how will the data be handled in this step-by-step execution? Routes carry other structures in order to make the integration work. Let's see.

There was one line of code left to comment about in the last example. The `setBody(constant("Hello World"))` is the only line where you actually set data in your route. Let's see how data is handled within routes.

In the previous section, I said: *"Those objects in memory will react to events being triggered to or by the from() endpoint."*. In this case, when I'm talking about an event, I mean that the timer triggered, but it could be an incoming HTTP request, a file hitting a directory, or another triggered action from different endpoints. What is important is that when that happens, an object called `Exchange` is created. This object is the data representation of the route execution. This means that every time the timer triggers, a new `Exchange` will be created, and that object will be available until that execution is finished. Take a look at the `Exchange` representation on Figure 1-4.

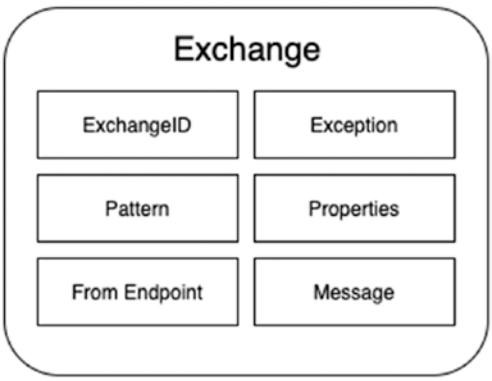

Figure 1-4. *Exchange representation*

The above representation shows the main attributes available in an `Exchange` object. All of them are important, but if you want to understand what `setBody(constant("Hello World"))` does, you must focus on the message.

Every endpoint you hit in your route chain has the potential to change the `Exchange` state, and most of the time they will do it by interacting with the `Message` attribute.

The `message` object represents the data coming and going, to and from the different endpoints within your route. Look at the representation of the `message` object in Figure 1-5.

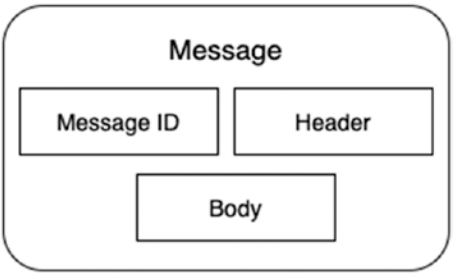

Figure 1-5. *Message object representation*

The `message` object is an abstraction to help deal with different types of data in a standardized manner. If you take an HTTP request, for example, it has headers and URL parameters that are metadata describing characteristics of the communication or just adding information in a key/value format, but it also has a body that could be a file, text, JSON, or many other formats. The `message` object has a very similar structure, but it is flexible enough to also represent other types of data as binary files, JMS messages, database returns, and so on.

Continuing with the HTTP request example, headers and URL parameters would be parsed to message header attributes and the HTTP body would became a `message` body.

When you did `setBody(constant("Hello World"))` you changed the message object in the exchange, by setting the string "`Hello World`" as the body attribute.

There is one thing left to explain. What does `constant("Hello World")` mean?

Expression Languages

The route class is just a blueprint, so it won't execute more than one time. So how do we deal with data dynamically? One possible answer is expression languages.

The `setBody()` method receives as an argument an object of the type **Expression**. This happens because this route step could be static or it could change depending on what data is going through the route, and that needs to be evaluated during the route creation. In this case, you want that every time a new event is triggered by the timer, the body message should be set to "`Hello World`". To do that, you used the method `constant()`. This method allows you to set a static value as a constant, in this case a string value, or to get a value at runtime and also use it as a constant. No matter what is being executed, the value will always be the same.

The `constant()` method is not the only way of dealing with the exchange data. There are other expression languages that fit different purposes. All available ELs in Camel are listed in Table 1-1.

Table 1-1. *Camel-Supported Expression Languages*

Bean Method	Constant	CSimple	DataSonnet	Exchange Property
Xquery	Xpath	XML Tokenize	Tokenize	SpEL
ExchangeProperty	File	Groovy	Header	HL7 Terser
jOOR	JsonPath	MVEL	OGNL	Ref
Simple				

> You will see examples with other expression languages in the future.

Now that you have a complete understanding of how the `Hello World` example works, you need to run this code. But there is a missing piece. How do you pack and run this code? For that you need to understand Quarkus first.

Quarkus

You will use cloud native principles and you will distribute your applications as container images, but this is the last step in the process. How do you handle the application dependencies? How do you compile Java classes? How do you run Java code? To answer these questions, you need Quarkus.

You are almost getting to the point where you can run Camel applications. You have a basic understanding of how Camel works. Now you will tackle the base framework.

Quarkus is an open source project released in 2018. It was made especially for the Kubernetes world, creating a mechanism to make Java development for Kubernetes a lot easier, and also dealing with Java "old" problems.

To understand why Quarkus is important, you need to understand the Java "old" problems. Let's talk about history.

Java Evolution

Let's step back and understand how Java was used for enterprise applications before Quarkus came into place.

Java was first released in 1995, almost 26 years ago. It is safe to say that the world has changed a lot since then, let alone the IT industry.

The idea of having a virtual machine capable of executing bytecodes, giving developers the possibility of writing code that could run in any operating system, was brilliant. It created a huge community around the Java language, making it one of the most, or maybe *the* most, popular programming language. Another feature that made huge impact on its popularity was the capacity to self-manage memory allocation, freeing developers of dealing with pointers to allocate memory space. But everything good comes with a price. The JVM (Java Virtual Machine), which is the "native" program responsible for translating Java bytecodes to machine code, requires computer resources to exist and at that point in history it was ok to spent a few megabytes on a virtual machine structure.

The majority of the enterprise application written, and here I'm talking somewhere between 2000 and 2010, were deployed in an application server. Websphere, JBoss, and Web Logic were huge back then, and I dare to say they still are, but not as much as in their glory days. Application servers provided centralized capabilities such as security, resource sharing, configuration management, scalability, logging, and so on, with a very small price to pay: a few megabytes for the virtual machine and a few megabytes for the application server code itself, besides additional CPU usage. That price would be diluted if you could deploy a bunch of applications in the same server.

To make them highly available, a system administrator would create a cluster for that particular application server, deploying every application at least twice, once for each node of the cluster.

Even though you could achieve high availability, scalability wasn't necessarily easy, and definitely not cheap. You had the option to scale everything by adding another node identical to what you had in your cluster, which sometimes would scale applications that didn't need to be scaled, therefore spending resources without necessity. You also could create different profiles and different clusters for specific applications because there were application that couldn't be scaled and required a profile of their own. Here you could have ended in a situation where

you need to have a single application per application server because the characteristics of the applications were too different from each other, making harder to plan their lifecycle together.

At this point, the price to pay to have an application server started to get higher and higher, even with AS vendors trying to make their platform as modular as possible to dodge those problems. Applications started to evolve in a different way to solve those situations.

Microservices

The cloud native way is often built on top of the *microservices architecture*. Here I will describe what it is and how it relates to the Java evolution, and most importantly, the Java framework ecosystem.

As you saw in the last section, to scale application servers wasn't an easy task. It took a specific strategy depending on your application and a lot of computational resources. Another problem was dealing with application library dependencies.

One of the things that makes the Java community so strong is how easy it is to distribute and obtain reusable libraries. Using tools like Gradle, Maven, or even Maven's older brother Ant, it was possible to pack your code into a jar/war/ear and incorporate the dependencies your application needed, or you could just deploy your dependencies straight to your application server and share them with every application on that server. This mechanism was used by numerous Java projects. Nothing was just created. Everything was reused when possible. That's fine, until you have to put different applications on the same application server.

In the begging of the application server era, handling dependency conflicts was chaotic. You could, and still can, have applications using the same library but different versions of it, and the versions may be completely incompatible. It was a true class-loading hell. Who back then didn't receive an `NoSuchMethodError` exception? Of course application servers have evolved to deal with these issues. They created isolation