Chapter 1. Introduction

In this introductory chapter, we set the scene for the rest of the book by explaining a few of the core Kubernetes concepts used for designing and implementing container-based cloud-native applications. Understanding these new abstractions, and the related principles and patterns from this book, are key to building distributed applications made automatable by cloud-native platforms.

This chapter is not a prerequisite for understanding the patterns described later. Readers familiar with Kubernetes concepts can skip it and jump straight into the pattern category of interest.

The Path to Cloud Native

The most popular application architecture on the cloud-native platforms such as Kubernetes is the microservices style. This software development technique tackles software complexity through modularization of business capabilities and trading development complexity for operational complexity.

As part of the microservices movement, there is a significant amount of theory and supplemental techniques for creating microservices from scratch or for splitting monoliths into microservices. Most of these practices are based on the *Domain-Driven Design* book by Eric Evans (Addison-Wesley) and the concepts of bounded contexts and aggregates. *Bounded contexts* deal with large models by dividing them into different components, and *aggregates* help further to group bounded contexts into modules with defined transaction boundaries. However, in addition to these business domain considerations, for every distributed system—whether it is based on microservices or not—there are also numerous technical concerns around its organization, structure, and runtime behavior.

Containers and container orchestrators such as Kubernetes provide many new primitives and abstractions to address the concerns of distributed applications, and here we discuss the various options to consider when putting a distributed system into Kubernetes.

Throughout this book, we look at container and platform interactions by treating the containers as black boxes. However, we created this section to emphasize the importance of what goes into containers. Containers and cloud-native platforms bring tremendous benefits to your distributed applications, but if all you put into containers is rubbish, you will get distributed rubbish at scale. Figure 1-1 shows the mixture of the skills required for creating good cloud-native applications.

Figure 1-1. The path to cloud native

At a high level, there are multiple abstraction levels in a cloud-native application that require different design considerations:

- At the lowest *code level*, every variable you define, every method you create, and every class you decide to instantiate plays a role in the long-term maintenance of the application. No matter what container technology and orchestration platform you use, the development team and the artifacts they create will have the most impact. It is important to grow developers who strive to write clean code, have the right amount of automated tests, constantly refactor to improve code quality, and are software craftsmen at heart.
- *Domain-Driven Design* is about approaching software design from a business perspective with the intention of keeping the architecture as close to the real world as possible. This approach works best for object-oriented programming languages, but there are also other good ways to model and design software for real-world problems. A model with the right business and transaction boundaries, easy-to-consume interfaces, and rich APIs is the foundation for successful containerization and automation later.
- The *microservices architectural style* very quickly evolved to become the norm, and it provides valuable principles and practices for designing changing distributed applications. Applying these principles lets you create implementations that are

optimized for scale, resiliency, and pace of change, which are common requirements for any modern software today.

- *Containers* were very quickly adopted as the standard way of packaging and running distributed applications. Creating modular, reusable containers that are good cloud-native citizens is another fundamental prerequisite. With a growing number of containers in every organization comes the need to manage them using more effective methods and tools. *Cloud native* is a relatively new term used to describe principles, patterns, and tools to automate containerized microservices at scale. We use *cloud native* interchangeably with *Kubernetes*, which is the most popular open source cloud-native platform available today.

In this book, we are not covering clean code, domain-driven design, or microservices. We are focusing only on the patterns and practices addressing the concerns of the container orchestration. But for these patterns to be effective, your application needs to be designed well from the inside by applying clean code practices, domain-driven design, microservices patterns, and other relevant design techniques.

Distributed Primitives

To explain what we mean by new abstractions and primitives, here we compare them with the well-known object-oriented programming (OOP), and Java specifically. In the OOP universe, we have concepts such as class, object, package, inheritance, encapsulation, and polymorphism. Then the Java runtime provides specific features and guarantees on how it manages the lifecycle of our objects and the application as a whole.

The Java language and the Java Virtual Machine (JVM) provide local, in-process building blocks for creating applications. Kubernetes adds an entirely new dimension to this well-known mindset by offering a new set of distributed primitives and runtime for building distributed systems that spread across multiple nodes and processes. With Kubernetes at hand, we don’t rely only on the local primitives to implement the whole application behavior.

We still need to use the object-oriented building blocks to create the components of the distributed application, but we can also use Kubernetes primitives for some of the application behaviors. Table 1-1 shows how various development concepts are realized differently with local and distributed primitives.

Concept	**Local primitive**
Behavior encapsulation	Class
Behavior instance	Object

Concept	Local primitive
Unit of reuse	*.jar*
Composition	Class A contains Class B
Inheritance	Class A extends Class B
Deployment unit	*.jar/.war/.ear*
Buildtime/Runtime isolation	Module, Package, Class
Initialization preconditions	Constructor
Postinitialization trigger	Init-method
Predestroy trigger	Destroy-method
Cleanup procedure	`finalize()`, shutdown hook
Asynchronous & parallel execution	`ThreadPoolExecutor`, `Fork`
Periodic task	`Timer`, `ScheduledExecutor`
Background task	Daemon thread
Configuration management	`System.getenv()`, `Propert`

[a] Defer (or de-init) containers are not yet implemented, but there is a proposal on the way t

Concept	Local primitive
lifecycle hooks in Chapter 5, *Managed Lifecycle*.	

Table 1-1. Local and distribu

The in-process primitives and the distributed primitives have commonalities, but they are not directly comparable and replaceable. They operate at different abstraction levels and have different preconditions and guarantees. Some primitives are supposed to be used together. For example, we still have to use classes to create objects and put them into container images. However, some other primitives such as CronJob in Kubernetes can replace the `ExecutorService` behavior in Java completely.

Next, let’s see a few distributed abstractions and primitives from Kubernetes that are especially interesting for application developers.

Containers

Containers are the building blocks for Kubernetes-based cloud-native applications. If we make a comparison with OOP and Java, container images are like classes, and containers are like objects. The same way we can extend classes to reuse and alter behavior, we can have container images that extend other container images to reuse and alter behavior. The same way we can do object composition and use functionality, we can do container compositions by putting containers into a Pod and using collaborating containers.

If we continue the comparison, Kubernetes would be like the JVM but spread over multiple hosts, and would be responsible for running and managing the containers. Init containers would be something like object constructors; DaemonSets would be similar to daemon threads that run in the background (like the Java Garbage Collector, for example). A Pod would be something similar to an Inversion of Control (IoC) context (Spring Framework, for example), where multiple running objects share a managed lifecycle and can access each other directly.

The parallel doesn’t go much further, but the point is that containers play a fundamental role in Kubernetes, and creating modularized, reusable, single-purpose container images is fundamental to the long-term success of any project and even the containers’ ecosystem as a whole. Apart from the technical characteristics of a container image that provide packaging and isolation, what does a container represent and what is its purpose in the context of a distributed application? Here are a few suggestions on how to look at containers:

- A container image is the unit of functionality that addresses a single concern.
- A container image is owned by one team and has a release cycle.

- A container image is self-contained and defines and carries its runtime dependencies.
- A container image is immutable, and once it is built, it does not change; it is configured.
- A container image has defined runtime dependencies and resource requirements.
- A container image has well-defined APIs to expose its functionality.
- A container runs typically as a single Unix process.
- A container is disposable and safe to scale up or down at any moment.

In addition to all these characteristics, a proper container image is modular. It is parameterized and created for reuse in the different environments it is going to run. But it is also parameterized for its various use cases. Having small, modular, and reusable container images leads to the creation of more specialized and stable container images in the long term, similar to a great reusable library in the programming language world.