Build Serverless Apps on Kubernetes with Knative

Build, deploy, and manage serverless applications on Kubernetes

Amit Deshpande
Anuj Gupta
Ashish Saxena

Build Serverless Apps on Kubernetes with Knative

Build, deploy, and manage serverless applications on Kubernetes

Amit Deshpande

Anuj Gupta

Ashish Saxena

www.bpbonline.com

First published: 2024

Published by BPB Online

WeWork

119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55515-797

www.bpbonline.com

Forewords

In recent years, serverless computing has emerged as a revolutionary paradigm, transforming the way we build and deploy applications. With its promise of scalability, reduced operational overhead, and pay-per-use pricing, serverless architecture has captured the imagination of developers and organizations alike. Among the many frameworks and platforms available for serverless development, Knative stands out as a powerful and flexible open-source solution. The ability to build and deploy applications without worrying about infrastructure management has revolutionized the way developers approach software development.

This book, "Serverless Apps with Knative" is a comprehensive resource that delves into the depths of Knative, empowering readers to harness its capabilities and unlock the full potential of serverless computing. With a blend of theoretical concepts, practical examples, and hands-on exercises, this book equips both beginners and experienced practitioners with the knowledge and skills needed to develop, deploy, and manage serverless applications using Knative.

The journey begins in Chapter where readers are introduced to the fundamental concepts of serverless computing and provided with an overview of Knative's build components. This chapter sets the stage for the exploration of Knative's capabilities in the subsequent chapters. By the end of Chapter readers will have a solid understanding of serving, eventing, event sourcing, and event consumption in a serverless context.

Chapters 2 and 3 focus on the installation and configuration of Knative. In Chapter readers will learn various methods to install and configure Knative, gaining insights into the background of installation, different installation approaches, and essential configuration concepts. The chapter also provides step-by-step recipes to guide readers through the installation process. Chapter 3 builds upon this foundation, introducing peripheral tools and frameworks for implementing DevSecOps and observability within the Knative serverless architecture. Through detailed recipes, readers will learn how to provision GitHub repositories for GitOps, install tools like ArgoCD, Prometheus, Grafana, Loki, and Jaeger, and verify the installation of these essential components.

The subsequent chapters dive deeper into Knative's core functionalities. Chapter 4 introduces the Knative CLI, guiding readers through its installation, customization, and plugin concepts. Chapter 5 provides an overview of Knative Functions, explaining the simple programming model they offer. Through practical recipes, readers will learn to implement and deploy use cases using Knative CLI, kubectl, and YAML configurations.

Chapter 6 explores Knative Eventing, focusing on understanding event-driven architectures and how Knative enables event-driven communication. Readers will gain insights into creating custom resources for Knative Eventing CRDs and deploying Apache Kafka clusters for event sourcing. This chapter also covers autoscaling with Apache Kafka and Knative Eventing, enabling readers to leverage these powerful features for handling event-driven workloads effectively.

Chapter 7 introduces routers and autoscaling in the context of Knative. Readers will discover how to deploy multiple versions of a service and distribute traffic between them. The chapter also covers blue-green deployments and canary release patterns, empowering readers to adopt advanced deployment strategies with confidence.

The book culminates in Chapter where patterns and best practices for utilizing Knative are explored. This chapter offers invaluable insights into using Istio as the default networking layer for Knative, implementing GitOps with GitHub actions and Argo CD for CI/CD, and achieving observability with log aggregation using Loki, as well as utilizing Jaeger, Prometheus, and Grafana.

"Serverless Apps with Knative" is a comprehensive and practical resource that equips readers with the necessary skills to leverage the power of Knative for building scalable and efficient serverless applications. Whether you are a developer, architect, or IT professional, this book will empower you to embrace serverless computing and harness the full potential of Knative.

I commend the authors for their meticulous attention to detail, comprehensive coverage of the subject matter, and their ability to make complex concepts accessible to readers of all levels. I am confident that this book will serve as an invaluable guide in your journey to mastering Knative and unlocking the true potential of serverless architecture.

Happy reading and serverless coding!

— A B Vijay Kumar

IBM Fellow

CTO Hybrid Cloud Services

IBM

Second Foreword

As a Practice leader & Thought leader in the Hybrid Multi Cloud team at IBM, I am responsible for partnering with multiple clients across industries in their Business modernization strategies; and developing skills, capabilities, tools & assets to facilitate & accelerate their Cloud transformation journeys. Through my 26 years of career, I have been a constant innovator, embracing new technologies that have changed & disrupted the IT industry in a positive way. As an IBM Master Inventor with 15+ issued & 6+ filed patents, and 30+ IP.com technical publications, I have been an advocate of exploring new technologies to drive efficiencies, productivity, environment sustainability and quality.

Serverless computing has numerous advantages, as it allows application developers to focus only on the application code delivering business value without worrying about underlying infrastructure, capacity planning, scalability, and so on. Knative is one such novel upcoming framework in Serverless technology that is changing the way applications are built and / or modernized. Knative is a Cloud Native Computing Foundation (CNCF) incubation open-source project, supported by major companies like IBM, Google, VMWare, RedHat. It provides a Kubernetes based serverless container platform to build, deploy & manage serverless workloads. It abstracts away many of the complexities associated with infrastructure management. By providing a cloud agnostic framework, it provides for developers to leverage innovation from across cloud providers, shift from one provider to another easily eliminating vendor lock in, and cater to a multi / Hybrid Cloud environment.

Knative Serving, one of the two key components, provides a way to deploy & run containerized serverless workloads on Kubernetes cluster without worrying about infrastructure management. It handles the tasks related to managing the lifecycle of the containerized application, routing of traffic and revision control. It also scales the number of instances based on the traffic automatically, and can even scale down to zero when there is no traffic.

Knative Eventing, the other key component, provides the users a set of APIs to use event-driven architecture for Serverless applications. These APIs can be used to create components that route events from producers to consumers that receive events. Creation, parsing, sending & receiving of events is done in a cloud agnostic way.

This book introduces the reader to the concept of Serverless computing from a beginner's point of view, clarifying the concepts, detailing out the serverless offerings from the different cloud providers and breaking the myths related to the technology. It then introduces Knative and how it helps implement Serverless containers. The book talks in detail about its two key components (Serving & Eventing), Knative functions with hands-on code recipes, installing & configuring Knative including setting up GitOps, Observability and concepts like Autoscaling, Scale to zero, Revisions, Traffic Splitting between revisions and so on. The book provides detailed examples which will provide a deep understanding of the subject and equip the readers with all the information needed to develop production ready applications with Knative. In summary, whether you are an Architect, Developer, Tester or a Business leader, this book has something for you.

I am thrilled to see a book like this which starts from the basics, covering the fundamentals of Serverless & Knative in particular, and then deep dives into the various aspects of Knative. It also outlines the architectural perspectives of Knative to me, as it traverses a learner's journey from Novice to an Experienced professional.

As someone who has led several Cloud modernization projects, I am confident that Serverless & Knative provide tremendous benefits to developers & leaders in Cloud development. This book provides a complete reference to anyone who wishes to leverage Knative, one of the finest technologies to have evolved recently.

Cheers !

— Deepak Gupta

IBM Master Inventor

IBM

Dedicated to

My wonderful kids:
Ansh
&
Anisha

About the Authors

Amit Deshpande is an Associate Partner & Executive Architect with IBM and has 21+ years of experience in the IT industry. He is a Thought Leader in the area of Digital Transformation & Hybrid Multi-Cloud Technologies. He has extensive experience in solutions & delivering multiple complex projects in Microservices/API, Integration, Event-Driven, Hybrid Cloud Architectures for various Banking, Manufacturing, and Automotive customers. He has been involved in the conceptualization and development of multiple Cloud Assets. Amit is an IBM Senior Certified Architect, Open Group Certified Distinguished Architect, Google Certified Professional Cloud Architect, and AWS Certified Solution Architect – Associate. Amit is a passionate mentor to several budding Architects in their journey in Architect Profession. He is an active member of the Architect Certification Review Board (CRB). He has also filed three patents in Kubernetes / Containerization area.

Anuj Gupta is an Executive Architect in Hybrid Cloud Manage with a specialization in Redhat-OpenShift Microservices, which is part of IBM Cloud Application Services. Prior to this role, he was part of IBM Application Innovation Lab and Export Blue, working as a Senior Architect across multiple LOBs and Portfolios. During his 17+ years of experience, he has been involved in defining technical architecture and end-to-end delivery for multiple projects across Open Subsurface Data Lake (OSDU Opensource with the Open Group) for IBM Data & AI, B2B payments, Card Authorization System, Enterprise Digital Analytics, AP and AR systems of large Financial Services clients. He has extensive experience in developing cloud-native applications based on OpenShift for various workloads (stateful and stateless) and also migration of various monolith applications from VSS to IaaS and then to OpenShift using microservice architecture.

Ashish Saxena is currently working as an Application Architect at IBM and has 15+ years of experience in the IT industry. He is an SME for Digital Transformation & Hybrid Cloud Technologies. He has extensive experience in solutions and designing applications using microservices and Event Driven architecture for various Telecommunication, Retail, and Banking customers. Ashish is a certified Azure Solution Architect Expert.

About the Reviewer

Gaurav is a passionate technology leader and hands-on technologist, with a track record of driving technical innovation. Gaurav has delivered solutions for enterprises and start-ups operating in leadership, management, architectural, and development capacities. Gaurav has over 26 years of experience, collaborating with some of the most well-known technologies like Java, Microsoft, Angular, React, Python, PHP etc. pertaining to the domains Medical, Media, Construction, Gaming, Finance, ATM, Supply-chain, and so on. Gaurav has a doctorate in computer science (Machine Learning) from California Public University. Gaurav is a Microsoft MVP award recipient. He is a Mentor of Change with AIM NITI Aayog, Govt. of India, Business Coach with Business Blaster, Govt of NCT of Delhi. He is a lifetime member of the Computer Society of India (CSI), an advisory member and senior mentor at IndiaMentor. He has authored books across-the-technologies. Recently, Gaurav has recognized as a world record holder for writing books in exceptional technologies.

Acknowledgement

I would like to take this opportunity to express my profound gratitude to my family – my parents, my wonderful wife Vidyuta, and my brother Amogh. Their unwavering support and encouragement have been instrumental in my journey.

I extend my heartfelt thanks to my esteemed friends and colleagues, Anuj Gupta and Ashish Saxena, for their invaluable contributions as co-authors of this book. Their extensive industry experience and technical prowess have added immense value to the book.

My sincere appreciation goes to the exceptional team at BPB Publications for their guidance and expertise throughout the book's development process. The contributions from reviewers, technical experts, and editors have been indispensable.

I am deeply grateful to my mentors, notably A B Vijay Kumar and Deepak Gupta, who have served as a constant source of inspiration and guided me towards the path of becoming an author. I would also like to express my gratitude to IBM and my management for their encouragement and support in writing this book.

Finally, I would like to extend my appreciation to all the readers who have taken an interest in this book and for their support in bringing it to fruition.

Preface

In the ever-evolving landscape of software development, there has been a continuous search for tools and methodologies that streamline the process of creating and deploying applications. As cloud computing has become an integral part of modern development workflows, the focus has shifted towards harnessing its full potential, leading to the emergence of serverless architecture. Among the various technologies that support this paradigm, Knative has emerged as a powerful and versatile solution for deploying and managing serverless containers.

This book aims to provide a comprehensive guide to understanding, implementing, and optimizing Knative serverless containers for developers, DevOps engineers, and IT professionals. We will start by introducing the core concepts of serverless architecture and its benefits, followed by an in-depth exploration of Knative, its components, and how it fits within the broader Kubernetes ecosystem.

Through a series of practical examples and case studies, we will demonstrate how to build, deploy, and manage serverless containers using Knative. We will cover topics such as setting up a development environment, creating custom serverless applications, integrating with other cloud-native tools and services, and best practices for monitoring, logging, and troubleshooting.

In addition, we will delve into advanced topics such as scaling, and performance optimization, helping you gain a solid understanding of how to deploy and maintain high-performing, resilient serverless applications using Knative.

Whether you are new to serverless computing, a seasoned professional looking to expand your knowledge, or an organization considering adopting Knative, this book will serve as a valuable resource to guide you through the intricacies of serverless containers and their effective management.

As the field of serverless computing continues to evolve, it is our hope that this book will equip you with the knowledge and confidence to navigate and harness the power of Knative serverless containers, unlocking new possibilities and efficiencies in your software development journey.

Happy reading, and here's to a future of seamless, scalable, and efficient serverless applications!

Chapter 1: Serverless and Knative in a Nutshell – provides a detailed overview of the evolution of serverless, how it works, and its key advantages. The chapter also provides the reader with a point of view on when serverless should be considered and its pitfalls. Furthermore, the chapter covers serverless offerings, Function as a Service (FaaS), and Serverless Containers, along with an introduction to the Knative project.

Chapter 2: Installation and Configuration of Knative – Part I - presents detailed steps to install and configure various Knative pre-requisite software like Kubernetes (K8S) Cluster, Helm as a package manager, and Istio as a services mesh. The chapter explains the various ways of installing Knative and provides detailed installation and validation steps of Knative by deploying and running a pre-built app.

Chapter 3: Installation and Configuration – Part II - covers the installation of various monitoring and observability tools. The chapter explains the installation and validation steps for Kafka eventing and ArgoCD for GitOps. This chapter also covers the installation of various observability tools like Loki, Prometheus, Jaeger, and Grafana.

Chapter 4: Knative Functions – An Overview - allows the reader to learn fundamental concepts of Knative Functions by demonstrating how to create, build, deploy, and run Knative Functions on local and remote K8S Cluster using hands-on code recipes.

Chapter 5: Knative Serving - gives special attention to the Knative Serving component, demonstrating how to build, deploy and run serverless containers using Knative Serving on the local K8S Cluster. The chapter explains the Knative Serving concept based on a Case Study and hands-on code recipes.

Chapter 6: Knative Eventing - gives special attention to the Knative Eventing component, demonstrating how to build, deploy and run services using Knative Eventing on the local K8S Cluster. The chapter explains various Knative Eventing patterns with a Case Study and hands-on code recipes. This chapter also covers how Kafka can be used as a messaging service in Knative Eventing.

Chapter 7: Scaling and Routing - explains in detail how to scale the serverless application based on demand using a Case Study by deploying it on a remote K8S Cluster. This chapter also allows the reader to learn about various autoscaling configurations in Knative. Furthermore, the chapter covers the concepts of routing along with traffic splitting and switching using hands-on code recipes with detailed implementation of various Deployment Strategies like Blue-Green, Canary, and A/B Testing.

Chapter 8: Knative Best Practices – explains how to leverage the power of Knative effectively, the problems associated with services deployment, and their solution using GitOps strategy with a Case Study. This chapter covers managing the deployments using GitOps strategy and continuous deployment using ArgoCD. This chapter also covers Observability with the configuration of various tools like Loki, Prometheus, Jaeger, and Grafana using hands-on code recipes.

Table of Contents

C

HAPTER

1

Serverless and Knative in a Nutshell

Introduction

Serverless Computing is commonly known as Serverless. The name Serverless Computing or Serverless itself sounds like a paradox, is it not? We all know for a fact that, for any software to run, it needs basic infrastructure - compute, storage, and network. So, does the Serverless claims the ability to run software without any server infrastructure? Certainly not.

This chapter will walk you through the evolution of We will learn about the two main concepts of Serverless - Function-as-a-Service and Serverless We will also learn about the Knative project and how it helps to implement Serverless Containers.

Structure

In this chapter, we will discuss the following topics:

Introduction to Serverless

Function-as-a-Service

Serverless Containers

Introduction to the Knative project

Objectives

Upon completing this chapter, you will acquire a comprehensive grasp of Serverless Concepts and the distinctions between FaaS and Serverless Containers. Furthermore, you will be introduced to the Knative project.

Introduction to Serverless

Over the last three decades, tremendous innovation in server infrastructure has been seen. With the advent of the Internet, enterprises were scrambling to make information available round the clock from anywhere in the world during the last decade of the century. It all started with Physical Servers, also known as bare-metal. As depicted in the following figure, getting a bare-metal and deploying the application on it required a high lead time, a lot of effort, and a huge upfront Capital Expenditure

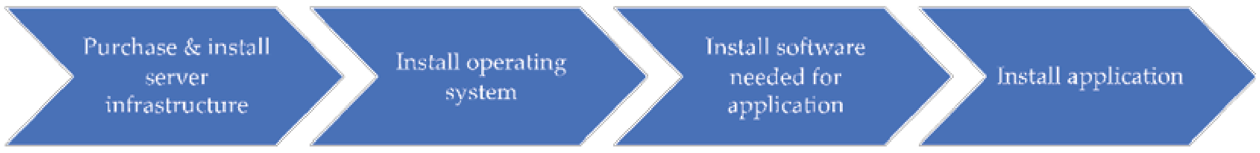

Purchase & install server infrastructure
Install operating system
Install software needed for application
Install application

Figure 1.1: Installing application on the physical server

Almost a decade later, such applications have grown significantly in numbers, and enterprises had to own/manage hundreds of bare-metal servers. The enterprises were looking for a solution that could improve the resource utilization of bare-metal servers. Then came the Virtual Machine technology which allows partitioning of the bare-metal, making it easy to adapt to dynamic workloads by reallocating resources. It also provided opportunities for public cloud providers to offer compute platforms on rent, an offering now popularly known as Infrastructure-as-a-Service As depicted in the following figure, VM technology reduced the lead time to provision server infrastructure significantly and made deploying applications on VM a bit easier, but it still needed a lot of configuration and setup to be done.

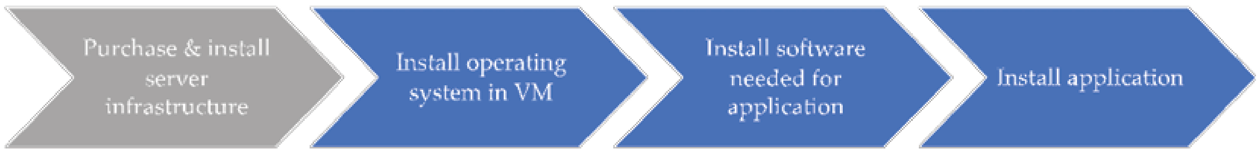

Purchase & install server infrastructure
Install operating system in VM
Install software needed for application
Install application

Figure 1.2: Installing application on Virtual Machine

A few years later, Docker splashed on the scene with the ability to package and run containers. It enabled multiple applications with different OS requirements to run on the same OS kernel as containers and provided an opportunity for further simplification and resource savings. Unlike VMs, containers need significantly less resource footprint, are blazingly fast to spin up or down, and need less overhead to manage. Shortly after, Kubernetes, an open-source container scheduler and orchestration tool, was launched, which now has become a default standard. As shown in Figure with containers, it was made possible to significantly reduce the lead time and scope to be able to deploy an application:

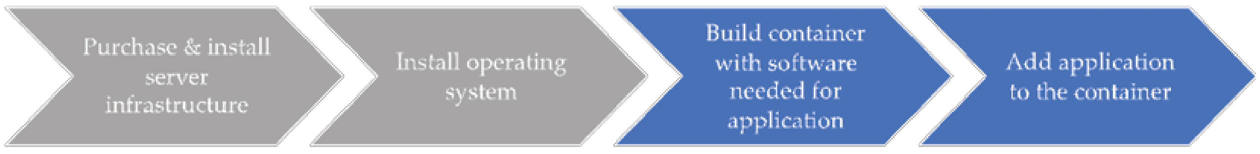

Purchase & install server infrastructure
Install operating system
Build container with software needed for application
Add application to the container

Figure 1.3: Installing the application as a container

Then came the concept of Serverless, which challenged the notion of the need for continuously running servers to serve the applications. Serverless provided an architectural approach for ephemeral infrastructure to serve an application that can come into existence on an incoming request and disappears immediately after serving it. Serverless allowed application developers to focus only on the application code delivering business value without worrying about underlying infrastructure, capacity planning, scalability, and so on. As depicted in Figure Serverless approach improved lead time drastically to build and deploy an application:

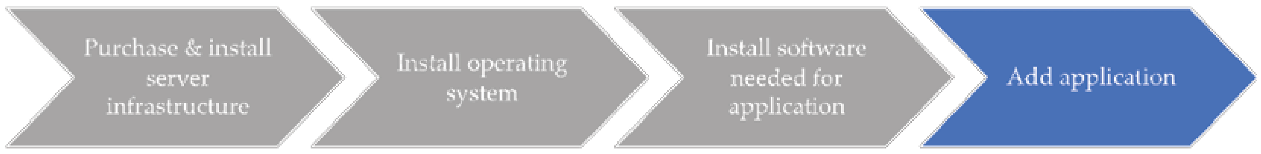

Purchase & install server infrastructure
Install operating system
Install software needed for application
Add application

Figure 1.4: Installing application on serverless platform

As we know, software applications seldom have constant load, and it keeps changing with time. So, we tend to either over or under provision the infrastructure. As shown in Figure physical servers and virtual servers provide very less elasticity, the provisioned infrastructure remains constant. This leads to wastage of idle resources when load is less (overprovisioning) and poor quality of service/lost business revenue in case of excess load (under provisioning). Please refer to the following figure:

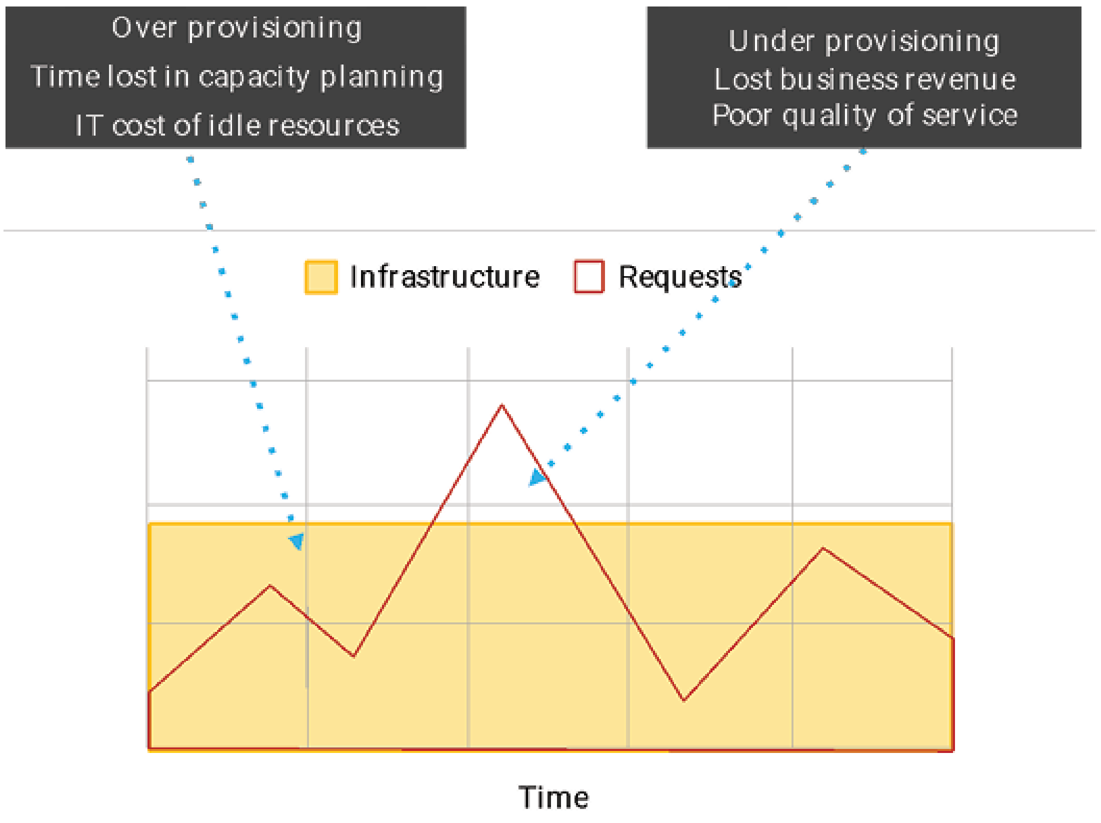


Figure 1.5: With bare metals and VMs

As shown in the following Figure Cloud computing, with its autoscaling feature, tries to solve this problem to some extent. Since scaling of VM infrastructure (up or down) takes few minutes, the minimum server capacity must be kept active all the time, which leads to the idle resources even when load is less. Please refer to the following figure:

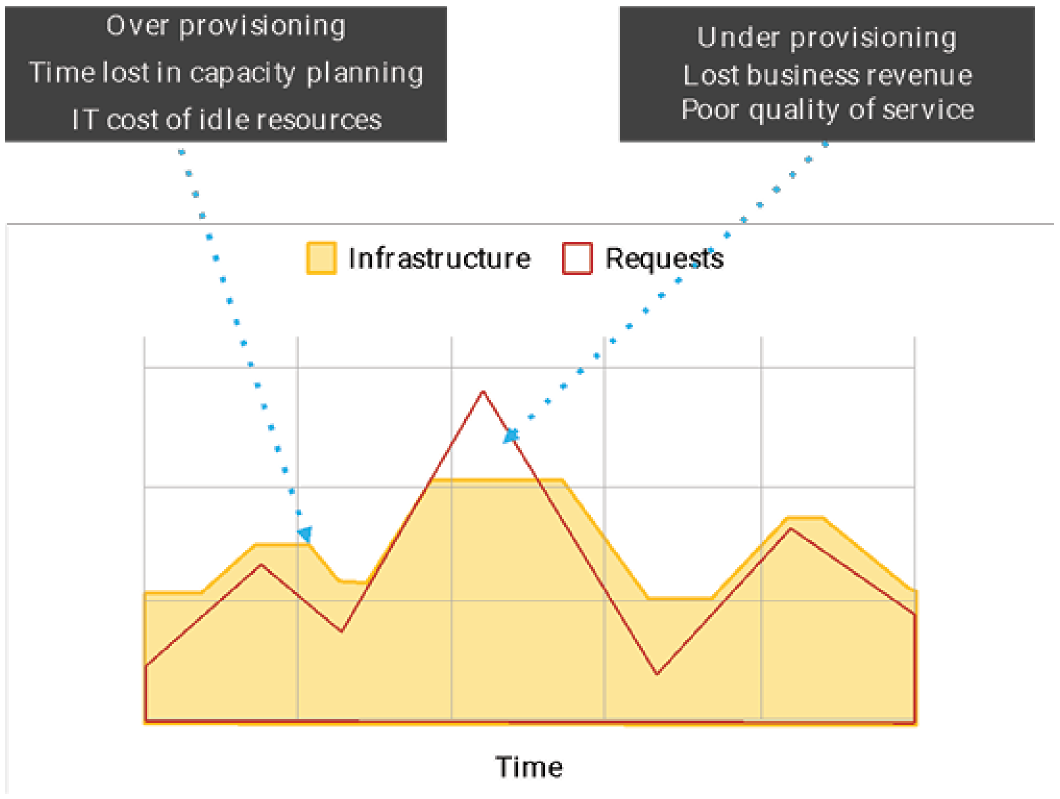


Figure 1.6: With auto-scaling on cloud

Serverless tries to match the infrastructure needed to the load on the application in real time as shown in the following Figure With this it is able to provide optimal usage of the infrastructure resources and save overall IT costs. And hence Serverless has become a very attractive proposition for Enterprises. Please refer to the following figure:

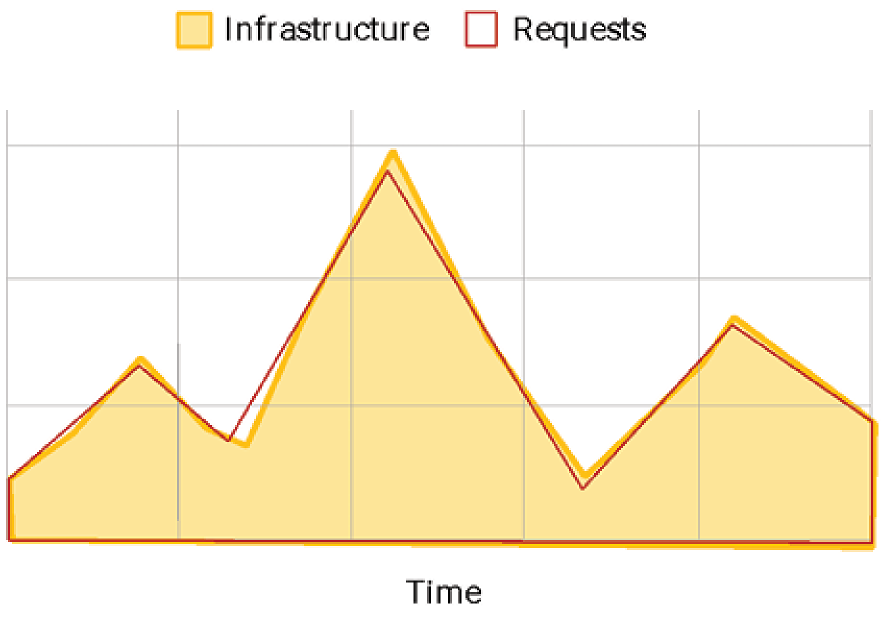


Figure With Serverless

Serverless

Let us get into a bit of detail about Serverless. So, what is Serverless? There are multiple definitions available. The one that fits the most is Serverless is a compelling paradigm for deployment of applications and services which is based on Event Driven Architecture, built-in scalability, taking application centric approach by completely abstracting the physical infrastructure and its management from the application developers. There is a long-standing belief about Serverless is no server infrastructure is needed. This is not true. All Serverless platforms allow the code to run on a trigger. To run the code the Serverless platforms need to provision server infrastructure till its execution is complete.

How serverless works

Let us take an example of a web application which has a front-end which gets loaded in a browser as shown in the following figure. When front-end sends a HTTP request, the serverless platform checks a backend application is available to serve this request. If the backend application is not available, then platform spins it up, which will serve the request. If multiple such requests start coming in and if one backend application is not able to serve them at the same time, the serverless platform automatically creates multiple instances of backend application. And if incoming requests go down, the serverless platform will scale down the backend application, even to zero. Please refer to the following figure:

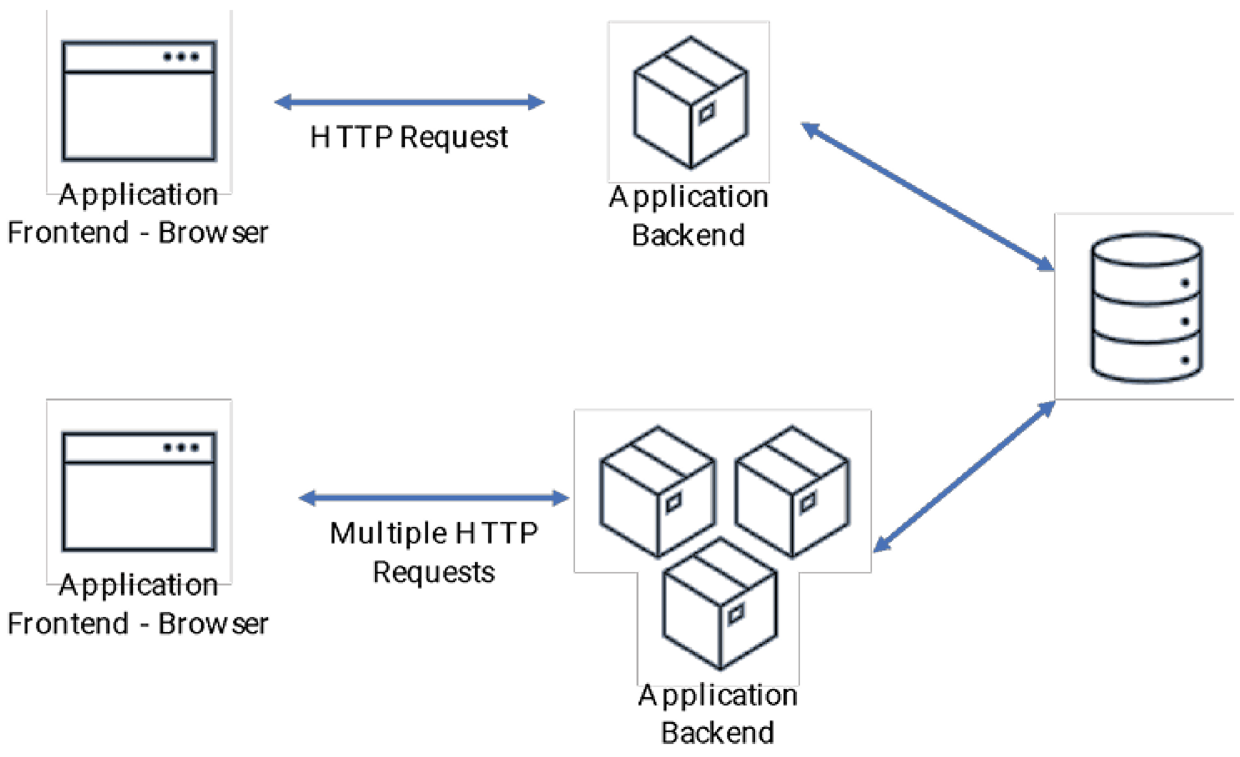


Figure 1.8: Serverless pattern for web application

Similarly, it works for event-driven applications as well, as shown in the following figure:

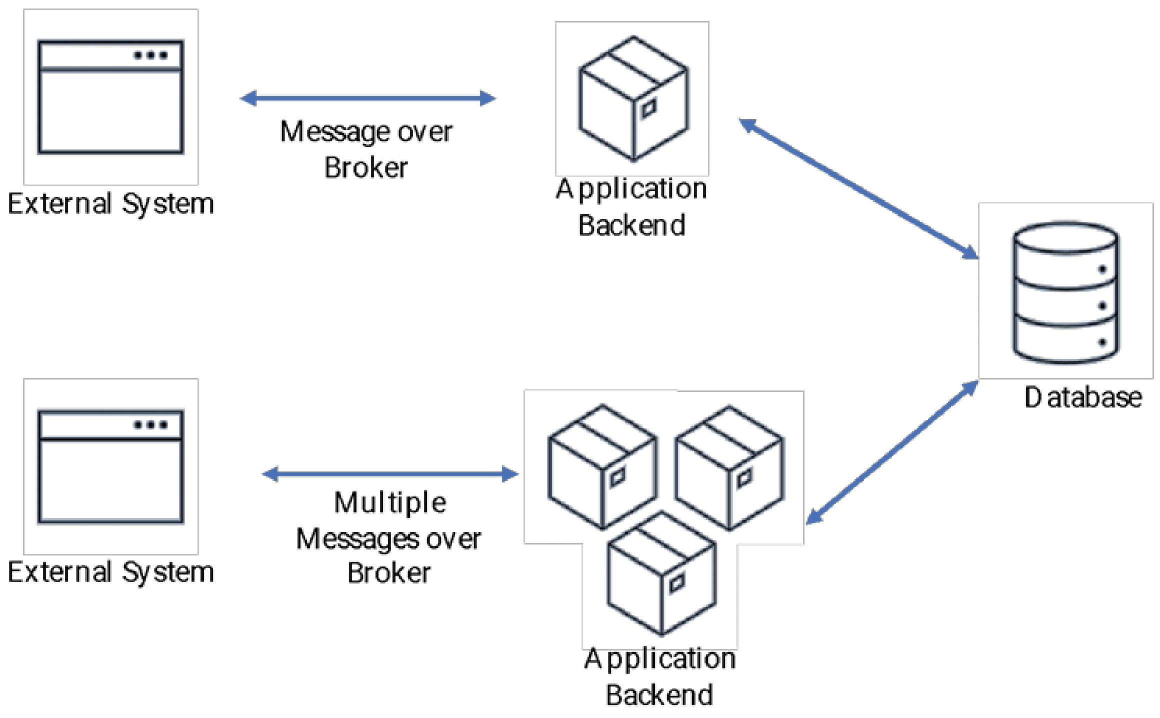


Figure 1.9: Serverless pattern for event driven application

Key advantages of serverless

Serverless is a simple concept with many advantages like:

Enables optimal usage of infrastructure If there is no workload on the application, the serverless platform scales it down to zero. This means no infrastructure is blocked / utilized in such cases, which allows it to be used for other applications in need.

Inherently scalable architecture: The serverless architecture mandates platform to be scalable by default. The serverless applications developers do not have to define how much it needs to be scaled. The application can be scaled to the maximum limit of the platform and can be scaled down to zero if there is no workload.

Eliminates the need for maintaining/managing the infrastructure: the application developers are freed from infrastructure related tasks like setting up auto-scaling, load balancers, SSL certificates, and so on. for the application, the serverless platform takes care of these.

Enables development teams to focus on development and delivery of core business values to the teams become more agile and innovate faster as Serverless provides flexible infrastructure, easier development, shorter development cycle and build powerful polyglot applications best suited for customer needs.

Serverless should be considered for

Serverless is an architectural pattern, it is not a silver bullet and should not be considered for any scenario without proper due diligence. The following are a few scenarios (not limited to) that are best suited for serverless patterns:

Application with unpredictable load or bursts of requests

Applications with seasonal workloads with varying peaks

Applications requiring event-driven architecture and loosely coupled system design

Applications with a lot of idle time

Applications with stateless microservices design

Applications with short running functions and / or need high system resources for processing.

Key limitations/shortcomings of serverless

Following are some of the key disadvantages or shortcomings of the serverless pattern:

Cold When there is no active load on the serverless application, the serverless platform will scale it down to zero and will release all infrastructure resources used by that application. So, when the first request comes in for the application, the application needs to be instantiated which leads to slower response time. This performance issue with serverless applications is also known as cold start. Once the application is instantiated (also known as warm), it can handle incoming requests/events with much faster response times.

State As serverless applications can be killed by the platform at any time, these applications effectively need to be stateless, and they need to depend upon another component for state management.

Local For serverless application development, developers need to rely on unit tests as end-to-end integration testing locally is significantly difficult. This is a result of difficulty in replicating serverless platform on local environment of developer.

Monitoring and Monitoring and debugging serverless applications is quite difficult and time consuming. That is mainly because of every time a serverless application instance spins up it creates a new version of itself.

Also there are tools available for remote debugging for serverless applications.

Serverless is multi-tenant by definition, as same/available infrastructure will be used to instantiate needed serverless application when trigger/event arrives for the same. This might be a security compliance issue for some organizations.

Scenarios not suited for serverless

Similarly, following are the few scenarios for which serverless patterns should not be considered:

Application which has use cases which mandate to have stateful services

Applications with long tunning tasks / services

Applications with stringent compliance requirements

Complex applications which take long time to load / initialize

Applications with very low response time (sub-milliseconds) requirements

Serverless 1.0 - Function-as-a-Service

The Public Cloud providers also popularly known as Hyperscalers were one of the first to identify the power of Serverless and they came up with Serverless offering which now popularly known as Function-as-a-service, a.k.a. FaaS. FaaS allows one to execute code in response to an event without need to manage complex infrastructure typically needed to run such applications. This allowed application developers to focus completely on development of the business functionalities and delivering business value.

The following table shows popular FaaS offerings by various Hyperscalers:

Table 1.1: FaaS offerings by hyperscalers

All hyperscalers FaaS offerings have matured over the period of time. They now provide:

Choice of various runtimes

Multiple modes of invocation (event triggers, api, messaging, and so on.)

Support for environment variables, logging and monitoring

Extended environment support for database, storage, integration with other cloud services

All Hyperscalers provide almost similar functionalities and common features for their FaaS offering and they all provide very cheap pricing! Just to give an example AWS offers first 1M Lambda executions FREE and 20 cents for next 1M executions. All other Hyperscalers offer similar pricing models. Although FaaS has enabled variety of use cases, it still lacks enterprise computing needs. Some of the key shortcomings of FaaS are:

Resource FaaS providers / Hyperscaler platforms have defined limitations on resources (CPU, I/O, memory, and so on) under which FaaS functions need to work.

Limited execution By definition FaaS function cannot run indefinitely. The allowed execution time ~5 mins which varies by the Hyperscaler

Limited The orchestration/integration options available are hyperscaler specific

Limited local development Local development and debugging tooling is mostly provided by hyperscaler and limited.

Vendor FaaS offerings have hyperscaler specific implementation which developers need to comply with. This makes porting functions across FaaS platforms difficult.

Serverless 2.0 - Serverless Containers

In comes the Serverless Container technology and as name suggests, brings the best of both paradigms:

Enable abstracting applications from underlying infrastructure helping enterprises to innovate faster.

Applications can be packaged as OCI compliant containers that can run anywhere removing vendor lock-in.

Serverless container technology makes use of container orchestrator platforms like Kubernetes. It allows to package of the code as a container image and provides deployment mechanism on container orchestrator platform (Kubernetes). Then serverless container platform takes care of starting/stopping/scaling of the deployed container as per the workload. And it solves most of the shortcomings of FaaS, mentioned in the previous section, as

Developers can work with choice of their run times, local development environment and package their code as container image – something they are most familiar with

All the advantages of serverless and FaaS are still there

All the orchestration and integration options provided by the container orchestrator are available to use

Removes vendor lock-in as code is packaged as containerized image which can run anywhere

With this background, now let us come to the main topic of this book – Knative which is a leading framework for building applications using Serverless Container technology.

Introduction to Knative

Knative is an open source, enterprise level and Kubernetes based serverless container platform for deploying and managing serverless and event driven applications. Knative is a Cloud Native Computing Foundation incubation project strongly backed by RedHat, Google, IBM and others. Knative means Kubernetes native which is very important. It makes the Knative cloud provider agnostic framework – if there is a Kubernetes environment available, Knative will work as well, eliminating vendor lock-in issues.

The following figure shows high level architecture of Knative. Knative provides a layer on top of Kubernetes and provides Kubernetes-native mechanism to manage serverless containerized applications on Kubernetes:

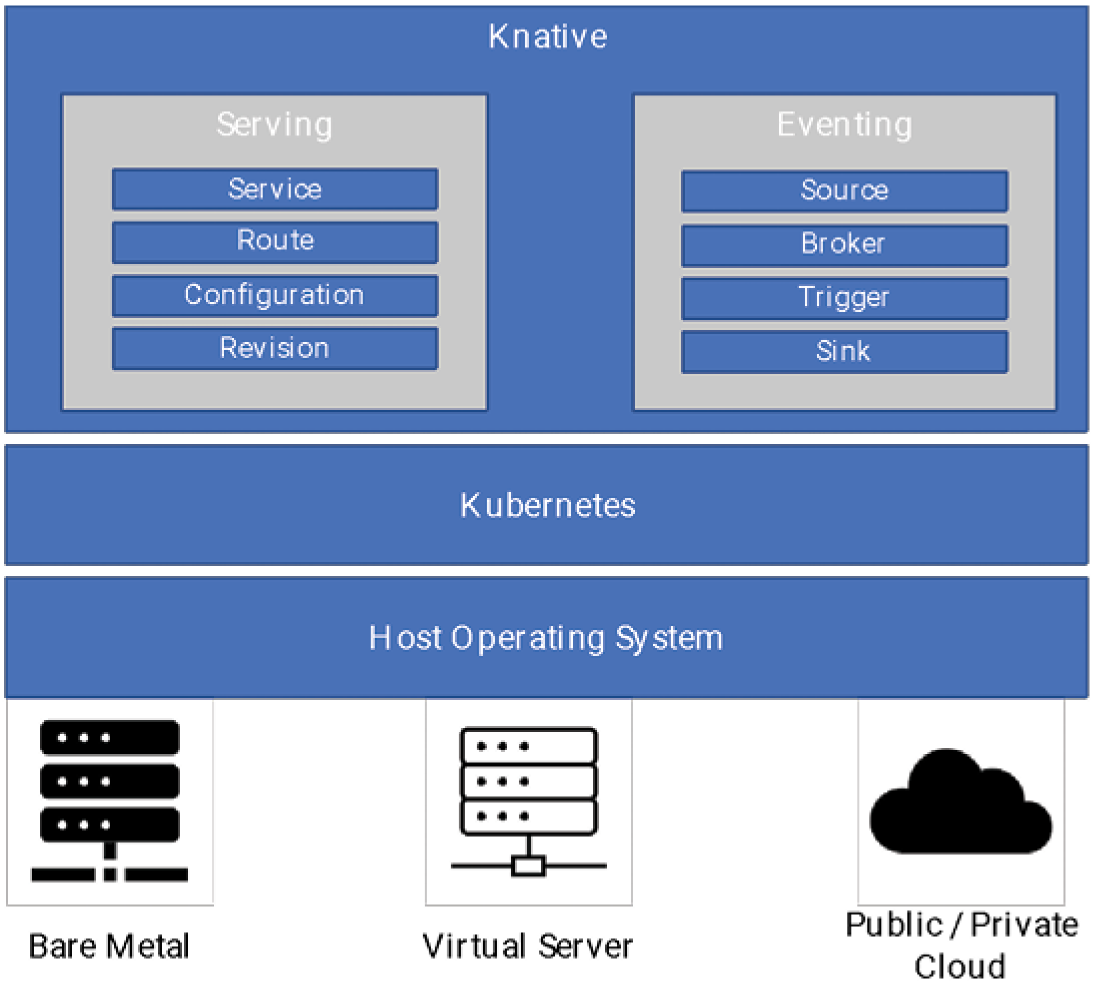


Figure 1.10: Knative architecture

Knative has two main components: Serving and Eventing.

Knative Serving

Knative Serving provides mechanism to deploy and run containerized serverless workloads on Kubernetes cluster without having to deal with cluster or server management. It handles tasks such as revision control, traffic routing and autoscaling for deployed serverless workloads. It has four objects which are defined as Custom Resource Definitions as shown in the following figure:

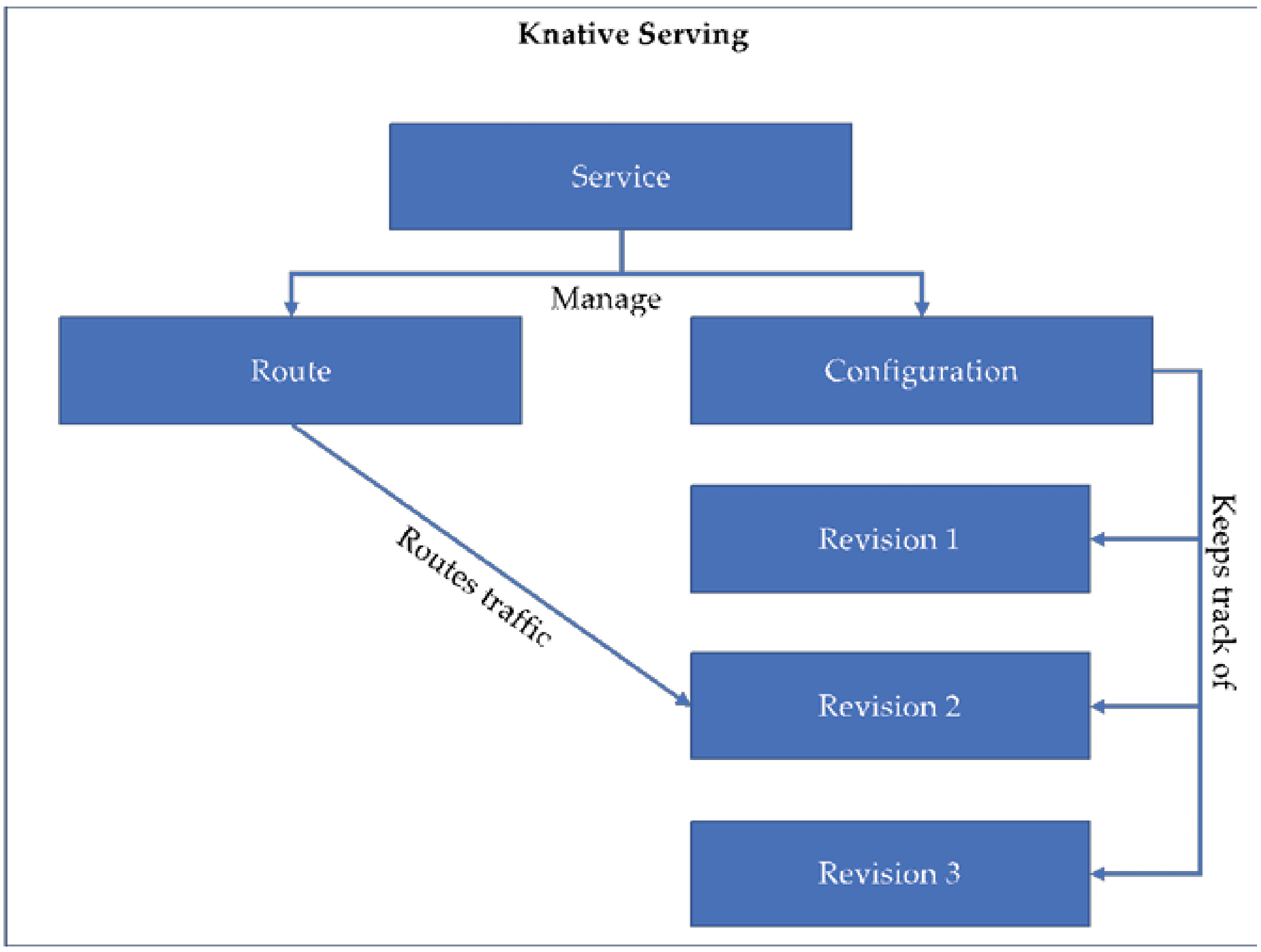


Figure 1.11: Knative serving

Let us understand the Knative Serving objects and how they interact with each other

Service Manages the entire lifecycle of the containerized application deployed using Knative. Service is also responsible for creating routes and configurations. A service can be defined to route traffic to any available Revisions of the deployed application.

Route Provides a network endpoint to a service that is backed by one or more revisions. It also helps to manage traffic in various ways.

Configuration Describes the desired latest revision state and tracks the status of revisions of the deployed application. If the configuration spec is updated, a new Revision is created.

Revision Is a point-in-time snapshot of a deployed application. Revisions are immutable objects and can be scaled up or down according to the incoming traffic.

Knative Eventing

Knative Eventing is a set of APIs that provides a mechanism to use event-driven architecture for Serverless applications. These APIs can be used to create components that route events from event producers to event consumers, also known as sinks, that receive events. These events conform to the CloudEvents specifications, which enable creating, parsing, sending, receiving events in any programming language and in cloud-agnostic way. The following image depicts few of the Knative Eventing building blocks and how they interact with each other to build an eventing

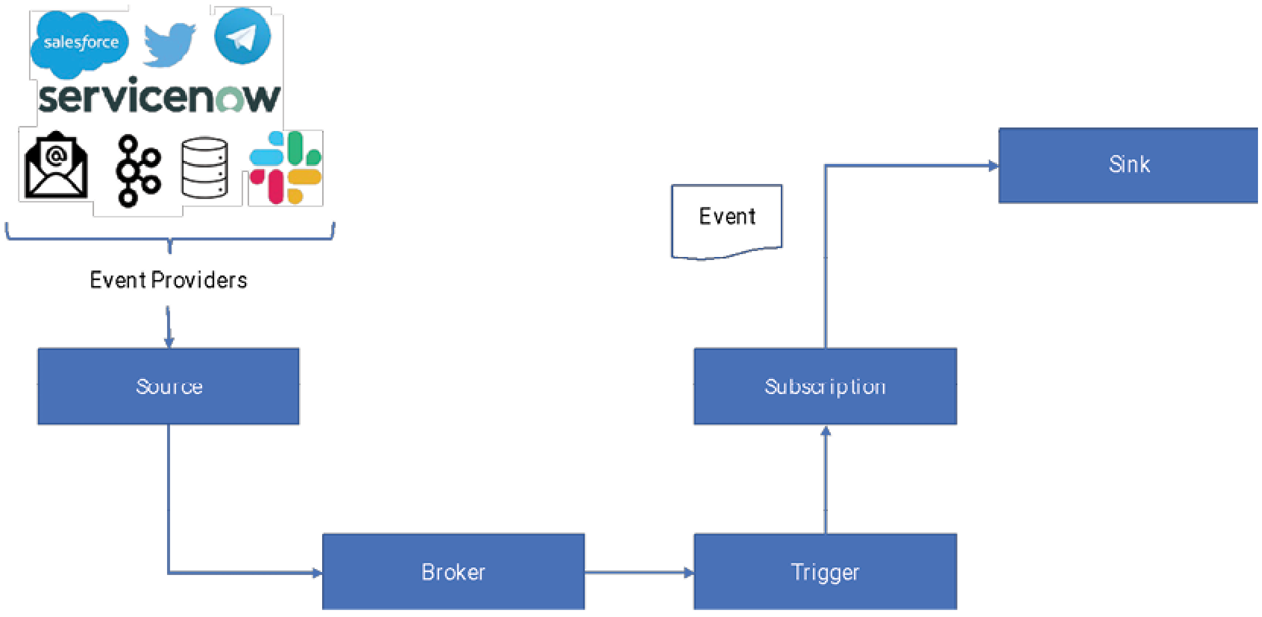


Figure 1.12: Knative Eventing

Let us have a look at these Knative Eventing building blocks

Source: A Kubernetes Custom Resource that register interest in a class of events from a particular system

Broker: A Kubernetes CR that defines an "event mesh" which provides a discoverable endpoint for event ingress, and send / trigger events to multiple subscribers.

Trigger: A desire to subscribe to events of a given Broker or event consumer using a Filter.

Filter: Applied to a Broker in order to allow types of events to be selected.

Channel: is an interface between event source and the subscriber. It can store the incoming event and distribute them to the subscribers.

Subscription: Services can subscribe to a channel using Subscription to be able to start receiving events.

Sink: An addressable / callable resource that can receive incoming events from other sources.

Note: Addressable resources: Able to receive and acknowledge an event delivered over HTTP to an address defined in the status.address.url field of the event.

Callable resources: Able to receive an event delivered over HTTP and transform it, returning 0 or 1 new event in the HTTP response payload.

We will get into much more detail for both Knative Serving and Eventing in subsequent chapters of this book.

Knative features

Traditional Kubernetes deployment has quite a few steps, which involve creating long configuration files typically in YAML format describing various moving parts like pod, service, deployment, replicaset, and so on.

In contrast, Knative requires only one resource file called Knative Service, which literally says, “run this image.” Once this file is applied to the cluster, the Knative Operator automatically creates the required resources – service, route, configuration, and revision. With Knative, developers do not have to worry about infrastructure at all.

Some of the other Knative features are

Scale to zero in case of no traffic and scale from zero in case of a spike in traffic, providing optimal usage of infrastructure resources

Progressive Progressive rollout strategy (blue/green, Canary deployments, and so on) made easy with Knative as it provides Configuration and Revisions

Knative provides for integration with various event sources and handling of events by triggering handlers

All Knative components are Kubernetes native and hence can be easily integrated and extended.

Conclusion

In this chapter, we covered how innovation led to major shifts in infrastructure which have led to the rising of the Cloud and now the Serverless platforms. We also looked at how Serverless started as a FaaS and its advantages as well as shortcomings. We then covered the latest Serverless offering, which is Serverless Containers, and how it tries to overcome the shortcomings of FaaS. Then we had a look at the leading Serverless Container project – Knative, its architecture, and its main components – Knative Serving and Knative Eventing.

In the next chapter, we will get our hands dirty and start with installation of Knative and related components.

Multiple choice questions

Which of the following is not the advantage of Serverless?

Inherently scalable architecture

Focus on delivery of core business values to the customer.

Does not provide elastic infrastructure based on incoming requests.

Optimal usage of Infrastructure

Which technology does Knative make use of?

Containers

Runtimes

C++

Python

Which of the following statement is True about Knative?

Knative always results in cloud vendor lock-in.

Knative is proprietary technology.

Knative applications cannot be hosted on public cloud.

Knative is open source and cloud provider agnostic framework.

Answers

c

a

d

Key terms

A large technology companies that operate massive, highly scalable data centers and cloud computing infrastructures to provide services and resources on a global scale. Examples: Google, Amazon, Microsoft

CAPEX (Capital Refers to the upfront investment in physical infrastructure, hardware and software licenses required for building and maintaining data centers, while Opex (Operational Expenditure) pertains to the ongoing operational costs, such as maintenance, utilities, and personnel, associated with running and managing the cloud services.

C
HAPTER
2

Installation and Configuration of Knative – Part I

Introduction

There are various approaches to installing and configuring Knative on a K8S Cluster. This chapter will walk you through the installation and configuration of various pre-requisite software to get started with Knative.

Structure

In this chapter, we will discuss the following topics:

Kubernetes cluster

Exploring and understanding various methods to get Kubernetes cluster

Recipes

Installing kubectl CLI.

Validation of Kubernetes cluster

Kubernetes package manager

Understanding Helm and its need

Recipes

Installing Helm on Linux

Validate Helm installation

Istio an overview

Validate Knative Serving through pre-built images.

Validate Knative Eventing through pre-built images.

Sending and verifying cloud events

Objectives

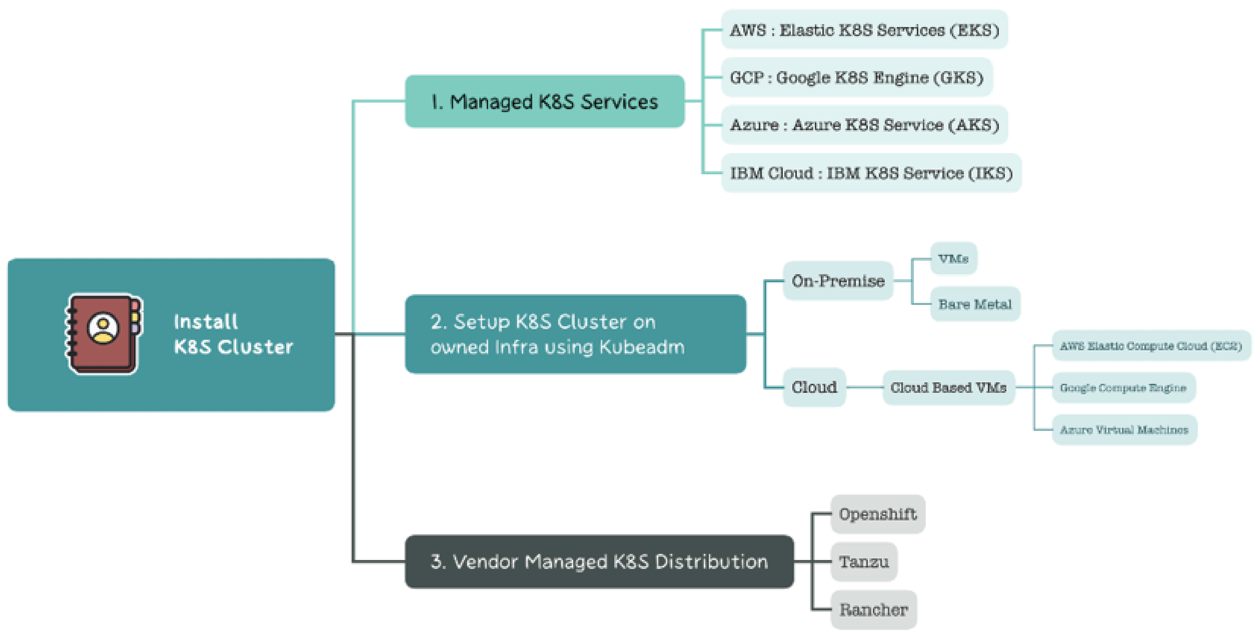
By the end of this chapter, you should have a setup of your Kubernetes cluster with Knative components installed. You will also learn about various installation methods and Knative dependencies.

Kubernetes cluster

A Kubernetes cluster is a group of nodes (physical or virtual machines) that run containerized applications and work together to provide the Kubernetes services necessary to manage those applications.

Explore and understand various methods to get Kubernetes cluster

Knative helps you to run serverless workloads in a Kubernetes-native way. To learn Knative and try exercises in the upcoming chapters, it is very important to first explore and understand the various methods to procure/install the Kubernetes Cluster. You can get the K8S cluster from one of the methods described in Figure which explains various pathways for installing the K8S Cluster:



Install K8S Cluster
1. Managed K8S Services
AWS : Elastic K8S Services (EKS)
GCP : Google K8S Engine (GKS)
Azure : Azure K8S Service (AKS)
IBM Cloud : IBM K8S Service (IKS)
2. Setup K8S Cluster on owned Infra using Kubeadm
On-Premise
VMs
Bare Metal
Cloud
Cloud Based VMs
AWS Elastic Compute Cloud (EC2)
Google Compute Engine
Azure Virtual Machines
3. Vendor Managed K8S Distribution
Openshift
Tanzu
Rancher

Figure 2.1: Various methods to install K8S cluster

Managed K8S A Managed K8S Service such as AWS Elastic Kubernetes Service Google Kubernetes Engine Azure Kubernetes Service IBM K8S Service can be procured. These services allow you to create and manage a cluster without having to worry about the underlying infrastructure.

Setup K8S cluster on owned Using the Command-Line Interface like kubeadm and automation tools like Ansible, Terraform a K8S cluster can be installed/deployed on owned infrastructure, which can be either on on-premises or on Cloud. Within each of public cloud, cloud-based Virtual Machine services such as AWS Elastic Compute Cloud Google Compute Engine and Azure Virtual Machines can be leveraged to set up the K8S Cluster. Within on-premises, one can use VMs or bare metal servers to create the K8S cluster. For the local development environment, options like minikube and microk8s can be leveraged to create and run a single-node K8S cluster. This can be used for local development and testing purpose.

Vendor managed K8S The K8S distribution like Red Hat OpenShift, Rancher, and VMWare Tanzu provide additional features and tools to create, manage and scale K8S cluster.

For experimentation/prototyping/learning purposes, a Local/Single Node machine having 3 CPUs and 8GB memory can be used.

In this book, for all the Knative exercises and recipes, we will use the Managed K8S cluster (Method - 1 explained above) from one of the major cloud providers, which can host and scale any production-grade workload. The configuration used is 3 nodes, and each node is 4X16 (4 CPUs, 16GB RAM) with Ubuntu 20.04. K8S version used is 1.25.x. You can use either options 1 or 2 to get the K8S cluster setup.

Install kubectl CLI

kubectl is a CLI for running commands against K8S clusters. It allows you to run and manage application deployments in a K8S environment. You can use kubectl to deploy, inspect, and manage the state of your applications, as well as to view and modify the configuration of your cluster.

To install kubectl on Ubuntu 20.04, follow these steps:

#Update the package manager cache:

```
sudo apt-get update
```

#Install the required dependencies:

```
sudo apt-get install -y apt-transport-https
```

#Add the GPG key for the official Kubernetes package repository:

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
```

#Add the Kubernetes package repository to your system and update cache manager again

```
echo

"deb http://apt.kubernetes.io/ kubernetes-xenial main"

 | sudo tee /etc/apt/sources.list.d/kubernetes.list

sudo apt-get update

#Install kubectl:

sudo apt-get install -y kubectl
```

You should now have kubectl installed on your system. To verify the installation, run the following command:

```
kubectl version --client
```

The output will be like the below, the exact version#, git commits, and build date may differ based on the time CLI was installed:

```
user@virtualserver01:/
# kubectl version --client

WARNING: This version information is deprecated and will be replaced
with
 the
output

from
```

```
 kubectl
version

--short.  Use --output=yaml|json to get the full version.



Client

Version
: version.Info{Major:
"1"
, Minor:
"26"
, GitVersion:
"v1.26.0"
, GitCommit:
"b46a3f887ca979b1a5d14fd39cb1af43e7e5d12d"
, GitTreeState:
"clean"
, BuildDate:
"2022-12-08T19:58:30Z"
, GoVersion:
"go1.19.4"
, Compiler:
"gc"
, Platform:
"linux/amd64"
}

Kustomize
Version
```

```
: v4
.5.7
```

For the setup of the K8S context of the cluster, which is a management services cluster in this case, you will have to issue a few commands like login to the respective cloud/hyperscaler to authenticate and authorize, followed by setting up the kube config context in the local machine. Once this step is done successfully, you can verify using the below commands:

Note: If the kubectl version command is issued, the output will also display the Server version of the K8S Cluster. In the above case, since we have added -- client to the kubectl version common, it only returned the Client version.

This time the output will include server details having version#, git commit#, and build date right below the client details:

```
user@virtualserver01:/
# kubectl version

WARNING: This version information is deprecated and will be replaced
with the output from kubectl version –short.  Use –
output=yaml|json to get the full version.

Client Version: version.Info{Major:
”1”
, Minor:
”26”
```

```
, GitVersion:
”v1.26.0”
, GitCommit:

”b46a3f887ca979b1a5d14fd39cb1af43e7e5d12d”
, GitTreeState:
”clean”
, BuildDate:
”2022-12-08T19:58:30Z”
, GoVersion:
”go1.19.4”
, Compiler:
”gc”
, Platform:
”linux/amd64”
}

Kustomize Version: v4.5.7

Server Version: version.Info{Major:
”1”
, Minor:
”25”
, GitVersion:
”v1.25.5+IKS”
, GitCommit:
”7108fa3bb8b37cd6b31affc82dc45cee3c9493f6”
, GitTreeState:
”clean”
, BuildDate:
”2022-12-09T06:35:21Z”
```

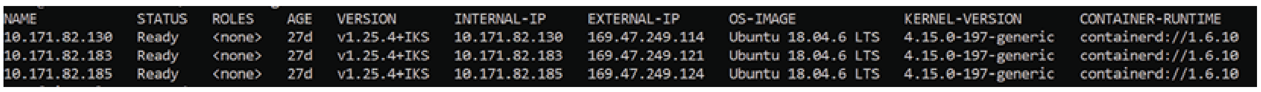
```
, GoVersion:
"go1.19.4"
, Compiler:

"gc"
, Platform:
"linux/amd64"
}

user@virtualserver01:/
#
```

Validation of the Kubernetes cluster

To validate the cluster further and get more information about the cluster below commands can be issued. The output will show all the nodes in the cluster and their corresponding name, status, Age, server version, build, network details, and operating system:

# to get cluster’s node details and network details

```
kubectl get nodes -o wide
```



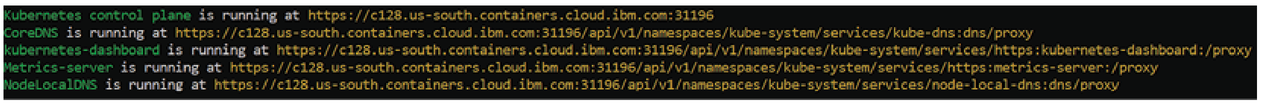
```
NAME           STATUS   ROLES    AGE   VERSION       INTERNAL-IP    EXTERNAL-IP      OS-IMAGE             KERNEL-VERSION       CONTAINER-RUNTIME
10.171.82.130  Ready    <none>   27d   v1.25.4+IKS   10.171.82.130  169.47.249.114   Ubuntu 18.04.6 LTS   4.15.0-197-generic   containerd://1.6.10
10.171.82.183  Ready    <none>   27d   v1.25.4+IKS   10.171.82.183  169.47.249.121   Ubuntu 18.04.6 LTS   4.15.0-197-generic   containerd://1.6.10
10.171.82.185  Ready    <none>   27d   v1.25.4+IKS   10.171.82.185  169.47.249.124   Ubuntu 18.04.6 LTS   4.15.0-197-generic   containerd://1.6.10
```

Figure 2.2: Example output for K8S cluster validation

# to get cluster's various API endpoints like K8S control plane, core dns, metr ics server and k8s-dashboard,

```
kubectl cluster-info
```

The following image shows the cluster information around K8S control plane version, metrics server and K8S dashboard:



```
Kubernetes control plane is running at https://c128.us-south.containers.cloud.ibm.com:31196
CoreDNS is running at https://c128.us-south.containers.cloud.ibm.com:31196/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
kubernetes-dashboard is running at https://c128.us-south.containers.cloud.ibm.com:31196/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy
Metrics-server is running at https://c128.us-south.containers.cloud.ibm.com:31196/api/v1/namespaces/kube-system/services/https:metrics-server:/proxy
NodeLocalDNS is running at https://c128.us-south.containers.cloud.ibm.com:31196/api/v1/namespaces/kube-system/services/node-local-dns:dns/proxy
```

Figure 2.3: Example output for K8S cluster info validation

Kubernetes package manager

Kubernetes package manager is a tool that simplifies the installation process, and management of K8S applications and their dependencies. It also helps to automate the deployment of complex applications on K8S by providing an easy way to install, upgrade, and manage K8S resources such as deployments, services, and configuration files. There are several package manager tools like Helm, Kustomize, Operator Framework, and so on.

We will be using Helm.

Understanding Helm and its need

Helm is a package manager for K8S. It is used to manage K8S resources, such as applications, services, and their dependencies. Helm helps you deploy applications and services to K8S clusters, and it provides a way to package, configure, and manage complex applications in a reproducible manner.

In Helm, a package is called a chart. A chart is a collection of files that describe a related set of K8S resources. It includes all the necessary K8S manifest files and may also include scripts, templates, and other files needed to deploy and manage the application.

To use Helm, you first install it on your local machine, and then you can use it to search for, download, and install charts from repositories. Helm also provides a Command-Line Interface that you can use to manage charts and perform other tasks, such as upgrading and uninstalling charts.

Helm is a powerful tool for managing applications on K8S, and it is widely used in the K8S community. It is an open-source project, and it is available for free under the Apache 2.0 license.

We will use Helm to install and manage some of the pre-requisite components for Knative, like Istio, Argo CD, Prometheus, and so on., in upcoming chapters and exercises.

Helm installation

The Helm installable binary for the desired version and operating system can be downloaded from the https://github.com/helm/helm/releases

Below are the specific instructions used for installing Helm on Ubuntu 20.04.

#download helm for ubuntu 20.04 (linux
 amd64).

```
wget https://get.helm.sh/helm-v3.10.3-linux-amd64.tar.gz
```

#untar/unzip downloaded file

```
tar -zxvf helm-v3.10.3-linux-amd64.tar.gz
```

#move file to the usr/local/bin so can be executed from any path

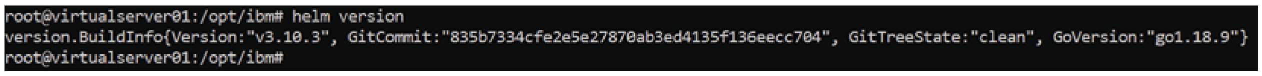
```
sudo mv linux-amd64/helm /usr/
local
/bin/helm
```

Verify Helm installation

helm

version

The following image displays the Helm version information:



```
root@virtualserver01:/opt/ibm# helm version
version.BuildInfo{Version:"v3.10.3", GitCommit:"835b7334cfe2e5e27870ab3ed4135f136eecc704", GitTreeState:"clean", GoVersion:"go1.18.9"}
root@virtualserver01:/opt/ibm#
```

Figure 2.4: Example output for Helm version validation

Istio

Istio is a service mesh that provides advanced traffic management, observability, and security features. It includes its own ingress controller, Istio Gateway, which can be used to route incoming traffic to services in the mesh. In the following sections, we will go through the alternatives and why choose Istio for Knative for gateway.

Why Istio for Knative

Let us start with understanding K8S Ingress. K8S Ingress is a resource that allows external traffic to reach the services in a K8S cluster. It acts as a gateway for incoming traffic, routing it to the appropriate service based on the host and path of the incoming request. Ingress can also be used to provide additional functionality such as authentication, SSL termination, and URL rewriting. Additionally, it can be used to expose multiple services on the same IP address and port, using host and path-based routing to direct traffic to the appropriate service. There are several options for deploying an Ingress gateway for K8S like:

Ingress Controller using nginx, HAProxy

K8S Service Type Loadbalancer

NodePort

ClusterIP

Considering the Knative's requirement for HTTP/2, gRPC, autoscaling, traffic splitting, rate limiting, and canary deployments, it is recommended to use open-source ingress controllers for K8S like

Is a service mesh that provides advanced traffic management, observability, and security features. It includes its own ingress controller, Istio Gateway, which can be used to route incoming traffic to services in the mesh.

Is built on top of Envoy and offers a wide range of functionality for traffic management, including load balancing, rate limiting, and authentication. It also supports service meshes like Istio and can integrate with other service discovery and configuration management tools like Consul and ConfigMap.

Is a Kubernetes-native API Gateway built on top of the Envoy proxy. It is designed to be easy to use and provides features such as authentication, rate limiting, and canary deployments.

Is a Kubernetes-native ingress controller that is built on top of the Envoy proxy. It is designed to be simple and lightweight, and focuses on providing a consistent configuration for routing and load balancing in K8S.

Is an ingress controller built on top of the Envoy proxy. It is focused on providing a simple, consistent configuration for routing and load balancing in K8S.

All of them are powerful Ingress controllers but the choice may depend on the specific requirements of the user, for example if the user wants a simple and lightweight solution Kourier will be a good fit, if the user needs advanced features like traffic management, observability, and security Istio will be the best fit. In this chapter we will install Istio as Ingress Controller for Knative considering its most widely used in production K8S environment.

[Istio installation]

To install Istio on K8S cluster using Helm, you will need to have Helm installed on your local machine which is already covered in earlier sections of this chapter. Here are the specific steps and instructions used for installing Istio:

#Add the Istio Helm repository to your local Helm installation by running the following command

```
helm repo add istio https://istio-release.storage.googleapis.com/charts
```

#Update your local Helm chart repository by running the following comm and

```
helm repo update
```

#Create a namespace for Istio components in your Kubernetes cluster:

```
kubectl create namespace istio-system
```

#Install the Istio base chart which contains cluster-wide resources used by the Istio control plane:

```
helm install istio-base istio/base -n istio-system
```

#Install the Istio discovery chart which deploys the istiod service:

```
helm install istiod istio/istiod -n istio-system --
wait
```

#Install an ingress gateway:

```
kubectl create namespace istio-ingress
```

```
kubectl label namespace istio-ingress istio-injection=enabled
```

```
helm install istio-ingress istio/gateway -n istio-ingress --
wait
```

Verify Istio installation

The installation of Istio can be verified through below steps:

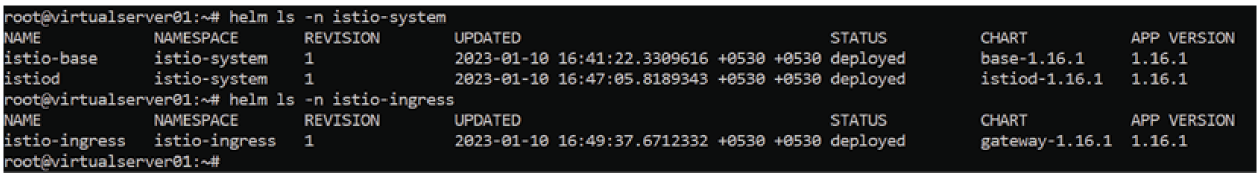
#verify istio-base and istiod system

```
helm ls -n istio-system
```

#verify istio-gateway

```
helm ls -n istio-ingress
```

The following image displays the successful installation of istio package through helm.



```
root@virtualserver01:~# helm ls -n istio-system
NAME            NAMESPACE       REVISION        UPDATED                                 STATUS          CHART           APP VERSION
istio-base      istio-system    1               2023-01-10 16:41:22.3309616 +0530 +0530 deployed        base-1.16.1     1.16.1
istiod          istio-system    1               2023-01-10 16:47:05.8189343 +0530 +0530 deployed        istiod-1.16.1   1.16.1
root@virtualserver01:~# helm ls -n istio-ingress
NAME            NAMESPACE       REVISION        UPDATED                                 STATUS          CHART           APP VERSION
istio-ingress   istio-ingress   1               2023-01-10 16:49:37.6712332 +0530 +0530 deployed        gateway-1.16.1  1.16.1
root@virtualserver01:~#
```

Figure 2.5: Example output for Istio validation

The output of above commands will show the Helm resource their status, chart version and app version. In the above case the, the app Istio base and gateway of version 1.16.1 are successfully deployed.

So, now we have K8S cluster running with Istio gateway. We also have installed command line tools - kubectl and Helm. Let us now dive deep to understand the Knative components and their installation.

Tip: You can use Open K8S Integrated Development Environment (IDE) like lens desktop and K8S Web Dashboard to better manage and gain visibility to your cluster.

Knative installation

The installation process for Knative can be complex, depending on the configuration and requirements of your environment. It is important to follow the installation instructions carefully and ensure that all prerequisites are met before installing Knative.

Explore and understand various approaches to installing Knative

There are below ways in which Knative can be installed. Let us go through each of them in brief:

This is a simplified installation process that uses pre-configured YAML files and Helm charts to set up a basic Knative environment on a K8S cluster. This method is intended for users who are new to Knative and want to get started quickly, without having to manually configure each component. It can be used to set up a development or test environment but may not be suitable for production use due to the lack of customization options. The environment set up from quickstart is for experimental use only.

YAML based YAML-based installation of Knative is a method for installing and configuring the various components of the Knative platform on a K8S cluster using YAML files. These files define the desired state of the cluster, including the configuration of the Knative components such as the Knative Serving, Eventing and Serving operators. The files can be modified to customize the installation, such as to change the resources allocated to a component or to enable or disable specific features.

Operator based This method of installing and managing the Knative components using K8S operators. Operators are a method of packaging, deploying, and managing a K8S application. They provide a way to automate the management of the application's lifecycle, including scaling, upgrades, and rollbacks. In the case of Knative, the operator is responsible for installing and managing the various Knative components, such as the Serving and Eventing components. This method of installation allows for more fine-grained control and customization of the Knative installation.

Vendor managed Knative

Red Hat OpenShift This is a fully managed Knative service that runs on Red Hat's OpenShift platform and allows developers to build and deploy containerized applications without having to worry about infrastructure.

Google Cloud Run for This is a fully managed Knative service that allows developers to deploy and run containerized applications on Google Cloud without having to worry about infrastructure.

IBM Cloud This is a fully managed Knative service that allows developers to build and deploy containerized applications on IBM Cloud without having to worry about infrastructure.

For this book we are going to use YAML based installation for Knative components. Refer to the Chapter Serverless and Knative in a Nutshell to read more about Knative components.

Install Knative

Knative has two components – Serving and Eventing which have been discussed in detail Chapter Serverless and Knative in a Here, we will be focusing on installation of the same. Both components can be installed by using YAML based approach which has been discussed as one of the approaches for installation in earlier section.

Install Knative Serving with YAML

To install the Knative Serving component, please follow below steps:

Execute the following Custom Resource Definitions and Custom Resources for Knative serving:

#Install custom resources definition for knative serving

```
kubectl apply -f https://raw.githubusercontent.com/serverless-apps-with-knative/code-recipe/main/chapter-2/install-knative/serving/1-serving-crds.yaml
```

#Install the core components of knative serving

```
kubectl apply -f https://raw.githubusercontent.com/serverless-apps-with-knative/code-recipe/main/chapter-2/install-knative/serving/2-serving-core.yaml
```

The successful execution of below commands can be verified using following command:

```
kubectl get pods -n knative-serving
```

The following output will show Knative Serving components (activator, autoscaler, controller, domain-mapping, domain mapping webhook and webhook) in running one healthy pod for each:

```
$
kubectl
get
pods
-n
knative-serving



NAME
READY
STATUS
RESTARTS
AGE

activator-7cbbfbc85-glcqp
1
/1
Running
0
3m29s

autoscaler-8f986cff-ph4n8
1
/1
Running
0
3m27s
```

```
controller-58dfb45d74-b8fp2
1
/1
Running
0
3m25s

domain-mapping-5d8db49bf6-cz297
1
/1
Running
0
3m23s


domainmapping-webhook-584476fd67-mtrjj
1
/1
Running
0
3m22s

webhook-6d5c55fd8c-29q4m
1
/1
Running
0
3m18s

$
```

Since Istio is already setup, in earlier section of the chapter and same is being used as networking layer so no need to install the same again. Please proceed with step 3.

Configure Istio to create a gateway and also map it to Knative service for routing following below steps:

```
#apply net-isito to configure istio gateway and map knative
 service

kubectl apply -f https://raw.githubusercontent.com/serverless-apps-with-knative/code-recipe/main/chapter-2/install-knative/serving/3-net-istio.yaml
```

Configure Magic DNS by leveraging Knative Job configures Knative to use sslip.io as a default DNS suffix.

```
#apply magic dns sslip.io

kubectl apply -f https://raw.githubusercontent.com/serverless-apps-with-knative/code-recipe/main/chapter-2/install-knative/serving/4-serving-default-domain.yaml
```

The successful execution of preceding commands can be confirmed through the following verification:

```
kubectl get pods -n knative-serving
```

The following output will show two additional pods, net-istio-controller, and net-sitio-webhook, in running and healthy state

Fetch the External IP address or CNAME by running the following command. The information will be used later to verify installed application or access the gateway.

#Fetch the External IP address or CNAME

```
kubectl get svc -n istio-ingress
```

Configure Append External IP or CNAME with add it to ConfigMap

Given, 1.2.3.4 is the External IP, you can edit the resource using following commands:

#Edit the CM to add DNS (external-ip.sslip.io)

```
kubectl edit
cm
config-domain -n knative-serving
```

Change content to: Notice the last line in following example content:

```
apiVersion:


v1
```

```
kind:

ConfigMap

metadata:

name:

config-domain

namespace:

knative-serving

data:

# sslip.io is a "magic" DNS provider, which resolves all DNS lookups for:

# *.{ip}.sslip.io to {ip}.

1.2.3.4 .sslip.io:

""
```

Install Knative Eventing with YAML

To install the Knative Eventing component, please follow below steps:

Execute the following Custom Resource Definitions and Custom Resources for Knative Eventing:

#Install custom resources definition for Knative Eventing

```
kubectl apply -f https://raw.githubusercontent.com/serverless-apps-with-knative/code-recipe/main/chapter-2/install-knative/eventing/1-eventing-crds.yaml
```

#Install the core components of Knative serving

```
kubectl apply -f https://raw.githubusercontent.com/serverless-apps-with-knative/code-recipe/main/chapter-2/install-knative/eventing/2-eventing-core.yaml
```

The successful execution of step 1 can be verified using following command:

```
kubectl get pods -n knative-eventing
```

You will notice 2 new pods getting added: eventing-controller and eventing webhook in running and healthy state.

Example

```
user@virtualserver01:~$
kubectl
get
pods
-n
knative-eventing

NAME

READY
STATUS
RESTARTS
AGE

eventing-controller-7b95f495bf-5lpdq
1
/1
Running
2
(7d14h
ago)
7d14h

eventing-webhook-8db49d6cc-d6cvj
1
/1
Running
```

```
2
(7d14h
ago)
7d14hs
```

Install an in-memory implementation of Channel by running the command:

#Install in-memory implementation of channel

```
kubectl apply -f https://raw.githubusercontent.com/serverless-apps-with-knative/code-recipe/main/chapter-2/install-knative/eventing/3-in-memory-channel.yaml
```

Install this implementation of Broker by running the following command:

#Install broker implementation

```
kubectl apply -f https://raw.githubusercontent.com/serverless-apps-with-knative/code-recipe/main/chapter-2/install-knative/eventing/4-mt-channel-broker.yaml
```

You will notice 5 new pods getting added: mt-broker-ingress in running and healthy state.

Example Output:

```
user@virtualserver01:~$
```

```
kubectl
get
pods
-n
knative-eventing

NAME
READY
STATUS
RESTARTS
AGE

eventing-controller-7b95f495bf-5lpdq
1
/1
Running
2
(7d14h
ago)
7d14h

eventing-webhook-8db49d6cc-d6cvj
1
/1

Running
2
(7d14h
ago)
7d14h
```

```
imc-controller-bf8c4d6f5-j8ndf
1
/1
Running
0
6d21h

imc-dispatcher-5688655bb-2f6ft
1
/1
Running
0
6d21h

mt-broker-controller-98f7b598b-l8wns
1
/1
Running
0
6d22h

mt-broker-filter-58c89f8487-p8r7l
1
/1
Running
0
6d22h



mt-broker-ingress-69bd8fccf6-sbvqs
1
```

```
/1
Running
0
6d22h
```

Install Knative CLI

For validation and future ease of managing upcoming Knative resources, one should install Knative Knative CLI is a Command-Line Interface tool that allows developers to interact with and manage the Knative resources on a K8S cluster. It provides a set of commands to deploy, configure, and scale Knative services, as well as to view logs and status information. Knative CLI is built on top of the K8S CLI (kubectl) and uses the same configuration and authentication mechanisms. The following are the steps to install Knative CLI on Ubuntu 20.04:

```
wget https://github.com/knative/client/releases/download/knative-
v1.8.1/kn-linux-amd64

mv kn-linux-amd64 kn

chmod +x kn

mv kn /usr/
local
/bin
# use sudo if required.

kn version
```

Validate Knative installation

In this section, we will verify the installation of Knative Serving and Eventing by quickly deploying a pre-built application.

Let us start with Knative Serving validation first. The app is pre-built, and image is available at docker.io/knativedemo/helloworld-java-spring

To Deploy the app, follow either YAML or Knative approach

Follow the steps for deploy using YAML:

Create a new file service.yaml file with below content:

```
apiVersion:
serving.knative.dev/v1

kind:
Service

metadata:

name:
helloworld-java-spring

namespace:
```

```
default

spec:

template:

spec:

containers:

-
image:
docker.io/knativedemo/helloworld-java-spring

env:

-
name:

TARGET

value:
"Hello Knative
Book Sample v1"
```

Apply the preceding YAML file

```
kubectl apply -f service.yaml
```

OR

Follow the steps for deploy using

```
kn service create helloworld-java-spring --
image=docker.io/knativedemo/helloworld-java-spring --env TARGET=
"Hello Knative Book Sample v1"
```

This will wait until your service is deployed and ready, and ultimately it will print the URL through which you can access the service.

To verify the app, follow below steps:

Find the domain URL for your application using following command:

```
kubectl get ksvc helloworld-java-spring  --output=custom-
columns=NAME:.metadata.name,URL:.status.url
```

Example

```
book@
LAPTOP-
780
KFQ65:~$ kubectl
get

 ksvc helloworld-java-spring  --output=custom-
columns=NAME:.metadata.name,URL:.status.url
```

```
NAME                    URL

helloworld-java-spring   http://helloworld-java-spring.knative-
serving.169.48.166.178.sslip.io

book@
LAPTOP-
780
KFQ65:~$
```

Make the request to above URL which is our case wither in browser or through curl. You should get following output:

Example Output:

Hello Knative Book Sample v1!

OR

Follow the steps to find domain url and verify using

```
curl $(kn service
describe
 helloworld-
java
-spring -o
url
)
```

Example Output:

```
Hello Knative Book Sample v1!
```

Let us start with Knative Eventing validation now. The app is pre-built and image is available at docker.io/knativedemo/helloworld-python

To Deploy the app, follow the steps:

Apply the following YAML file creating the deployment, svc and triggers:

```
kubectl apply -f https://raw.githubusercontent.com/serverless-apps-with-knative/code-recipe/main/chapter-2/verify-knative/eventing/sample-app.yaml
```

Verify the using following commands:

```
kubectl --namespace knative-samples get deployments helloworld-python
```

```
kubectl --namespace knative-samples get svc helloworld-python
```

The pods should be up and running in healthy state.

Verify if Knative Eventing Trigger is Ready state. Make sure that Ready=true

```
kubectl -n knative-samples get trigger helloworld-python
```

```
kubectl -n knative-samples get trigger helloworld-python
```

Example output: Ready is true in the below output:

```
book@

LAPTOP-
780
KFQ65:~$ kubectl -n knative-samples
get
 trigger helloworld-python

NAME              BROKER    SUBSCRIBER_URI                          AGE
 READY   REASON

helloworld-python
default
   http:
//helloworld-python.knative-samples.svc.cluster.local   7d2h   True

book@
LAPTOP-
780
KFQ65:~$
```

Now, you have deployed the application, and have verified that the namespace, sample application and trigger are ready, you can send a CloudEvent.

Send and verify the Cloud Events, follow below steps:

Deploy a curl pod and SSH into it:

```
kubectl --namespace knative-samples run curl --
image=radial/busyboxplus:curl -it
```

Run the following command in the SSH terminal:

```
curl -v

"broker-ingress.knative-eventing.svc.cluster.local/native-samples/default"
 \

-X POST \

-H
"Ce-Id: 536808d3-88be-4077-9d7a-a3f162705f79"
 \

-H
"Ce-specversion: 0.3"
 \

-H
"Ce-Type: dev.knative.samples.helloworld"
 \
-H
“Ce-Source: dev.knative.samples/helloworldsource”
 \


-H
“Content-Type: application/json”
 \


-d
‘{“msg”:”Hello World from the curl pod.”}’
```

```
Exit
```

Verify pod logs if event is received by app using following:

```
kubectl --namespace knative-samples logs -l app=helloworld-python --
tail=50
```

Example output:

```
[2023-01-11
06


:55:16,016]
WARNING in helloworld:
b'{"msg":"Hello
World
from
the
curl
pod."}'

172.30.226.233
-
-
[11/Jan/2023
06
:55:16]
"POST / HTTP/1.1"
200
-

[2023-01-11
07
:02:02,585]
WARNING in helloworld:
b'{"msg":"Hello
World
from
the
curl
pod."}'

172.30.226.233
-

-
[11/Jan/2023
07
:02:02]
"POST / HTTP/1.1"
```

200

-

Conclusion

In this chapter, we have covered the installation and configuration of various Knative's pre-requisites software and its components. We have installed kubectl CLI, K8S Cluster, Helm, Istio, kn CLI, Knative Serving, Knative Eventing. We also understood their significance and various ways of installation. Then we ran sample pre-built apps to verify Serving and Eventing components.

In next chapter, we will take up the installation of the next set of pre-requisite tools which will help in implementing GitOps and Observability for Knative.

Multiple choice questions

Which statement is true for kubectl?

kubectl is CLI to run and manage K8S clusters.

kubectl is a python library to having util functions for K8S.

kubectl CLI only works in MAC OS.

kubectl helps in controlling the ingress traffic to cluster.

Which of them cannot be ingress traffic controller for Knative having envoy proxy?

Contour

Nginx

Ambassador

Gloo

Istio

Kourier

Helm is __________ ?

a CLI to manage to K8S cluster.

a Package Manager for K8S

a monitoring agent for K8S

Answers

a

b

b

Key terms

Kubernetes CRD (Custom Resource Definition) is an extension mechanism that enables users to define custom resource types in Kubernetes, while CR (Custom Resource) refers to an instance of that custom resource type created based on the defined CRD.

Kubernetes Package commonly known as is a popular open-source tool that simplifies the deployment, management, and versioning of applications and services on Kubernetes by providing a convenient way to package, distribute, and install pre-configured sets of Kubernetes resources called "charts."

C

HAPTER

3

Installation and Configuration – Part II

Introduction

In previous chapter, we learnt about various tools like Knative CLI, Knative Serving, Knative Eventing and so on. These tools help us to get started with Knative. In this chapter we will discuss and provide installation of various monitoring and observability tools. We will also discuss about significance of these tools in Knative.

Structure

In this chapter, we will discuss the following topics:

Broker

Explore and understand various broker options for Knative.

Installation options for Kafka broker

Recipes

Install Kafka

Validation of Kafka cluster

GitOps with Argo CD

Understanding GitOps and its need

Understanding Argo CD

Recipes

Install Argo CD

Objective

By the end of this chapter, you should be able to understand, setup, and install Messaging tool like kafka broker, GitOps tools like Argo CD, and Observability stack.

Broker

A message broker provides a mechanism for asynchronous communication, where the sender and receiver do not need to be active at the same time. The sender publishes messages to a designated destination, and the receiver subscribes to that destination to receive the messages. This decoupling of sender and receiver allows for greater flexibility, scalability, and fault tolerance in distributed systems. It is a software system or middleware that facilitates communication and coordination between distributed applications or services.

Brokers for Knative and its need

Brokers are a pivotal component in Knative because they provide a mechanism for decoupling event sources from event consumers. Brokers act as intermediaries that receive events from event sources and forward them to interested event consumers. This allows greater flexibility and scalability in event-driven systems, as event sources and consumers can be added or removed without impacting other components of the system. Additionally, brokers in Knative provide features such as filtering, batching, and deduplication of events, which can help simplify application logic and reduce the amount of code that developers need to write.

Following are the options to implement the brokers for Knative:

In-memory A simple broker that is included with Knative and uses memory to store events. Refer Install Knative Eventing section in Chapter Installation and Configuration of Knative to understand installation steps of In-memory broker.

NATS A broker that uses NATS streaming as a backend for storing events.

Kafka A broker that uses Apache Kafka as a backend for storing events.

With manage services on the public cloud, one can use the following brokers

Amazon Simple Notification Service (SNS) A broker that uses Amazon SNS as a backend for storing events.

Azure Event Grid A broker that uses Azure Event Grid as a backend for storing events.

Google Cloud Pub/Sub A broker that uses Google Cloud Pub/Sub as a backend for storing events.

IBM Event An Apache Kafka based managed service broker from IBM.

Among the preceding options for implementing the Broker, the one which best suits for production workload for Knative is Kafka considering it is open source has no cloud vendor lock-in, is scalable, and can be spun up as part of the K8S Cluster itself. In the following section, we will learn the ways the Kafka Cluster can be spun up in K8S Cluster.

Installation options for Kafka broker

There are broadly three approaches in which Kafka broker can be installed in K8S Cluster, which are described in the following sections. The choice of installation method may depend on factors such as ease of deployment, customization options, and support requirements.

Using individual YAML/Manifest It involves installing the Kafka components like Apache Kafka broker, Zookeeper, and so on., using individual YAML files. It also involves customizing and configuring these files manually, which makes managing and administration a very complex affair compared to the other options.

Helm Helm charts are K8S package managers, and we have already read in detail in Chapter Installation and Configuration of Knative while installing Istio. There are several helm charts that are available to deploy and manage Kafka Clusters in K8S. One of the examples is from the Bitnami helm chart described in the following location https://bitnami.com/stack/kafka/helm

Operator Operator framework allows a seamless way to manage the lifecycle of software packages in K8S. The following are the two options for Operator-based installation:

open-source project that provides a way to run Apache Kafka on K8S. Strimzi is a K8S operator that can be installed on a K8S cluster to manage

Kafka clusters and topics.

Confluent K8S operator provided by which is a company that offers a commercial version of Kafka. The Confluent operator can be installed on a K8S cluster to manage Kafka clusters and other related Confluent components.

We will be using Operator framework-based installation using Strimzi in the following section considering the ease of the deployment and management of Kafka Cluster on K8S.

Installation and validation of Kafka

Kafka Cluster can be created using following instructions:

Before deploying Strimzi Cluster Operator, we will first create K8S namespace using following command:

#create namespace

```
kubectl create namespace kafka
```

Now, use following command to deploy Strimzi Cluster Operator. This command will apply Strimzi Custom Resource Definitions which define schemas used for custom resources like Kafka, Kafka Topic and so on. This will also download and install YAML files for ClusterRoles and ClusterRoleBindings in Kafka namespace:

#create CRDs, cluster role binding and cluster roles.

You can follow the below commands to verify logs if Strimizi operator was installed successfully:

#verify if pods are up and running

```
kubectl get pod -n kafka –watch
```

#verify logs if operator / CRDs installed

```
kubectl logs deployment/strimzi-cluster-operator -n kafka -f
```

Once the operator is installed, and pods are in a healthy state, it will watch or monitor for new custom resources and create the Kafka cluster, topics, or users that correspond to those Custom Resources

Create an Apache Kafka cluster with one node for Apache Zookeeper and one node for Apache Kafka:

# Apply the `Kafka` Cluster CR file

```
kubectl apply -f https://strimzi.io/examples/latest/kafka/kafka-persistent-single.yaml -n kafka
```

Validate pods are ready

#validate if pods ready

```
kubectl
wait
 kafka/my-cluster
–
for
=condition=Ready –timeout=300s -n kafka
```

Validate setup/installation by sending and receiving the messages:

#while cluster is running, try sending the message through simple produce
r

Validate if the message was received in a different terminal:

```
#while cluster is running, try receiving the message through simple consu
mer in another terminal

kubectl -n kafka run kafka-consumer -ti --
image=quay.io/strimzi/kafka:0.33.1-kafka-3.3.2 --rm=
true
 --restart=Never -- bin/kafka-console-consumer.sh --bootstrap-server my-
cluster-kafka-bootstrap:9092 --topic my-topic --from-beginning
```

Your Kafka Cluster is ready, and you are ready to cruise further with Knative.

GitOps with Argo CD

GitOps is a software development approach that uses Git as a single source of truth for both application code and infrastructure configuration. Argo CD is one of the open source tools implementing the same.

Understanding GitOps and its need

In a GitOps workflow, all changes to infrastructure and application code are committed to a Git repository, and a tool such as Argo CD is used to automatically apply those changes to the running environment. This approach provides a standardized, auditable, and version-controlled way to manage deployments while reducing the risk of human error and enabling more frequent and reliable updates.

Argo CD is an open-source GitOps tool that automates the deployment of applications to K8S clusters. It uses a Git repository to manage the desired state of the K8S resources and continuously monitors the actual state of the resources. When changes are detected in the Git repository, Argo CD automatically synchronizes the desired state with the actual state of the resources in the Cluster.

Argo CD can be used effectively with Knative to help manage and deploy serverless applications. The following are the reasons why Argo CD is a good fit for Knative:

Simplified Argo CD simplifies the deployment process by automating the rollout of new code changes to production, reducing the risk of human error and streamlining the deployment process.

Multi-cluster Argo CD is designed to manage deployments across multiple K8S Clusters, which can be useful for larger-scale or multi-region deployments.

GitOps Argo CD uses a GitOps workflow, which means that application deployments are triggered by commits to a Git repository, making it easier to manage and audit changes to the deployment process.

Application Argo CD provides a clear view of the deployed applications and their configuration, making it easier to manage and troubleshoot issues that arise during the deployment process.

Integration with other Argo CD can be easily integrated with other K8S tools, such as Prometheus and Grafana, to provide additional monitoring and observability features.

By using Argo CD with Knative, teams can benefit from a simplified, automated deployment process that reduces the risk of errors and improves visibility into the deployed applications. This can help teams to move more quickly, with greater confidence in their ability to manage and deploy serverless applications at scale.

Installation of Argo CD

Create a namespace for Argo CD

#create a namespace argocd

```
kubectl create namespace argocd
```

Apply the following manifest file for Argo

#Apply manifest file having argo resources

```
kubectl apply -n argocd -
f https://raw.githubusercontent.com/argoproj/argo-
cd/stable/manifests/install.yaml
```

Verify if Pods are running fine

#Verify if pods are running fine

```
kubectl get pods -n argocd -w
```

Sample output

```
argocd-application-controller-0
```

```
1
/1

Running



0


31s


argocd-applicationset-controller-7cdf6fc77b-mgkhw


1
/1

Running


0


42s


argocd-dex-server-7d687968db-9qp8l


1
/1

Running


0
```

```
41s

argocd-notifications-controller-79dccb6f68-cdtjv

1
/1

Running

0

39s

argocd-redis-74c8c9c8c6-tbg2p

1
/1

Running

0

38s

argocd-repo-server-98d6d47cc-pjwnb

1
/1

Running
```

```
0

36s

argocd-server-567b595777-8smsd


1
/1

Running

0

33s
```

Once all pods are in a running state and use the following commands to expose the argo-cd You can optionally use port-forward as well

```
kubectl
 patch svc argocd-server -n argocd -p ‹{
"spec"
: {
"type"
:
"LoadBalancer"
}}'
```

OR

```
kubectl port-forward svc/argocd-server -n argocd 8080:443
```

Access the UI at loadbalancer or localhost based on the approach you used in the preceding steps. Use the following password to access the protected Argo CD admin UI console.

```
kubectl -n argocd get secret argocd-initial-admin-secret -
o jsonpath=”{.data.password}” | base64 -d;
echo
```

Observability

Observability in software refers to the ability to gain insights and understand the internal state and behavior of a software system during runtime. It encompasses the collection, analysis, and visualization of various metrics, logs, traces, and other telemetry data from a software system to gain visibility into its performance, reliability, and behavior.

Understanding observability and its significance in Knative

Observability is an important consideration for Knative because it provides a way to monitor and understand the behavior of Knative applications and the underlying infrastructure. Knative applications are typically composed of multiple components, such as event sources, brokers, and services, which can make it difficult to debug and analyze the issues or understand the performance of the whole system.

Understanding the Knative observability stack

As shown in Figure Knative typically includes tools for collecting, storing, and analyzing metrics, logs, and traces from Knative applications and the K8S infrastructure. Some common components of an observability stack for Knative include:

Metrics collection and This involves collecting and storing metrics from Knative components, such as the number of requests and response times for services. Common tools for this include Prometheus and Grafana.

Log collection and This involves collecting logs from Knative components and analyzing them to identify issues or troubleshoot problems. Common tools for this include Elasticsearch and Kibana.

Tracing and distributed involves tracing requests as they flow through a Knative application to understand the end-to-end performance and identify issues. Common tools for this include Jaeger and Zipkin.

By using a good observability stack, developers and operators can gain insights into the behavior of their Knative applications and make informed decisions about how to optimize performance, diagnose issues, and improve the overall user experience:

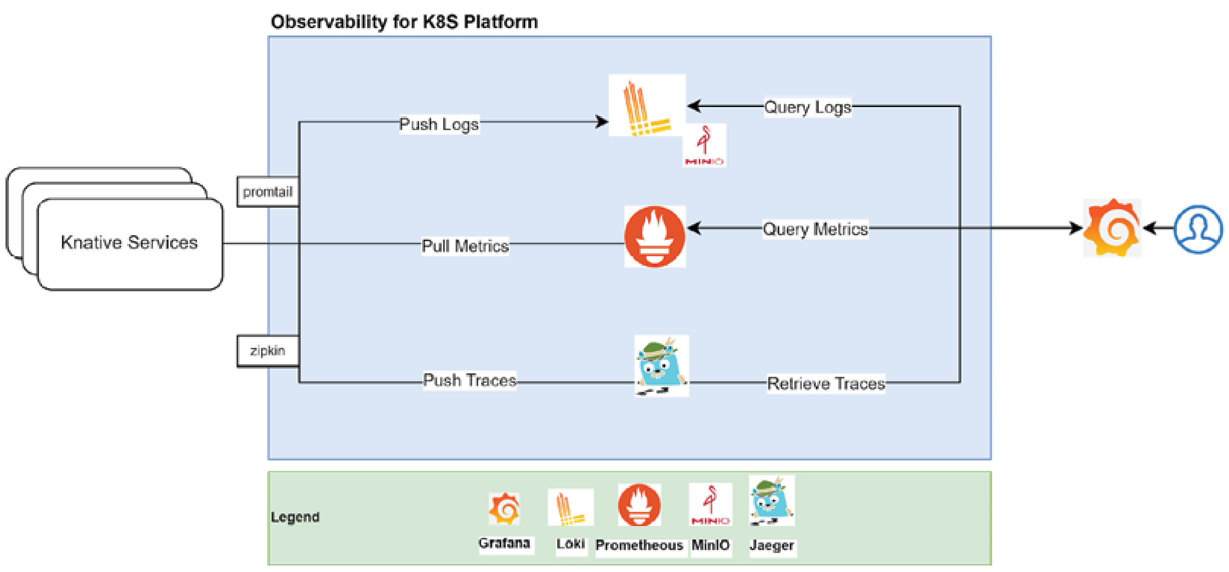


Figure 3.1: Observability stack for Knative

Now, let's discuss and understand some of Observability tools for metrics, log collection and tracing, we will also provide steps of installing these tools in next section.

Loki

Loki is a horizontally-scalable, highly-available, multi-tenant log aggregation system. Loki can be used to store and query logs from the applications as well as infrastructure. Loki can be installed on the K8S Cluster by following the installation guide provided in the official Loki documentation, which we will follow in the next section. Post installation, it requires configuration for log storage and query. It provides a range of configuration options that you can use to specify how logs are to be stored, indexed, and queried.

Installing Loki

To install Loki on the K8S Cluster, we will be using Helm charts. To proceed further, one should have a helm installed on their local machine, which is already covered in the earlier Chapter Installation and Configuration of Knative.

The Grafana helm charts present several alternatives for Loki to choose from. In this case, we will go with installing grafana/loki charts. It is always a good practice to keep the separate environment of our logging stack as this gives more control over the permission and helps in keeping resource consumption in check. Let us start by downloading charts and creating new namespace:

```
# Add grafana helm charts

helm repo add grafana https://grafana.github.io/helm-charts

#update helm charts using the following code

helm repo update

# create a new namespace for loki

kubectl create ns loki
```

#to install Loki

```
helm upgrade --install --namespace loki logging grafana/loki -
f values.yml --
set

 loki.auth_enabled=
false
```

Prometheus

Prometheus is a popular open-source monitoring tool that can collect metrics from your applications and infrastructure. Once you have installed Prometheus, you need to instrument your applications to expose the relevant metrics that you want to monitor. Prometheus provides a range of client libraries that you can use to instrument your code, as well as an API that you can use to expose custom metrics.

Installing Prometheus

#Create a namespace for Prometheus

```
kubectl create ns prometheus
```

#Add charts repository for Prometheus

```
helm install prom-logging prometheus-community/prometheus -
n prometheus
```

Jaeger

It is an open-source, end-to-end distributed tracing system that is designed to help developers and DevOps teams to troubleshoot and optimize complex distributed systems. When deployed in a K8S Cluster, Jaeger provides a powerful observability stack that can help you to monitor, analyze and troubleshoot the behavior of your microservices.

[Installing Jaeger]

```
# Jaeger mainfests are being installed in istio-system namespace.

kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.17/samples/addons/jaeger.yaml -n istio-system
```

Grafana

Grafana is a popular open-source data visualization tool that can be used to visualize metrics and logs from your applications and infrastructure. It provides visualization options, including dashboards, alerts, and plugins.

To fully leverage the capabilities of the monitoring stack, you need to integrate Grafana with Prometheus, Loki, and Jaeger. Grafana provides a range of data sources that you can use to connect to these systems, and you can use the built-in query language to create custom visualizations and alerts.

Installing Grafana

```
#Install Grafana using below downloaded charts if not done in prior step

helm repo add grafana https://grafana.github.io/helm-charts

#To install Grafana from the charts

helm upgrade --install --namespace=loki loki-grafana grafana/grafana
```

This completes the understanding and installation of the tools required to enable observability for Knative.

Conclusion

In this chapter, we have covered the rest of the installation and configuration of various Knative's pre-requisites software and its components. We have installed Kafka broker, Argo CD for GitOps, and Observability stack. We also understood their significance and various ways of installation.

In the next chapter, we will take a deep dive into the Knative functions, their significance, and how to implement them.

Multiple choice questions

ArgoCD is a tool to implement:

DevOps in GitOps way

Control Plane in Kubernetes

Cloud agnostic storage

None of these

Which of the following combination is not the correct one from observability perspective?

Logs - Loki

Metrics - Prometheus

Traces - Prometheus

Traces - Jaeger

Answers

a

c

Key terms

The Operator Framework in Kubernetes is a collection of tools and best practices for building, deploying, and managing complex applications as Kubernetes Operators, which are custom controllers that automate the management of applications and services on Kubernetes.

GitOps is a DevOps approach in which the desired state of a system's infrastructure and applications is declared in Git repositories, and changes are automatically applied to the live environment through continuous deployment pipeline.

C
HAPTER
4

Knative Functions – An Overview

Introduction

In Chapter Serverless and Knative in a nutshell, we learned about the Knative project on a high level. We also learned Knative installation and related components in Chapter Installation and Configuration of Knative and Chapter Setting Up DevSecOps and Observability.

In this chapter, we will deep dive into Knative We will learn how to create, build, deploy, and run Knative Functions using func CLI using hands-on code recipes.

Structure

In this chapter, we will discuss the following topics:

Introduction to Knative Functions

Installing Knative Function

Creating function

Building and deploying the function

Objectives

At the end of this chapter, you should have learned all about Knative Functions. You should be able to create, build and deploy Knative Functions on local as well as remote K8S Cluster using Knative's func CLI.

Knative Functions

In Chapter Serverless and Knative in a nutshell, we learnt about Function-as-a-Service and its benefits. Hyperscalers provide managed FaaS service to developers to build Serverless Functions. Knative provides an alternative to managed FaaS service from the Hyperscalars, which is Knative Functions. Knative provides an easy way to create, build and deploy self-managed Functions on K8S cluster. For this, there is no need for developers to have in-depth knowledge of Kubernetes, containers, and Knative as well. And the best part is these Functions can be executed/tested on a local machine without need of K8S or Knative running.

To create and manage Function workflows, the func CLI or kn func plugin needs to be installed. Upon building a Function, an Open Container Initiative image is generated and pushed to the container registry of choice.

Knative provides a functions template that helps in generating a boilerplate Function project. This helps developer to focus on writing business logic and increases productivity. Function templates are available in the following languages

Node.js

Python

Go

Quarkus

Rust

Spring Boot

TypeScript

In this chapter, we will use the Spring Boot template to create a simple Function. We will run this Function also, we will deploy the same on the K8S Cluster.

Installing Knative functions

Knative Functions can be installed on the local machine in two ways:

Install func CLI

Install the kn func plugin

Both work in a very similar fashion and are pretty straightforward to install. Follow the instructions on https://knative.dev/docs/functions/install-func/ to install any of them on a local machine as per the Operating System. We will use func CLI in this chapter.

func CLI installation can be verified using the following command, which should give an installed version of the func CLI:

Let us go ahead and create a Function.

Creating function

Let us create a function using func CLI. Use the func create command to generate a boilerplate project of a Spring Boot-based Knative Function:

Let us now look at the arguments of the above command. -l denotes the language to be used for the generated Function as the first argument; it could be any of the languages from the list of supported languages mentioned in the previous section. The second argument is the name of the Function. After executing the above command, you should see the output:

The is an absolute path where Function project code is generated, which is a Maven project. Now, let us take a closer look at the generated code starting with pom.xml as shown in the following snippet. It uses Spring Boot version 3.0.2 and Spring Cloud version 2022.0.0:

```
<
project

xmlns
=
"http://maven.apache.org/POM/4.0.0"
```

```
xmlns:xsi
=

"http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation
=
"http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven
-4.0.0.xsd"
>

  <
modelVersion
>4.0.0
modelVersion
>

  <
parent
>

    <
groupId
>org.springframework.boot
groupId
>

    <
artifactId
>spring-boot-starter-parent
```

```
artifactId
>

    <
version
>3.0.2
version
>

    <
relativePath
/>

parent
>

  <
groupId
>com.example.events
groupId
>

  <
artifactId
>function
artifactId
>
```

```
  <
version
>0.0.1-SNAPSHOT
version

>

  <
name
>Spring Cloud Function::Http Example
name
>

  <
description
>A Spring Cloud Function, Http Example
description
>

  <
properties
>

    <
java.version
>17
java.version
>

    <
```

spring-cloud.version
>2022.0.0
spring-cloud.version
>

properties
>

  <
dependencyManagement
>

    <
dependencies
>

      <
dependency
>

        <
groupId
>org.springframework.cloud
groupId
>

        <
artifactId

>spring-cloud-dependencies
artifactId
>

<
version

>${spring-cloud.version}
version
>

<
type
>pom
type
>

<
scope
>import
scope
>

dependency
>

dependencies
>

```
dependencyManagement
>
```

The func CLI function template generates the Spring Boot main class with a function which simply prints the request headers and payload (if getting passed). This is just a boilerplate example of which func CLI function template generates, which can be modified as per the actual business requirements.

```
package
 functions;

import
 org.springframework.boot.SpringApplication;

import
 org.springframework.boot.autoconfigure.SpringBootApplication;

import
 org.springframework.context.annotation.Bean;

import
 org.springframework.messaging.Message;

import
 java.util.function.Function;
```

```
@SpringBootApplication

public

class

CloudFunctionApplication
 {




public

static

void

main
(String[] args) {

    SpringApplication.run(CloudFunctionApplication.
class
,
args
);

  }

@Bean
```

```
public
 Function, String> echo() {

return
 (inputMessage) -> {

var
 stringBuilder =
new
 StringBuilder();



      inputMessage.getHeaders()

        .forEach((key, value) -> {

          stringBuilder.append(key).append(
": "
).append(value).append(
" "
);

        });

var
 payload = inputMessage.getPayload();

if
```

```
 (!payload.isBlank()) {

        stringBuilder.append(
"echo: "
).append(payload);

      }

return
 stringBuilder.toString();

    };



  }

}
```

The above files are normal Java maven project files. The only new type of file func CLI generates is which is used as configuration for the function project. func CLI makes use of values specified in func.yaml while executing its commands. Many of the fields of func.yaml are populated automatically when create, build, and deploy commands of func CLI are executed:

```
specVersion:

0.35.0
```

```
name:

hello-function

runtime:

springboot

registry:

""

image:

""

imageDigest:



""

created:

2023-02
-11T20:24:45.366219+05:30

build:

buildpacks:
```

```
[]

builder:

""

buildEnvs:

-

name:

BP_NATIVE_IMAGE

value:

"false"


-

name:

BP_JVM_VERSION

value:

"17"

-
```

```
name:

BP_MAVEN_BUILD_ARGUMENTS

value:

-Pnative

-Dmaven.test.skip=true

--no-transfer-progress

package

run:

volumes:



[]

envs:

[]

deploy:

namespace:
```

```
""

remote:

false

annotations:

{}

options:

{}

labels:

[]



healthEndpoints:

liveness:

/actuator/health

readiness:

/actuator/health
```

The following are some of the important fields of func.yaml file:

Specifies the name of the function, also used as the Knative service name when the function is deployed.

Specifies the language runtime for the function. For example, Spring Boot, as shown in the above example.

Specifies image registry URL for function image

Image name of the function. It gets populated automatically after the function is built.

Field contains the SHA256 hash of the image manifest and is populated automatically after the Function is deployed. Do not modify this value.

Building function

After making desired changes to the Spring Boot project as per the business requirements, func CLI helps to build the Knative Function. The outcome of this build process is an OCI container image of the developed function, which is then pushed to the container registry. Two types of builds are supported. They have been discussed below:

Local build

As the name suggests, local build builds a container image of the Function locally without deploying it on the cluster. Use the following command to build our Function locally.

The above command accepts any container registry name like local docker registry, Google Cloud Registry, IBM Cloud Registry, and so on. In our case, we have set up and provided a local registry in the command above.

To set up a local registry, you can follow https://docs.docker.com/registry/deploying/

Note: Knative Function works with Docker or Podman only. You will not be able to build or run Knative Functions locally if Docker or Podman is not installed on your machine.

You will see the following output on a successful build:

Now if we look at the func.yaml file once again, we will see registry and image details are embedded automatically by the func CLI as

```
name:
```

```
hello-function

registry:
localhost:5000

imageDigest:
""

build:

builder:
pack
buildEnvs:
-
name:
BP_NATIVE_IMAGE
```

```
-
name:
BP_JVM_VERSION

-
name:
BP_MAVEN_BUILD_ARGUMENTS

run:

envs:
[]
deploy:
namespace:
```

""

annotations:

{}

labels:

[]

healthEndpoints:

liveness:

/actuator/health

Use the following command to run this Function locally:

You should see the following output on the successful execution of the command above:

Function

```
already
built.
Use
--build
to
force
a
rebuild.
Function
started
on
port
8080
Setting
Active
```

Processor

Count

to

4

Calculating

JVM

memory

based

on

2542392K

available

memory

For

more

information

on

this

calculation,

see

https://paketo.io/docs/reference/java-reference/#memory-calculator

Calculated JVM Memory Configuration:

-XX:MaxDirectMemorySize=10M

-Xmx2146641K

-XX:MaxMetaspaceSize=88550K

-XX:ReservedCodeCacheSize=240M

-Xss1M

(Total

Memory:

2542392K,

Thread Count:

50

,

Loaded Class Count:

13220
,

Headroom:

0
%)

Enabling

Java

Native

Memory

Tracking

Adding

124

container

CA

```
certificates

to

JVM

truststore

Spring

Cloud

Bindings

Enabled

Picked up JAVA_TOOL_OPTIONS:

-Djava.security.properties=/layers/paketo-buildpacks_bellsoft-
liberica/java-security-properties/java-security.properties

-XX:+ExitOnOutOfMemoryError

-XX:ActiveProcessorCount=4



-XX:MaxDirectMemorySize=10M

-Xmx2146641K

-XX:MaxMetaspaceSize=88550K
```

```
-XX:ReservedCodeCacheSize=240M

-Xss1M

-XX:+UnlockDiagnosticVMOptions

-XX:NativeMemoryTracking=summary

-XX:+PrintNMTStatistics

-Dorg.springframework.cloud.bindings.boot.enable=true

.

____

_

__

_

_



/\\

/

___'_
```

```
__
_
_(_)_
__
__
_
\
\
\
\
(
(
)\____
|
'_ | '_|
|

'_ \/ _` | \ \ \ \

\\/  ___)| |_)| | | | | || (_| |  ) ) ) )

 ‹
|_____|

.__|_|

|_|_|

|_\__,

|

/

/

/

/

=========|_|==============|___/=/_/_/_/

::

Spring Boot ::
```

```
(v3.0.2)

2023-03
-05T16:29:06.621Z

INFO

1

---

[

main]

functions.CloudFunctionApplication       :

Starting

CloudFunctionApplication

v0.0.1-SNAPSHOT


using

Java

17.0.6

with
```

```
PID

1

(/workspace/BOOT-INF/classes

started

by

cnb

in

/workspace)

2023-03
-05T16:29:06.626Z

INFO


1

---

[

main]

functions.CloudFunctionApplication       :
```

```
No

active

profile

set,

falling back to 1 default profile:

"default"

2023-03
-05T16:29:10.000Z

INFO

1

---



[

main]

o.s.b.w.embedded.tomcat.TomcatWebServer  :

Tomcat

initialized
```

```
with

port(s):

8080

(http)


2023-03
-05T16:29:10.032Z

INFO

1

---

[



main]

o.apache.catalina.core.StandardService   :

Starting

service

[Tomcat]
```

```
2023-03
-05T16:29:10.033Z

INFO

1

---

[

main]

o.apache.catalina.core.StandardEngine    : Starting Servlet engine:

[Apache

Tomcat/10.1.5]



2023-03
-05T16:29:10.297Z

INFO

1

---

[

main]
```

```
o.a.c.c.C.[Tomcat].[localhost].[/]

:

Initializing

Spring

embedded

WebApplicationContext

03
-05T16:29:10.313Z

INFO

1

---

[

main]

w.s.c.ServletWebServerApplicationContext : Root WebApplicationContex
t:

initialization
```

```
completed

in

3516

ms

2023-03
-05T16:29:12.286Z

INFO

1

---

[

main]

o.s.c.f.web.mvc.FunctionHandlerMapping   : FunctionCatalog:

org.springframework.cloud.function.context.catalog.BeanFactoryAwareFu
nctionRegistry@78b7f805

2023-03
-05T16:29:12.362Z

INFO
```

```
1

---

[

main]

o.s.b.a.e.web.EndpointLinksResolver      :

Exposing

1

endpoint(s)



beneath

base

path

'/actuator'

2023-03
-05T16:29:12.594Z

INFO

1
```

```
---

[

main]

o.s.b.w.embedded.tomcat.TomcatWebServer  :

Tomcat

started

on


port(s):

8080

(http)

with

context

path

''


2023-03
-05T16:29:12.642Z
```

```
INFO

1

---

[

main]

functions.CloudFunctionApplication       :

Started


CloudFunctionApplication

in

6.984

seconds

(process

running

for

8.467
)
```

The above output shows your Function successfully started running as a service and will be able to serve requests on You can also use the following command to this Function locally

```
func

invoke
```

This command should generate the following output

```
Received response

content-length


:
25

host
:
localhost
:
8080

content-type
: application/json
id
: a4e2eec1-ecff-e7d2-
2957
-c730e7b97896
```

```
uri
: /
accept-encoding
: gzip
user-agent
: Go-http-client/
1.1

timestamp
:
1678034297984

echo
: {

"message"
:
"Hello World"
}
```

You can also send test data to this newly developed Function using the --data flag, as shown in the following snippet

```
func

invoke --data “Hello Knative function!"
```

The above command should generate the output as:

```
Received response
```

```
content-length:
23
 host: localhost:
8080
 content-
type
: application/json id:
56
d633ed-dd11-f4a4
-353
d-d5e24693bf3a uri:
/ accept-encoding: gzip user-agent: Go-http-client/
1.1
 timestamp:
1678034746440
 echo: Hello Knative

function
!
```

Remote build

This will push your created image to your registry, and the Function is deployed as a Knative service to the cluster. To deploy the Function on the cluster, use the following command.

```
func

deploy
 --
build
=
false
```

Note: The func remote build command will use the Kubernetes cluster, which is set in the kubectl config. You can use the kubectl config set-cluster command in case you want to use/set a particular cluster for the remote build of the Knative Function.

The above command generates the output as

```
⬆

Deploying

function
```

```
to

the

cluster

✅

Function

deployed in
namespace

"default"

and
 exposed at URL:

http
:
//hello-function.default.127.0.0.1.sslip.io
```

To verify the Function deployed on the Kubernetes cluster using the remote build, you can use the func invoke command as

```
func

invoke
```

The following output verifies that your Function deployed successfully on the cluster. You can see the host header value, which is your Functions route.

```
Received response



x-request-id:
46929
acc
-4564-4
f26
-8850
-e3792043dfa8 content-length:
25
 x-forwarded-proto: http host: hello-
function
.
default
.127.0.0.1.
sslip
.
io

content
-
type
:
application
/
json
```

```
id
: 1595
bd2f
-59

ca
-
bf31
-3881-4
e81f1236bbd

k
-
proxy
-
request
:
activator

uri
: /
accept
-
encoding
:
gzip

forwarded
:
for
=10.244.0.12;proto=http user-agent: Go-http-client/
```

```
1.1
 timestamp:
1678036170154



echo
: {
"message"
:
"Hello World"
}
```

To check Function details, use the func describe command as

```
kn
func

describe
```

Output generated would have Function name, namespace, and routes as shown in the following snippet

```
Function

name:

hello-function

Function
```

```
is built in image:

Function

is deployed in namespace:



default

Routes
:

http
:
//hello-function.default.127.0.0.1.sslip.io
```

You can also invoke the deployed Function using the route url given in the above output of the func describe command on the browser of your choice

```
http
:
//hello-function.default.127.0.0.1.sslip.io
```

Finally, if we look at the func.yaml file again, we will see the image digest embedded along with registry and image details automatically by the CLI, as shown in the following snippet:

```
specVersion:
```

```
0.35.0

name:

hello-function

runtime:

springboot



registry:

localhost:5000/hello-function

image:

localhost:5000/hello-function:latest

imageDigest:

sha256:cdc930883c662fa044e405059012c6c85eaf253c84bea278e9744ce0
5ffa9ccf

created:

2023-03
-05T21:40:03.090159+05:30

build:
```

```
buildpacks:

[]

builder:

pack

buildEnvs:



-

name:

BP_NATIVE_IMAGE

value:

"false"

-

name:

BP_JVM_VERSION

value:
```

```
"17"

-

name:

BP_MAVEN_BUILD_ARGUMENTS

value:


-Pnative

-Dmaven.test.skip=true

--no-transfer-progress

package

run:

volumes:

[]

envs:

[]

deploy:
```

```
namespace:

default

remote:

false

annotations:


{}

options:

{}

labels:

[]

healthEndpoints:

liveness:

/actuator/health

readiness:
```

/actuator/health

Conclusion

Knative Function is a developer-friendly, self-managed Function-as-a-Services with very good language support. It has got a few limitations, like dependency on Docker and Podman and limited documentation. In this chapter, we covered how to install, build, deploy, and run Knative Function. We also covered how developers can execute Knative Function on their local machine.

In the next chapter, we will start learning Knative Serving in detail. We will also start building a use case using Knative Serverless.

Multiple choice questions

Knative Functions require in-depth knowledge of Knative and Kubernetes.

True

False

Knative function templates are available only in Springboot and Node.js language.

True

False

Answers

b

b

Key terms

Image Registry: An Image Registry in Knative Functions is a container image repository that stores the container images used for running serverless functions. It allows developers to store, version, and distribute their function images, making them accessible for deployment and execution in the serverless environment.

Image Digest: In Knative Functions, an Image Digest is a unique identifier that represents the specific content of a container image. It is derived from the image's contents and serves as a fingerprint of the image's state. By using Image Digests, Knative ensures that the correct version of the function is deployed and executed, providing integrity and version tracking for the function images.

C
HAPTER
5

Knative Serving

Introduction

In previous chapters we learned about Knative Functions. How to create, build, deploy and run Knative Functions using func CLI using hands-on code recipes.

In this chapter we will understand the concept of Knative We will learn how to build, deploy, and run serverless containers using Knative Serving on Kubernetes. We will discuss and implement a case study to understand the concept using hands-on code recipes.

Structure

In this chapter, we will discuss the following topics:

Knative Serving

Benefits of Knative Serving

Case study – Online order processing system

Functional architecture

Deployment architecture

Flow diagram

Recipes

Service implementation

Build and deploy services

Validate services

Knative serving – Services runtime behavior

Objectives

By the end of this chapter, you should have a clear understanding of Knative Serving and should be able to build, and deploy Knative Serving based services on Kubernetes cluster.

Knative Serving

Knative Serving provides a mechanism to deploy, run, and manage serverless, cloud-native workloads on Kubernetes. Knative Serving has objects defined as Kubernetes Custom Resource Definitions The following are primary custom resources of Knative Serving:

Please refer Knative Serving section of Chapter Serverless and Knative in a Nutshell, for detailed understanding of custom resources.

Following are some good examples where Knative Serving could be useful:

For rapid serverless containers deployments on Kubernetes

Pods autoscaling, including scale down to zero

Multiple networking layers support like Kourier, Gloo, Istio for integration into existing environment

Benefits of Knative Serving

To understand the benefits of Knative Serving, we need to understand how Knative Serving eases out the process of application deployment on Kubernetes Cluster. Let us first understand the process of a simple application deployment on Kubernetes cluster without Knative. The following Kubernetes artifacts are required to deploy an application on cluster:

A deployment to manage ReplicaSets which creates Pods in the background.

A service to expose your application to the cluster.

A route to expose your application outside the cluster.

An HorizontalPodAutoscaler to support autoscaling.

So, there is lot to consider for a simple application deployment on Kubernetes cluster and somehow it slows down the core development process.

With Knative you need to create only one resource that is a Knative Service, using Knative command line interface with kn create name> command. Or by using a service YAML file and applying that using kubectl Knative by default creates all the necessary resources required to run and expose your application externally on Kubernetes cluster. Figure 5.1 depicts the typical deployment of Knative Service and its associated resources:

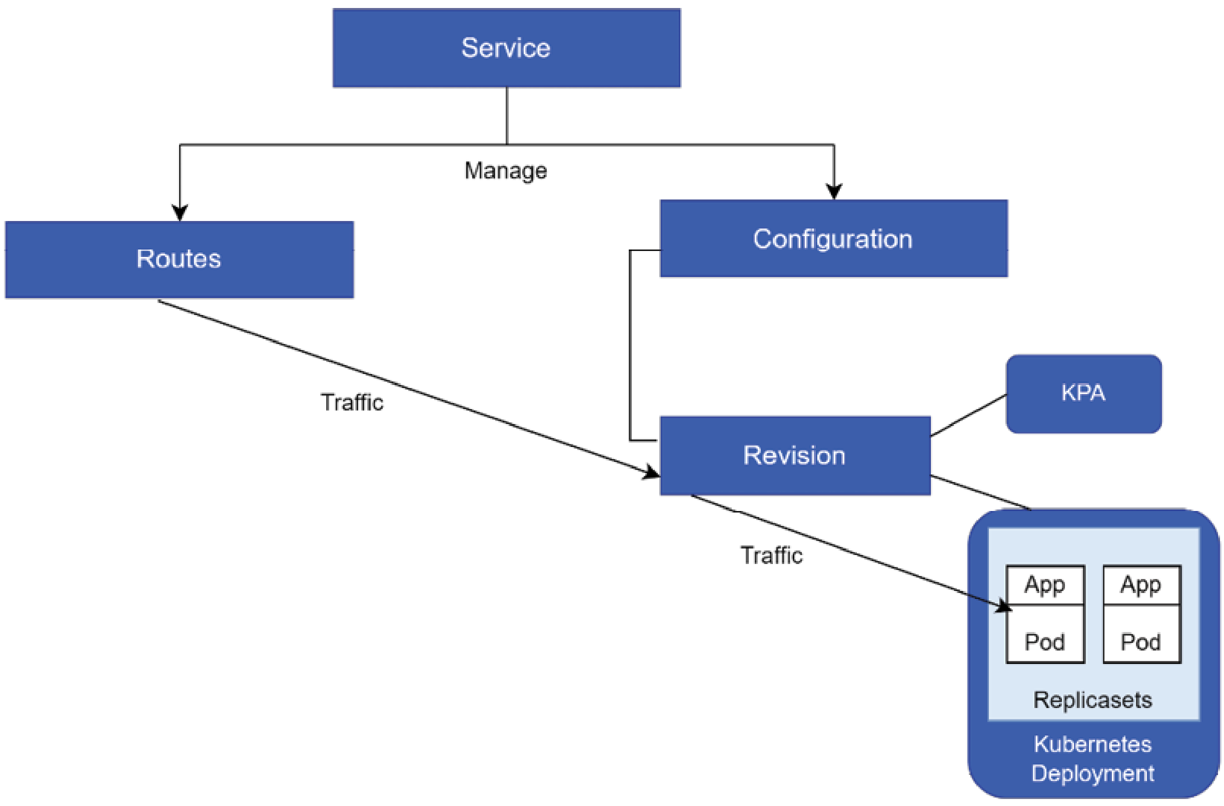


Figure 5.1: Knative serving resources

Let us understand the benefits of Knative Serving in detail:

Simplified deployment

The Knative service automatically creates routes and configurations. Configuration keeps track of revisions; each has Kubernetes Deployment and Knative Pod Autoscaler You can access all these created resources in your Kubernetes cluster using certain set of commands that we discuss later in the chapter. You can use the exposed Route URL to access your application.

Autoscaling and scale to zero

As we can see in the Figure 5.1 above, traffic reaches Kubernetes cluster through Knative Route. By default, the route sends 100% of traffic to the latest revision. Knative Pod Autoscaler KPA continuously monitors the number of incoming requests to the revision and automatically scales pods up and down, and when there is no traffic going to your application, it scales number of Pods to zero.

Easier release management via revisions

Knative configuration CRD manages multiple revisions. Revisions can be defined as point-in-time snapshots of your code and configuration. Suppose you want to update your application configuration, update configuration using kn service update or in service YAML file and apply. This will create the latest revision and by default 100% of traffic via Routes reaches latest revision.

Knative also allows you to split traffic between multiple revisions as depicted in the Figure Suppose you are rolling out a new release, which creates Revision-2 and you want to test some scenarios before sending out all traffic to new release. You can configure Knative Service to send small amount of traffic let us say 20% to new Revision-2 and 80% will be served from your stable release Revision-1. You can gradually increase traffic on Revision-2 and once you are sure by running all your tests, you can configure to serve 100% of traffic from new version.

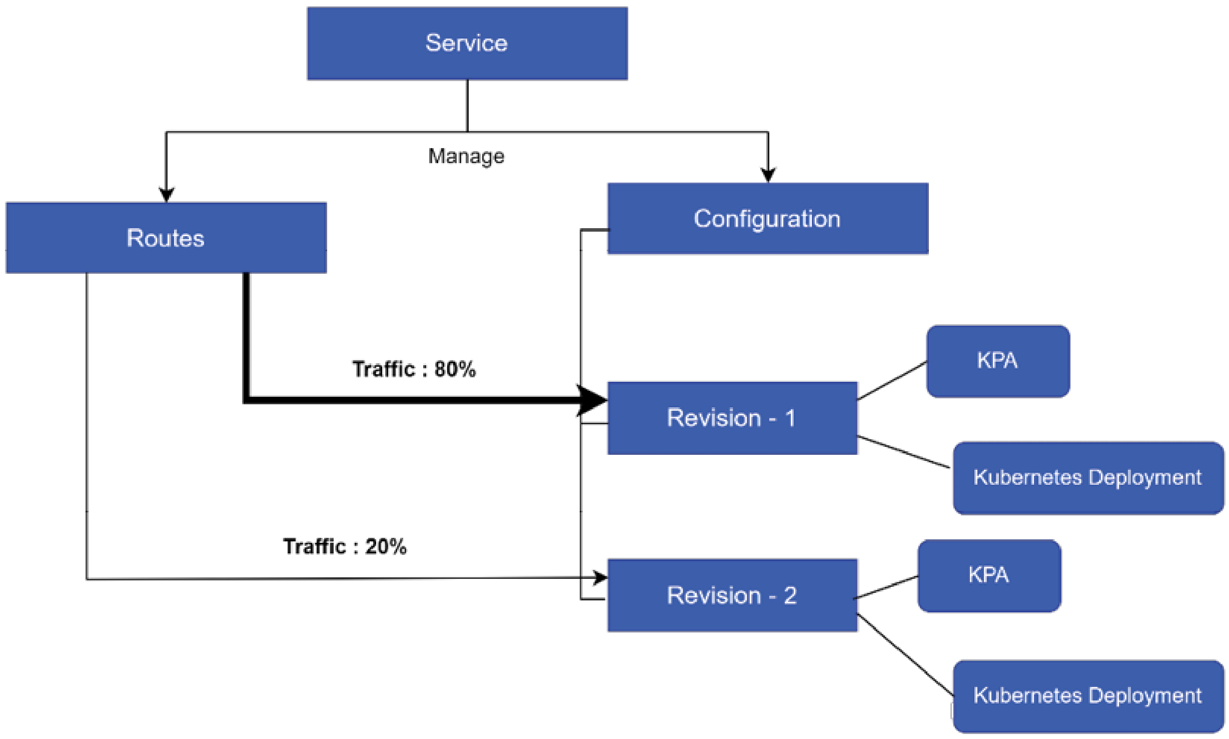


Figure 5.2: Knative serving revisions and traffic split

Easier rollback:

Since Knative configuration CRD manages multiple revisions, you can jump back to any version of your application. This will help in easier rollback of applications.

To understand the Knative Serving concept better, let us implement a simple use case of Online order processing system. We will implement important Knative Serving features and observe run time behaviors. Let us begin with implementing the use case.

Case study – Online order processing system

This case study involves creating simple microservices that process orders placed for a product by an online buyer. Our system accepts and captures customer orders, allocates purchased products from inventory and deducts product cost from customer's wallet. To implement the use case, we will be creating three microservices, order product service and customer service used to fulfill the requested order. To keep the implementation simple, we have taken following assumptions:

We will create backend services only used to fulfil orders.

Users who purchase products pre-exist in the database.

There is no actual payment integration. There are dummy payments made from user pre-loaded wallet amount saved in database.

In memory database used to store customers, products, and order details.

No admin service exposed to add/update customer and product details in database.

Functional architecture

To build this simple system, we require the following services:

Order This service provides restful APIs responsible for accepting product purchase requests from the customer. This service captures orders requested by the customer and its status. This service is also responsible for updating orders and status to product and customer service.

Product This service is responsible for allocating products requested by customers from the product inventory. This service captures product information and manages product inventory. This service updates order service on successful allocation of product purchased.

Customer This service is responsible for deducting purchased product costs from customer wallet. This service captures customer information and manages their wallet amount. This service is responsible for updating order service on successful deduction of amount from customer wallet.

Figure 5.3 illustrates how these components work together:

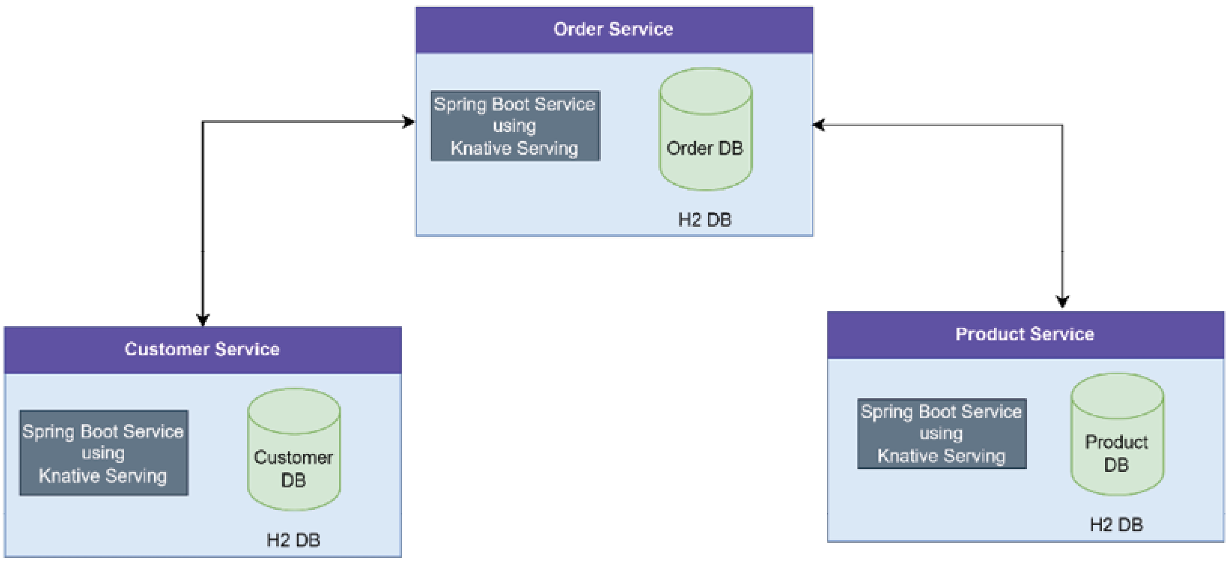

Order Service
Spring Boot Service using Knative Serving
Order DB
H2 DB
Customer Service
Spring Boot Service using Knative Serving
Customer DB
H2 DB
Product Service
Spring Boot Service using Knative Serving
Product DB
H2 DB

Figure 5.3: Order processing system – functional architecture

While building the final architecture we made several architectural decisions. Let us review them:

Microservices We decided to create all the services on microservices architecture, so that it is easier to manage and work independently. We choose Spring Boot as it is a proven framework for microservices development to develop services along with Java 11 as a development language. We will use Spring cloud functions to publish functions as a service

Storage: We use In-memory H2 database to store data about orders, customers, and products. Since this is a test use case, we will be using this H2 to store information.

Service communication: We decided to use HTTP based communication to communicate with different services in the System

Let us know look at the deployment architecture on Kubernetes Cluster. The following section assumes you have a good understanding of Kubernetes.

Deployment architecture

Figure 5.4 illustrates the deployment architecture of our components in Kubernetes cluster:

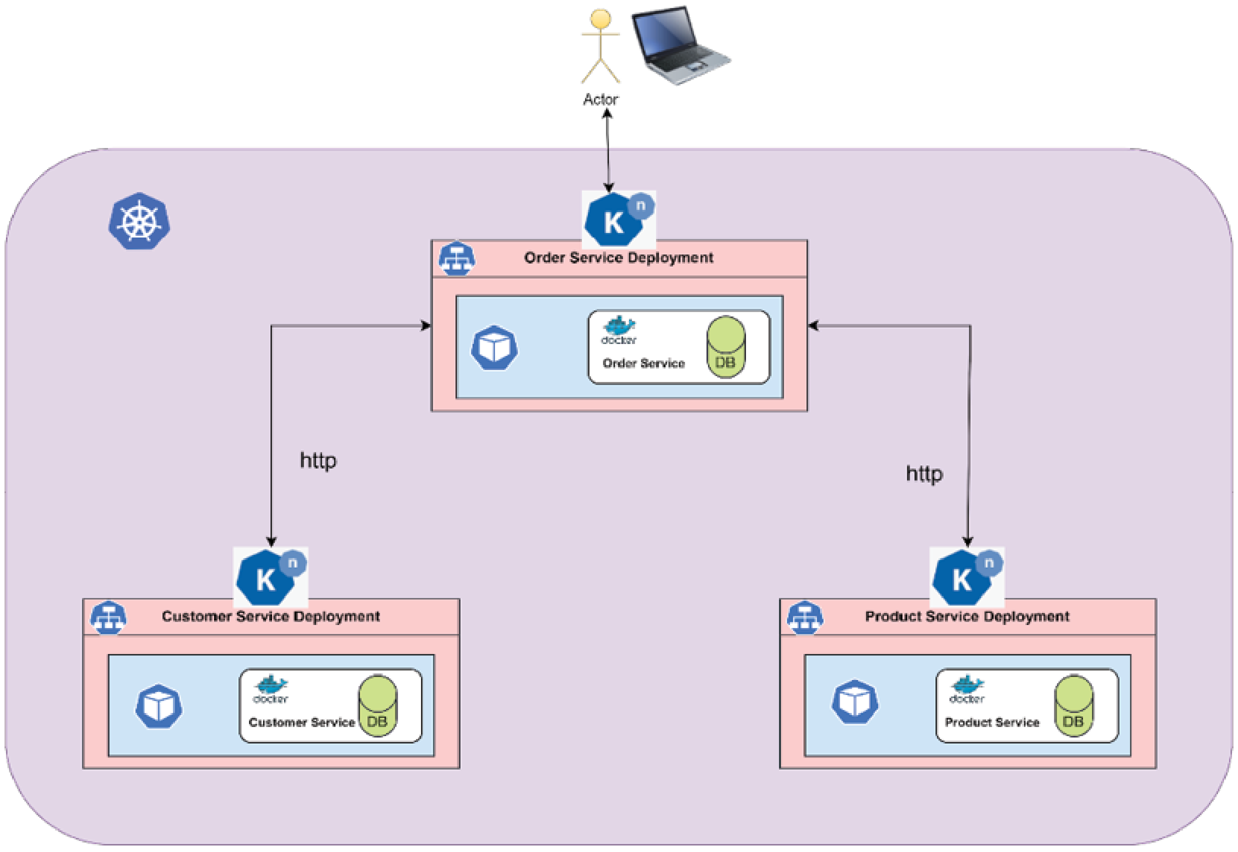


Figure 5.4: Order processing system – deployment architecture

Let us understand the above diagram in detail:

Order service deployment: This deployment has Knative Serving implementation of Order Service using Spring Boot with Spring cloud function. This service exposes endpoint for accepting orders from users. This service maintains user order and its status in H2 database. This service communicates with Product and Customer service for order fulfilment.

Product service deployment: This deployment has Knative Serving implementation of product service using Spring Boot with Spring cloud function and Spring Kafka. This service maintains and manages product inventory in H2 database. This service updates Order Service about product status.

Customer service deployment: This deployment has Knative Serving implementation for customer service using Spring Boot with Spring Cloud function. This service maintains customer information in H2 database. This service updates order service about product cost deduction from customer wallet.

Now that, we have understood the details of deployment architecture of our order processing system. Let us now look at the way our services communicate with each other.

Flow diagram

Figure 5.5 illustrates the way and the sequence in which our services communicate with each other:

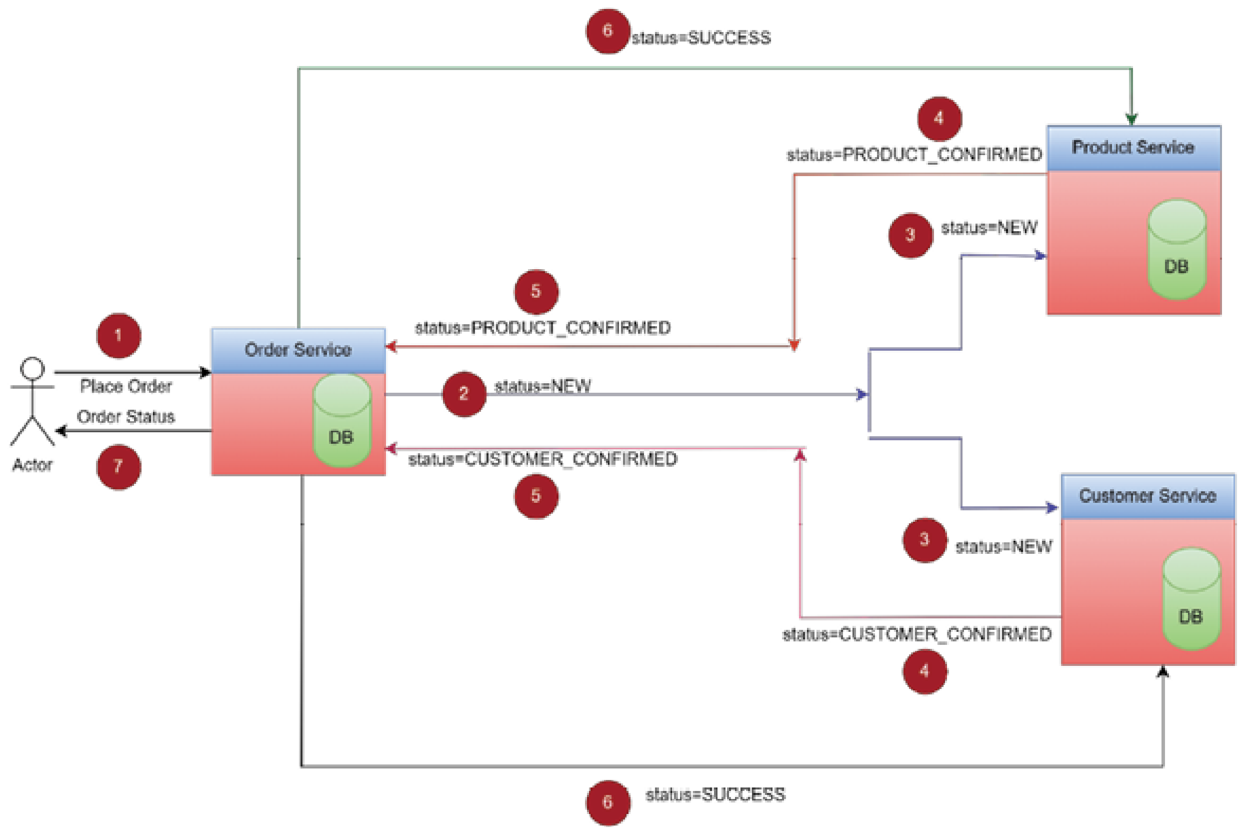


Figure 5.5: Order processing system – Flow diagram

Let us understand above flow diagram in detail:

User places an order with user (ID, amount) and product details (product ID, product count).

Request reaches order service, which updates order DB with order details, maintains order status=NEW and calls product and customer service over http with order details.

Product service updates Product DB to block product and Customer Service updates Customer DB to block amount.

Product service and customer service call order service over http with status=PRODUCT_CONFIRMED and status=CUSTOMER_CONFIRMED respectively to confirm order.

Order service updates order DB as order status=SUCCESS

Order service with calls Product Service to update product inventory and Customer Service to update customer amount.

Order service finally returns the response back to user as order accepted.

Let us go ahead now and create our services.

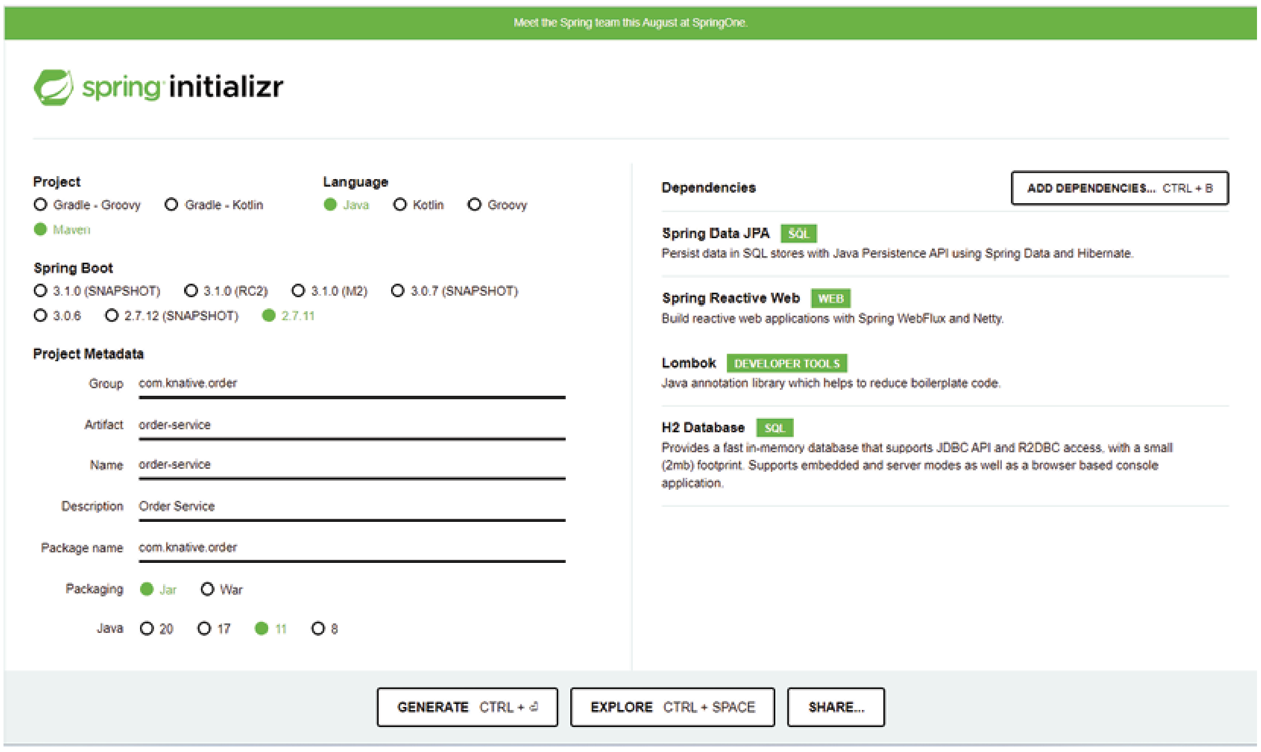
Service Implementation

So far we have defined functional and deployment architecture for our use case. We have also laid out the way the services will interact with each other. Let us now start implementing them as per the architecture laid out and using technologies as per architectural decisions.

Order service

Assuming you already have Knative Serving installed on our system to Chapters 2 and 3 for detailed installation and verification steps with a fair understanding of Maven, Spring Boot and Spring Cloud Functions.

Let us implement Order Service using Spring Initializr which generated boilerplate code and required packages as a zip file. You can import the zip file in any IDE of your choice and start building the service. Figure 5.6 is from the Spring Initializr website; to implement our service we have chosen development language as Java version 11 and Spring Boot version 2.7.11.


Meet the Spring team this August at SpringOne.

spring initializr

Project: Gradle - Groovy, Gradle - Kotlin, [✓ Maven]

Language: [✓ Java], Kotlin, Groovy

Spring Boot: 3.1.0 (SNAPSHOT), 3.1.0 (RC2), 3.1.0 (M2), 3.0.7 (SNAPSHOT), 3.0.6, 2.7.12 (SNAPSHOT), [✓ 2.7.11]

Project Metadata

Group: com.knative.order
Artifact: order-service
Name: order-service
Description: Order Service
Package name: com.knative.order
Packaging: [✓ Jar], War
Java: 20, 17, [✓ 11], 8

Dependencies

Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

Spring Reactive Web WEB
Build reactive web applications with Spring WebFlux and Netty.

Lombok DEVELOPER TOOLS
Java annotation library which helps to reduce boilerplate code.

H2 Database SQL
Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Figure 5.6: Order processing system – Spring initalizr

The following are the main dependencies we have chosen to implement this service:

For database connection and operations.

To expose functions as a service.

H2 As an in-memory database. Use the following link to learn and understand more on Spring boot in-memory database H2.

H2 DB is not recommended for production environment.

Code generator for minimizing the code.

The following snippet is from generated pom.xml with dependencies:

org.springframework.boot

spring-boot-starter-data-jpa

<
dependency>

org.springframework.boot

spring-boot-starter-webflux

org.springframework.cloud

spring-cloud-starter-function-web
artifactId>

org.projectlombok

<
artifactId>lombok

true

com.h2database

h2

<
scope>runtime

```
<
dependencies>

org.springframework.cloud

${spring-cloud.version}

pom

<
scope>import
```

Now to create Order we will start adding our own classes and configurations in the generated code. Following are few important classes and configurations added to complete the service. Refer to the GitHub repository of the book to clone and download full Order Service code.

An Order Database, having order_details table contains order information and status. Table 5.1 shows the table structure:

Table 5.1 : Order Details

A model class OrderDetails represents database table which is used to save order details.

```
@Entity
@Table(name = "order_details")
@Getter
@Setter
@NoArgsConstructor
@ToString

public class OrderDetails {
 @Id
 private Integer id;
 @Column(name = "customer_id")
 private Integer custId;

 @Column(name = "product_id")

 private Integer prodId;

 @Column(name = "amount")

 private Integer amount;

 @Column(name = "product_count")

 private Integer prodCount;

 @Column(name = "product_order_status")

 private String productOrderStatus;


 @Column(name = "customer_order_status")

 private String customerOrderStatus;

 @Column(name = "order_status")

 private String orderStatus;

}
```

An OrderStatus class which holds the order status constants:

```
public enum OrderStatus {

 NEW,

 IN_PROGRESS,

 CUSTOMER_CONFIRMED,

 PRODUCT_CONFIRMED,

 CONFIRMED

}
```

Let us understand above status:

Initial order request by user

When an order accepted by order service

When order confirmed by customer service

When order is confirmed by product service

When order service updates customer and product service after successful completion

Following snippets are from OrderApplication class which contains the main method that starts application context. This class also contains functions place() and confirm() which are exposed as a service.

Function place() is used to accept users order request and update customer and product services.

```
@Bean

public
 Function> place(){



 }

private
 ResponseEntity doPlaceOrder(OrderDetails orderDetails) {




  orderDetails.setOrderStatus(orderDetails.getOrderStatus() !=
null
 ? orderDetails.getOrderStatus() : OrderStatus.NEW.toString());

  String orderStatus = orderDetails.getOrderStatus();

  LOGGER.info(
"order-service :: Order Status in Request :: "
 + orderStatus);
if(OrderStatus.NEW.toString().equalsIgnoreCase(orderStatus)) {

this.callCustomer(orderDetails);

this.callproduct(orderDetails);

        .statusMessage(
"Request Processed Successfully")

        .build(), HttpStatus.ACCEPTED);

  }

return
 new ResponseEntity<>
(OrderResponse.builder().status(OrderStatus.IN_PROGRESS.toString())

          .build(), HttpStatus.ACCEPTED);

 }
```

Function confirm() is used to check status of product and customer service for order confirmation and update final order details with status in database.

```
@Bean

public
 Function confirm(){



 }

private

 String doConfirmOrder(OrderDetails orderDetails) {



  LOGGER.info(
"order-service :: Order Status  :: "
 + orderDetails.getOrderStatus());

  Optional order = orderDetailsRepository.findById(orderDetails.getId());

if(order.isPresent()) {

//In progress Order

   String orderStatus = orderDetails.getOrderStatus();

if(OrderStatus.CUSTOMER_CONFIRMED.toString().equalsIgnoreCase(ord
erStatus)) {
   order.
get().setCustomerOrderStatus(orderStatus);

   orderDetailsRepository.save(order.
get());

   order.
get().setOrderStatus(OrderStatus.IN_PROGRESS.toString());

   order.
get().setCustomerOrderStatus(order.get().getCustomerOrderStatus());

  }

  order = orderDetailsRepository.findById(orderDetails.getId());

if
(OrderStatus.CUSTOMER_CONFIRMED.toString().equalsIgnoreCase(order
.get().getCustomerOrderStatus()) &&

   order.
get().setOrderStatus(OrderStatus.CONFIRMED.toString());
```

```
this.callCustomer(order.get());



   LOGGER.info(
"Final Order Status :: "
 + order.
get());

   }

  }


 }
```

Following application.yml file holds important properties along with database information:

```
spring.cloud.function.web.path:

/orders

spring:

h2:
```

datasource:

username:

sa

password:

driverClassName:

org.h2.Driver

jpa:

defer-datasource-initialization:

true

generate-ddl:

true

hibernate:

ddl-auto:

create-drop

Now with the help of application.yml the functions place() and confirm() are exposed as a service and are accessible at end points /orders/place and /orders/confirm respectively.

Create Dockerfile with following content to build Docker image:

```
FROM openjdk:11-jre-slim

ARG JAR_FILE=target/*.jar

COPY ${JAR_FILE} order-service.jar

ENTRYPOINT ["java","-jar","order-service.jar"]
```

This completes the implementation of Order Later in this chapter, we will build and deploy this as Knative Serving service on Kubernetes Cluster.

Now let us start implementing our next service that is, Product Service which takes care of Products Inventory of our Order Processing System.

Product service

We will implement this service in a similar fashion as Order Service. Please refer and follow Order Service implementation steps.

Here, we will discuss a few important components used to implement product service functionalities. For full code access please refer to the book's GitHub repository.

A Product Database, having product_details table contains information about Products. Table 5.2 shows the table structure:

Table 5.2 : Product details

Table 5.2 will be created in H2 database and product related data will be inserted upon application start-up.

A ProductDetails class is created to map Java properties with product_table columns:

```
@Entity

@Table(name = "product_details")

@Getter
```

```
@Setter

@NoArgsConstructor

@ToString

public class ProductDetails {

 @Id

 private Integer id;



 @Column (name = "product_name")

 private String prodName;

 @Column (name = "product_available
"
)

 private Integer productAvailable;

 @Column (name = "product_blocked")

 private Integer productBlocked;

}
```

A ProductOrderStatus class which holds the product status constants:

```
public

enum

ProductOrderStatus
 {

NEW
,

PRODUCT_CONFIRMED


,

CONFIRMED


}
```

Let us understand each one of the above elements:

Initial o‹rder request by user

When product service confirms the order

order service updates product service after successful fulfilment of order

A Spring Boot main class ProductApplication is created to start up Product Service Application Context using Java main method. This class also contains function block() with business logic for product and will be exposed as a REST Endpoint that Order Service uses for processing requests. This function blocks products for users in Product Inventory and updates product count upon final confirmation from Order Service. Following is the code snippet of block() function:

```
@Bean

public Consumer block(){



 }



private void doBlockProduct(OrderDetails orderDetails) {



  LOGGER.info(
"product-service :: Order Status :: "
 +  orderDetails.getOrderStatus());

if(orderDetails != null) {

        ProductDetails productDetails = productDetailsRepository.findById(
orderDetails.getProdId()).orElseThrow();
```

```
            productDetails.setProductBlocked(productDetails.getProductBloc
ked() + orderDetails.getProdCount());

            productDetails.setProductAvailable(productDetails.getProductAva
ilable() - orderDetails.getProdCount());

            orderDetails.setOrderStatus(ProductOrderStatus.PRODUCT_CO
NFIRMED.toString());

            productDetailsRepository.save(productDetails);



this.callOrder(orderDetails);



            productDetails.setProductBlocked(productDetails.getProductBloc
ked() - orderDetails.getProdCount());

            productDetailsRepository.save(productDetails);

        }



       }

   }
```

Function block() is accessible as a service using URI /products/block.

Finally create Dockerfile which we will use later to build image and push this into registry.

Now let us start implementing next service that is, Customer Service which performs customer related operations in our order processing

Customer service

This service is implemented in a similar fashion as order and product services using similar dependencies.

Here, we will discuss a few important components used to implement customer service functionalities. For the full code access please refer to the book's GitHub repository.

A Customer having a customer_details table contains information about Products. Table 5.3 will be created in H2 database and products related data will be inserted upon application startup:

Table 5.3: Customer details

A CustomerDetails class is created to map Java properties with Customer DB table columns using JPA.

```
@Entity

@Table(name = "customer_details")

@Getter

@Setter
```

```
@NoArgsConstructor

@ToString

public class CustomerDetails {

@Id



 private Integer id;

@Column(name = "customer_name")

 private String custName;

@Column(name = "wallet_amount")

 private Integer walletAmount;

@Column(name = "wallet_amount_blocked")

 private Integer walletAmountBlocked;

}
```

An enum CustomerOrderStatus which holds the service status constants:

```
NEW,

CUSTOMER_CONFIRMED,


CONFIRMED

}
```

Let us understand each one of the above:

Initial order requested by user.

When customer service confirms the order.

When order service updates customer service upon successful fulfilment of order.

A Spring Boot main class CustomerApplication is created to start up customer service application context using Java main method. This class also contains function blockAmount() having business logic for customer service, which will be exposed as a REST Endpoint and is used by Order Service for processing requests. This function blocks amounts for users from the customer wallet and updates amount upon final confirmation from Order Service. Following is the code snippet of blockAmount() function:

```
@Bean
```

```
public Consumer blockAmount(){



 }



private void doBlockAmount(OrderDetails orderDetails) {

  LOGGER.info(
"Customer-service :: Order Details :: "
 +orderDetails.toString());



if(orderDetails != null) {

        CustomerDetails customerDetails = customerDetailsRepository.find
ById(orderDetails.getCustId()).orElseThrow();

if
 (CustomerOrderStatus.NEW.toString().equalsIgnoreCase(orderDetails.get
OrderStatus())) {

            customerDetails.setWalletAmountBlocked(customerDetails.getWa
lletAmountBlocked() + orderDetails.getAmount());

            customerDetails.setWalletAmount(customerDetails.getWalletAmo
unt() - orderDetails.getAmount());
```

```
            }

            orderDetails.setOrderStatus(CustomerOrderStatus.CUSTOMER_
CONFIRMED.toString());



            customerDetailsRepository.save(customerDetails);


this.callOrder(orderDetails);


        }
else

if
 (CustomerOrderStatus.CONFIRMED.toString().equalsIgnoreCase(order
Details.getOrderStatus())) {


            customerDetails.setWalletAmountBlocked(customerDetails.getWa
lletAmountBlocked() - orderDetails.getAmount());


            customerDetailsRepository.save(customerDetails);


        }



       }


 }
```

The function blockAmout() created is accessible using URI

We will use Docker to build a customer service image and push this into the registry.

Till now we have built all the required services. In the next section, we will build and deploy our services as Knative Serving component on Kubernetes Cluster.

Build and deploy

Now that, we have completed the implementation our services, let us build them using Maven and Docker and deploy them as Knative Serving service on Kubernetes Cluster.

Order service

Navigate to your project directory and run following Maven command to build Order Service as jar:

```
mvn clean install
```

Following is the expected output:

```
[INFO]

Installing

order-service-0.0.1-SNAPSHOT.jar

to
 /
com/knative/order/order-service/0.0.1-SNAPSHOT/order-service-0.0.1-
SNAPSHOT.jar



[INFO]

------------------------------------------------------------------------
```

Navigate to the created Dockerfile in your project directory and run following command to create Docker image:

```
docker build -t knativedemo/order-service .
```

Following is expected output on successful execution of command:

```
 => exporting to image                                  .3s

 => => exporting layers                         .3s

 => => writing image sha256:0db17358c2df72df743b6d393e0cda8b3a709
d2dc89249abc24e2a2463dc4ed6          0.0s

 => => naming to docker.io/knativedemo/order-service
```

Next step is to push the image to an image registry. We have setup Docker Hub as image registry to push images. You can use registry of your own choice as well like Google Cloud Registry and so on.

```
docker push knativedemo/order-service
```

The command will generate following output:

```
The push refers to repository [docker.io/knativedemo/order-service]
```

```
ef9e1367b730: Pushed

d7802b8508af: Layer already exists

e3abdc2e9252: Layer already exists

eafe6e032dbd: Layer already exists

92a4e8a3140f: Layer already exists


latest: digest: sha256:686e89c77880d3a375e11a4b648ac36d788c9057abc
eed4cf52c7f361c7923c5 size: 1371
```

Now, that we have completed all the required steps to build Order let us deploy it on the Kubernetes cluster. We are using default Kubernetes namespace for deployment. To deploy order service, run the following command:

```
--image docker.io/knativedemo/order-service \

--port 8080 \

--env customer-service-ep=http://customer-
service.default.svc.cluster.local \

--env product-service-ep=http://product-service.default.svc.cluster.local
```

The above Knative command creates a service with the name order-service by pulling the image from Docker repository and running on port 8080. We are also setting environment variables available to service after successful deployment. The command will start deployment of our Order Service on cluster and generates following output:

```
http://order-service.default.127.0.0.1.sslip.io
```

This completes the deployment Order

Product and customer service

Product and customer service can be built in a similar way like Order Service using Maven and Docker. Refer Order Service deployment steps to build them. Let us deploy product services on Kubernetes cluster by running following command:

```
--image docker.io/knativedemo/product-service \

--port 8080 \

--env order-service-ep=http://order-service.default.svc.cluster.local
```

Following is the expected output:

```
http://product-service.default.127.0.0.1.sslip.io
```

Similarly, deploy customer service by running following command:

```
--image docker.io/knativedemo/customer-service \

--port 8080 \
--env order-service-ep=http://order-service.default.svc.cluster.local
```

Following is the expected output:

```
http://customer-service.default.127.0.0.1.sslip.io
```

Finally, verify that all our services are deployed successfully and are ready to serve by running following command:

```
Kn services ls
```

Following is the expected output:

This completes the verification of all our services required for our use case.

With this we have completed the creation and deployment of all three services order service, product service and customer service for implementation our use case Order Processing System.

Validate services

Now we will do a simple demonstration of how our services worked together to complete the order requested by a customer. We will check logs generated against of each of the service.

Let us start by creating a new order request using our Order Service REST Endpoint with the following request:

```
{

    "id": 1,

    "custId": 1,

    "prodId": 1,

    "amount": 100,

    "prodCount": 1

}
```

Following curl command executes Order Service, and we can see response retuned along with statusCodeValue as

```
-H 'Content-Type:application/json' \
```

Let us go through the following logs generated by Product and Customer Service:

```
INFO

1

---

[nio-8080-exec-2]

c.knative.customer.CustomerApplication   : Customer-
service :: Order Status ::

NEW

INFO

1
```

```
---

[nio-8080-exec-5]



c.knative.customer.CustomerApplication   : Customer-
service :: Order details ::

CustomerDetails(id=1,

custName=Cust1,

walletAmount=49900,

walletAmountBlocked=0)



INFO

1

---

[nio-8080-exec-2]

com.knative.product.ProductApplication   : product-
service :: Order Status ::

NEW
```

```
INFO

1

---

[nio-8080-exec-5]

com.knative.product.ProductApplication   : product-
service ::Order details ::

ProductDetails(id=1,

prodName=Product1,

productAvailable=999,

productBlocked=0)
```

In both logs, we can observe the initial and final order status provided by Order Service to both Customer and Product Service.

Now, let us go through logs generated by Order Service. We can see new order requests received by order service and finally confirmation received from Product Service as PRODUCT_CONFIRMED and Customer Service as CUSTOMER_CONFIRMED to update the final status of order as

```
Invoking function:
 confirmwith input
type:
 class com.knative.order.model.OrderDetails

INFO 1
 --- [nio-
8080-exec-8] com.knative.order.OrderApplication       : order-
service :: Order Status  :: PRODUCT_CONFIRMED
```

So, that is the complete validation of our Case Study.

Knative Serving – Services runtime behavior

As we discussed initially in this chapter, we just need to focus on service development, and the rest like creation of routes, revisions, and autoscaling are being handled by Knative Serving.

Let us first verify the creation of Knative custom resources against our services:

Verify automatically by Knative using the following command:

```
kn routes list
```

URLs returned in the following output confirm the creation of routes against each service which will be used as REST Endpoints:

```
customer-service

http://customer-service.default.127.0.0.1.sslip.io

True



product-service
```

http://product-service.default.127.0.0.1.sslip.io

True

traffic with the following command:

```
kn revision ls
```

The following output confirms the latest revisions created and serving 100% of traffic:

```
customer-service-00001

customer-service

100
%

1

3h24m

3

OK

/
```

4

True

product-service-00001

product-service

100
%

1

3h23m

3

OK

/

4

True

We will see the magic of Knative Serving now by showing, how our Pods scale up upon a request and scales down to zero when there is no request.

Let us once again check status of our services using following command:

Kn services ls

The output of the command shows our services are ready to serve requests:

customer-service

http://customer-service.default.127.0.0.1.sslip.io

customer-service-00001

3h49m

3

OK

/

3

True

product-service

http://product-service.default.127.0.0.1.sslip.io product-service-00001

3h48m

3

OK

/

3

True

Let us now check status of our Pods using the following command:

The following output shows no resources in the namespace, that is, all Pods are scaled down to zero because of no requests on services:

Let us generate an order using the following request and check the pod’s status once again:

-H 'Content-Type:application/json' \

Finally, analyze the following output to see power of Serverless containers using Knative Serving and check how our Pods scale up on receiving a request and again down to zero when there is no request:

As you can see, in line number 2, no Pods are running and upon receiving a new order request. Order Service Pods start scaling up (line number 4) and once Order service calls Customer and Product Service endpoints, both the services Pods start scaling up (line numbers 11 and After completing the order and again on no request, Pods start terminating (line numbers 25, 26, 27) and finally scale down to zero (line number 29).

Conclusion

In this chapter we covered the concept of Knative Serving and understood its various features. We implemented a case study, Online Order Processing System using Microservices Architecture and implemented various concepts of Knative Serving. We also saw the power of Knative Serving using live examples of Pods scaling up and down to zero automatically.

In the next chapter, we will get to learn about Knative Eventing and related concepts. We will enhance our case study by implementing Knative Eventing concepts and features.

Multiple choice questions

What does Knative Serving provide in a Kubernetes environment?

Networking capabilities for the cluster.

A serverless platform to run applications.

Automatic scaling of Kubernetes nodes.

Tools for monitoring resource utilization.

Which component of Knative automatically scales the number of pods based on incoming traffic?

Knative Eventing

Knative Build

Knative Serving

Knative Networking

Answers

b

c

Key terms

HPA Horizontal Pod Autoscaler, which automatically adjusts the number of replicas for a Knative Service based on CPU utilization or custom metrics, ensuring efficient resource utilization.

KPA stands for Knative Pod Autoscaler, a component that dynamically scales the number of pods in a Knative Service to handle incoming traffic based on the defined concurrency targets, enabling automatic scaling for event-driven workloads.

C HAPTER 6

Knative Eventing

Introduction

In the previous chapter, we learned about Knative Serving, detailing how to create, build, deploy, run services built using Knative Serving and demonstrated with hands-on code recipes.

In this chapter, we will understand the concept of Knative We will learn how to build, deploy, and run services using Knative Eventing on Kubernetes. We will enhance our use case – order processing system to understand the concepts of various Knative Eventing patterns using hands-on code recipes.

Structure

In this chapter we will discuss the following topics:

Knative Eventing

Knative Eventing patterns

Source to Sink pattern

Broker and Trigger pattern

Case study – Online order processing system

Functional architecture

Deployment architecture

Order processing system with Source to Sink pattern

Recipes

Service implementation

Build and deploy services

Event source installation

Create Event sources

Validate services

Order processing system with Broker and Triggers pattern

Recipes

Service implementation

Build and deploy services

Broker installation

Create event source

Create triggers

Validate services

Objectives

By the end of this chapter, you should have clear understanding of Knative Eventing and should be able to build, and deploy Knative Eventing based services on Kubernetes Cluster.

Knative Eventing

As described in Chapter Knative Eventing is a set of APIs that provides the mechanism to use event-driven architecture for Serverless applications.

These APIs are composed of the following:

Event source

Broker

Filter

Trigger

Subscription

Sink

Knative Eventing enables the development of asynchronous applications through event delivery from anywhere. It is primarily focused on three main areas:

Developer It provides a uniform experience for developers when sourcing events from systems which enables them to develop, support and maintain event driven applications. It also provides standard experience in event creation and consumption using the Cloud Event standards. Refer https://cloudevents.io/ to learn more about Cloud

Event delivery: It simplifies the event delivery mechanism and provides a pluggable architecture which helps developers to choose the correct event delivery pattern for their requirements.

Event sourcing: It provides tools and specifications for creating events and integrating in Cloud Event environment.

Refer Knative Eventing section of Chapter Serverless and Knative in a Nutshell, to better understand the above components.

Knative Eventing patterns

The following are primary Eventing patterns with Knative Eventing:

Source to Sink pattern

This is the basic pattern with Knative Eventing which contains two Knative Eventing components:

Event source: Event source acts as a link between an event producer and an event sink. Table 6.1 lists few important event sources maintained by Knative:

Knative:
Knative: Knative: Knative: Knative: Knative: Knative: Knative: Knative: Knative: Knative: Knative: Knative: Knative: Knative: Knative:
Knative: Knative: Knative: Knative: Knative: Knative: Knative: Knative: Knative: Knative: Knative: Knative: Knative: Knative: Knative:
Knative: Knative: Knative: Knative: Knative: Knative: Knative: Knative: Knative: Knative:
Knative: Knative: Knative: Knative: Knative:

Table 6.1: Event Sources supported by Knative

Sink: Sink could be a Knative Service, a broker or a channel that receives events from event Sources as shown in Figure

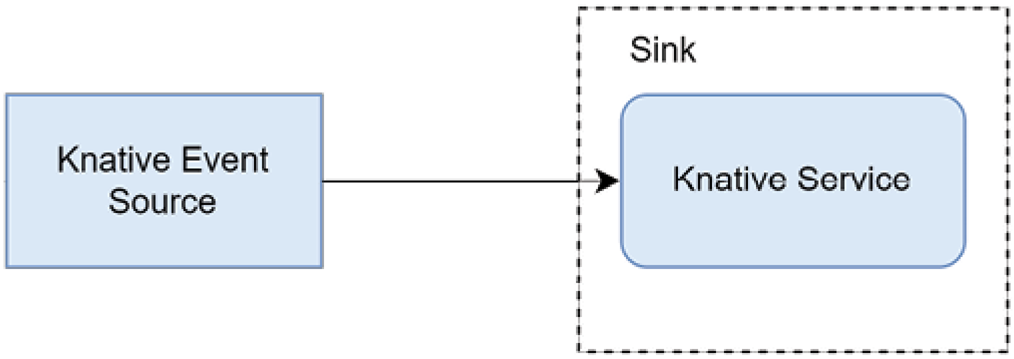


Figure 6.1: Source to Sink pattern

Broker and Trigger pattern

This pattern supports filtering of events. This implies that out of multiple events from broker, it delivers only those events to a subscribed service where it shows interest. As shown in Figure this pattern consists of:

Brokers which is a Kubernetes Custom resource and acts as a sink for event sources. The Brokers in Table 6.2 are supported by Knative:

Table 6.2 : Brokers supported by Knative

subscribe to a broker with filtering configuration on cloud events. They route filtered events to its configured subscription service.

Event source which acts as a link between event producer and broker.

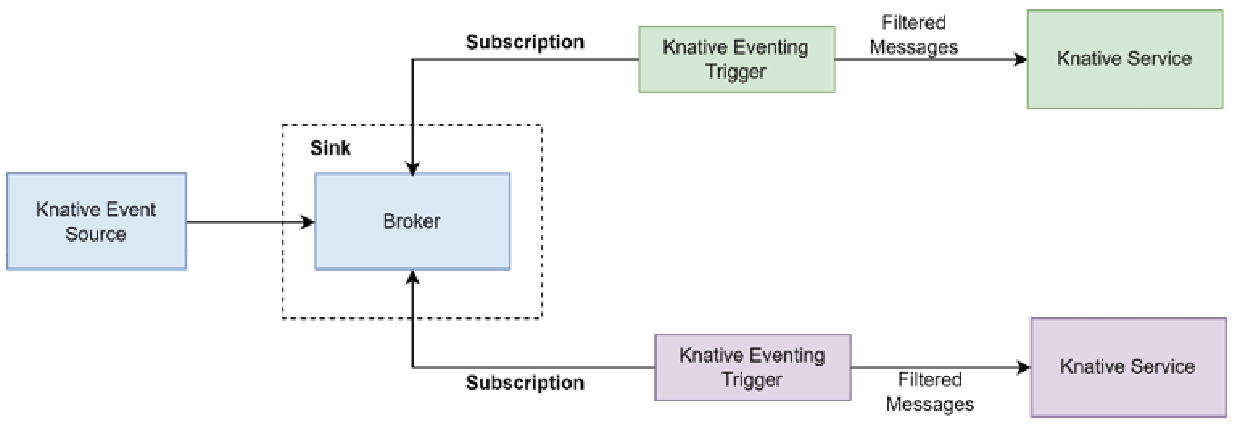


Figure 6.2: Broker and Triggers pattern.

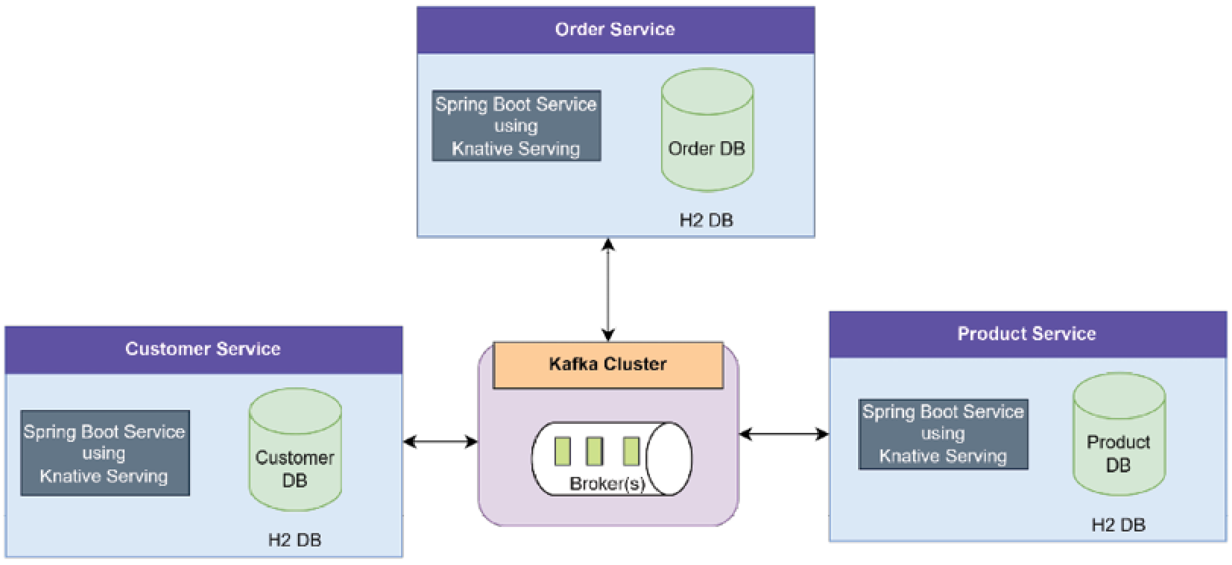
We will enhance our use case order processing system from the previous chapter with Knative Eventing and apply patterns discussed above to understand Eventing concepts better.

Case study – Online order processing system

We already described this case study in the previous chapter. The core services order, product and customer service required to build the system will remain same. From here onwards in each section, we only discuss the changes required to implement this use case using Knative Eventing.

Functional architecture

Figure 6.3 illustrates how the various components of our use case work together. As you can see, we introduced a messaging component Apache Kafka for async communication between services. Events published to this component will be consumed by services of the system for further processing:



Order Service
Spring Boot Service using Knative Serving
Order DB
H2 DB
Customer Service
Spring Boot Service using Knative Serving
Customer DB
H2 DB
Kafka Cluster
Broker(s)
Product Service
Spring Boot Service using Knative Serving
Product DB
H2 DB

Figure 6.3: Order processing system – Functional architecture

While building the final architecture we have taken various architectural decisions. Let us review them:

Micro services: We decided to create all the services on microservices architecture so that it is easier to manage and work independently. We choose Spring Boot as it is a proven framework for microservices development to develop services, along with Java 11 as a development language. We will use Spring cloud functions to publish functions as a service.

Storage: We use In-memory H2 database to store data about orders, customers, and products. Since this is a test use case, we will be using this H2 to store information.

Service communication: We decided to use Apache Kafka for asynchronous communication between various services of the system.

Deployment architecture

Figure 6.4 illustrates the deployment architecture of our components in the Kubernetes cluster. A key change from the previous chapters is the introduction of asynchronous communication between services using Apache Kafka:

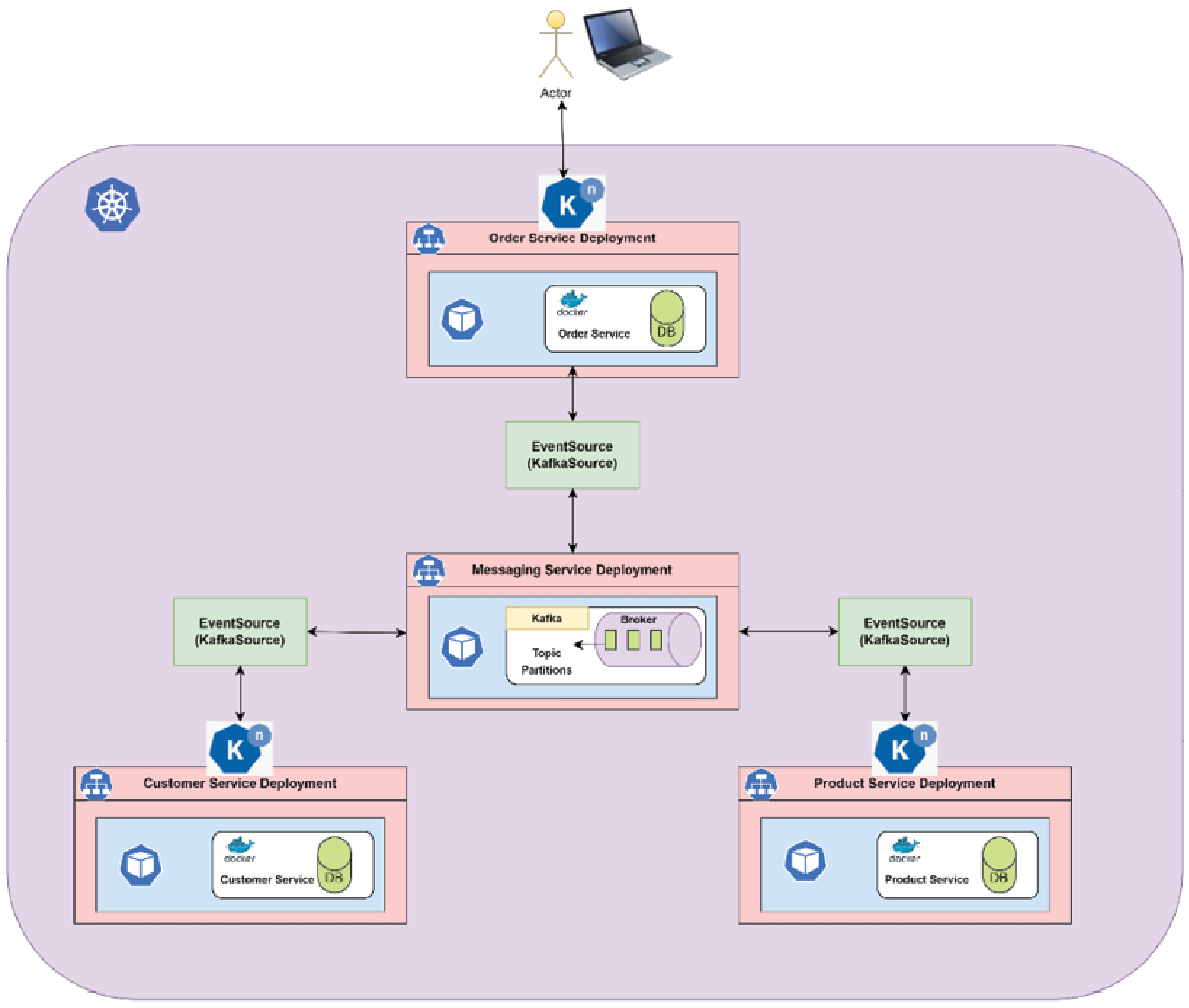


Figure 6.4: Order processing system – Deployment architecture

We will discuss about components which are added to our use case for Knative Eventing as follows:

Refer Chapter Knative Serving, for detailed description of other components.

Messaging Service deployment: This deployment has Apache Kafka Cluster which contains Brokers as an instance of Kafka and holds partitions of Kafka Topics.

Let us now understand Knative Eventing and its various patterns by applying them in our order processing system use case. As illustrated in previous chapter of Knative Serving, our use case remains same and the services required to implement the system also remain same. There will be changes in communication patterns. In Chapter Knative Serving, our services communicate over HTTP, now with Knative Eventing we will implement the use case with event driven architecture and communication will be done asynchronously using Kafka as a messaging service. We will discuss about changes required at code level in each service to implement system using Knative Eventing.

Order processing system with Source to Sink pattern

Till now, we understood functional and deployment architecture of our services. Let us now start implementing them using Knative Eventing Source to Sink pattern. As per our architectural decisions, we will use Apache Kafka as messaging service and as a communication mechanism between our services. If you have not installed Kafka yet, please refer Apache Kafka Installation steps from Chapter Installation and Configuration of Knative.

Service implementation

Figure 6.5 illustrates the flow of new order using Source to Sink pattern:

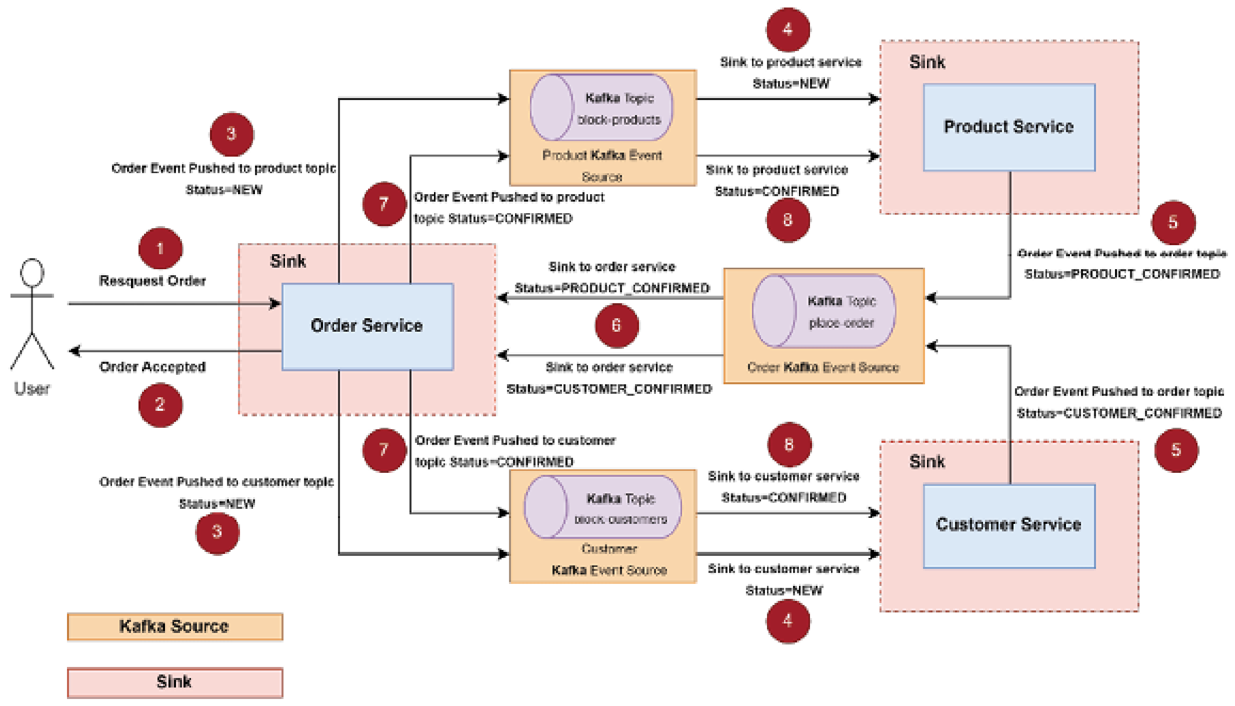


Figure 6.5: Order processing system – Source to Sink pattern

Let us understand the flow first and later we will discuss about required code changes in the services:

User purchases a product which generates a new order request in the system.

Order Service accepts request and responds back to user with request accepted.

Order service then generates CloudEvents and pushes the event to Kafka topics block-products and

The product event source then reads the message from Kafka Topic and delivers this to its configured Sink, product service. Similarly, the customer event source reads the message from Kafka Topic and delivers it to its Sink customer service. We will discuss about these configurations later in the chapter.

Product and customer service does the required processing and pushes the events stating their confirmation in place-order Kafka Topic.

Order event source then reads message from Kafka Topic and delivers this to its configured Sink, Order Service.

Order service once again pushes events to Kafka topics block-products and block-amount confirming order status.

Finally, Product and customer event source reads and delivers messages to its configured Sink, product and customer service respectively.

Refer the book’s Github repository to download or clone the code.

The following changes have been done in services and configuration to implement Knative Eventing using Source to Sink pattern. We have added Spring Cloud Kafka Streams for sending messages to Apache Kafka. The following dependencies are added in pom.xml of all the Services:

org.springframework.cloud

spring-cloud-starter-stream-kafka

dependency>

io.cloudevents

2.3.0

io.cloudevents

cloudevents-kafka

io.cloudevents

cloudevents-http-basic

2.4.2

An OrderSourceToSink class is added in order service which contains a function place() exposed as a service which accepts orders and responds back to user. This function also saves order in database, generates cloud event from the request and pushes the generated event to block-product and block-amount Kafka topics as highlighted in the

```
@Bean

public

Function> place()
 throws JsonMappingException, JsonProcessingException {



}



    LOGGER.info("order-service-es :: Order Request Recieved :: "

     + order.toString());



String orderStatus = order.getOrderStatus();

    LOGGER.info("order-service-es :: Order Status in Request :: "
     + orderStatus);
    CloudEvent event = null;

    try
     {

    event = orderUtil.generateCloudEvent(order);

} catch
 (JsonProcessingException e) {

    LOGGER.error("Error in creating order :: ", e);

}


if(OrderStatus.NEW.toString().equalsIgnoreCase(orderStatus)) {

    orderDetailsRepository.save(order);

    kafkaTemplate.send(TOPIC_NAME_PRODUCT, event);


    kafkaTemplate.send(TOPIC_NAME_CUSTOMER, event);

    return

    new
 ResponseEntity<>
(OrderResponse.builder().status(OrderStatus.IN_PROGRESS.toString()).stat
usMessage(
    "Request Processed Successfully").build(), HttpStatus.ACCEPTED);

}

    return

    new
     ResponseEntity<>
(OrderResponse.builder().status(OrderStatus.IN_PROGRES
S.toString()).statusMessage(
    "Request Processed Successfully").build(), HttpStatus.BAD_REQUEST);

}
```

This class also contains a Consumer Function confirm() which consumes events from product and customer service, processes the events and finally pushes the events with final order status to block-product and block-amount topics as highlighted:

```
@Bean

public

Consumer> confirm(){

 }
```

```
OrderDetails
 orderDetails = msg.getPayload();

LOGGER.info("order-service-es :: Message Recieved  :: "+ orderDetails);

LOGGER.info("order-service-
es :: Order Status  :: "+ orderDetails.getOrderStatus());

Optional order = orderDetailsRepository.findById(orderDetails.getId());

if(order.isPresent()) {

if(OrderStatus.
CUSTOMER_CONFIRMED.toString().equalsIgnoreCase(orderStatus)) {

   order.
get().setCustomerOrderStatus(orderStatus);

   orderDetailsRepository.save(order.
get());

   order.
```

```
get().setOrderStatus(OrderStatus.
IN_PROGRESS.toString());



   order.
get().setCustomerOrderStatus(order.get().getCustomerOrderStatus());



   }

   order = orderDetailsRepository.findById(orderDetails.getId());


if(
OrderStatus.CUSTOMER_CONFIRMED.toString().equalsIgnoreCase(order.
get().getCustomerOrderStatus()) &&

OrderStatus.PRODUCT_CONFIRMED.toString().equalsIgnoreCase(order.ge
t().getProductOrderStatus())) {

   order.
get().setOrderStatus(
OrderStatus.CONFIRMED.toString());


LOGGER.info("Final Order Status :: "
 + order.
get());
```

```
kafkaTemplate.send(TOPIC_NAME_CUSTOMER, orderUtil.generateCloud
Event(order.get(), "call-customer"));



   }
catch
 (
JsonProcessingException
 e) {



LOGGER.error("Error Occurred :: ", e);

   }

  }

 }

}
```

A new class CustomerSourceToSink was added in customer service with the function blockAmount() which consumes Events generated by order service and process request. This function also pushes events to place-order topics consumes by Order Service:

```
@Bean
```

```
public
 Consumer> blockAmount(){



 }



  OrderDetails orderDetails = msg !=
null
 ? msg.getPayload() :

null;



  LOGGER.info(
"Customer-service-es :: Order Status"
 +  orderDetails.getOrderStatus());

if(orderDetails != null) {

        CustomerDetails customerDetails = customerDetailsRepository.findByI
d(orderDetails.getCustId()).orElseThrow();



            customerDetails.setWalletAmountBlocked(customerDetails.getWallet
AmountBlocked() + orderDetails.getAmount());
```

```
            customerDetails.setWalletAmount(customerDetails.getWalletAmount
() - orderDetails.getAmount());

            orderDetails.setOrderStatus(CustomerOrderStatus.CUSTOMER_CO
NFIRMED.toString());

            customerDetailsRepository.save(customerDetails);



try
 {



   }
catch
 (JsonProcessingException e) {



   }



            customerDetails.setWalletAmountBlocked(customerDetails.getWallet
AmountBlocked() - orderDetails.getAmount());

            customerDetailsRepository.save(customerDetails);

        }

        LOGGER.info(
"Customer-service-es :: Order details :: "
 + customerDetails);


        }




 }
```

Similarly, a new class ProductSourceToSink was added in Product Service with the function block() which consumes Events generated by order service and process request. This function also pushes events to place-order topics consumed by Order Service:

```
@Bean


public
 Consumer> block(){




 }




  OrderDetails orderDetails = msg !=
null
 ? msg.getPayload() :
null;




  LOGGER.info(
"product-service-es :: Order Status :: "
```

```
 +  orderDetails.getOrderStatus());




        ProductDetails productDetails = productDetailsRepository.findById(ord
erDetails.getProdId()).orElseThrow();



            productDetails.setProductBlocked(productDetails.getProductBlocked
() + orderDetails.getProdCount());

            productDetails.setProductAvailable(productDetails.getProductAvaila
ble() - orderDetails.getProdCount());

            orderDetails.setOrderStatus(ProductOrderStatus.PRODUCT_CONFI
RMED.toString());

            productDetailsRepository.save(productDetails);

try
 {



   }
catch
 (JsonProcessingException e) {
    }



            productDetails.setProductBlocked(productDetails.getProductBlocked
() - orderDetails.getProdCount());

            productDetailsRepository.save(productDetails);

        }



       }

    }
```

Let us now deploy of our services on cluster:

Build and deploy services

Please refer Chapter Knative Serving, for steps to build services. As mentioned there, build all the three services using Maven and Docker. Upon successful build and pushing image to docker repository, let us deploy our services.

Deploy order service: Following command will deploy order service with minimum one pod running on Kubernetes Cluster:

```
--image docker.io/knativedemo/order-service-eventing \

--port 8080 \

-a autoscaling.knative.dev/min-scale=1
```

Following is the expected output:

```
http://order-service-eventing.default.127.0.0.1.sslip.io
```

Deploy product Following command will deploy product service with minimum one pod running on Kubernetes Cluster.

```
--image docker.io/knativedemo/product-service-eventing \

--port 8080 \

-a autoscaling.knative.dev/min-scale=1
```

Following is the expected output:

```
http://product-service-eventing.default.127.0.0.1
.sslip.io
```

Deploy customer Following command will deploy customer service on Kubernetes Cluster:

```
--image docker.io/knativedemo/customer-service-eventing \

--port 8080 \

-a autoscaling.knative.dev/min-scale=1
```

Following is the expected output:

http://customer-service-eventing.default.127.0.0.1.sslip.io

Following command will list services created along with its routes and state:

After successful deployment of services, we need to create Apache Kafka Event Source.

Event source installation

Since we are using Apache Kafka, we will first install Apcahe Kafka Source which will read messages from Apache Kafka topics and send them to its configured sink. Following are the installation steps:

Install Kafka Source Controller using the following

Install Kafka Source data the following

Verify the successful installation by entering the following command:

Output must contain kafka-contoller and kafka-source-dispatcher pods running along with other pods:

kafka-controller-749dd9cfc5-jb2pr

1/1

Running

0

4d21h

After successful installation of Kafka Event Source, let us create event sources required for our services.

Create event sources

Now we will create event sources required for our System. We will create following three event sources:

Product Event Kafka reading the messages from block-product topic and delivering events to its configured sink URI /products/block as illustrated in Figure

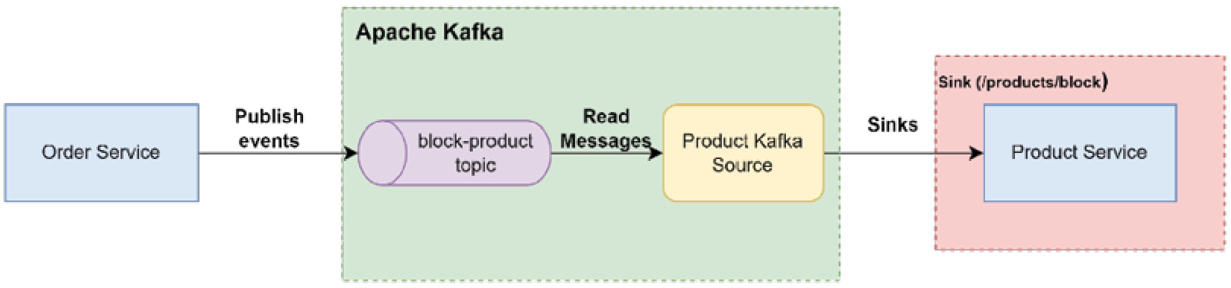


Figure 6.6: Product event Kafka Source to Sink

To create the Product Kafka Source, the following YAML file should be created. This will create block-products topic along with Product Event source with name You can also create your own topic separately and provide details of it in YAML file. This YAML will also have details about Kafka Bootstrap servers (Refer Chapter Installation and Configuration of Knative, for Kafka Bootstrap servers) along with Sink, product service with sink URI

apiVersion: sources.knative.dev/v1beta1

kind: KafkaSource

```
metadata:

name: order-product-event-source
spec:
bootstrapServers:

topics:
  - block-products
sink:
ref:
apiVersion: serving.knative.dev/v1
kind: Service
name: product-service-eventing
uri: /products/block
```

Apply the YAML by entering following command:

Following output is expected on successful completion of command:

Customer Event Kafka Source reading the messages from block-customers topic and delivering events to its configured sink URI /products/blockAmount as illustrated in Figure

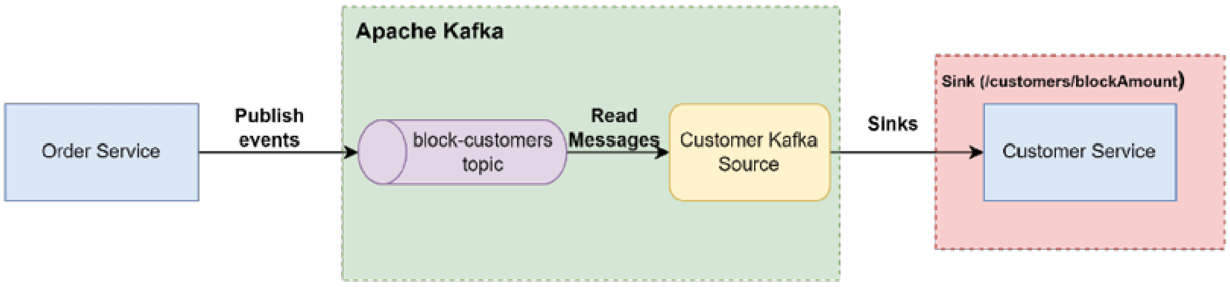


Figure 6.7: Customer Event Kafka Source and Sink

To create the Customer Kafka Source, the following YAML file will be used:

```
apiVersion: sources.knative.dev/v1beta1

kind: KafkaSource

metadata:

name: order-customer-event-source

spec:
```

bootstrapServers:

topics
:

sink:

ref:

apiVersion: serving.knative.dev/v1

kind: Service

name: customer-service-eventing

Apply the YAML by entering following command:

Following output is expected on successful completion of command:

kafkasource.sources.knative.dev/order-customer-event-source created

Order Event Kafka reading the messages from place-orders topic and delivering events to its configured sink URI illustrated in Figure

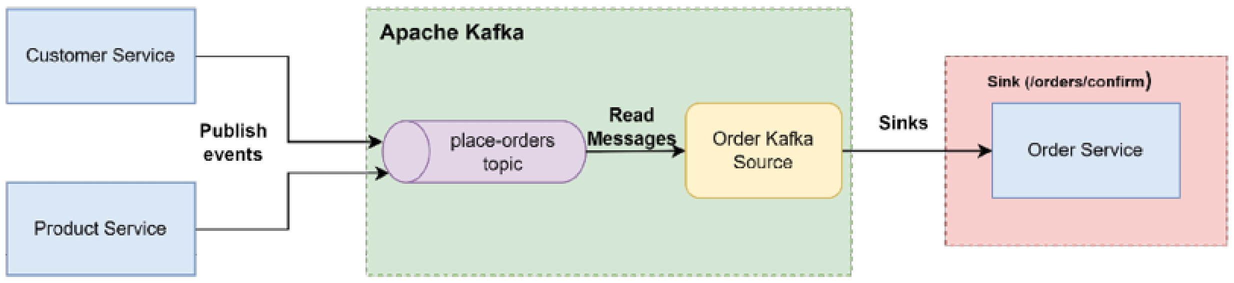


Figure 6.8: Order Event Kafka Source and Sink

Following YAML file will be used to create Order Event Source:

```
apiVersion: sources.knative.dev/v1beta1

kind: KafkaSource

metadata:

name: order-event-source

spec:

bootstrapServers:



topics:
```

```
sink:

ref:

apiVersion: serving.knative.dev/v1

kind: Service

name: order-service-eventing

uri: /orders/confirm
```

Apply the YAML by entering the following command:

Following output is expected on successful completion of command:

```
kafkasource.sources.knative.dev/order-event-source created
```

You can verify all the event sources created by entering following command:

This will give you the list of Event Sources created and their state, type and the configured Sink details:

order-customer-event-source

KafkaSource

kafkasources.sources.knative.dev

ksvc:customer-service-eventing

True

order-product-event-source

KafkaSource

kafkasources.sources.knative.dev

ksvc:product-service-eventing

True

You can also use kubectl to list sources by entering following command:

This command also lists sources with their state and topics from which it reads messages:

```
kafkasource.sources.knative.dev/order-customer-event-source   ["block-
customers"]   ["my-cluster-kafka-bootstrap.kafka:9092"]   True           2d12h
```

```
kafkasource.sources.knative.dev/order-product-event-source    ["block-
products"]    ["my-cluster-kafka-bootstrap.kafka:9092"]   True           8m28s
```

Till now, we have successfully built and deployed our services, we also created event sources. It is time to validate our services and observe runtime behaviour.

Validate services

Now we will validate our services by doing a simple demonstration of how our services worked together to complete the order requested by a customer. We will check logs generated against of each of the service.

Let us start by creating a new order request using our order service REST Endpoint http://order-service-eventing.127.0.0.1.sslip.io/orders/place with following request:

```
{
   "id": 1,
   "custId": 1,
   "prodId": 1,
   "amount": 100,
   "prodCount": 1
}
```

Following curl command executes Order Service, and we can see the following response retuned along with statusCodeValue as

```
-H 'Content-Type:application/json' \
```

Let us go through the following logs generated by Customer and Product Service:

```
INFO

1

---

[nio-8080-exec-9]

c.k.es.customer.CustomerSourceToSink     : Customer-service-
es :: Order Status ::

NEW

INFO

1
```

```
---

[io-8080-exec-10]



c.k.es.customer.CustomerSourceToSink     : Customer-service-
es :: Order Details ::

OrderDetails(id=5,

custId=1,

amount=100,

orderStatus=CONFIRMED)



INFO

1

---

[io-8080-exec-10]

c.k.es.customer.CustomerSourceToSink     : Customer-service-
es :: Order details ::

CustomerDetails(id=1,

custName=Cust1,
walletAmount=49900,

walletAmountBlocked=0)

INFO

1

---

[nio-8080-exec-5]

c.k.es.product.ProductSourceToSink       : product-service-
es :: Order Status ::

NEW

INFO

1

---

[nio-8080-exec-8]

c.k.es.product.ProductSourceToSink       : product-service-
es :: Order Recieved ::
```

```
OrderDetails(id=5,

prodId=1,

prodCount=1,

orderStatus=CONFIRMED)


INFO

1

---

[nio-8080-exec-8]

c.k.es.product.ProductSourceToSink       : product-service-
es :: order details ::

ProductDetails(id=1,

prodName=Product1,

productAvailable=999,

productBlocked=0)
```

In both the logs, we can observe initial and final order status provided by order service to both Customer and Product Service.

Now, let us go through logs generated by Order Service. We can see new order request received by order service and finally the confirmation received from product service as PRODUCT_CONFIRMED and customer service as CUSTOMER_CONFIRMED to update final status of order as

```
INFO

1

---

[nio-8080-exec-4]

com.knative.es.order.OrderSourceToSink   : order-service-
es :: Order Request Recieved ::

OrderDetails(id=5,

custId=1,

prodId=1,

amount=100,

prodCount=1,
productOrderStatus=null,

customerOrderStatus=null,

orderStatus=null)



INFO

1

---

[nio-8080-exec-5]

com.knative.es.order.OrderSourceToSink   : order-service-
es :: Message Recieved  ::

OrderDetails(id=5,

custId=null,

prodId=1,

amount=null,

prodCount=1,

productOrderStatus=null,
customerOrderStatus=null,

orderStatus=PRODUCT_CONFIRMED)



INFO

1

---

[nio-8080-exec-6]

com.knative.es.order.OrderSourceToSink   : order-service-
es :: Message Recieved  ::

OrderDetails(id=5,

custId=1,

prodId=null,

amount=100,

prodCount=null,

productOrderStatus=null,



customerOrderStatus=null,
orderStatus=CUSTOMER_CONFIRMED)

INFO

1

---

[nio-8080-exec-6]

com.knative.es.order.OrderSourceToSink   : Final Order Status ::

OrderDetails(id=5,

custId=1,

prodId=1,

amount=100,

prodCount=1,

productOrderStatus=PRODUCT_CONFIRMED,

customerOrderStatus=CUSTOMER_CONFIRMED,


orderStatus=CONFIRMED)
```

Now let us see the messages produced at different topics:

block-products topic is associated with Product Event source and the following messages is produced when the request is processed. The first one is for new order details and second is final order status message published by Order Service.

```
kubectl -n kafka run kafka-consumer -ti --image=strimzi/kafka:0.14.0-kafka-2.3.0 --rm=true --restart=Never -- bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9092 --topic block-products
```

```
If you don't see a command prompt, try pressing enter.

{"id":1,"custId":1,"prodId":1,"amount":100,"prodCount":1,"productOrderStatus":null,"customerOrderStatus":null,"orderStatus":"NEW"}

{"id":1,"custId":1,"prodId":1,"amount":100,"prodCount":1,"productOrderStatus":"PRODUCT_CONFIRMED","customerOrderStatus":"CUSTOMER_CONFIRMED","orderStatus":"CONFIRMED"}
```

block-customers topic is associated with Customer Event source and here too we can see new order details and final order status events published by Order

```
kubectl -n kafka run kafka-consumer -ti --image=strimzi/kafka:0.14.0-kafka-2.3.0 --rm=true --restart=Never -- bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9092 --topic block-customers
```

```
If you don't see a command prompt, try pressing enter.

{"id":1,"custId":1,"prodId":1,"amount":100,"prodCount":1,"productOrder
Status":null,"customerOrderStatus":null,"orderStatus":"NEW"}

{"id":1,"custId":1,"prodId":1,"amount":100,"prodCount":1,"productOrder
Status":"PRODUCT_CONFIRMED","customerOrderStatus":"CUSTOM
ER_CONFIRMED","orderStatus":"CONFIRMED"}
```

place-order topic is associated with Order Event Source. As we can see in following output, there are two messages published to this topic, first one by product service to confirm order and second by customer service to confirm the order:

```
kubectl -n kafka run kafka-consumer -ti --image=strimzi/kafka:0.14.0-
kafka-2.3.0 --rm=true --restart=Never -- bin/kafka-console-consumer.sh --
bootstrap-server my-cluster-kafka-bootstrap:9092 --topic place-order
```

```
If you don't see a command prompt, try pressing enter.

{"id":1,"custId":1,"amount":100,"orderStatus":"CUSTOMER_CONFIRM
ED"}

{"id":1,"prodId":1,"prodCount":1,"orderStatus":"PRODUCT_CONFIRM
ED"}
```

So, that is the complete validation of our Use Case using Knative Eventing with Source to Sink pattern. Here we observed, how our Kafka

Sources reads messages from Kafka topics and delivers them to its configured Sink. In the next section, we will implement and validate same use case using Broker and Triggers Pattern.

Order processing system with Broker and Triggers Pattern

To understand Source to Sink pattern, let us now implement our case study using Knative Eventing Broker and Trigger pattern.

Service implementation

Figure 6.9 illustrates the flow of new order using Broker and Trigger pattern:

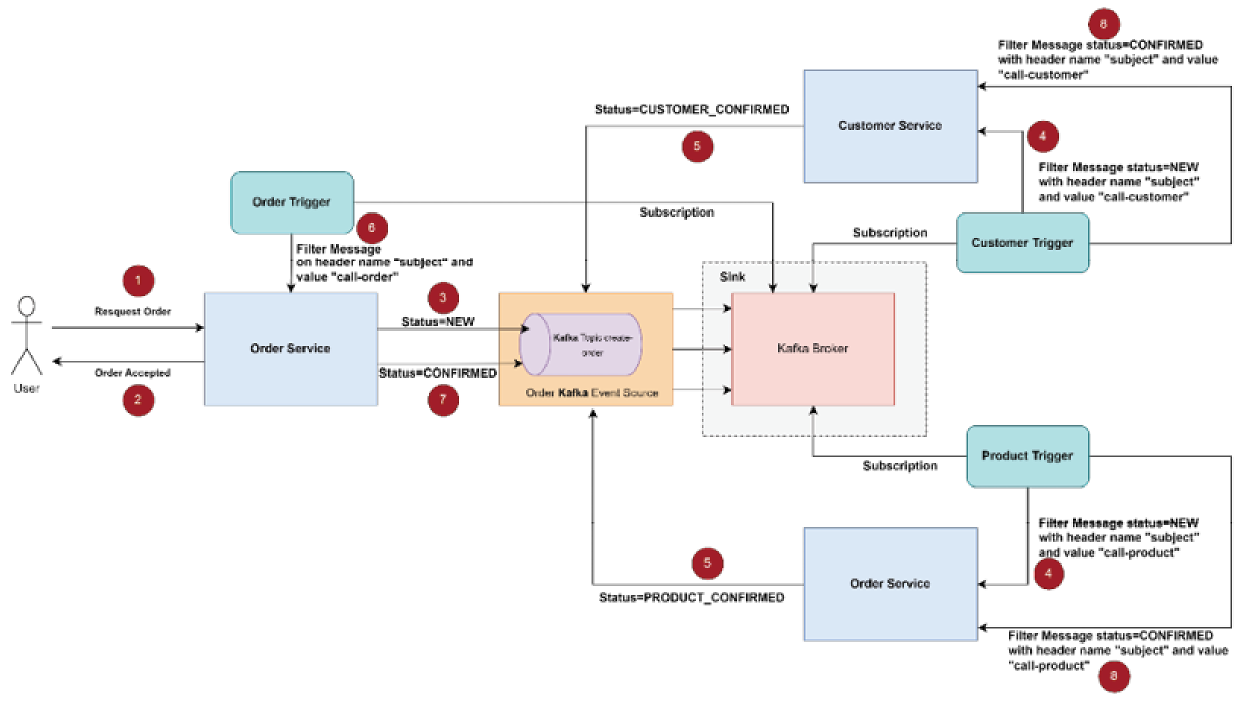

8
Filter Message status=CONFIRMED with header name "subject" and value "call-customer"
Status=CUSTOMER_CONFIRMED
Customer Service
5
4
Filter Message status=NEW with header name "subject" and value "call-customer"
Order Trigger
Subscription
Subscription
Customer Trigger
6
Filter Message on header name "subject" and value "call-order"
Sink
1
Request Order
User
Order Accepted
2
Order Service
3
Status=NEW
Status=CONFIRMED
7
Kafka Topic create-order
Order Kafka Event Source
Kafka Broker
Product Trigger
Subscription
Filter Message status=NEW with header name "subject" and value "call-product"
4
5
Status=PRODUCT_CONFIRMED
Order Service
Filter Message status=CONFIRMED with header name "subject" and value "call-product"
8

Figure 6.9: Order processing system – Broker and Triggers Pattern

Let us understand the flow and later we will discuss about required code changes in the services:

User purchases a product which generates a new order request in the system.

Order service accepts request and responds back to user with request accepted.

Order service then generates CloudEventspushed two events to Kafka topics create-orders with value and Refer https://cloudevents.io/ to learn more about CloudEvents and its

Order event source then reads the message from Kafka Topic and delivers this to its configured Sink, Kafka Broker. We will discuss about these configurations later in the chapter.

Product events using ce-subject header value as call-product and delivers them to its subscriber reference product service for processing. Similarly, Customer events with ce-subject header value as call-customer and delivers them to its subscriber reference customer service for further processing.

Customer and product service then publish events again to create-order Kafka Topic and deliver this to its configured Sink, Kafka Broker.

Order Triggers filters events with ce-subject header value as call-order and delivers them to its subscriber reference order service for further processing. After processing, order service once again pushed events to Kafka topics create-order confirming order status.

Finally, Product Triggers and Customer Triggers once again filter events based on ce-subject header and deliver them to its subscriber references Product and customer services respectively for further processing.

Refer to the book’s Github repository to download or clone the code.

The following changes have been done in services and configuration to implement Knative Eventing with Broker and Triggers pattern:

An OrderBrokerTrigger class added in order service which contains a function create() exposed as a service. It accepts orders and responds back to user. This function also saves orders in database, generates CloudEvents for Product and customer service with ce-subject header value as call-product and call-customer respectively and pushes the generated event to create-order Kafka topics as

```
@Bean

public

Function> create() throws JsonMappingException, JsonProcessingException
 {



 }

doCreateOrder(OrderDetails order){

  LOGGER.info(
"order-service-es :: Order Request Recieved BT :: "
 + order.toString());



  String orderStatus = order.getOrderStatus();




if(OrderStatus.NEW.toString().equalsIgnoreCase(orderStatus)) {
   orderDetailsRepository.save(order);

try
 {

kafkaTemplate.send(TOPIC_NAME_BT, orderUtil.generateCloudEvent(orde
r, "call-customer"));

    kafkaTemplate.send(TOPIC_NAME_BT, orderUtil.generateCloudEvent(or
der,
"call-product"));



    LOGGER.error(
"Error while pushing message in topic :: ", e);

   }

return

new
 ResponseEntity<>
(OrderResponse.builder().status(OrderStatus.IN_PROGRESS.toString())



        .build(), HttpStatus.ACCEPTED);

  }
```

```
return

new
 ResponseEntity<>
(OrderResponse.builder().status(OrderStatus.IN_PROGRESS.toString())



       .build(), HttpStatus.BAD_REQUEST);

 }
```

This class also contains a Consumer Function confirmOrder() which consumes events from Product and customer service, processes the events and finally pushes the events with final order status again to create-header topics as highlighted:

```
@Bean

public

Consumer
Object>> confirmOrder(){




 }

private
 void doConfrimOrder(
```

Message