

Continuous Delivery for Kubernetes



Mauricio Salatino



MEAP Edition
Manning Early Access Program
Continuous Delivery for Kubernetes
Version 6

Copyright 2022 Manning Publications

For more information on this and other Manning titles go to
manning.com

welcome

Thanks for purchasing the MEAP for *Continuous Delivery for Kubernetes*. To get the most of this book, you'll need to feel comfortable with the basics of Kubernetes and Containers. Although the book doesn't focus on any specific programming language, and teaching is transferable between languages, examples are based on Java and Go.

When I started working with Kubernetes, it was a new container orchestration system and the entire community was learning the ropes. At the time, there weren't many tools available to build, deploy and monitor cloud-native applications; users needed to do a lot of tasks by hand or creating custom scripts. Nowadays, the ecosystem is vibrant, and choosing from a myriad of tools available is hard and time-consuming. This book focuses on highlighting some of the most common challenges that you will face when building applications that will run on top of Kubernetes. Because I consider myself a forever learning software engineer (aka a developer), these tools are evaluated through the lens of practitioners and architects that are tasked to release software in a continuous fashion.

The tools covered in this book have been chosen based on their adoption, their relevance in the Kubernetes community, and their way of solving particular challenges that are inherent to the Cloud. This uses Kubernetes as the base layer which users can rely on and none of the tools included in the book relies on Cloud-Provider-specific services, which makes this book a good guide for planning and implementing applications that can be deployed into multiple Cloud Providers.

I hope that the projects covered in this book give enough guidance to tackle medium to advanced challenges and help you evaluate other projects available in the Cloud Native landscape. I encourage you to join the Open Source communities for the projects that you feel most passionate about; there is a constant stream of experience and valuable wisdom that you find there. At the end of the day, these projects are where the innovation is happening.

Please be sure to post any questions, comments, or suggestions you have about the book in the [liveBook discussion forum](#). Your feedback is essential in developing the best book possible.

—Mauricio “Salaboy” Salatino

brief contents

- 1 Cloud-Native Continuous Delivery*
- 2 Cloud-Native Application Challenges*
- 3 Service and Environment Pipelines*
- 4 Multi-Cloud Infrastructure*
- 5 Release Strategies*
- 6 Events for Cloud-Native Integrations*
- 7 Functions for Kubernetes*
- 8 Building a Platform designed for CD*
- 9 The road to Continuous & Progressive Delivery*

1

Cloud-Native Continuous Delivery

This chapter covers

- An introduction to Cloud-Native Applications, how they are architected and how they are related to Kubernetes
- An introduction to the goal of Continuous Delivery practices: Why these practices essential to deliver a continuous stream of value to your end-users
- Introduces a “walking skeleton” project that will be used throughout the chapters and explains how the rest of the book works

Building modern Cloud-Native applications is hard. You end up building highly complex distributed applications that are continuously evolving on top of a tech stack that is continuously changing. Delivering software has always been challenging; delivering software efficiently and reliably is still considered a holy grail by many. In today’s world, how fast you deliver new features to your users/customers can become a real differentiator from your competition; hence it is becoming a priority for companies from all industries to change the way they work, how they organise teams and how they architect and deliver software.

This book focuses on applying Continuous Delivery practices to modern Cloud-Native environments using Kubernetes as the target platform. This book aims to be very practical in showing how you can use these tools with the primary goal of delivering software reliably and efficiently.

To make sure that the practices and tools are efficient, and you are continuously improving, you first need to measure how things are currently working. This book relies on the research presented by another great book called *Accelerate* (<https://www.amazon.com/Accelerate-Software-Performing-Technology-Organizations/dp/1942788339>), where a statistical analysis has been done to understand which key metrics help organizations to improve their performance.

The tools introduced in this book have been chosen to solve specific challenges that you will face when building Cloud-Native applications. Some of these tools solve very technical and

architectural challenges. Some cover how teams will improve collaboration. But all the tools presented have a shared goal: “help you to deliver robust and reliable software to your customers”.

This first chapter covers some basic concepts and definitions that are used throughout this book, giving you context about where the tools presented in the following chapters can be useful and how they were designed. Before we can deep dive into the specifics of Cloud-Native Continuous Delivery, let’s get started checking where you are in your Cloud-Native journey.

1.1 Are you Cloud-Native?

Let’s go straight to the point, you will see a lot about Kubernetes in this book but you need to understand that you can implement Cloud-Native applications without using Kubernetes. Similarly, you can apply Continuous Delivery practices without Kubernetes. Still, this book aims to deliver a practical experience on a real technology stack that is widely available today, so the reader can experience the advantages of Continuous Delivery first hand.

Let’s get the definitions out of the way, Cloud-Native is a very overloaded term, and while you shouldn’t worry too much about it, it is essential to understand why this book makes use of concrete tools that run on top of Kubernetes using containers.

A good definition of the term can be found in VMWare site by Joe Beda (Co-Founder, Kubernetes and Principal Engineer, VMware) <https://tanzu.vmware.com/cloud-native>

“Cloud-Native is structuring teams, culture, and technology to utilise automation and architectures to manage complexity and unlock velocity.”

As you can see, there is much more than the technology associated with the term *Cloud-Native*. There is a people and culture angle to it, that pushes us to reevaluate how we are building software. This book, while covering technology, will make a lot of references to practices that can speed up the process of creating and delivering Cloud-Native applications.

On the technical side, Cloud-Native applications are heavily influenced by the “12-factor apps” principles (<https://12factor.net>) which were defined to leverage cloud computing infrastructure. These principles were created way before Kubernetes existed and served to establish recommended practices for when building distributed applications. With these principles, you can separate services to be worked by different teams all using the same assumptions on how these services will work and interact with each other. These 12-factors are:

- [I. Codebase](#)
One codebase tracked in revision control, many deploys
- [II. Dependencies](#)
Explicitly declare and isolate dependencies
- [III. Config](#)
Store config in the environment
- [IV. Backing services](#)
Treat backing services as attached resources
- [V. Build, release, run](#)
Strictly separate build and run stages
- [VI. Processes](#)
Execute the app as one or more stateless processes
- [VII. Port binding](#)
Export services via port binding
- [VIII. Concurrency](#)
Scale out via the process model
- [IX. Disposability](#)
Maximize robustness with fast startup and graceful shutdown
- [X. Dev/prod parity](#)
Keep development, staging, and production as similar as possible
- [XI. Logs](#)
Treat logs as event streams
- [XII. Admin processes](#)
Run admin/management tasks as one-off processes

By following these principles, you are aiming to manage and reduce the complexity of building distributed applications, for example, by scoping a smaller and more focused set of functionalities into what is known as a microservice. These principles guide you to build stateless microservices (VI and VIII) that can be scaled by creating new instances (replicas) of the service to handle more load. By having smaller microservices, you end up having more services for your applications. This forces you to have a clear scope for each microservice, where the source code is going to be stored and versioned (I) and its dependencies (II and IV). Having more moving pieces (microservices), you will need to rely on automation to build, test and deploy (V) each service, having a clear strategy becomes a must from day one. Now you need to manage an entire fleet of running services, instead of just one big ship (monolith), which requires you to have visibility on what is going on (XI) and plan accordingly for cases when things go wrong (XII). Finally, to catch

production issues early (X), it is highly recommended to work and regularly performs testing on environments that are as close as possible to your production environment.

The term Cloud-Native is also strongly related to container technologies (such as Docker) as containers by design follow best practices from the “12-factor apps” principles as they were designed with Cloud-Native applications and cloud infrastructures in mind. Once again, you can implement Cloud-Native patterns without using containers. Still, for the sake of simplicity, in this book, Cloud-Native services, “12-factor apps” and microservices are all going to be packaged as containers, and these terms will be used as synonyms.

If you are following the “12-factor apps” principles, you and your teams are going to be building a set of services that have a different lifecycle and can evolve independently. No matter the size of your teams, you will need to organise people and tools around these services.

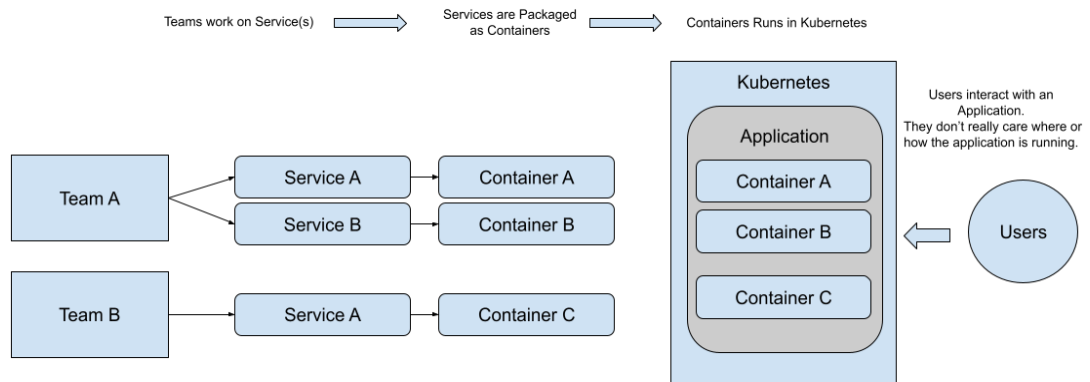


Figure 1.1 Teams, Services, Containers composing applications for end-users

When you have multiple teams working on different services, you will end up with tenths or even hundreds of services. When you have 3 or 4 containers and a set of computers to run them, it is possible to decide where these services will run manually, but when the amount of services grows, and your data centre needs to scale you will need to automate this job. That is precisely the job of a container orchestrator, which will decide for you based on the size and utilisation of your cluster (machines in your datacenter) where your services will run.

The industry already chose Kubernetes to become the defacto standard for containers orchestration. You will find a Kubernetes managed service in every major Cloud-Provider and On-Prem services offered by companies such as Red Hat, VMWare and others.

Kubernetes provides a set of abstractions to deal with a group of computing resources (usually referred to as a cluster, imagine a data center) as a single computer. Dealing with a single computer simplifies the operations as you can rely on Kubernetes to make the right placements of your workloads based on the state of the cluster. Developers can focus on deploying applications, Kubernetes will take the burden of placing them where it is more appropriate.

Kubernetes provides a developer and operations friendly declarative REST API to interact with these abstractions. Developers and Operations can interact with different Kubernetes Clusters by

using a CLI (command-line interface) called `kubectl` or directly calling the REST APIs exposed by each cluster.

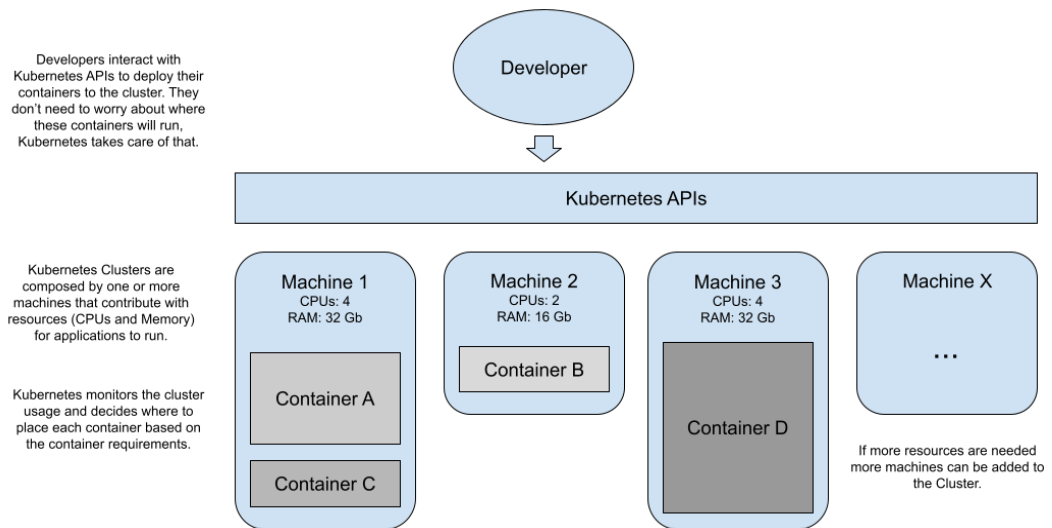


Figure 1.2 Kubernetes lets you focus on running your services.

Kubernetes is in charge of deciding where your containers will run (in which specific machine), based on the cluster utilisation and other characteristics that you can tune. Most of the time, as a developer, you are only interested in having your applications running, not where they run.

Both **Kubernetes in Action** and **Core Kubernetes** are highly recommended books if you are interested in learning how to work with Kubernetes and how Kubernetes works internally.

1.1.1 Challenges of delivering Cloud-Native applications

While working with Kubernetes, containers and 12-factor applications you will quickly notice that a lot is going on. If it is your first time in such environments, there are loads of challenges to tackle to even understand how pieces fit together.

From the architecture point of view, your (micro)services now rely on other components that are not co-located in the same machine. These services need to be configured to understand the environment where they are running and these services will need to follow certain conventions and best practices, so they can work together.

These services will be owned and evolved independently by different teams; collaboration and constant communication are needed when changes that might affect other teams and services are going to happen.

To run each of these services (in Kubernetes) now, you need to take into account creating, building and testing containers and Kubernetes YAML files. It is not enough to produce a binary for your application, you now need to consider how your application will be containerized, configured and deployed to a Kubernetes cluster.

Sooner or later, you will realise that there is no Application anymore, now you have a set of independent services that are evolving at a different pace.

Without having the right tests in place, you will spend weeks (including weekends) trying to understand why one of these services is bringing the whole system down.

Without having the right measurements in place you will never know if a change besides adding a feature, is slowing down your entire customer experience or if it is going to cause some downtime in the future.

It is also important to understand that for your users it doesn't really matter if you are using Kubernetes, Containers or even building Cloud-Native applications, all they look for are new and better applications that help them to solve their problems. To make sure that value to the user is always our true north, we rely on Continuous Delivery practices. Continuous Delivery (CD for short), as presented in the Continuous Delivery Book (**a reference to be added**), is a set of practices that helps teams to deliver valuable software to their users in a reliable and efficient way. CD focuses heavily on automation and measuring our delivery processes, to always keep improving. Explaining all of the practices involved in CD is beyond what we want to cover in this chapter; for now, I'll simply explain the **goals** behind CD and why you should care about learning and apply CD to your day to day software practice. In the following section, we will focus on understanding in more detail Continuous Delivery goals and how they relate to Cloud-Native applications. We will go over the practices in the following chapters when we deep dive into some of the tools that were created to help us apply these practices to our projects.

1.2 Continuous Delivery Goals

Delivering valuable software to your customers/users in an efficient way should be your main goal. While building Cloud-Native applications, these become challenging as you are not dealing with a single application, you are now dealing with complex distributed applications and multiple teams delivering features at different paces.

For the remainder of the book, the following goal of Continuous Delivery is going to be used, to guide the selection of different projects and tools for your teams to use:

From the Book Continuous Delivery

"Goal: Deliver useful, working software to users as quickly as possible"

Focus on Reducing cycle time (The time from deciding to make a change to having it available to users)"

This goal definition come from the Continuous Delivery book wrote by Jez Humble and David Farley (2010 https://www.amazon.co.uk/Continuous-Delivery-Deployment-Automation-Addison-Wesley/dp/0321601912/ref=asc_df_0321601912/?tag=googshopuk-21&linkCode=df0&hvadid=310913487979&hvpos=&hvnetw=g&hvrnd=15021799242967420863&hvpone=&hvptwo=&hvqmt=&hvdev=c&hvdvcmdl=&hvlocint=&hvllocphy=9045944&hvtargid=pla-434514065498&pssc=1&th=1&pssc=1). The book, on purpose, doesn't go deep into any technologies besides naming them, and because it was written more than 10 years ago, the Cloud and Kubernetes didn't exist in the way that exists today.

There are some significant areas covered by Humble and Farley's book which you will read about here, such as:

- **Deployment pipeline:** all the steps needed to create and publish the software artifacts for our application's services.
- **Environment Management:** how to create and manage different environments to develop, test, and host the application for our customers/users.
- **Release Management:** the process to verify and validate new releases for your services.
- **Configuration Management:** how to manage configuration changes across environments in an efficient and secure way.

This book, *Cloud-Native Continuous Delivery*, aims to be a practical guide where you can experience the concepts described by Humble and Farley in their book first hand, with simple tools and a working example that you can modify to test different aspects of Continuous Delivery.

To benefit (and have some return of investments) from adopting Kubernetes, re-architecting your applications and running your workloads is not enough. You can only fully leverage Kubernetes design principles if your organisation delivers more and better software to their users faster.

In such a way, the Cloud-Native Continuous Delivery goal can be stated as follows:

“Deliver useful, working software to users as quickly as possible by organising teams to build and deploy in an automated way Cloud-Native applications that run in cloud-agnostic setup.”

This goal implies multiple teams working on different parts of these Cloud-Native applications that can be deployed to different cloud providers to avoid vendor lock-in. It also means the fact that Cloud-Native applications are more complex than old monoliths, but this inherent complexity also unlocks velocity, scalability and resilience if managed correctly.

1.2.1 Are you doing Continuous Delivery already?

Continuous Delivery is all about speeding up the feedback loop from the moment you release something to your users until the team can act on that feedback and implement the change or new feature requested. To be efficient and reliable high-quality software, you need to automate a big part of this process.

I often hear people stating that they are already doing Continuous Delivery; hence this section gives a quick overview about what the remainder of the book will be covering so you can map your current situation with some of these points:

- **Every change needs to trigger the feedback loop:** there are four main things that you need to monitor for changes and verify that these changes are not breaking the application:
 - **Code:** if you change source code that will be built and run, you need to trigger the build, test and release process for every change
 - **Configuration:** if something in the configuration changes, you need to re-test and make sure that these changes broke nothing.
 - **Environment:** if the environment where you run the application changes, you need to re-test and verify that the application is still behaving as expected. Here is where you control and monitor which version of the Operating System are you using, which version of Kubernetes is being used in every node of your cluster, etc.

- **Data structures:** if a data structure changes in your application, you need to verify that the application keeps working as expected, as data represent a very valuable asset, every change needs to be correctly verified. This also involves a process for deciding how backward compatible is the change and how the migration between the old and new structure will work.
- **The feedback loop needs to be fast:** the faster the feedback loop is, the quicker you can act on it, and the smaller the changes are. To make the feedback loop faster, most of the verifications need to be automated by applying a Continuous Integration approach. Usually, you will find the following kind of tests required to verify these changes:
 - **Unit Tests:** at each project/service level, these tests can run fast (under 10 seconds) and verify that the internal logic of the services works. Usually, you avoid contacting databases or external services here, to prevent long-running tests. A developer should run these tests before pushing any changes.
 - **Integration/Component Tests:** these tests take longer as they interact with other components. But verify that these interactions are still working. For these tests, components can be mocked, and this book covers “Consumer Contract Testing” to verify that new versions of the services are not breaking the application when their interfaces (contracts) change.
 - **Acceptance Tests:** verify that the application is doing what it is supposed to do from a business perspective. Usually, this is verified at the service level, instead of at the User Interface level, but there are different techniques to cover different angles. These tests are executed on top of the entire application, this requires a whole environment to be created and configured with the version of the service that includes the new change, and it can take more time to run.
 - **Manual Testing:** This is performed by a team that is going to test the application in an environment similar to production. Ideally, these testers should be testing what the users are going to get. These tests are prolonged, as they require people to go over the application.
- **Everything needs to be measured:** to make sure that we are going in the right direction and you keep delivering high-quality software to your users, you need to track how much time and resources this feedback loop is taking you from start to finish. Here are some key measurements that will help you to understand how good you are. Based on the DORA report about the State of DevOps (<https://services.google.com/fh/files/misc/state-of-devops-2019.pdf>) you can measure:
 - **How frequent your code deployments are:** how often are you deploying new versions to your production environment?
 - **Lead time from committing changes to deploying your service:** how much time does it take you from committing a change to version control to having those changes deployed in your production environment?

- **Time to recover from incidents:** how much time does it take you to fix a service (or a set of services) that are misbehaving since the issue is reported until the system is again in a stable state?
- **Change failure rate:** how often do you deploy new versions that cause problems in the production environment?

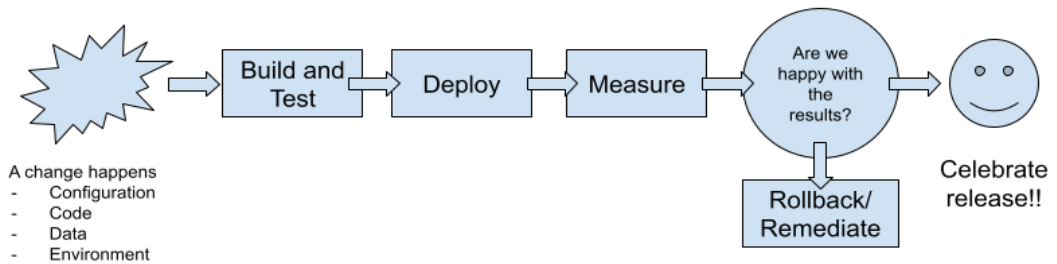


Figure 1.3 Fast feedback loops help you to accelerate your deliveries

There are high chances that you are doing some of these things already, but unless you are measuring, it becomes impossible to assert if your changes are being useful to your users or not.

The main objective of this book is to show how Continuous Delivery can be achieved in a Cloud-Native environment. This helps you to deal with the complex nature of distributed teams working on distributed applications. For this exact same reason, an example is needed, and by example, I don't mean a typical "hello world" for each technology that I will mention in the following chapters.

In order to be convinced that continuous delivery can be achieved for your company or scenario, you need to see an example that you can map almost one to one with your daily challenges. The following section introduces an example that will be used throughout this book. To highlight some tools and frameworks we need more than a simple example, hence we use the term "walking skeleton" which represents a fully functional application that contains enough components and functionality to work end-to-end. A "walking skeleton" is supposed to highlight the defined architecture and how components interact with each other. This "walking skeleton" pushes you to define which frameworks, target platform and which tools are you going to use to deliver your software. Following chapters will deep dive into different characteristics of the walking skeleton, which was created in an Open Source way, for you to run in your own environment and use as a playground for testing new technologies before applying them to your own projects.

1.3 The need for a "walking skeleton"

In the Kubernetes ecosystem, it is common to need at least to integrate 10 or more projects or frameworks in order to deliver a simple PoC (proof of concept). From how you build these projects into containers that can run inside Kubernetes to how to route traffic to the REST endpoints provided in each of these containers. If you want to experiment with new projects to see if they fit into your own ecosystem you end up building a PoC to validate your understanding of how this shiny new project works and how it is going to save your and your teams' time.

For this book, I have created a simple “Walking Skeleton”, which is a Cloud-Native application that goes beyond being a simple PoC and allows you to explore how different architectural patterns can be applied and how different tools and frameworks can be integrated, without the need of changing your own projects for the sake of experimentation.

The main purpose of this walking skeleton is to highlight how to solve very specific challenges from the architectural point of view and from the delivery practices angle. You should be able to map how these challenges are solved in the sample Cloud-Native application to your specific domain. Challenges are not always going to be the same, but I hope to highlight the principles behind each proposed solution and approach taken to guide your own decisions.

With this walking skeleton, you can also figure out what is the minimum viable product that you need and deploy it quickly to a production environment where you can improve from there. By taking the walking skeleton all the way to a production environment you can get valuable insights for what you will need for other services and from an infrastructure perspective. It can also help your teams to understand what it takes to work with these projects and how and where things can go wrong.

The technology stack used to build the “walking skeleton” is not important in my opinion, it is more important to understand how the pieces fit together and what tools and practices can be used to enable each team behind a service (or a set of services) to evolve in a safe and efficient way.

1.3.1 Building a Conference Platform

During this book, you will be working with a Conference Platform application. This conference platform can be deployed in a different environment to serve different conference events when needed. This platform relies on containers, Kubernetes and tools that will work in any major Cloud-Providers as well as On-Prem Kubernetes installations.

This is how the application main page looks like:

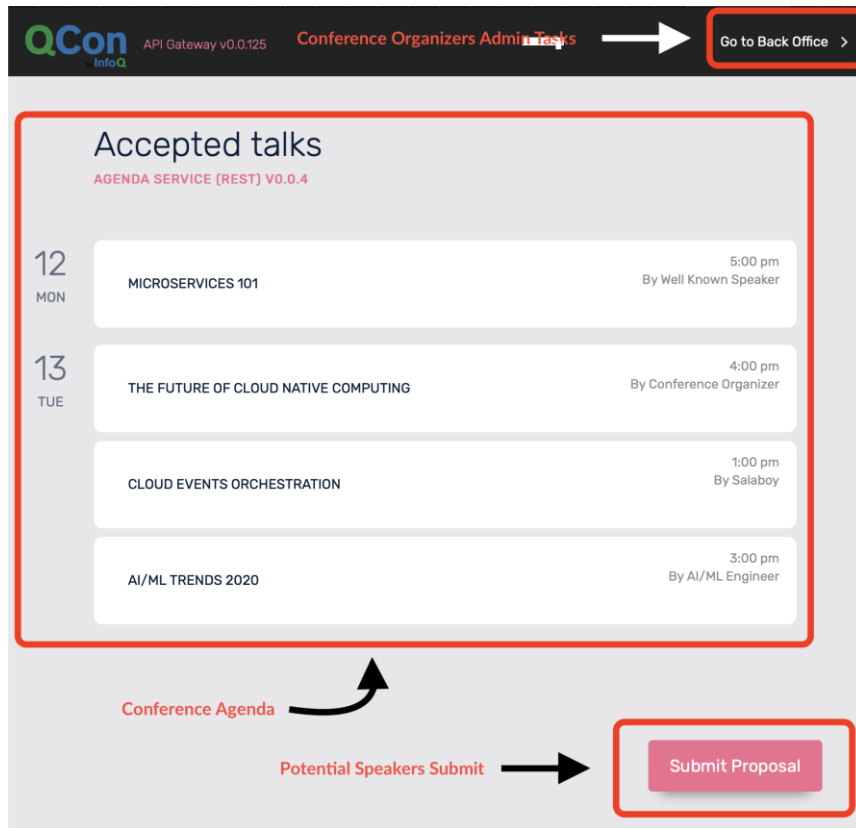


Figure 1.4 Conference Platform Main Site

The conference application lists all the approved submissions on the Agenda Page. The main page will also allow potential speakers to submit proposals while the “Call for Proposals” window is still open. There is also a Back Office section for the organizers to review proposals and do admin tasks while organizing the conference:

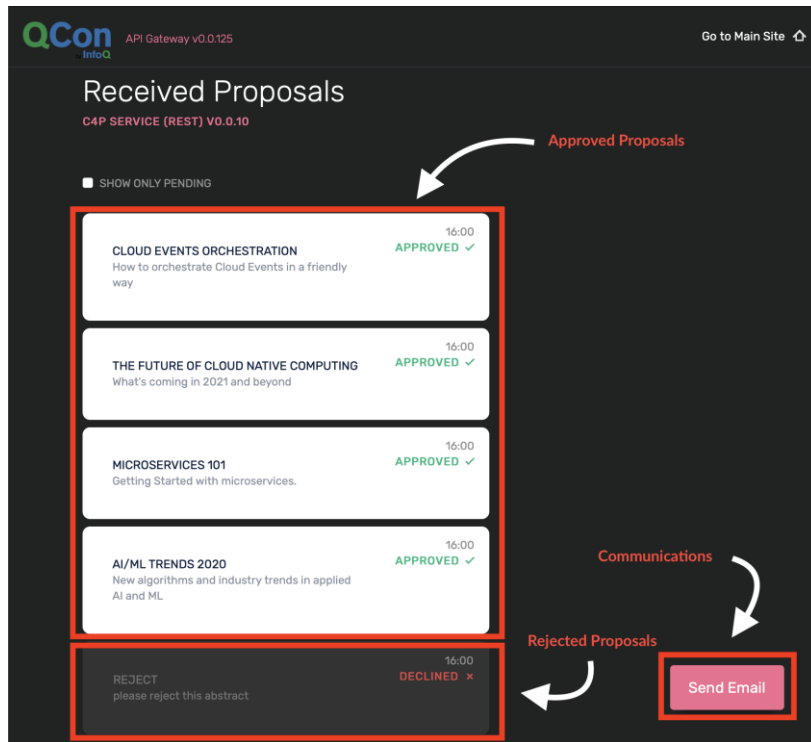


Figure 1.5 Conference Platform Back Office Page

This application is composed of a set of services that have different responsibilities:

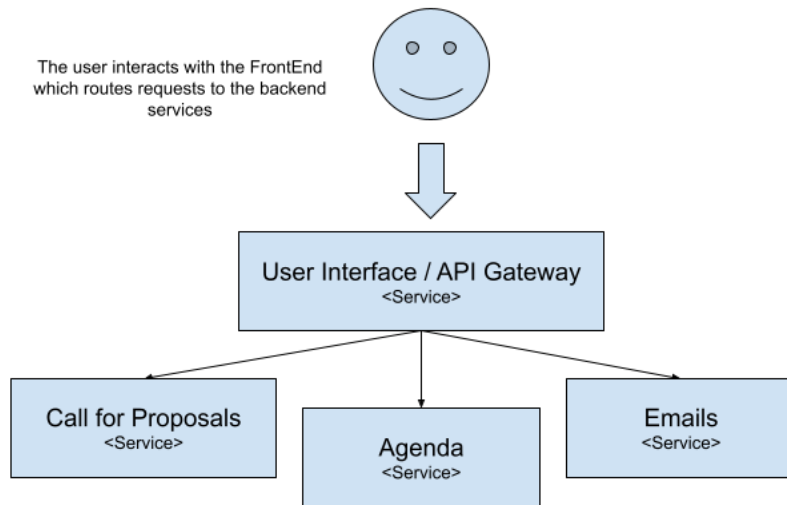


Figure 1.6 Conference Platform Services

Figure 1.6 shows the main components of the application that you control, in other words, the services that you can change and that you are in charge of delivering. These services functionally compose the application, here is a brief description of each service:

- **User Interface/API Gateway:** This service serves as the main entry point for your users to access the application. For this reason, the service hosts the HTML, JavaScript and CSS files that will be downloaded by the client's browser interacting with the application.
- **Agenda Service:** This service deals with listing all the talks that were approved for the conference. This Service needs to be highly available during the conference dates, as the attendees will be hitting this service several times during the day to move between sessions.
- **Email Service:** This service is just a facade exposing rest endpoints to abstract an SMTP email service that needs to be configured in the infrastructure where the application is running.
- **Call for Proposals (C4P):** This service contains the logic to deal with "Call for proposals" use case (C4P for short) when the conference is being organized. As you can see in the following diagram, the C4P service calls both the Agenda and the Email Service, hence these two services are considered "downstream" services from the C4P service perspective.

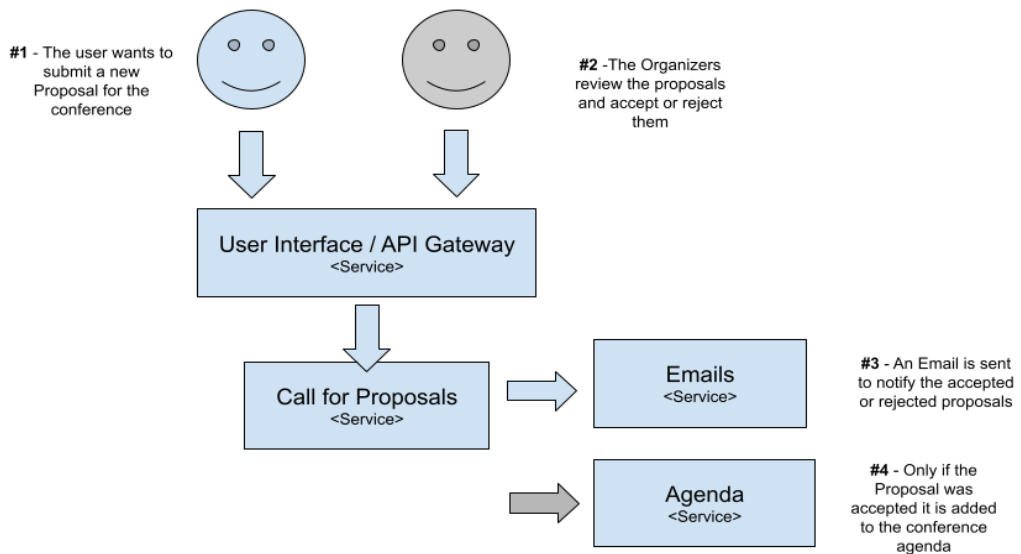


Figure 1.7 Call for Proposals Use Case

It is quite normal in Cloud-Native architectures to expose a single entry point for users to access the application, this is usually achieved using an API Gateway, which is in charge of routing requests to the backend services that are not exposed outside the cluster.

This simple application implements a set of well-defined use cases that are vital for the events to take place such as:

- **Call for Proposals:** potential speakers submit proposals that need to be validated by the conference organizers. If approved, the proposals are published in the conference agenda.
- **Attendee Registration:** attendees need to buy a ticket in order to attend the conference
- **Event Agenda (schedule):** host the approved proposals, times and descriptions.
- **Communications:** Sending organizers, attendees and sponsors emails.

While looking at how these use cases are implemented, you need to consider also how to coordinate across teams when new use cases will be implemented or when changes need to be introduced. For improving collaboration, you need visibility, you need to understand how the platform is working, the last few chapters of this book goes into integrating tools for monitoring, visualizing and orchestrating your services to understand and collect business metrics from your applications.

You also need to take into consideration the operation side of this Cloud-Native application. You can imagine that there will be a period when the application will open the "Call for Proposals" request for potential speakers to submit proposals, then closer to the conference date open the attendee registration page, etc.

During the length of the book, I will encourage you to experiment by adding new services and implementing new use cases. In chapter 2 when you deploy the application to a Kubernetes Cluster

you will inspect how these services are configured to work, how the data is flowing between the different services and how to scale the services.

By playing around with a fictional application, you are free to change each service internals, use different tools and compare results or even have different versions of each service to try in parallel. Each service provides all the resources needed for you to deploy these services to your own environment. In chapter 3, you will go deeper into each service to understand how to change the functionality and the behaviour.

By having this example application up and running you will be able to understand and experience with a concrete example of how to measure your Continuous Delivery practices such as:

- **Every change needs to trigger the feedback loop:** what kind of setup do you need to have in place to trigger these feedback loops? How do you reconcile different service feedback loops and how do you aggregate these changes when they happen? Part 2 of the book covers how to define and implement a Deployment Pipeline.
- **The feedback loop needs to be fast:** where do you test? what to test? and how to stop promoting artifacts when tests go wrong? Part 3 of the book covers how Consumer-Driven Contract testing can help us to answer some of these questions
- **Everything needs to be measured:** what to measure?, when to measure? and how to make sure that new changes are not taking your application in the wrong direction. Part 2 and 3 combined will help us to understand where to collect the right measurements using an event-driven approach to see how we are performing.

Before moving forward to actually deploy these Cloud-Native conference platform, it is important to mention some of the main differences with having all these functionalities bundled up in a single monolithic application.

The Cloud-Native Conference Platform was created based on a monolith application that was serving the same purposes but it had several drawbacks. The monolithic application implemented exactly the same use cases, but it suffered from several drawbacks discussed in the next section.

1.3.2 Differences with a Monolith

Understanding the differences between having a single monolithic application instead of a fully distributed set of services is key to grasp why the increased complexity is worth the effort. If you are still working with monolithic applications that you want to split up to use a distributed approach, this section highlights the main differences that you will encounter between a monolith and a set of services implementing more or less the same functionalities from the end-user perspective.

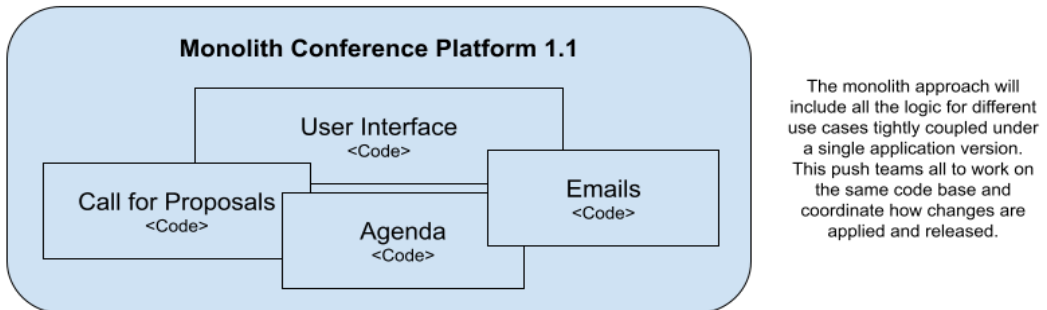


Figure 1.8 The monolith approach

Functionally wise they are the same, you can do the same amount of use cases, but the monolith application presented some drawbacks that you might be experiencing with your monolith applications already, the following points compare the Cloud-Native approach with the previous monolith implementation:

- **Now services can evolve independently, teams are empowered to go faster, there is no bottleneck at the code-base level:** In the monolithic application, there was a single source code repository for different teams to work on, there was a single Continuous Integration pipeline for the project which was slow and teams were using feature branches that caused problems with complex merges.
- **Now the platform can scale differently for different scenarios:** from a scalability perspective, each individual service can be now scaled depending on the load level that they experience. With the monolith application, the operations team could only create new instances of the entire application if they needed to scale just a single functionality. Having fine-grained control on how different functionalities can be scaled can be a big differentiator for your use case, but you need to do your due diligence.
- **The Cloud-Native version is much more complex, as it is a distributed system:** but it leverages the flexibility and characteristics of the cloud infrastructure much better, allowing the operation teams to use tools to manage this complexity and the day to day operations. When building monolithic applications it was much more common to create your in-house mechanism to operate your large applications. In Cloud-Native environments, there are a lot of tools provided by Cloud Providers and Open Source projects that can be used to operate and monitor Cloud-Native applications.
- **Each service can be built using a different programming language or different frameworks:** with the monolith, application developers were stuck in old versions of libraries, as changing or upgrading a library usually involved large refactorings and the whole application needed to be tested to guarantee that the application will not break. When working in a Cloud-Native approach, services are not forced to use a single technology stack. This allowed teams to be more autonomous in choosing their own tools which for some situations can speed up the delivery times.
- **All or nothing with the monolith:** if the monolith application went down, the entire application was down and users would not be able to access anything. With the Cloud-Native version of the application, users can still access the application even if services are going down. The example “walking skeleton” shows how to support degraded services by adopting some popular tools. By using Kubernetes, which was designed to monitor your services and take actions in the case a service is misbehaving, the platform will try to self-heal your applications.
- **Each conference event required a different version of the monolith:** when dealing with different conference events, each conference required a version of the monolith that was slightly different from the other events, this caused divergence of codebases and duplication of the entire project. Most of the changes done for a conference were lost when the event was done. In the Cloud-Native approach, we promote reusability by having fine-grained services that can be swapped avoiding duplication of the whole platform.

While the monolith application is much more simple to operate and develop than the Cloud-Native application, the remainder of the book focuses on showing you how to deal with the complexity of building a distributed application. We'll do that by looking at how to adopt the right tools and practices, which will unlock your teams to be more independent and efficient while promoting resiliency and robustness to your applications.

1.4 How the rest of the book works

In this book, you will be working to make sure that you maximize the value that is being delivered by the teams working on the different services of the application while reducing the cycle time, by using the right tools for the job.

While this book chose to exemplify some concrete tools, each tool will be dissected to highlight where the value/usefulness and concepts that each tool was built upon. Different tools can be used to achieve the same results, but I prefer to show a concrete tool working than just concepts floating in the air.

This book takes you on a journey where different tools will be applied to our walking skeleton to improve the application architecture and how changes can be efficiently delivered and measured.

- **Chapter 2: Cloud Native Application Challenges:** This chapter deploys the walking skeleton to a Kubernetes Cluster and explores its services and how they connect to each other. This chapter describes how to use Helm, a package manager for Kubernetes to package, distribute and install our applications into a cluster. Once the basics of the walking-skeleton are understood and you can interact with the application the chapter goes into discussing all the challenges presented by this simple application. By understanding the challenges early on in the process of developing the application, we can plan and experiment with different tools to mitigate or reduce the impact of these challenges.
- **Chapter 3: Service and Environment Pipelines:** While the previous chapter describes the challenges of the application running, we need to understand how each service of the application is built, released and published so the application can be installed in local and remote Kubernetes Clusters. This chapter also covers how we can manage different environments and their configurations so our applications can be developed, tested and hosted for our users in a reliable and reproducible way. This chapter uses Tekton as an example of a CI/CD Pipeline Engine that can help us to automate the building, releasing and deployment of our Cloud Native applications. This chapter also introduces other tools like Jenkins X which aims to solve most of our CI/CD needs for Kubernetes.
- **Chapter 4: Multi-Cloud Infrastructure:** This chapter takes a stab at defining where the line between our application's services and infrastructure is. Every application will need Databases, Message Brokers, Email Servers to be provisioned and configured in a way that our services can connect to them. This chapter looks at how Crossplane can help us to solve some challenges associated with provisioning infrastructure in a Cloud Provider agnostic way.
- **Chapter 5: Canary Releases:** If you are making changes to your services but not releasing them to production you are probably wasting time. This chapter covers Canary Releases using Knative Serving, to allow your teams to deploy multiple versions of your services at the same time, allowing you to test different features included in different versions without exposing those versions to all your live users.
- **Chapter 6: Event-Driven Features for decoupling services:** While service to service interactions using REST is the most widely understood approach to make two services communicate and share information, using an Event-Driven approach might bring some benefits for certain scenarios. This chapter covers how you can use Knative Eventing to communicate services using events and how these events can be produced and consumed in a very decoupled way.
- **Chapter 7:**

Without further ado, let's jump to the next chapter to get the walking skeleton up and running inside a Kubernetes Cluster.

1.5 Summary

- Cloud-Native applications follow the 12-factor principles to build robust and resilient distributed applications that can scale. You are going to be using Docker containers and Kubernetes to run these distributed applications.
- The Continuous Delivery practices goal aims to reduce the cycle time between a decision for a change is made and the change is live for the end-users.
- Kubernetes is the common denominator between Cloud Providers and it gives us a useful set of abstractions to build Cloud-Native applications but is just the starting point, not the end goal.
- A walking skeleton can be used to find out high-risk or challenging aspects of a Cloud-Native application. A Cloud-Native Conference Platform is going to be used as a walking skeleton throughout the book to exemplify how different Open Source projects help us to speed up delivering software to your end-users.

1.5.1 Why Kubernetes?

There are several reasons why it makes sense focus on Kubernetes as the target platform when building Cloud-Native applications, here are the most relevant when it comes to evaluating cloud providers and where the industry is going:

- **Kubernetes is the common denominator between Cloud Providers and On-Prem workloads:** Every major Cloud-Provider provides a Kubernetes Managed offering (AKS, GKE, EKS, TKG, LKE) enabling developers with a shared common set of APIs that they can use, regarding the Cloud-Provider selected by their company. The Kubernetes communities are also leading the way defining the tools and best practices on how to build the next generation of Cloud-Native applications.
- **Kubernetes for exploring and delivering multi-cloud solutions, Cloud-Provider agnostic solutions:** If your company is looking into deploying the applications that are being produced to multiple cloud providers, maybe to avoid vendor lock-in, you end up in a position where you cannot rely too much on cloud-provider specific solutions and tools. In such situations, developers will want to build abstractions that allow them to move each application and the application infrastructure to a different Cloud-Provider. If that's the case, you can rely on Kubernetes abstractions to deal with the Cloud-Infrastructure and with other abstractions built on top such as Knative to deal with application-specific topics such as messaging. This book covers Knative in chapters X and X.
- **Kubernetes for building your SaaS platform, for delivering your cloud services:** If your company is planning to offer a *Software as a Service* platform to their customers, were your customers are not interested in actually knowing where the platform is running, you might want to leverage Kubernetes and its growing community to hire resources to build such platform. There are a huge amount of companies relying on Kubernetes to provide end-user services that you won't feel alone.
- **Kubernetes offers a vibrant, fast-paced, multi-vendor ecosystem:** Kubernetes is not just another Open Source project, as mentioned before every major Cloud-Provider is today a big part of the Kubernetes Community. Cloud-Providers (such as Amazon, Google, Microsoft) are committed to upgrading their customers to Kubernetes latest releases so no-

one is left behind. This promotes the collaboration also on building tooling and more high-level tools for developers and team to use, relying on Kubernetes APIs and not Cloud-Provider specific tools.

- **Kubernetes offers a transition from On-Prem Cloud and Hybrid to Public Cloud:** If you are in a heavily audited environment where you need to keep data inside your country or inside your datacenters, Kubernetes offers you a path to build applications that can be hosted in your Kubernetes On-Prem installation and easily moved to a Public Cloud when Cloud-Providers provides you all the assurances about data and security. Hybrid alternatives, such as running part of your workloads On-Prem and the rest on Public Cloud are also offered by different vendors.

Having listed the good characteristics of Kubernetes, it is also important to recognize that Kubernetes is not a silver bullet and it will not solve all your problems, in some situations, it might cause you more problems on top of the ones that you already have. So, I need to mention when not to use Kubernetes.

1.5.2 When not Kubernetes?

Kubernetes is not a silver bullet, and you always need to evaluate if it is worth the effort. There is still a case for heavily adopting Cloud-Provider specific tools to achieve the practices and goals described in this book.

One of the first challenges that you will face when looking to adopt Kubernetes is the steep-learning curve for developers to understand this ecosystem. There are heated discussions around if developers need to actually understand Kubernetes in order to do their daily work. Bringing entire teams up to speed is a challenge, a challenge that can be easily solved by training. Knowledge sharing about a completely new ecosystem and tools is key for organizations to leverage new technologies. It is important to clearly recognize that training needs to happen in your teams to enable your teams to understand and use every day the projects that are covered in this book.

Cases, when you have teams with skills that are related to a particular Cloud-Provider and short deadlines that don't include Kubernetes training, might push your teams to deliver using Cloud-Provider specific tools.

If you have no reason to move to another Cloud-Provider or if you are already running in your own infrastructure On-Prem, it might be challenging to add Kubernetes into the mix.

When you are evaluating to adopt Kubernetes, it is not the only thing that you need to consider. Kubernetes on its own it is not enough, you need to heavily research which tools are your developers going to use, hence the reason for this book, and how are you going to deliver software.

At the end of the day, Kubernetes is an abstraction, and with abstractions comes lack of specialisation in certain areas. Over the last year, most of these rough edges have been solved by Cloud Providers, giving their Kubernetes users a first-class citizen experience where tools such as centralized logging, monitoring and other cloud-specific tools had been integrated with their Kubernetes managed services.

This book relies on the fact that Kubernetes is adopted by major Cloud-Providers and that you value that for your business. But as mentioned before, Kubernetes is just the starting point, and you need to think through how are your teams going to work with Kubernetes to delivery fantastic software to your customers or users. I hope this book helps you to understand how to evaluate

different Open Source projects and their characteristics to save time when working with Kubernetes, no matter if you are working on a Cloud-Provider, in an On-Prem setup or aiming for a hybrid approach.

2

Cloud-Native Application Challenges

This chapter covers

- Working with a Cloud-Native application running in a Kubernetes Cluster
- Choosing between local and remote clusters
- Exploring the application to understand the main components and Kubernetes Resources
- How to break a Cloud Native application and understanding the challenges of distributed applications

When I want to try something new, a framework, a new tool, or just a new application, I tend to be impatient; I want to see it running right away. Then, when it is running, I want to be able to dig deeper and understand how it is working. I tend to break things to experiment and validate that I understand how these tools, frameworks or applications are internally working. That is the sort of approach we'll take in this chapter!

To have a Cloud Native application up and running you will need a Kubernetes Cluster. In this chapter, you are going to work with a local Kubernetes Cluster using a project called KIND (Kubernetes IN Docker - <https://kind.sigs.k8s.io/>). This local cluster will allow you to deploy applications locally for development and experimentation purposes. To install a set of microservices you will be using Helm, a project that helps us to package, deploy and distribute Kubernetes applications. You will be installing the walking skeleton services introduced in Chapter 1 which implements a Conference Platform application.

Once the services for the Conference Platform are up and running you will inspect its Kubernetes resources to understand how the application was architected and its inner workings by using `kubectl`. Once you get an overview of the main pieces inside the application, you will jump ahead to try to break the application, finding common challenges and pitfalls that your Cloud Native applications can face. This chapter covers the basics of running Cloud-Native applications in a modern technology stack based on Kubernetes highlighting the good and the bad that comes with developing, deploying and maintaining distributed applications. The following chapters tackle these

associated challenges by looking into projects whose main focus is to speed up and make more efficient the delivery of your projects.

2.1 Running the walking skeleton

To understand the innate challenges of Cloud-Native applications, we need to be able to experiment with a simple example that we can control, configure, and break for educational purposes. To run a Cloud-Native application, in this case, the walking skeleton introduced in chapter 1, you need a Kubernetes cluster. Where this cluster is going to be installed and who will be responsible for setting it up is the first question that developers will have. It is quite common for developers to want to run things locally, in their laptop or workstation, and with Kubernetes, this is definitely possible... but is it optimal? Let's analyze the advantages and disadvantages of running a local cluster against other options.

2.1.1 Choosing the best Kubernetes environment for you

This section doesn't cover a comprehensive list of all the available Kubernetes flavours, but it focuses on common patterns on how Kubernetes clusters can be provisioned and managed.

There are three possible alternatives. All them with advantages and drawbacks:

- **Local Kubernetes in your laptop/desktop computer:**

I tend to discourage people from using Kubernetes running on their laptops, as you will see in the rest of the book, running your software in similar environments to production is highly recommended to avoid issues that can be summed up as "but it works on my laptop". These issues are most of the time caused by the fact that when you run Kubernetes on your laptop, you are not running on top of a real cluster of machines. Hence, there is no network, no round-trips and no real load balancing.

- **Pros:** lightweight, fast to get started, good for testing, experimenting and local development. Good for running small applications.
- **Cons:** not a real cluster, it behaves differently, reduced hardware to run workloads. You will not be able to run a large application on your laptop.

- **On-Premise Kubernetes in your data centre:**

This is a typical option for companies where they have private clouds. This approach requires the company to have a dedicated team and hardware to create, maintain and operate these Clusters. If your company is mature enough, it might have a self-service platform that allows users to request new Kubernetes Clusters on demand.

- **Pros:** real cluster on top of real hardware, will behave closer to how a production cluster will work. You will have a clear picture of which features are available for your applications to use in your environments.
- **Cons:** it requires a mature operation team to set up clusters and gives credentials to users, it requires to have dedicated hardware for developers to work on their experiments

- **Managed Service Kubernetes offering in a Cloud Provider:**

I tend to be in favour of this approach, as using a Cloud Provider service allows you to pay for what you use, and services like Google Kubernetes Engine (GKE), Azure AKS and AWS EKS are all built with a self-service approach in mind, enabling developers to spin up new Kubernetes Cluster quickly. There are two primary considerations:

- 1) You need to choose one and have an account with a big credit card to pay for what your teams are going to consume, this might involve setting up some caps in the budget and defining who has access. By selecting a Cloud Provider, you might be going into a vendor lock-in situation, if you are not careful.
- 2) Everything is remote, and for some people, this is too big of a change. It takes time for developers to adapt, as the tools and most of the workloads will be running remotely. This is also an advantage, as the environments used by your developers and the applications that they are deploying are going to behave as if they were running in a production environment.
 - **Pros:** You are working with real (fully-fledged) clusters, you can define how much resources you need for your tasks, when you are done you can delete it to release resources. You don't need to invest in hardware upfront.
 - **Cons:** you need a potentially big credit card, you need your developers to work against remote clusters and services.

A final recommendation is to check the following repository which contains free Kubernetes credits in major Cloud Providers: <https://github.com/learnk8s/free-kubernetes> . I've created this repository to keep an updated list of these free trials that you can use to get all the examples in the book up and running on top of real infrastructure.

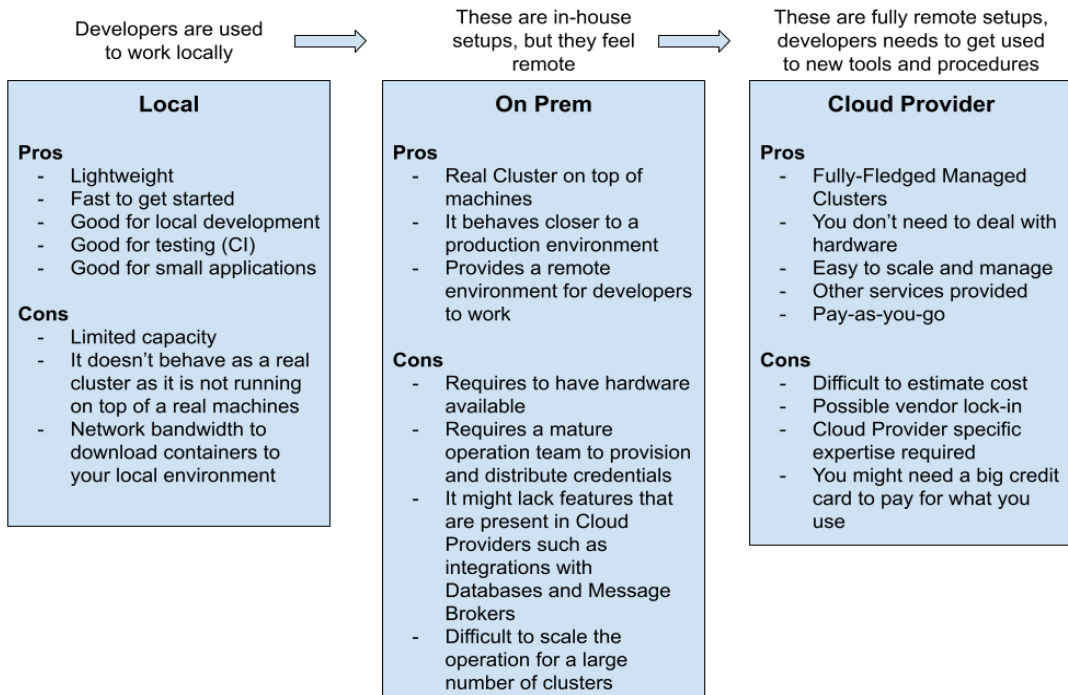


Figure 2.1 Kubernetes Cluster Local vs Remote setups

While these three options are all valid and have drawbacks, in the next sections, you will be using Kubernetes KIND (Kubernetes IN Docker: <https://kind.sigs.k8s.io/>) to deploy the walking skeleton introduced in chapter one in a local Kubernetes environment running in your laptop/pc.

2.1.2 Installing Kubernetes Kind locally

There are several options to create Local Kubernetes Clusters for development. This book has chosen KIND because it supports different platforms and has ease of customization to run your clusters with minimum dependencies, as KIND doesn't require you to download a Virtual Machine. In the following chapters, you will also use Google Kubernetes Engine to run bigger and larger projects, highlighting the power of relying on the Kubernetes API to run your workloads on top of very different hardware.

For practical reasons, having access to a local Kubernetes environment can help you to get started. It is essential to understand that most of the steps are not tied to Kubernetes KIND in any way, meaning that you can run the same commands against a remote Kubernetes Cluster (on-prem or in a cloud provider). If you have access to a fully-fledged Kubernetes Cluster I encourage you to use that one instead, you can skip the following section on KIND and move straight away to 2.X: Installing the application with Helm.

For the examples in this section to work you need to have installed:

- **Docker**, follow the documentation provided on their website to install: <https://docs.docker.com/get-docker/>
- **Kubernetes KinD** (Kubernetes in Docker), follow the documentation provided on their website to install KinD in your laptop: <https://kind.sigs.k8s.io/docs/user/quick-start/#installation>
- `kubect1`, follow the documentation provided in the official Kubernetes site to install `kubect1` <https://kubernetes.io/docs/tasks/tools/>
- **Helm**, you can find the instructions to install Helm in their website: <https://helm.sh/docs/intro/install/>

Once we have everything installed, we can start working with KIND, which is a project that enables you to run local Kubernetes clusters, using Docker container “nodes”.

In this section, you will be creating a local Kubernetes Cluster in your laptop/pc and setting it up so you can access the applications running inside it.

By using KIND, you can quickly provision a Kubernetes Cluster for running and testing your applications; hence it makes a lot of sense when working with applications composed of several services to use a tool like this to run integration tests as part of your Continuous Integration pipelines.

Once you have `kind` installed in your environment, you can create clusters by running a single line in the terminal.

The cluster you are going to create will be called `dev`, and will have four nodes, three workers, and a master node (control plane), as seen in Figure 2.2. We want to be able to see in which nodes our application services are placed inside our Kubernetes Cluster.

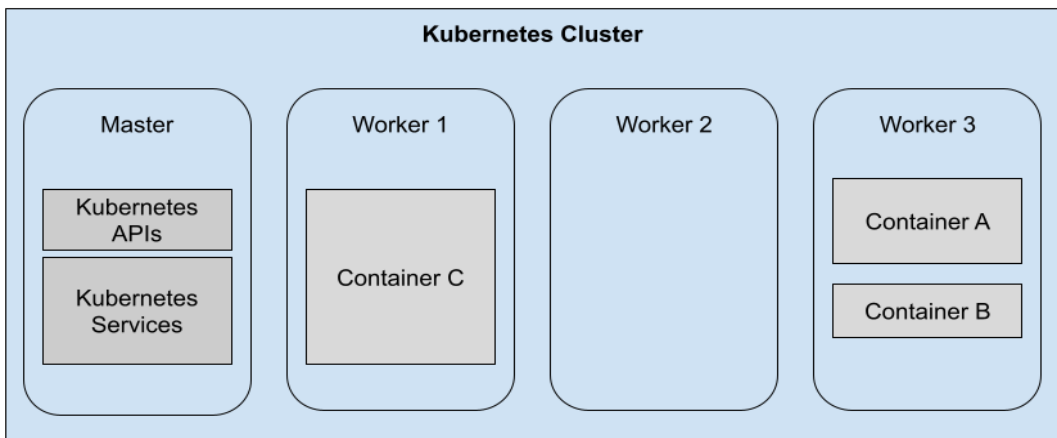


Figure 2.2 Kubernetes Cluster topology

KinD will simulate a real cluster conformed by a set of machines or virtual machines, in this case, each Node will be a Docker Container. When you deploy an application on top of these nodes, Kubernetes will decide where the containers for the application will run based on the overall cluster

utilization. Kubernetes will also deal with failures of these nodes to minimize your applications downtimes. Because you are running a local Kubernetes cluster, this has limitations, such as your laptop/pc available CPUs and Memory. In real life clusters, each of these nodes is a different physical or virtual machine that can be running in different locations to maximize resilience.

You can create the cluster by running the following command in the terminal:

```
cat <<EOF | kind create cluster --name dev --config=-
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  kubeadmConfigPatches:
  - |
    kind: InitConfiguration
    nodeRegistration:
      kubeletExtraArgs:
        node-labels: "ingress-ready=true"
  extraPortMappings:
  - containerPort: 80
    hostPort: 80
    protocol: TCP
  - containerPort: 443
    hostPort: 443
    protocol: TCP
- role: worker
- role: worker
- role: worker
EOF
```

You can copy the previous command and following commands from GitHub: <https://github.com/salaboy/from-monolith-to-k8s/blob/master/kind/README.md>

Notice that besides creating a cluster you will also need to set up an Ingress Controller (hence the labels in the control plane node: `node-labels: "ingress-ready=true"` and some port-mappings to route traffic from your laptop to the services running inside the cluster.

You should see something similar to Figure 2.3 after you run the previous command:

```

Creating cluster "dev" ...
✓ Ensuring node image (kindest/node:v1.21.1) 🖼️
✓ Preparing nodes 📦 📦 📦 📦
✓ Writing configuration 📄
✓ Starting control-plane 🚦
✓ Installing CNI 🛠️
✓ Installing StorageClass 🗄️
✓ Joining worker nodes 🚗
Set kubectl context to "kind-dev"
You can now use your cluster with:

kubectl cluster-info --context kind-dev

Have a nice day! 🌞

```

Figure 2.3 KIND cluster created

To connect your `kubectl` CLI tool with this newly created, you might need to run:

```
kubectl cluster-info --context kind-dev
```

You should see something similar to:

```

Kubernetes control plane is running at https://127.0.0.1:60714
CoreDNS is running at https://127.0.0.1:60714/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

```

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

Figure 2.4 Setting the context for `kubectl`

Once you have connected with the cluster you can start interacting with it. For example, you can check the cluster nodes by running:

```
kubectl get nodes -owide
```

The output of running that command should look similar to this:

NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP	EXTERNAL-IP	OS-IMAGE	KERNEL-VERSION	CONTAINER-RUNTIME
dev-control-plane	Ready	control-plane,master	6m37s	v1.21.1	172.18.0.6	<none>	Ubuntu 21.04	5.10.25-linuxkit	containerd://1.5.2
dev-worker	Ready	<none>	6m7s	v1.21.1	172.18.0.2	<none>	Ubuntu 21.04	5.10.25-linuxkit	containerd://1.5.2
dev-worker2	Ready	<none>	6m7s	v1.21.1	172.18.0.5	<none>	Ubuntu 21.04	5.10.25-linuxkit	containerd://1.5.2
dev-worker3	Ready	<none>	6m7s	v1.21.1	172.18.0.4	<none>	Ubuntu 21.04	5.10.25-linuxkit	containerd://1.5.2

```

salaboy> kubectl get nodes -o wide
NAME                STATUS    ROLES                  AGE      VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE      KERNEL-VERSION   CONTAINER-RUNTIME
dev-control-plane    Ready     control-plane,master   6m37s     v1.21.1   172.18.0.6    <none>         Ubuntu 21.04   5.10.25-linuxkit containerd://1.5.2
dev-worker           Ready     <none>                 6m7s      v1.21.1   172.18.0.2    <none>         Ubuntu 21.04   5.10.25-linuxkit containerd://1.5.2
dev-worker2          Ready     <none>                 6m7s      v1.21.1   172.18.0.5    <none>         Ubuntu 21.04   5.10.25-linuxkit containerd://1.5.2
dev-worker3          Ready     <none>                 6m7s      v1.21.1   172.18.0.4    <none>         Ubuntu 21.04   5.10.25-linuxkit containerd://1.5.2
salaboy>

```

Figure 2.5 Listing all Kubernetes Nodes

As you can see, your Kubernetes Cluster is composed of four nodes, and one of those is the control plane. Notice that you are using the `-o wide` flag to get more information about your nodes.

Finally, you will use NGINX Ingress Controller (more detailed instructions can be found here: <https://kind.sigs.k8s.io/docs/user/ingress/>) to route traffic from outside the Kubernetes Cluster to the applications that are running inside the cluster. There are a number of Ingress Controllers implementations that you can install to do this routing, but NGINX Ingress Controller is widely adopted and the most popular option. For a non-extensive list of available options, you can check the Kubernetes website: <https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/>. To install the NGINX Ingress Controller you need to run the following command:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/deploy/static/provider/kind/deploy.yaml
```

This command creates a set of resources inside our Kubernetes Cluster required to run the NGINX Ingress Controller in a new Kubernetes Namespace called `ingress-nginx`:

```

namespace/ingress-nginx created
serviceaccount/ingress-nginx created
configmap/ingress-nginx-controller created
clusterrole.rbac.authorization.k8s.io/ingress-nginx created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx created
role.rbac.authorization.k8s.io/ingress-nginx created
rolebinding.rbac.authorization.k8s.io/ingress-nginx created
service/ingress-nginx-controller-admission created
service/ingress-nginx-controller created
deployment.apps/ingress-nginx-controller created
ingressclass.networking.k8s.io/nginx created
validatingwebhookconfiguration.admissionregistration.k8s.io/ingress-nginx-admission created
serviceaccount/ingress-nginx-admission created
clusterrole.rbac.authorization.k8s.io/ingress-nginx-admission created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
role.rbac.authorization.k8s.io/ingress-nginx-admission created
rolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
job.batch/ingress-nginx-admission-create created
job.batch/ingress-nginx-admission-patch created

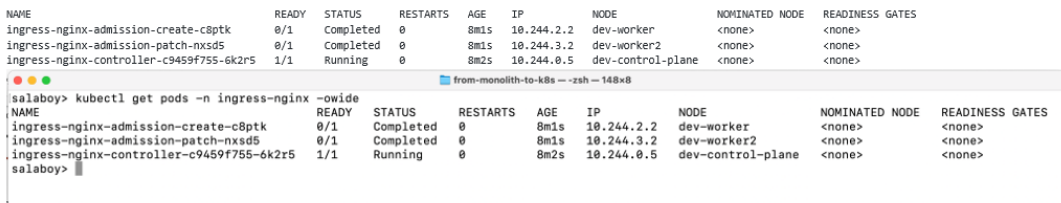
```

Figure 2.6 Installing NGINX Ingress Controller

As a side note, you can check where this Ingress Controller is running in your cluster by running:

```
kubectl get pods -n ingress-nginx -owide
```

The output of this command should look like:



NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS GATES
ingress-nginx-admission-create-c8ptk	0/1	Completed	0	8m1s	10.244.2.2	dev-worker	<none>	<none>
ingress-nginx-admission-patch-nxsd5	0/1	Completed	0	8m1s	10.244.3.2	dev-worker2	<none>	<none>
ingress-nginx-controller-c9459f755-6k2r5	1/1	Running	0	8m2s	10.244.0.5	dev-control-plane	<none>	<none>

Figure 2.7 Ingress Controller running in `control-plane` node

Here it can be seen that the Ingress Controller is running in the Control Plane node.

There you have it; your Cluster is up and running, and your `kubectl` command-line interface is configured to work against your new cluster! Now you are ready to install applications in your newly created cluster.

2.1.3 Installing the walking skeleton with the help of Helm

In this section, we are going to use Helm, a package manager for Kubernetes applications to install our Walking Skeleton into our freshly created Kubernetes Cluster. As we installed an Ingress Controller, we should be able to access the application from our Laptop's favourite browser.

To run containerized applications on top of Kubernetes you will need to have each of the services packaged as a container image, plus you will need to define how these containers will be configured to run in your Kubernetes cluster. To do so, Kubernetes allows you to define different kinds of resources (using YAML format) to configure how your containers will run and communicate with each other. The most common kinds of resources are:

- **Deployments:** declaratively define how many replicas of your container need to be up for your application to work correctly. Deployments also allows us to choose which container (or containers) do we want to run and how to these containers needs to be configured (using Environment Variables).
- **Services:** declaratively define a high-level abstraction to route traffic to the containers created by your deployments. It also acts as a load-balancer between the replicas inside your deployments. Services enable other services and applications inside the cluster to use the service name instead of the physical IP address of the containers to communicate, providing what is known as Service Discovery.
- **Ingress:** declaratively define a route to route traffic from outside the cluster to services inside the cluster. By using Ingress definitions, we can only expose the services that are required by client applications that run outside the cluster.
- **ConfigMap/Secrets:** declaratively define and store configuration objects to set up our services instances. Secrets are considered sensitive information that should have protected access.

If you have large applications with tens of services, these YAML files are going to be complex and hard to manage. Keeping track of the changes and deploying applications by applying these files using `kubectl` becomes a complex job. It is beyond the scope of this book to cover an in-detail view of these resources, as there are other books and online resources available. In this book, we will concentrate on how to deal with these resources for large applications and the tools that can help us with that task. In the following section, we will look into how Helm (<http://helm.sh>) can help us to package, distribute and manage these resources.

HELM BASICS

Helm was created to package all YAML files from a service or an entire application into packages called Charts.

To use Helm you create one of these charts (packages). A Helm Chart is defined by a set of files organized using a very specific directory structure. You can version these Charts to deal with configuration changes and new versions of your application/service.

As you can see in the following figure, a `Chart.yaml` file is required to define the Chart metadata such as name and version. The `templates` directory contains all of our YAML files required to deploy and configure our service/application. As the name of the directory indicates, the files inside the `templates` directory can include parameterizable values that you can replace when you are installing the chart into a specific environment. Finally, the `values.yaml` file

contains the default values for the parameterizable placeholders included in the templates. When installing a chart you can provide your own `values.yaml` file to override the defaults.

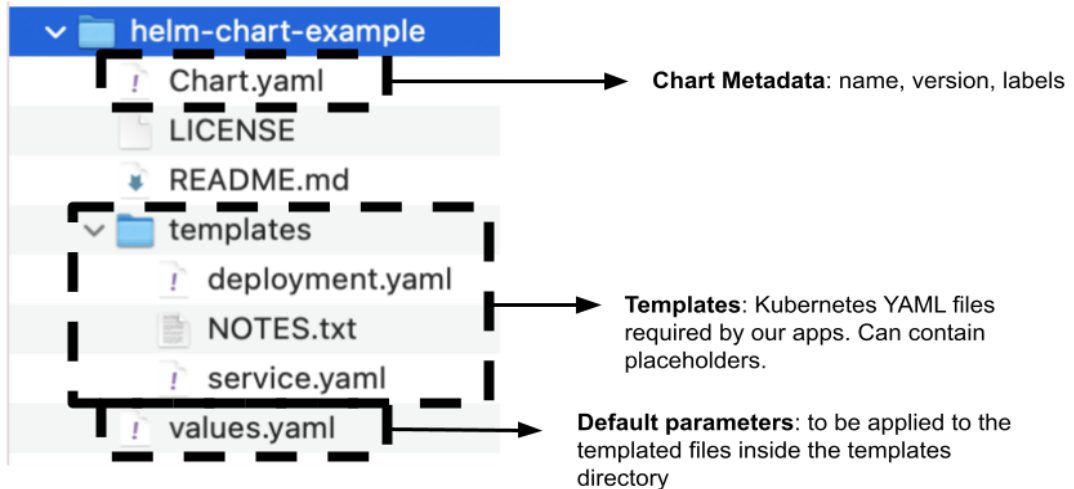


Figure 2.8 Simple Helm Chart

Helm also provides a command-line tool (`helm`) to package, search and install these packages. Helm Charts can be stored in “Helm Repositories” and distributed for other users to use. The most commonly used `helm` command that you need to learn how to use is `helm install <release name> <chart name>`. This will install the chart into a Kubernetes Cluster. In which cluster you might be wondering? Helm uses the same configuration used by `kubectl` to interact with the Kubernetes APIs, hence if you can connect with `kubectl` to your cluster Helm will be able to install charts in that cluster.

You can check out each of the files for this example chart here: <https://github.com/salaboy/helm-chart-example>. The `README.md` file also includes how to run the most common operations to package and install the chart in your own Kubernetes Cluster.

When you install a Chart into a Kubernetes, Helm will create a Release, that we can upgrade at any time if a new version of the chart is available. Helm will keep track of these releases, allowing us to roll back to previous releases if something goes wrong with a newer version of your application.

To install Helm Charts (packages/applications) you can add new repositories in the same location as where your applications are stored. For java developers, these repositories are like Maven Central, Nexus or Artifactory.

```
helm repo add fmtok8s https://salaboy.github.io/helm/
helm repo update
```

You should see the following output:

```

salaboy> helm repo add fmtok8s https://salaboy.github.io/helm/
"fmtok8s" has been added to your repositories
salaboy> helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "fmtok8s" chart repository
Update Complete. *Happy Helming!*

```

Figure 2.9 Adding custom Helm repository

The previous two lines added a new repository to your Helm installation called **fmtok8s**; the second one fetched a file describing all the available packages and their versions for each repo that you have registered. Now that you have installed a new repository, you don't need to install the chart from the chart source code and you can use the published version in the ``https://salaboy.github.io/helm/`` repository. Notice that Helm Chart Repositories can be created inside Github for small setups as I am doing in the following repository: <https://github.com/salaboy/helm>. Check the repository `README.md` for more details about how this is working.

Before jumping into installing our walking skeleton it is also important to know that Helm also provides dependency management between these packages, meaning that you can define that a Chart depends on one or more Charts and Helm will download and install these dependent charts when you install your (parent) Chart. This allows us to install multiple services and other components at the same time without the need to package all the YAML files together. You can define dependencies by adding a section to the `Chart.yaml` file, for example:

```

dependencies:
- name: postgresql
  repository: https://charts.bitnami.com/bitnami
  version: 10.8.0

```

If you are using dependencies, make sure that before ``helm package`` you run ``helm dependency build`` to fetch these charts. This will make your chart package standalone and not depend on other Chart repositories to be available for your application to work.

Now, let's jump to install our walking skeleton.

INSTALLING THE CONFERENCE PLATFORM WITH A SINGLE COMMAND

Now that your Helm installation fetched all the available packages from the **fmtok8s** Chart repository, you are ready to install the Conference Platform application, which was introduced in Chapter 1, Section X. This Conference Platform allows conference organizers to receive proposals from potential speakers, evaluate these proposals and keep an updated agenda with the approved submissions for the event. We will be using this application throughout the book to exemplify the challenges that you will face while building real-life applications. This application was built as a walking skeleton, which means it is not a complete application but has all the pieces required for some use cases to work and these pieces can be iterated further to support real-life scenarios. In the following sections, you will install the application into the cluster and interact with it to see how it behaves when it runs on top of Kubernetes.

Let's install the application with the following line:

```
helm install app fmtok8s/fmtok8s-app
```

You should see the following output:

```
NAME: app
LAST DEPLOYED: Sat Aug 28 13:52:48 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Cloud-Native Conference Platform V1

Chart Deployed: fmtok8s-app - 0.1.0
Release Name: app
```

Figure 2.10 Helm installed the chart `fmtok8s-app` version 0.1.0

`helm install` creates a Helm Release, which means that you have created an application instance, in this case, the instance is called `app`. With Helm, you can deploy multiple instances of the application if you want to. You can list Helm releases by running:

```
helm list
```

The output should look like this:

NAME	NAMESPACE	REVISION	UPDATED	STATUS	CHART	APP VERSION
app	default	1	2021-08-28 13:52:48.748112 +0100 BST	deployed	fmtok8s-app-0.1.0	0.1.0

Figure 2.11 List Helm releases

NOTE: If instead of using `helm install` you run `helm template` Helm will output the YAML files which will apply against the cluster. There are situations where you might want to do that instead of `helm install`, for example, if you want to override values that the Helm charts don't allow you to parameterize or apply any other transformations before sending the request to Kubernetes.

VERIFYING THAT THE APPLICATION IS UP AND RUNNING

Once the application is deployed, containers will be downloaded to your laptop to run, and this can take a while. You can monitor the progress by listing all the pods running in your cluster, once again, using the `-o wide` flag to get more information:

```
kubectl get pods -owide
```

The output should look like:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS GATES
app-fmtok8s-agenda-rest-7cfc4b5b4c-msqr9	1/1	Running	0	3h14m	10.244.3.6	dev-worker2	<none>	<none>
app-fmtok8s-api-gateway-8b4b6d56b-kr249	1/1	Running	0	3h14m	10.244.2.4	dev-worker	<none>	<none>
app-fmtok8s-c4p-rest-558b8d7bdd-c7768	1/1	Running	0	3h14m	10.244.3.5	dev-worker2	<none>	<none>
app-fmtok8s-email-rest-7795df6885-mv5m6	1/1	Running	0	3h14m	10.244.1.4	dev-worker3	<none>	<none>


```
salaboy> kubectl get pods -owide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS GATES
app-fmtok8s-agenda-rest-7cfc4b5b4c-msqr9	1/1	Running	0	3h14m	10.244.3.6	dev-worker2	<none>	<none>
app-fmtok8s-api-gateway-8b4b6d56b-kr249	1/1	Running	0	3h14m	10.244.2.4	dev-worker	<none>	<none>
app-fmtok8s-c4p-rest-558b8d7bdd-c7768	1/1	Running	0	3h14m	10.244.3.5	dev-worker2	<none>	<none>
app-fmtok8s-email-rest-7795df6885-mv5m6	1/1	Running	0	3h14m	10.244.1.4	dev-worker3	<none>	<none>
app-install-hhzf6	0/1	Completed	0	3h14m	10.244.1.5	dev-worker3	<none>	<none>

```
salaboy>
```

Figure 2.12 Listing application Pods

You need to pay attention to the **READY** and **STATUS** columns, where 1/1 in the **READY** column means that one replica of the `Pod` is running and one is expected to be running.

Notice that `Pod`s can be scheduled in different nodes. You can check this in the `NODE` column; this is Kubernetes efficiently using the cluster resources.

If all the `Pod`s are up and running, you've made it! The application is now up and running, and you can access it by pointing your favourite browser to: <http://localhost>

If you require, there is a step by step tutorial on how to deploy this application to a Kubernetes Cluster using Helm, which you can find at the following repository: <https://github.com/salaboy/from-monolith-to-k8s/tree/master/helm>.

2.1.4 Interacting with your application

In the previous section, we installed the application into our local Kubernetes Cluster. In this section we will quickly interact with the application to understand how the services are interacting to accomplish a simple use case: "Receiving and approving Proposals". Remember that you can access to the application by pointing your browsers to: <http://localhost>

The Conference Platform application should look like the following screenshot:

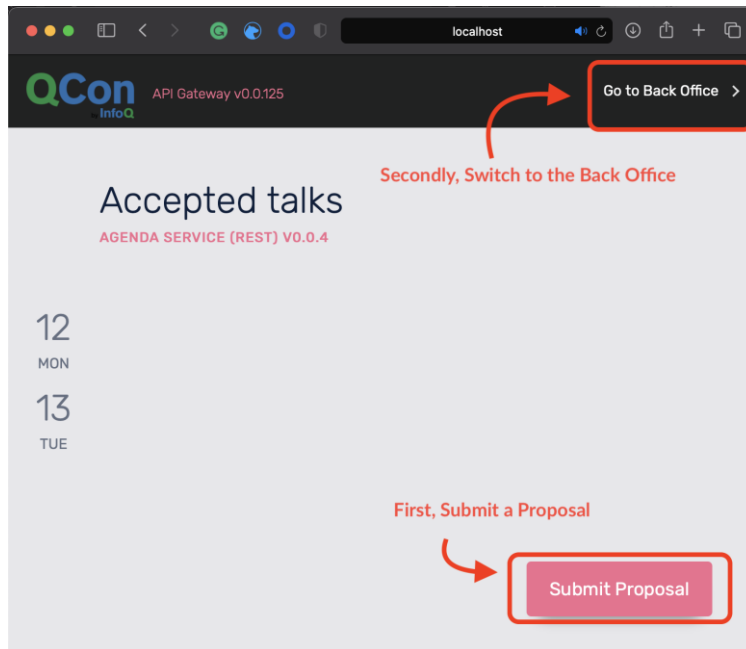


Figure 2.X Application Main Page

This application will help conference organizers and attendees to interact before and during the events.

The application Main Page lists all the talks scheduled for the conference. These talks were submitted by potential speakers and reviewed by the conference organisers. When you start the application for the first time, there will be no talks on the agenda, but you can now go ahead and 'Submit a Proposal'.

New Proposal
C4P SERVICE (REST) V0.0.10

TITLE
Microservices 101 ← Enter Title

AUTHOR
Well Known Speaker ← Enter Author

EMAIL
wellknow@speakers.org ← Enter Email

ABSTRACT
This presentation is about microservices basics. ← Enter Abstract

SEND ← Fill the Form and hit SEND

Figure 2.X Submitting a proposal for organisers to review

Notice that there are four fields (Title, Author, Email and Abstract) in the form that you need to fill to submit a proposal. The organisers will use this information to evaluate your proposal and get in touch with you via email if your proposal gets approved or rejected. Once the proposal is submitted, you can go to the 'Back Office' and 'Accept' or 'Reject' submitted proposals. You will be acting as a conference organiser on this screen:

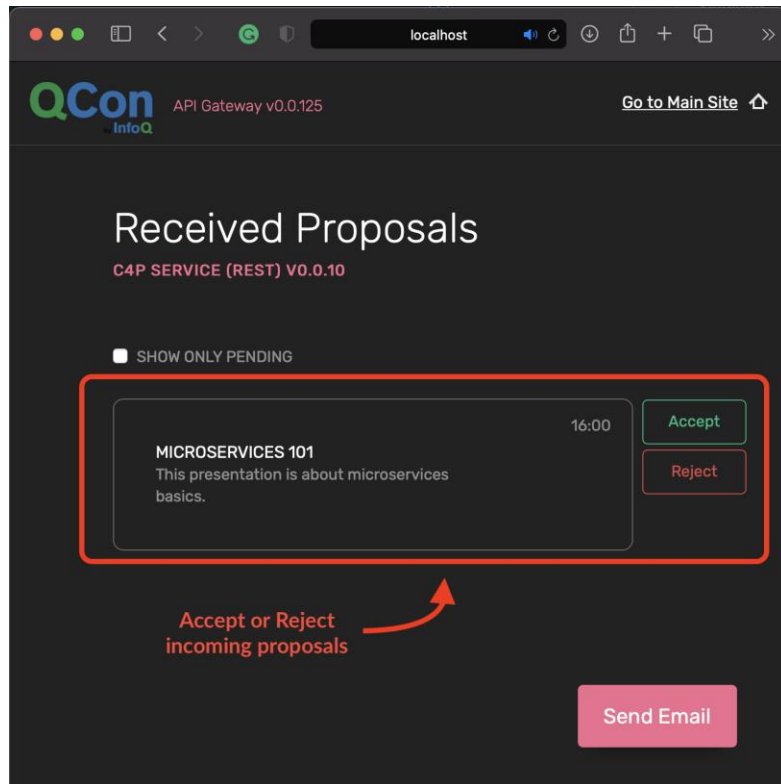


Figure 2.X Conference organizers can Accept or Reject incoming proposals

Accepted proposals will appear on the Main Page. Attendees who visit the page at this stage can have a glance at the conference main speakers.

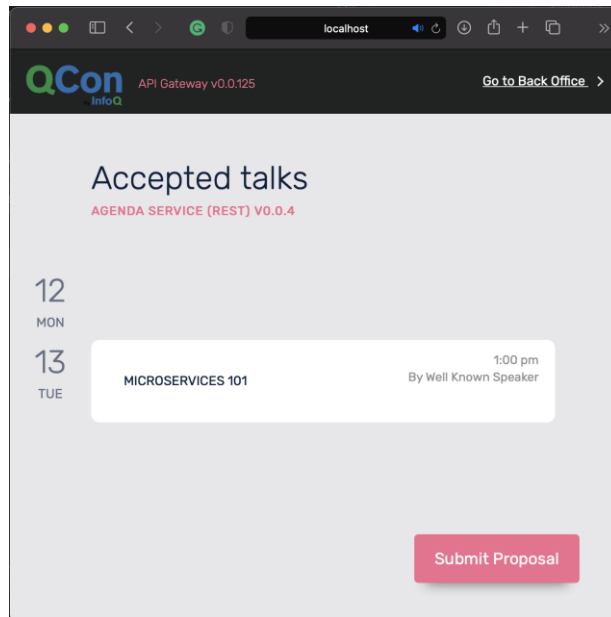


Figure 2.X Your proposal is now live on the agenda!

At this stage, the potential speaker should have received an email about the approval or rejection of his/her proposal. You can check this by looking at the Email Service logs, using `kubectl` from your terminal:

```
kubectl logs -f app-fmtok8s-email-rest-<POD_ID>
```


- The **container** that this deployment is using: notice that this is just a simple docker container, meaning that you can even run this container locally if you want to with ``docker run``. This is fundamental to troubleshooting problems.
- The number of **replicas** required by the deployment: for this example is set to 1, but you will change this in the next section. Having more replicas adds more resiliency to the application, as these replicas can go down. Kubernetes will spawn new instances to keep the number of desired replicas up at all times.
- The **resources allocation** for the container: depending on the load and the technology stack that you used to build your service, you will need to fine-tune how many resources Kubernetes allow your container to use.
- The status of the **'readiness'** and **'liveness' probes**: Kubernetes by default, will monitor the health of your container. It does that by executing two probes: 1) The ``readiness probe`` checks if the container is ready to answer requests 2) The ``liveness probe`` checks if the main process of the container is running.
- The rolling updates strategy defines how our Pods will be updated to avoid downtime to our users. With the RollingUpdateStrategy you can define how many replicas are allowed to go down while triggering and update to a newer version.

First, let's list all the available Deployments with:

```
kubect1 get deployments
```

With an output looking like this:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
app-fmtok8s-agenda-rest	1/1	1	1	3h20m
app-fmtok8s-api-gateway	1/1	1	1	3h20m
app-fmtok8s-c4p-rest	1/1	1	1	3h20m
app-fmtok8s-email-rest	1/1	1	1	3h20m

Figure 2.X Listing your application's Deployments

EXPLORING DEPLOYMENTS

In the following example, you will describe the API Gateway / User Interface deployment. You can describe each deployment in more detail with:

```
kubect1 describe deployment app-fmtok8s-api-gateway
```

```

salaboy> kubectl describe deployment app-fmtok8s-api-gateway
Name:          app-fmtok8s-api-gateway
Namespace:     default
CreationTimestamp: Fri, 25 Dec 2020 17:59:40 +0100
Labels:        app.kubernetes.io/managed-by=Helm
               chart=fmtok8s-api-gateway-0.0.125
               draft=draft-app
Annotations:   deployment.kubernetes.io/revision: 1
               meta.helm.sh/release-name: app
               meta.helm.sh/release-namespace: default
Selector:      app=app-fmtok8s-api-gateway
Replicas:      1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType:  RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=app-fmtok8s-api-gateway
           draft=draft-app
  Containers:
    fmtok8s-api-gateway:
      Image:   gcr.io/camunda-researchanddevelopment/fmtok8s-api-gateway:0.0.125
      Port:    8080/TCP
      Host Port: 0/TCP
      Limits:
        cpu:      1500m
        memory:   1768Mi
      Requests:
        cpu:       750m
        memory:    1Gi
      Liveness:    http-get http://:8080/actuator/health delay=60s timeout=1s period=10s #success=1 #failure=3
      Readiness:   http-get http://:8080/actuator/health delay=0s timeout=1s period=10s #success=1 #failure=3
      Environment:
        VERSION: 0.0.125
      Mounts:     <none>
      Volumes:    <none>
  Conditions:
    Type           Status  Reason
    ----           -
    Available       True    MinimumReplicasAvailable
    Progressing     True    NewReplicaSetAvailable
    OldReplicaSets: <none>
    NewReplicaSet:  app-fmtok8s-api-gateway-5bbb95bc74 (1/1 replicas created)
  Events:
    Type           Reason              Age    From          Message
    ----           -
    Normal          ScalingReplicaSet   12m    deployment-controller  Scaled up replica set app-fmtok8s-api-gateway-5bbb95bc74 to 1

```

Figure 2.X Describing a deployment to see its details

Describing deployments in this way will be helpful if for some reason the deployment is not working. For example, if the number of replicas required is not met, describing the resource will give you insights into where the problem might be. Always check at the bottom for the events associated with the resource to get more insights about the resource status, in this case, the deployment was scaled to have 1 replica 12 minutes ago.

As mentioned before, Deployments are also responsible for coordinating version or configuration upgrades and rollbacks. The deployment update strategy is set by default to "Rolling Update", which means that the deployment will incrementally upgrade Pods one after the other to minimize downtime. An alternative strategy can be set called "Recreate" which will shut down all the Pods and create new ones.

In contrast with Pods, Deployments are not ephemeral; hence if you create a `Deployment` it will be there for you to query no matter if the containers under the hood are failing. By default, when you create a Deployment resource, Kubernetes creates an intermediate resource for handling and checking the Deployment requested replicas.

REPLICASETS

Having multiple replicas of your containers is an important feature to be able to scale your applications. If your application is experiencing loads of traffic from your users, you can easily scale up the number of replicas of your services to accommodate all the incoming requests. In a similar way, if your application is not experiencing a large number of requests these replicas can be scaled down to save resources. The object created by Kubernetes is called `ReplicaSet`, and it can be queried by running:

```
kubectl get replicaset
```

The output should look like:

NAME	DESIRED	CURRENT	READY	AGE
app-fmtok8s-agenda-rest-7cfc4b5b4c	1	1	1	3h18m
app-fmtok8s-api-gateway-8b4b6d56b	1	1	1	3h18m
app-fmtok8s-c4p-rest-558b8d7bbd	1	1	1	3h18m
app-fmtok8s-email-rest-7795df6885	1	1	1	3h18m

Figure 2.X Listing Deployment's Replica Sets

These `ReplicaSet` objects are fully managed by the Deployment resource, and usually, you shouldn't need to deal with them.

ReplicaSets are also essential when dealing with Rolling Updates, and you can find more information about this topic here: <https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/>. You will be performing updates to the application with Helm in later chapters where these mechanisms will kick in.

If you want to change the number of replicas for a deployment you can use once again `kubectl` to do so:

```
kubectl scale --replicas=2 deployments/<DEPLOYMENT_ID>
```

You can try this out with the API Gateway deployment:

```
kubectl scale --replicas=2 deployments/app-fmtok8s-api-gateway
```

This command changes the deployment resource in Kubernetes and triggers the creation of a second replica for the API Gateway deployment. Increasing the number of replicas of your user-facing services is quite common as it is the service that all users will hit when visiting the conference page.

If you access the application again in your browser, at the top of the screen, right beside the API Gateway Service version number, you can see the Pod name and the Node where the Pod is running:

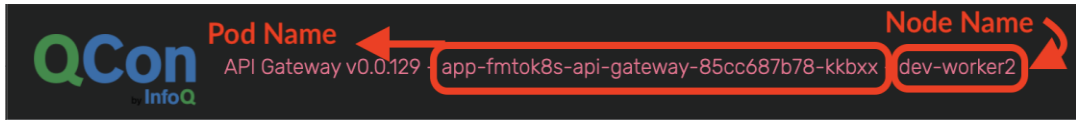


Figure 2.X First replica answering your request

If you now refresh the page, the second replica should answer your request:

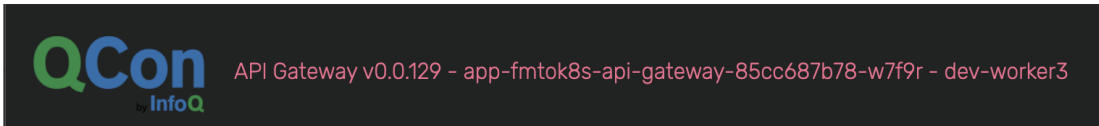


Figure 2.X Second replica is now answering; this replica might be on a different node

By default, Kubernetes will load-balance the requests between the replicas. Being able to scale by just changing the number of replicas, there is no need to deploy anything new, Kubernetes will take care of provisioning a new Pod (with a new container in it) to deal with more traffic. Kubernetes will also make sure that there is the amount of desired replicas at all times. You can test this by deleting one pod and watching how Kubernetes recreates it automatically.

2.2.2 Connecting services together

We have looked at Deployments, which are in charge of getting our containers up and running and keeping them that way, but so far, these containers can only be accessed inside the Kubernetes Cluster. If we want other services to interact with these containers we need to take a look at another Kubernetes resource called "Service". Kubernetes provides an advanced Service discovery mechanism that allows services to communicate with each other by just knowing their names. This is essential to connect a large number of services together without the need to use IP addresses of containers that can change over time.

EXPLORING SERVICES

For exposing your containers to other services, you need to use a `Kubernetes Service` resource. Each application service defines this `Service` resource, so other services and clients can connect to them. In Kubernetes, Services will be in charge of routing traffic to your application containers. These `Service`s represent a logical name that you can use to abstract where your containers are running. If you have multiple replicas of your containers, the Service resource will be in charge of load balance the traffic among all the replicas.

You can list all the services by running:

```
kubect1 get services
```

After running the command, you should see something like figure 2.X that follows:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
fmtok8s-agenda	ClusterIP	10.96.22.133	<none>	80/TCP	3h22m
fmtok8s-api-gateway	ClusterIP	10.96.2.222	<none>	80/TCP	3h22m
fmtok8s-c4p	ClusterIP	10.96.114.4	<none>	80/TCP	3h22m
fmtok8s-email	ClusterIP	10.96.215.48	<none>	80/TCP	3h22m
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	7h20m

Figure 2.X Listing application's services

And you can also describe a Service to see more information about it with:

```
kubect1 describe service fmtok8s-api-gateway
```

This should give you something, like we see in Figure 2.X. Services and Deployments, are linked by the Selector property, highlighted in the following image. In other words, the Service will route traffic to all the Pods created by a Deployment containing the label **app=app-fmtok8s-api-gateway**.

```
salaboy> k describe service fmtok8s-api-gateway
Name:          fmtok8s-api-gateway
Namespace:     default
Labels:        app.kubernetes.io/managed-by=Helm
               chart=fmtok8s-api-gateway-0.0.125
Annotations:   fabric8.io/expose: true
               fabric8.io/ingress.annotations: kubernetes.io/ingress.class: nginx
               meta.helm.sh/release-name: app
               meta.helm.sh/release-namespace: default
Selector:      app=app-fmtok8s-api-gateway
Type:          ClusterIP
IP:            10.96.28.159
Port:          http 80/TCP
TargetPort:    8080/TCP
Endpoints:     10.244.1.7:8080
Session Affinity: None
Events:        <none>
```

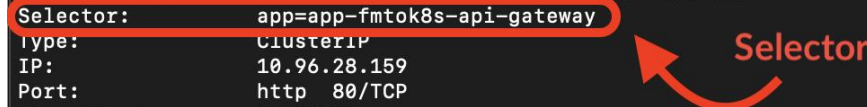


Figure 2.X Describing the API Gateway Service

SERVICE DISCOVERY IN KUBERNETES

By using Services, if your application service needs to send a request to any other service it can use the Kubernetes Services name and port, which in most cases, you can use port 80 if you are using HTTP requests, leaving the need to only use the Service name.

If you look at the source code of the services (you can find it here: <https://github.com/salaboy/fmtok8s-c4p-rest/blob/main/src/main/java/com/salaboy/conferences/c4p/rest/services/EmailService.java#L14>), you will see that HTTP requests are created against the service name, no IP addresses or Ports are needed.

Finally, if you want to expose your Services outside of the Kubernetes Cluster, you need an Ingress resource. As the name represents, this Kubernetes resource is in charge of routing traffic from outside the cluster to services that are inside the cluster. Usually, you will not expose multiple services, limiting the entry points for your applications.

You can get all the available Ingress resources by running the following command:

```
kubectl get ingress
```

The output should look like:

NAME	CLASS	HOSTS	ADDRESS	PORTS	AGE
frontend	<none>	*	localhost	80	4m32s

Figure 2.X Listing application's Ingress Resources

And then you can describe the Ingress resource in the same way as you did with other resource types to get more information about it:

```
kubectl describe ingress frontend
```

You should expect the output to look like this:

```
Name:          frontend
Namespace:     default
Address:       localhost
Default backend: default-http-backend:80 (<error: endpoints "default-http-backend" not found>)
Rules:
  Host      Path  Backends
  ----      -
  *
            /   fmk8s-api-gateway:80 (10.244.1.8:8080)
Annotations:  kubernetes.io/ingress.class: nginx
              meta.helm.sh/release-name: app
              meta.helm.sh/release-namespace: default
              nginx.ingress.kubernetes.io/rewrite-target: /

Events:
  Type    Reason    Age           From                      Message
  ----    -
  Normal  Sync      7m58s (x2 over 7m59s)  nginx-ingress-controller  Scheduled for sync
```

Figure 2.X Describing Ingress Resource

As you can see, Ingresses also uses Services name to route traffic. Also, for this to work, you need to have an Ingress Controller, as we installed when we created the KinD Cluster. If you are running in a Cloud Provider you might need to install an Ingress controller.

Check the following blog post to see different options of Ingress Controllers that are available for you to use: <https://docs.google.com/spreadsheets/u/2/d/191WWNpjJ2za6-nbG4ZoUMXMpUK8KICIOSvQB0f-oq3k/edit#gid=907731238>

With Ingresses you can configure a single entry-point and use path-based routing to redirect traffic to each service you need to expose. The previous Ingress resource in Figure 2.20 routes all the traffic sent to `/` to the `fmtok8s-api-gateway` service. Notice that Ingress rules are pretty simple and you shouldn't add any business logic routing at this level.

TROUBLESHOOTING INTERNAL SERVICES

Sometimes, it is important to access internal services to debug or troubleshoot services that are not working. For such situations, you can use the `kubectl port-forward` command to temporarily access services that are not exposed outside of the cluster using an Ingress resource. For example, to access the Agenda Service without going through the API Gateway you can use the following command:

```
kubectl port-forward svc/fmtok8s-agenda-rest 8080:80
```

You should see the following output, make sure that you don't kill the command:

```
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
```

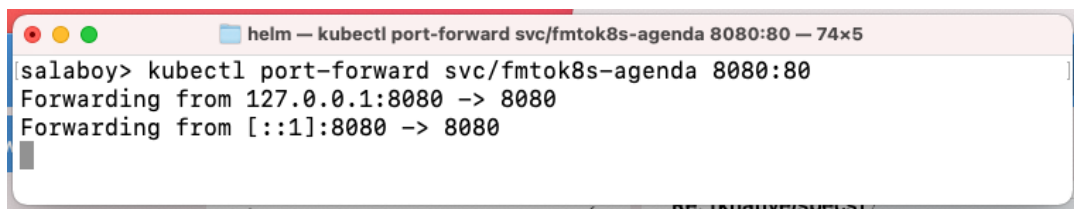


Figure 2.X `kubectl port-forward` allows you to expose a service for debugging purposes

And then using your browser, curl in a different tab or any other tool to point to `http://localhost:8080/info` to access the exposed Agenda Service. The following image shows how you can `curl` the Agenda Service info endpoint and print a pretty/colorful JSON payload with the help of `jq`, which you need to install separately.

```
curl localhost:8080/info | jq --color-output
{
  "name": "Agenda Service (REST)",
  "version": "v0.1.0",
  "source": "https://github.com/salaboy/fmtok8s-agenda-rest/releases/tag/v0.1.0",
  "podId": "app-fmtok8s-agenda-rest-7cfc4b5b4c-1lqr4",
  "podNamespace": "default",
  "podNodeName": "dev-worker"
}
```

Figure 2.X `curl localhost:8080` to access Agenda Service using `port-forward`

In this section, you have inspected the main Kubernetes Resources that were created to run your application's containers inside Kubernetes. By looking at these resources and their relationships, you will be able to troubleshoot problems when they arise.

For everyday operations, the `kubectl` command line tool might not be optimal and different dashboards can be used to explore and manage your Kubernetes workloads such as: k9s, Kubernetes Dashboard and Octant. You can check Appendix X where the same application is explored using Octant.

2.3 Cloud-Native applications Challenges

In contrast to a Monolithic application, which will go down entirely if something goes wrong, Cloud-Native applications shouldn't crash if a service goes down. Cloud-Native applications are designed for failure and should keep providing valuable functionality in the case of errors. A degraded service while fixing issues is better than having no access to the application at all. In this section, you will change some of the service configurations in Kubernetes to understand how the application will behave in different situations.

In some cases, application/service developers will need to make sure that they build their services to be resilient and some concerns will be solved by Kubernetes or the infrastructure.

This section covers some of the most common challenges associated with Cloud Native applications. I find it useful to know what are the things that are going to go wrong in advance rather than when I am already building and delivering the application. This is not an extensive list, it is just the beginning to make sure that you don't get stuck with problems that are widely known. The following sections will exemplify and highlight these challenges with the Conference platform.

- **Downtime is not allowed:** If you are building and running a Cloud-Native application on top of Kubernetes and you are still suffering from application downtime, then you are not capitalizing on the advantages of the technology stack that you are using.
- **Service's built-in resiliency:** downstream services will go down and you need to make sure that your services are prepared for that. Kubernetes helps with dynamic Service Discovery, but that is not enough for your application to be resilient.
- **Dealing with the application state is not trivial:** we have to understand each service infrastructural requirements to efficiently allow Kubernetes to scale up and down our services.
- **Data inconsistent data:** a common problem of working with distributed applications is that data is not stored in a single place and tends to be distributed. The application will need to be ready to deal with cases where different services have different views of the state of the world.
- **Understanding how the application is working (monitoring, tracing and telemetry):** having a clear understanding on how the application is performing and that it is doing what it is supposed to be doing is essential to quickly find problems when things go wrong.
- **Application Security and Identity Management:** dealing with users and security is always an after-thought. For distributed applications, having these aspects clearly documented and implemented early on will help you to refine the application requirements by defining "who can do what and when".

Let's start with the first of the challenges.

2.3.1 Downtime is not allowed

In section 2.2.x, you increased the number of replicas for the API Gateway; this is a great feature when your services are created based on the assumption that they will be scaled by the platform by creating new copies of the containers running the service. So, what happens when the service is not ready to handle replication, or when there are no replicas available for a given service?

You also scaled the API Gateway to have 2 replicas running all the time. This means that if one of the replicas stop running for any reason, Kubernetes will try to start another one until 2 are running.

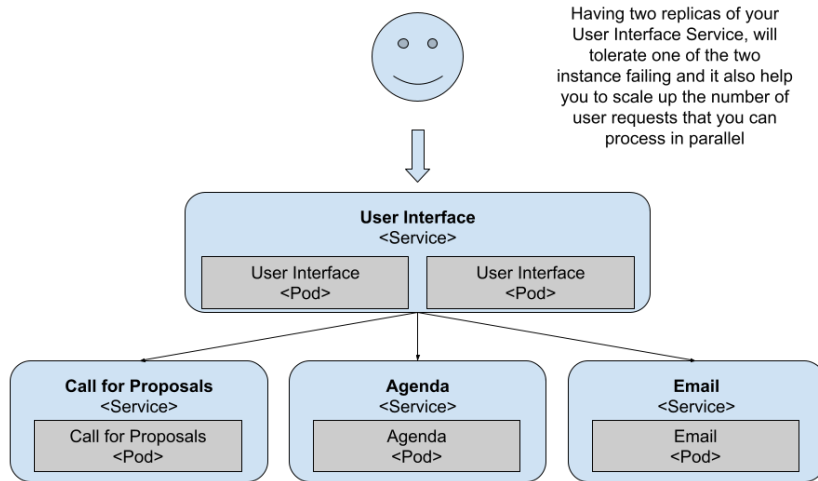


Figure 2.X Two replicas for the API Gateway / User Interface

You can quickly try this self-healing feature of Kubernetes by killing one of the two pods of the API Gateway. You can do this by running the following commands:

```
kubect1 get pods
```

```

salaboy — -zsh — 102x19
Last login: Wed Feb  3 09:12:53 on ttys006
[salaboy> k get pods
NAME                                READY   STATUS    RESTARTS   AGE
app-fmtok8s-agenda-rest-cfd77946f-zigrg  1/1     Running   0           23m
app-fmtok8s-api-gateway-76478fffd6-7jcq4  1/1     Running   0           3m29s
app-fmtok8s-api-gateway-76478fffd6-d6vh9  1/1     Running   0           23m
app-fmtok8s-c4p-rest-596b95594d-vb872    1/1     Running   0           23m
app-fmtok8s-email-rest-7d87866bf-hvwhm    1/1     Running   0           23m
salaboy>

```

Two Replicas are running

Figure 2.X Checking that the two replicas are up and running

Now, copy one of the two Pods Id and delete it:

```
kubectl delete pod <POD_ID>
```

Then list the pods again:

```
kubectl get pods
```

```

salaboy — -zsh — 102x19
[salaboy> k delete pod app-fmtok8s-api-gateway-76478fffd6-7jcq4
pod "app-fmtok8s-api-gateway-76478fffd6-7jcq4" deleted
[salaboy> k get pods
NAME                                READY   STATUS              RESTARTS   AGE
app-fmtok8s-agenda-rest-cfd77946f-zigrg  1/1     Running             0           29m
app-fmtok8s-api-gateway-76478fffd6-csnth  0/1     ContainerCreating   0           8s
app-fmtok8s-api-gateway-76478fffd6-d6vh9  1/1     Running             0           29m
app-fmtok8s-c4p-rest-596b95594d-vb872    1/1     Running             0           29m
app-fmtok8s-email-rest-7d87866bf-hvwhm    1/1     Running             0           29m
salaboy>

```

New replica created

Figure 2.X A new replica is automatically created by Kubernetes as soon as one goes down

You can see how Kubernetes (the ReplicaSet more specifically) immediately creates a new pod when it detects that there is only one running. While this new pod is being created and started, you

have a single replica answering your requests until the second one is up and running. This mechanism ensures that there are at least two replicas answering your users' requests.

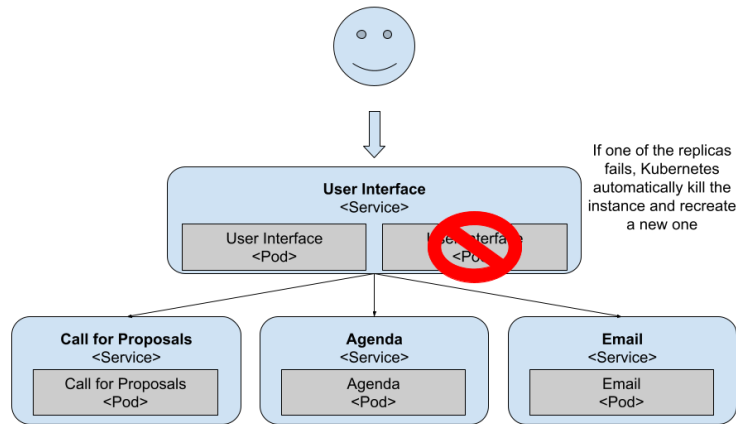


Figure 2.X As soon as Kubernetes detects one pod misbehaving it will kill it and create a new one

If you have a single replica, if you kill the running pod, you will have downtime in your application until the new container is created and ready to serve requests. You can revert back to a single replica with:

```
kubect1 scale --replicas=1 deployments/<DEPLOYMENT_ID>
```

Go ahead and try this out, delete only replica available for the API Gateway Pod:

```
kubect1 delete pod <POD_ID>
```

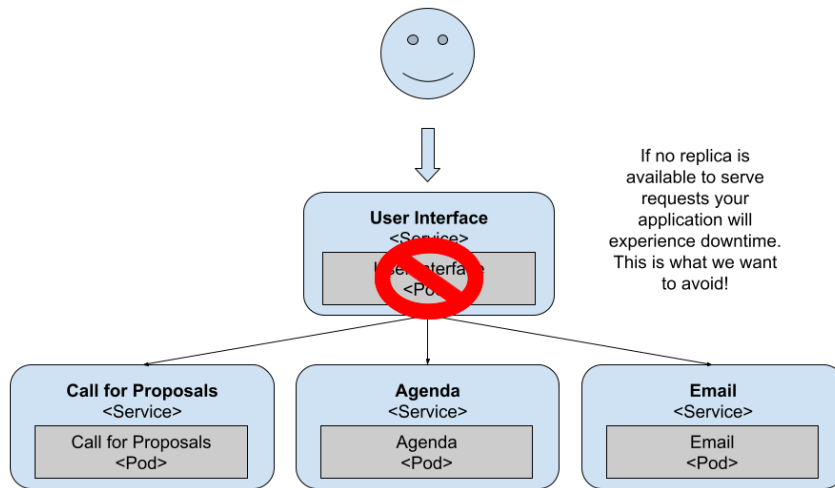


Figure 2.X With a single replica being restarted, there is no backup to answer user requests

Right after killing the pod, try to access the application by refreshing your browser (<http://localhost>). You should see "503 Service Temporarily Unavailable" in your browser, as the Ingress Controller (not shown in the previous figure for simplicity) cannot find a replica running behind the API Gateway service. If you wait for a bit, you will see the application come back up.

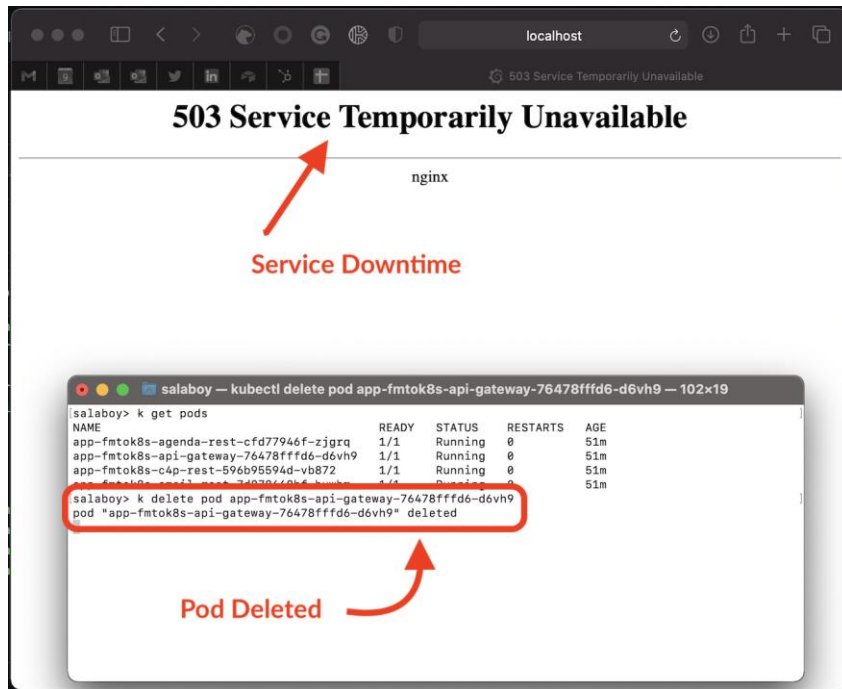


Figure 2.X With a single replica being restarted, there is no backup to answer user requests

This behaviour is to be expected, as the API Gateway Service is a user-facing service. If it goes down, users will not be able to access any functionality, hence having multiple replicas is recommended. From this perspective, we can assert that the API Gateway / FrontEnd service is the most important service of the entire application as our primary goal for our applications is to avoid downtime.

In summary, pay special attention to user-facing services exposed outside of your cluster. No matter if they are User Interfaces or just APIs, make sure that you have as many replicas as needed to deal with incoming requests. Having a single replica should be avoided for most use cases besides development.

2.3.2 Service's resilience built-in

But now, what happens if the other services go down? For example, the Agenda Service, which is just in charge of listing all the accepted proposals to the conference attendees.

This service is also critical, as the Agenda List is right there on the main page of the application. So, let's scale the service down:

```
kubectl scale --replicas=0 deployments/app-fmtok8s-agenda-rest
```

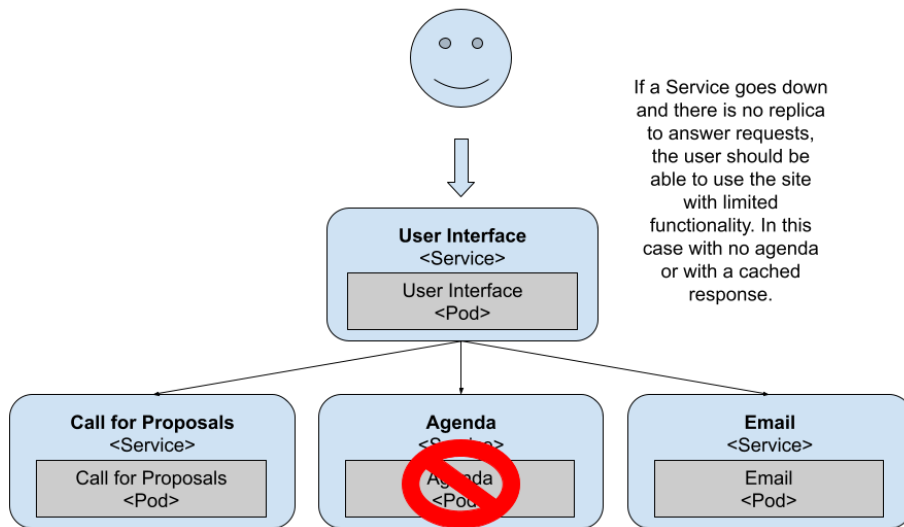


Figure 2.X No pods for the Agenda Service

Right after running this command, the container will be killed and the service will not have any container answering its requests.

Try refreshing the application in your browser:

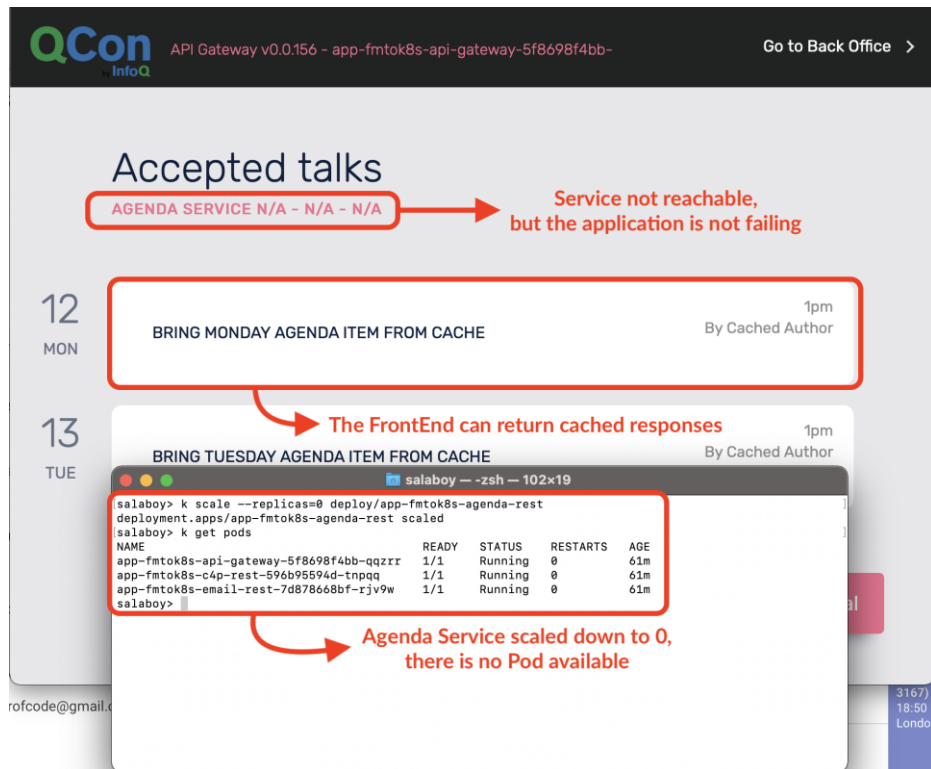


Figure 2.X With a single replica being restarted, there is no backup to answer user requests

As you can see, the application is still running, but the Agenda Service is not available right now. You can prepare your application for such scenarios; in this case, the API Gateway has a cached response to at least show something to the user. If for some reason the Agenda Service is down, at least the user will be able to access other services and other sections of the application. From the application perspective, it is important to not propagate the error back to the user. The user should be able to keep using other services of the application until the Agenda Service is restored.

You need to pay special attention when developing services that will run in Kubernetes as now your service is responsible for dealing with errors generated by downstream services. This is important to make sure that errors or services going down doesn't bring your entire application down. Having simple mechanisms as cached responses will make your applications more resilient and it will also allow you to incrementally upgrade these services without worrying about bringing everything down. Remember, downtime is not allowed.

2.3.3 Dealing with application state is not trivial

Let's scale it up back again to have a single replica:


```
kubectl scale --replicas=1 deployments/app-fmtok8s-agenda-rest
```

Let's create some proposals, so the Agenda has some data in it. You can do that by running the following command from the terminal:

```
curl -X POST http://localhost/api/test
```

This command creates some mock proposals, which you need to accept/reject in the Back Office section of the application. Go ahead and accept all of them.

You should now see all the proposals in the Agenda on the main page.

Now, what do you think will happen if we scale the Agenda Service up to two replicas?

```
kubectl scale --replicas=2 deployments/app-fmtok8s-agenda-rest
```

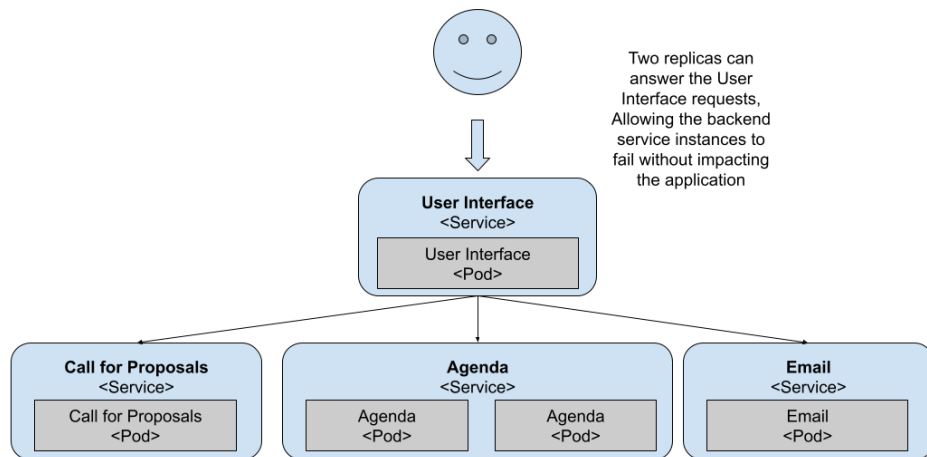


Figure 2.X Two replicas can now deal with more traffic

With two replicas dealing with your user requests, now the User Interface will have two instances to query. If you try refreshing the application multiple times, you will notice that once in a while, the agenda page will come back empty.

This is because the Agenda Service is keeping the Agenda Items in-memory and each Pod has a separate memory space. All your services should be stateless, meaning that you should store state in some kind of storage that can keep the state no matter which Pod tries to access it. Databases are commonly used for externalizing the state out of your services, allowing your services to be scaled independently.

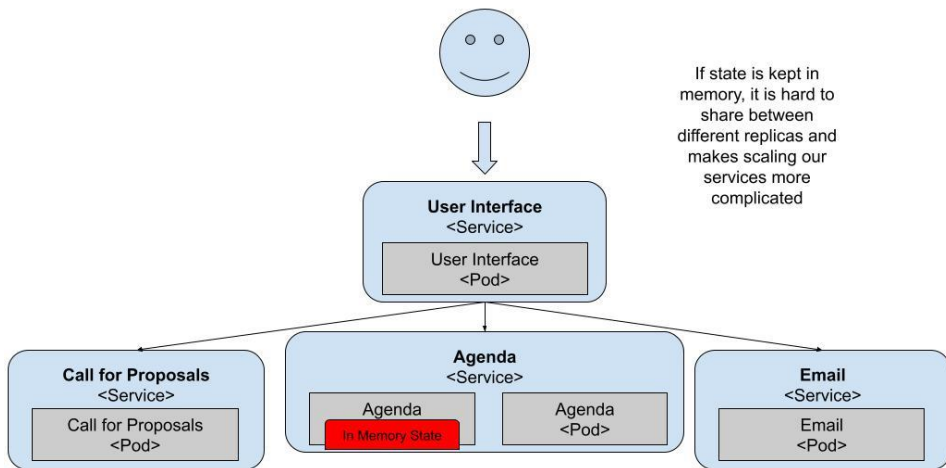


Figure 2.X The Agenda Service is keeping state in-memory

By scaling these services up, we have found an issue with the design of one of the application services. The Agenda Service is keeping state in-memory and that will affect the scaling capabilities from Kubernetes. Luckily for us, we can solve this by adding persistent storage.

For the previous services to keep the data, we can just install Redis and PostgreSQL to back the data for the Agenda and Call for Proposals service. In the same way that we installed the Conference Platform with Helm we can install Redis and PostgreSQL inside our cluster with Helm.

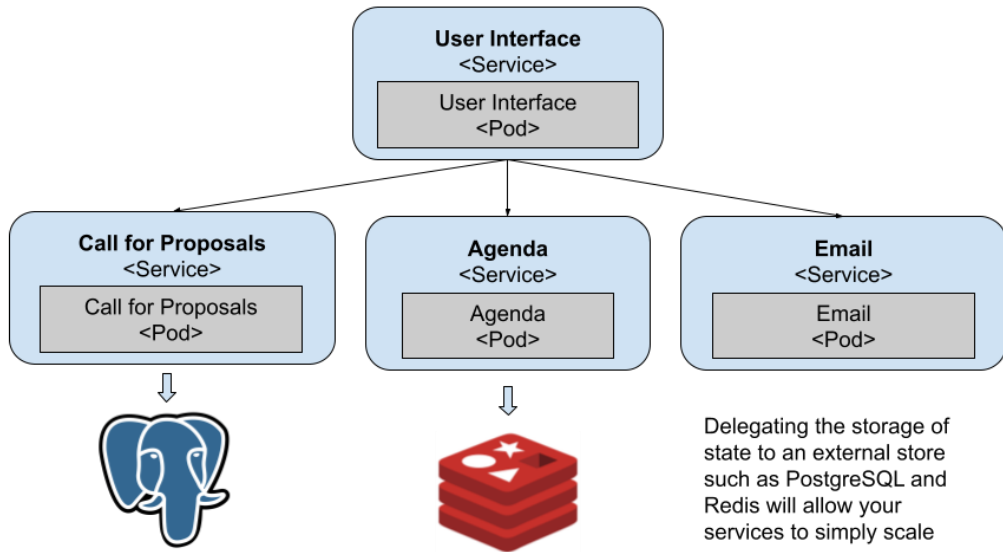


Figure 2.X Both data sensitive services use persistent stores

Understanding your service requirements will help you to plan and automate your infrastructural requirements such as Databases, message brokers, etc. The larger and more complex the application gets the more dependencies on these infrastructural components it will have.

2.3.4 Dealing with inconsistent data

Having stored data into a relational data store like PostgreSQL or a NoSQL approach like Redis doesn't solve the problem of having inconsistent data across different stores. As these stores should be hidden away by the service API, you will need to have mechanisms to check that the data that the services are handling is consistent. In distributed systems it is quite common to talk about "eventual consistency", meaning that eventually, the system will be consistent. Having eventual consistency is definitely better than not having consistency at all. For this example, one thing that we can build is a simple check mechanism that once in a while (imagine once a day) checks for the accepted talks in the Agenda Service to see if they have been approved in the Call for Proposal Service. If there is an entry that hasn't been approved by the Call for Proposal Service (C4P), then we can raise some alerts or send an email to the conference organizers.

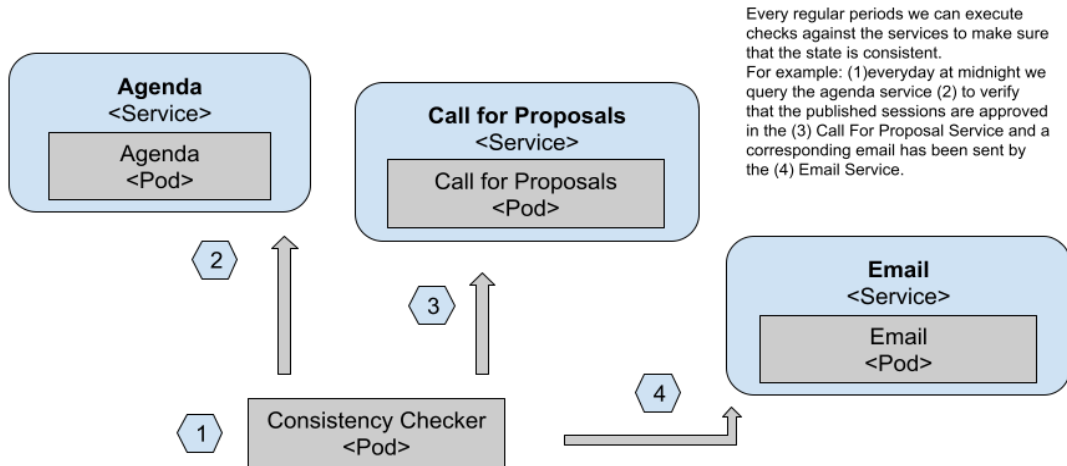


Figure 2.X Consistency checks can run as CronJobs

In figure 2.X we can see how a CronJob (1) will be executed every X period of time, depending on how important it is for us to fix consistency issues. Then it will proceed to query the Agenda Service public APIs (2) to check which accepted proposals are being listed and compare that with the Call for Proposals Service approved list (3). Finally, if any inconsistency is found, an email can be sent using the Email Service public APIs (4).

Think of the simple use case that this application was designed for, what other checks would you need? One that immediately comes to my mind is about verifying that emails were sent correctly for Rejection and Approved proposals. For this use case, emails are really important and we need to make sure that those emails were sent.

2.3.5 Understanding how the application is working

Distributed systems are complex beasts and fully understanding how they work from day one can help you to save time down the line when things go wrong. This has pushed the monitoring, tracing and telemetry communities really hard to come up with solutions that help us to understand how things are working at any given time.

The <https://opentelemetry.io/> OpenTelemetry community has evolved alongside Kubernetes and it can now provide most of the tools that you will need to monitor how your services are working. As stated on their website: "You can use it to instrument, generate, collect, and export telemetry data (metrics, logs, and traces) for analysis in order to understand your software's performance and behavior." It is important to notice that OpenTelemetry focuses on both the behavior and performance of your software as they both will impact your users and user experience.

From the behaviour point of view, you want to make sure that the application is doing what it is supposed to do and by that, you will need to understand which services are calling which other services or infrastructure to perform tasks.

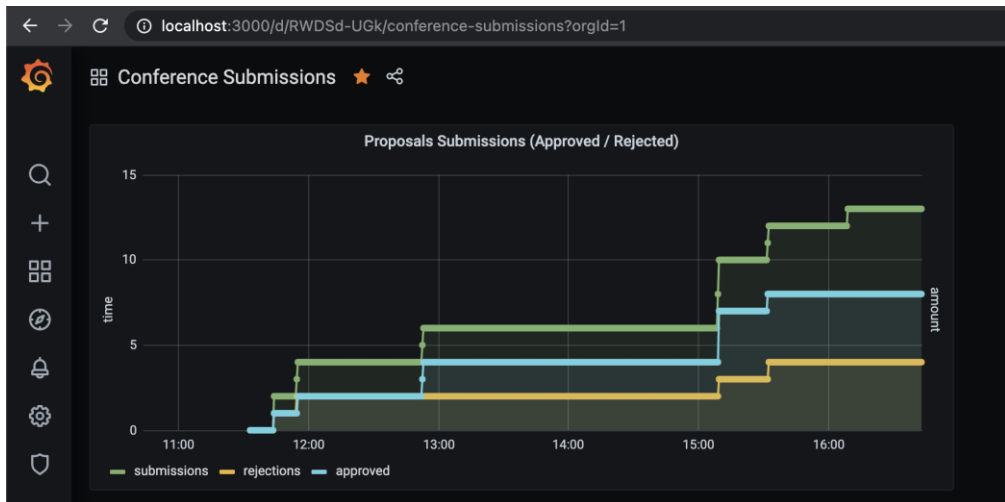


Figure 2.X Monitoring telemetry data with Prometheus & Grafana

Using Prometheus and Grafana allows us to not only see the service telemetry, but also build domain specific dashboards to highlight certain application level metrics, for example the amount of Approved vs Rejected proposals over time as shown in figure 2.x.

From the performance point of view, you need to make sure that services are respecting their Service Level Agreements (SLAs) which basically means that they are not taking too long to answer requests. If one of your services is misbehaving and taking more than usual, you want to be aware of that.

For tracing, you will need to modify your services if you are interested in understanding the internal operations and their performance. OpenTelemetry provides drop-in instrumentation libraries in most languages to externalize service metrics and traces.

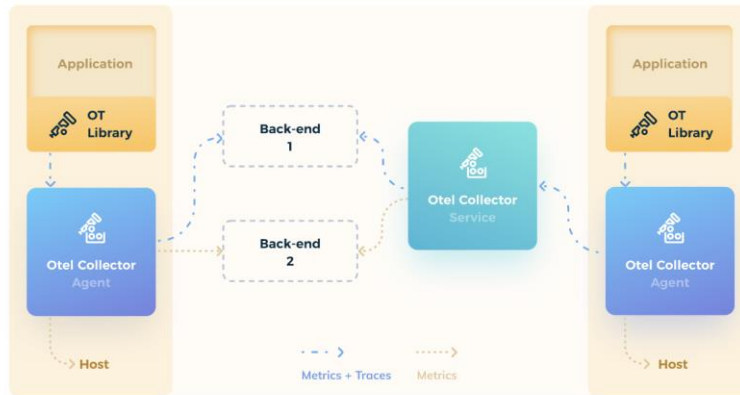


Figure 2.X OpenTelemetry architecture and library

Once the instrumentation libraries are included in our services we can check the traces with tools like Jaeger:

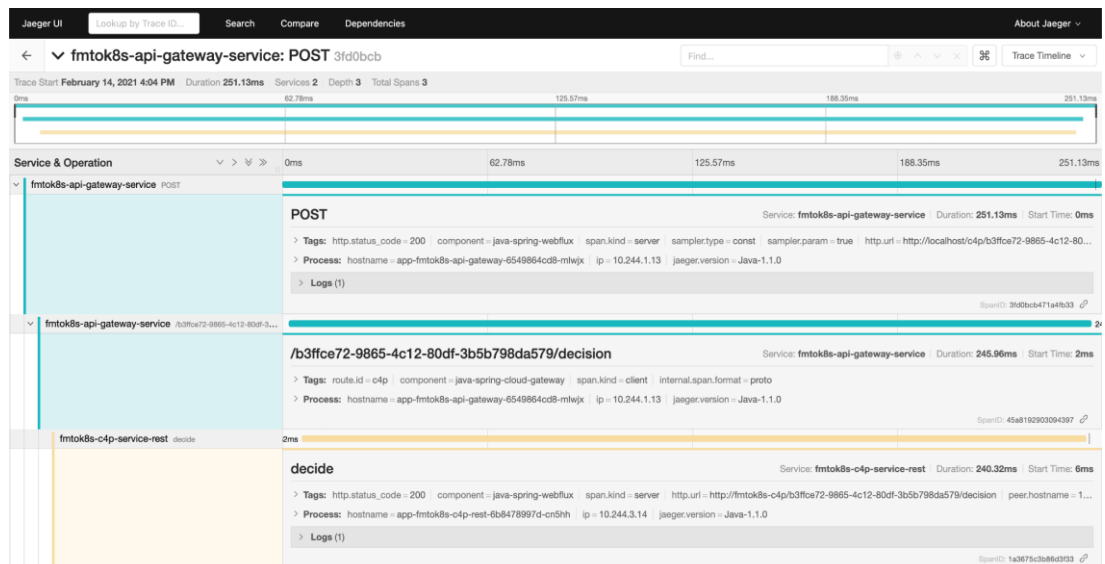


Figure 2.X Tracing diagram from Jaeger

You can see in the previous figure, the amount of time used by each service while processing a single user-request.

The recommendation here is: if you are creating a walking skeleton, make sure that it has OpenTelemetry built-in. If you push monitoring to later stages of the project it will be too late, things will go wrong and finding out who is responsible will take you too much time.

2.3.6 Application security and identity management

If you have ever built a Web application you know that providing identity management (user accounts and user identity) plus Authentication and Authorization is quite an endeavour. A simple way to break any application (cloud-native or not) is to perform actions that you are not supposed to do, such as deleting all the proposed presentations unless you are a conference organizer.

In distributed systems, this becomes challenging as well, as authorization and the user identity needs to be propagated across different services. In distributed architectures, it is quite common to have a component that generates requests on behalf of a user instead of exposing all the services for the user to interact directly. In our example, the API Gateway is this component. Most of the time you can use this external-facing component as the barrier between external and internal services. For this reason, it is quite common to configure the API Gateway to connect with an Authorization & Authentication provider commonly using the OAuth2 protocol.

On the Identity Management front, you have seen that the application itself doesn't handle users or their data and that is a good thing for regulations such as GDPR. We might want to allow users to use their social media accounts to login into our applications without the need for them to create a separate account. This is usually known as social logic.

A popular solution that brings both OAuth2 and Identity Management is Keycloak. An Open Source project created by Red Hat/JBoss which provides a one-stop-shop for Single Sign-On solution and advanced Identity Management. Alternatively, as we will see in Chapter 4, you can always use the Identity Management provided by your Cloud Provider.

Same as with Tracing and Monitoring, if you are planning to have users (and you will probably do, sooner or later) including Single Sign-On and identity management into the walking skeleton will push you to think the specifics of "who will be able to do what", refining your use case even more.

2.3.7 Other challenges

In the previous sections, we have covered a few common challenges that you will face while building Cloud Native applications, but these are not all. Can you think of other ways of breaking this first version of the application?

Notice that tackling the challenges discussed in this chapter will help, but there are other challenges related to how we deliver a continuously evolving application composed of a growing number of services.

In the next chapter, we will cover a more dynamic aspect of our distributed application with Pipelines to deliver new versions of our services, and following that, chapter 4 will explore how we can deal with infrastructural components that are vital for our application to work in a declarative way.

2.4 Summary

- Choosing between local and remote Kubernetes clusters requires serious considerations
 - You can use Kubernetes KIND to bootstrap a local Kubernetes Cluster to develop your application. The main drawback, is that your cluster is limited by your local resources (CPU and Memory) and the fact that it is not a real cluster of machines.
 - You can have an account in a Cloud Provider and do all development against a remote cluster. The main drawback of this approach is that most developers are not used to work remotely all the time and the fact that someone needs to pay for the remote resources.
- Helm helps you to package, distribute and install your Kubernetes applications, in this chapter you have fetched and installed an application into a local Kubernetes Cluster with a single command line
- Understanding which Kubernetes resources are created by your application gives you an idea about how the application will behave when things go wrong and what extra considerations are needed in real-life scenarios.
- Even with very simple applications you will be facing challenges that you will need to tackle one at the time. Knowing these challenges ahead of time help you to plan and architect your services with the right mindset.
- Having a walking skeleton helps you to try different scenarios and technologies in a controlled environment. In this chapter you have experimented with:
 - Scaling up and down your services to see first-hand how the application behaves when things go wrong
 - Keeping state is hard and we will need dedicated components to do this efficiently
 - Having at least 2 replicas for our services maximize the chances to avoid downtime. Making sure that the user-facing components are always up and running guarantees that even when things go wrong the user will be able to interact with parts of the application
 - Having fallbacks and built-in mechanisms to deal with problems when they arise makes your application as a whole more resilient.

3

Service and Environment Pipelines

This chapter covers

- The components needed to deliver Cloud Native applications to Kubernetes
- The advantages of creating and standardizing Service Pipelines for your services
- The benefits of creating Environment Pipelines to manage your environments such as Dev, Staging, Prod
- How projects like Tekton and Jenkins X help you to deliver software more efficiently

In the previous chapter, you installed and interacted with a simple distributed Conference Platform composed of 4 services. This chapter covers what it takes to deliver each of these components in a continuous delivery fashion by using Pipelines as delivery mechanisms. This chapter describes and shows in practice how each of these services can be built, packaged, released and published so they can run in your organization's environments.

This chapter introduces two main concepts: *Service Pipelines* and *Environment Pipelines*. Where the Service Pipeline takes care of all the steps needed to build your software from source code until the artefacts are ready to run. Environment Pipelines, in the other hand, covers the aspects of dealing with the installation and upgrade of new versions of each of these services into a live environment such as staging, testing and production.

This chapter is divided into 3 main sections:

- What does it take to deliver a Cloud-Native application?
- Pipelines
 - What is a Service Pipeline?
 - What is an Environment Pipeline?
- Cloud-Native Implementations
 - Tekton as the Cloud-Native Pipeline Engine
 - Jenkins X (<http://jenkins-x.io>) as a one stop shop for CI/CD in Kubernetes

3.1 What does it take to continuously deliver a Cloud-Native Application?

When working with Kubernetes, teams are now responsible for more moving pieces and tasks involving containers and how to run them in Kubernetes. These extra tasks don't come for free. Teams need to learn how to automate and optimise the steps required to have each service up and running. Tasks that were the responsibility of the operations teams are now becoming more and more the responsibility of the teams in charge of developing each of the individual services. New tools and new approaches give developers the power to develop, run and maintain the services they produce. The tools that we will look at in the second half of this chapter are designed to automate all the tasks involved, from source code to a service that is up and running inside Kubernetes.

This chapter is focused on describing the mechanisms needed to deliver software components (our application services) to multiple environments where these services will run. But before jumping into the tools, let's take a quick look at the challenges that we are facing.

Building and delivering cloud-native applications present significant challenges that teams must tackle:

- **Dealing with different teams** building different pieces of the application. This requires coordination between teams and making sure that services are designed in a way that the team responsible for a service is not blocking other teams progress or their ability to keep improving their services.
- We need to **support upgrading a service without breaking or stopping all the other services** that are running. If we want to achieve continuous delivery, services should be able to be upgraded independently without the fear of bringing down the entire application.
- **Storing and publishing several artifacts per service that can be accessed/downloaded from different environments, that might be in different regions.** If we are working in a cloud environment, all servers are remote and all produced artifacts need to be accessible for each of these servers to fetch. If you are working on an On-premise setup, all the repositories for storing these artifacts will need to be provisioned, configured and maintained in-house.
- **Managing and provisioning different environments for various purposes such as Development, Testing, Q&A and Production.** If you really want to speed up your development and testing initiatives, developers and teams should be able to provision these environments on-demand. Having environments configured as close as possible to the real Production environment will save you a lot of time in catching errors before they are hitting your live users.

As we saw in the previous chapter, the main paradigm shift when working with Cloud-Native applications is that there is no single code base for our application. Teams can work independently on their services, but this requires new approaches to compensate for the complexities of working with a distributed system. If teams will worry and waste time every time that a new Service needs to be added to the system, we are doing things wrong. End to end automation is necessary for teams to feel comfortable about adding or refactoring services. This automation is usually performed by what is commonly known as **Pipelines**. As shown in figure 3.X, these pipelines

describe what needs to be done to build and run our services and usually they can be executed without human intervention.

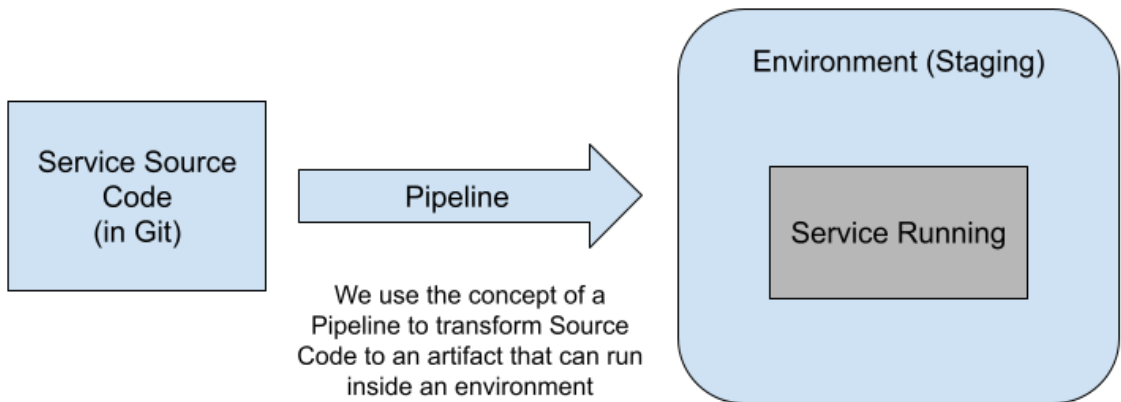


Figure 3.X from source to a running service using a Pipeline

But what are these Pipelines doing exactly? Do we need to create our own Pipelines from scratch? How do we implement these pipelines in our Projects? Do we need one or more pipelines to achieve this?

3.2 Pipelines

This chapter is focused on using Pipelines to build solutions that can be copied, shared and executed multiple times to produce the same results. Pipelines can be created for different purposes and it is quite common to define them as a set of steps (one after the other in sequence) that produce a set of expected outputs. Based on these outputs, these pipelines can be classified into different groups.

Most pipeline tools out there allow you to define pipelines as a collection of tasks (also known as steps or jobs) that will run a specific job or script to perform a concrete action. These steps can be anything, from running tests, copying code from one place to another, deploying software, provisioning virtual machines, etc.

Pipeline definitions can be executed by a component known as the Pipeline Engine, which is in charge of picking up the pipeline definition to create a new pipeline instance that runs each task. The tasks will be executed one after the other in sequence, and each task execution might generate data that can be shared with the following task. If there is an error in any of the steps involved with the pipeline, the pipeline stops and the pipeline state will be marked as in error (failed). If there are no errors, the pipeline execution (also known as pipeline instance) can be marked as successful. Depending on the pipeline definition and if the execution was successful, we should verify that the expected outputs were generated or produced.

In figure 3.x, we can see the Pipeline engine picking up our Pipeline definition and creating different instances that can be parameterized differently to have different outputs. For example,

Pipeline Instance 1 finished correctly while Pipeline Instance 2 is failing to finish executing all the tasks included in the definition. Pipeline Instance 3 in this case is still running.

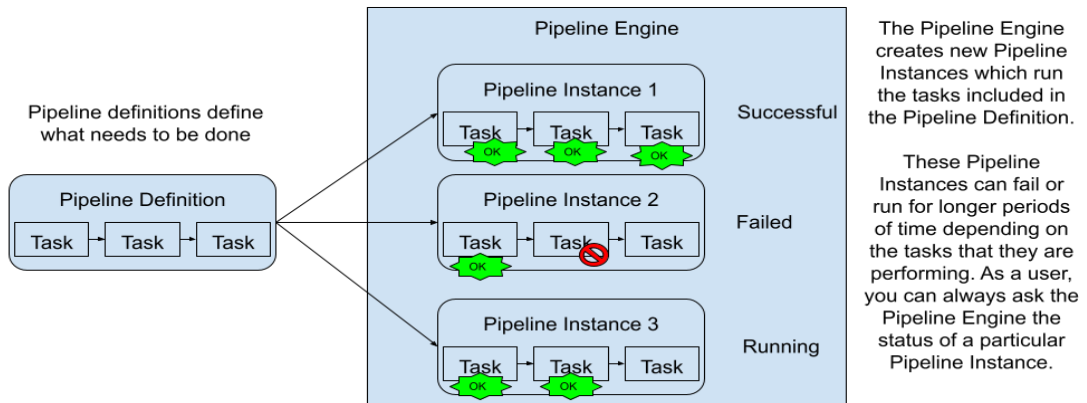


Figure 3.X A Pipeline definition can be instantiated by a Pipeline Engine multiple times

As expected, with Pipeline definitions we can create loads of different automation solutions, but how do these concepts apply to delivering Cloud-Native applications? For Cloud-Native applications, we have very concrete expectations about how to build, package, release and publish our software components (Services) and where these should be deployed. In the context of delivering cloud-native applications we can define two main kinds of pipelines:

- **Service Pipelines:** will take care of the building, unit testing, packaging and distributing (usually to an artifact repository) our software artifacts
- **Environment Pipelines:** will take care of deploying and updating all the services in a given environment such as staging, testing, production, etc.

Service and Environment Pipelines are executed on top of different resources and with different expectations. The following sections go into more detail about the steps that we will need to define for our service and environment pipelines, as these steps will be similar no matter which technology stack we are using. Let's take a look at the archetype of a Service Pipeline definition.

3.2.1 Service Pipelines

A Service pipeline is in charge of defining and executing all the steps required to build, package and distribute a service artifact so it can be deployed into an environment. A Service Pipeline is not responsible for deploying the newly created service artifact, but it can be responsible for notifying interested parties that there is a new version available for the service.

If you standardize how your services need to be built, packaged, and released, you can share the same pipeline definition for different services. You should try to avoid pushing each of your teams to define a completely different pipeline for each service, as they will be probably reinventing something that has been already defined, tested and improved by other teams.

As we will see in this section, there is a considerable amount of tasks that need to be performed and a set of conventions that when followed can reduce the amount of time required to perform these tasks.

The name “Service Pipeline” makes reference to the fact that each of our application services will have a pipeline that describes the tasks required for that particular service. If the services are similar and they are using a similar technology stack, it makes sense for the pipelines to look quite similar. One of the main objectives of these Service Pipelines is to contain enough detail so they can be run without any human intervention, automating all the tasks included in the pipeline end to end.

NOTE: it is tempting to think about creating a single pipeline for the entire application (collection of services), as we did with Monolith applications, but that defeats the purpose of independently updating each service at its own pace. You should avoid situations where you have a single pipeline defined for a set of services, as it will block your ability to release services independently.

CONVENTIONS WILL SAVE YOU TIME

Service Pipelines can be more opinionated on how they are structured and what is their reach, by following some of these strong opinions and conventions you can avoid pushing your teams to define every little detail and discover these conventions by trial and error. The following approaches have been proven to work:

- **Trunk Based Development:** the idea here is to make sure that what you have in the main branch of your source code repository is always ready to be released. You don't merge changes that break the build and release process of this branch. You only merge if the changes that you are merging are ready to be released. This approach also includes using feature branches, which allow developers to work on features without breaking the main branch. When the features are done and tested, developers can send Pull Requests (Change requests) for other developers to review and merge. This also means that when you merge something to the main branch, you can automatically create a new release of your service (and all the related artifacts). This creates a continuous stream of releases which is generated after each new feature is merged into the main branch. Because each release is consistent and has been tested you can then deploy this new release to an environment that contains all the other services of your application. This approach enables the team behind the Service to move forward and keep releasing without worrying about other services.
- **One Service/One Repository/One Pipeline:** you keep your service source code and all the configurations that need to be built, packaged, released and deployed into the same repository. This allows the team behind the service to push changes at any pace they want, without worrying about other services' source code. It is a common practice to have the source code in the same repository where you have the Dockerfile describing how the docker image should be created as well as the Kubernetes manifest required to deploy the service into a Kubernetes cluster. These configurations should include the pipeline definition that will be used to build and package your service.
- **Consumer-Driven Contract Testing:** Your service uses contracts to run tests against other services, unit testing an individual service shouldn't require having other services up and running. By creating Consumer-driven contracts each service can test its own functionality against other services APIs. If any of the downstream services is released a new contract is shared with all the upstream services so they can run their tests against the new version. Chapter XX will cover Consumer-Driven Contract testing in more detail.

If we take these practices and conventions into account, we can define the responsibility of a Service Pipeline as follows: "Transform source code to an artifact that can be deployed in an environment".

SERVICE PIPELINE STRUCTURE

With this definition in mind, let's take a look at what tasks are included in Service Pipelines for Cloud-Native applications that will run on Kubernetes:

- **Register to receive notifications about changes in the source code repository main branch:** (source version control system, nowadays a Git repository): if the source code changes, we need to create a new release. We create a new release by triggering the Service Pipeline.
- **Clone the source code from the repository:** to build the service, we need to clone the source code into a machine that has the tools to build/compile the source code into a binary format that can be executed.
- **Create a new tag for the new version to be released:** based on trunk-based development, every time that a change happens a new release can be created. This will help us to understand what is being deployed and what changes were included in each new

release.

- **Build & Test the source code:**
 - As part of the build process, most projects will execute unit tests and break the build if there are any failures
 - Depending on the technology stack that we are using, we will need to have a set of tools available for this step to happen, for example, compilers, dependencies, linters (static source code analyzers), etc.
- **Publish the binary artifacts into an artifact repository:** we need to make sure that these binaries are available for other systems to consume, including the next steps in the pipeline. This step involves copying the binary artifact to a different location over the network. This artifact will share the same version that the tag that was created in the repository, providing us with traceability from the binary to the source code that was used to produce it.
- **Building a container:** if we are building Cloud-Native services, we will need to build a container image. The most common way of doing this today is using Docker. This step requires the source code repository to have, for example, a Dockerfile defining how this container image needs to be built.
- **Publish the container into a container registry:** in the same way that we published the binary artifacts that were generated when building our service source code, we need to publish our container image into a centralized location where it can be accessed by others. This container image will have the same version as the tag that was created in the repository and the binary that was published. This helps us to clearly see which source code will run when you run the container image.
- **Lint, verify and package YAML files for Kubernetes Deployments (Helm can be used here):** if you are running these containers inside Kubernetes, you need to manage, store and version Kubernetes manifest that define how the containers are going to be deployed into a Kubernetes cluster. If you are using a package manager such as Helm, you can version the package with the same version used for the binaries and the container image.
- **(Optional) Publish these Kubernetes manifests to a centralized location:** if you are using Helm, it makes sense to push these Helm packages (called charts) to a centralized location. This will allow other tools to fetch these charts so they can be deployed in any number of Kubernetes clusters.
- **Notify interested parties about the new version of the service:** if we are trying to automate all the way from source to a service running, the Service Pipeline should be able to send a notification to all the interested services who might be waiting for new versions to be deployed.

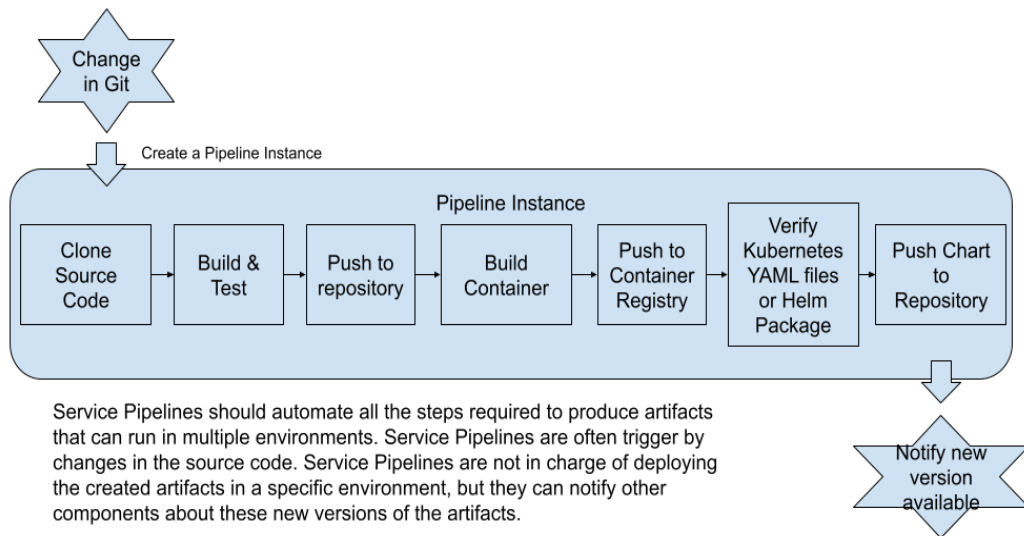


Figure 3.X Tasks expected for a service pipeline

The outcome of this pipeline is a set of artifacts that can be deployed to an environment to have the service up and running. The service itself needs to be built and packaged in a way that is not dependent on any specific environment. The service can depend on other services to be present in the environment to work, for example, infrastructural components such as databases and message brokers, or just other downstream services.

SERVICE PIPELINES IN REAL LIFE

In real life, this pipeline will need to run every time that you merge changes to the main branch of your repository, this is how it should work if you follow a trunk-based development approach:

- When you merge changes to your main branch this pipeline should run creating a new release for your software. This means that you shouldn't be merging code into your main branch if it is not releasable.
- For each of your feature branches, a very similar pipeline should run to verify that the changes in the branch can be built, tested and released. In modern environments, the concept of Github Pull Requests is used to run these pipelines, to make sure that before merging any "Pull Request" a pipeline validates the changes.

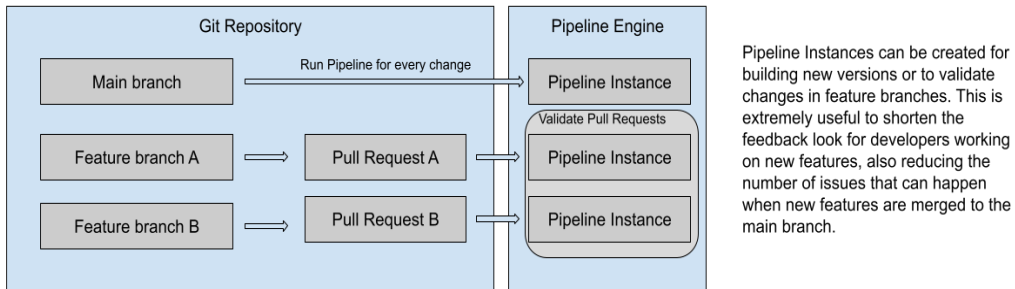


Figure 3.X Service pipelines for main branch and feature branches

This Service Pipeline, shown in figure 3.X represents the most common steps that you will need to execute every time you merge something to the main branch, but there are also some variations of this pipeline that you might need to run under different circumstances. Different events can kick off a pipeline execution, we can have slightly different pipelines for different purposes, such as:

- **Validate a change in a feature branch:** this pipeline can execute the same steps as the pipeline in the main branch, but the artifacts generated should include the branch name, maybe as a version or as part of the artifact name. Running a pipeline after every change might be too expensive and not needed all the time, so you might need to decide based on your needs.
- **Validate a Pull Request/Change Request:** The pipeline will validate that the Pull Request/Change Request changes are valid and that artifacts can be produced with the recent changes. Usually, the result of the pipeline can be notified back to the user in charge of merging the PR and also block the merging options if the pipeline is failing. This pipeline is used to validate that whatever is merged into the main branch is valid and can be released. Validating Pull Requests / Change Requests can be a good option to avoid running pipelines for every change in the feature branches, as when the developer(s) is ready to get feedback from the build system, can create a PR and that will trigger the pipeline. If developers made changes on top of the Pull Request, the pipeline would be retrIGGERED.

Despite small differences and optimizations that can be added to these pipelines, the behaviour and produced artifacts are mostly the same. These conventions and approaches rely on the pipelines executing enough tests to validate that the service that is being produced can be deployed to an environment.

SERVICE PIPELINES REQUIREMENTS

This section covers the infrastructural requirements for service pipelines to work as well as the contents of the source repository required for the pipeline to do its work.

Let's start with the infrastructural requirements that a service pipeline need to work:

- **Webhooks for source code changes notifications:** First of all, it needs to have access to register webhooks to the Git repository that has the source code of the service, so a pipeline instance can be created when a new change is merged into the main branch.
- **Artifact Repository available and Valid credentials to push the binary artifacts:** once the source code is built we need to push the newly created artifact to an artifact repository where all artifacts are stored. This requires having an artifact repository configured and the valid credentials to be able to push new artifacts to it.
- **Docker Registry and Valid credentials to push new container images:** in the same way as we need to push binary artifacts, we need to distribute our docker containers, so Kubernetes clusters can fetch the images when we want to provision a new instance of a service. Having a container registry available with valid credentials is needed to accomplish this step.
- **Helm Chart Repository and Valid Credentials:** Kubernetes manifest can be packaged and distributed as helm charts, if you are using Helm you will need to have a Helm Chart repository and valid credentials to be able to push these packages.

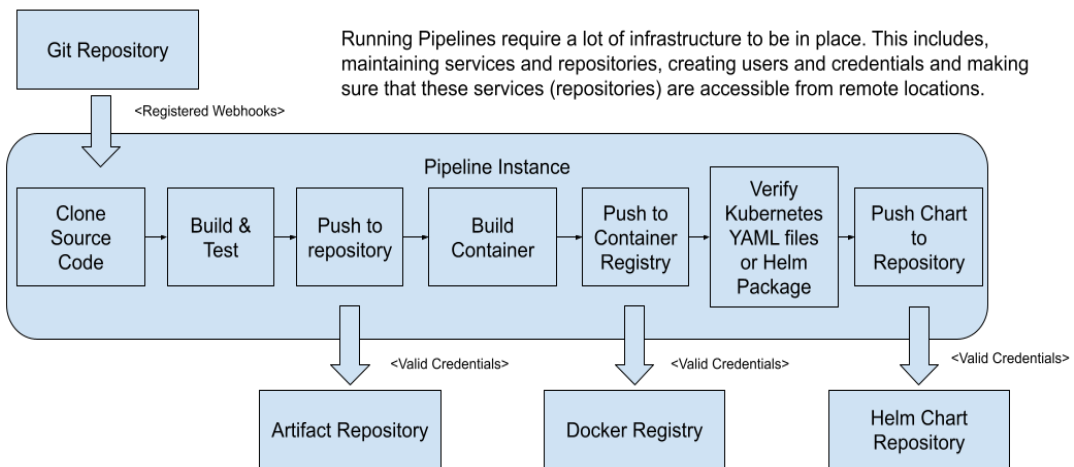


Figure 3.X Service pipelines required infrastructure

For Service Pipelines to do their job, the repository containing the service's source code also needs to have a Dockerfile or the ways to produce a container image and the necessary Kubernetes manifest to be able to deploy the service into Kubernetes. A common practice is to have a Helm chart definition of your service along with your service's source code, in other words, a Helm Chart per Service, as we will see in the following sections.

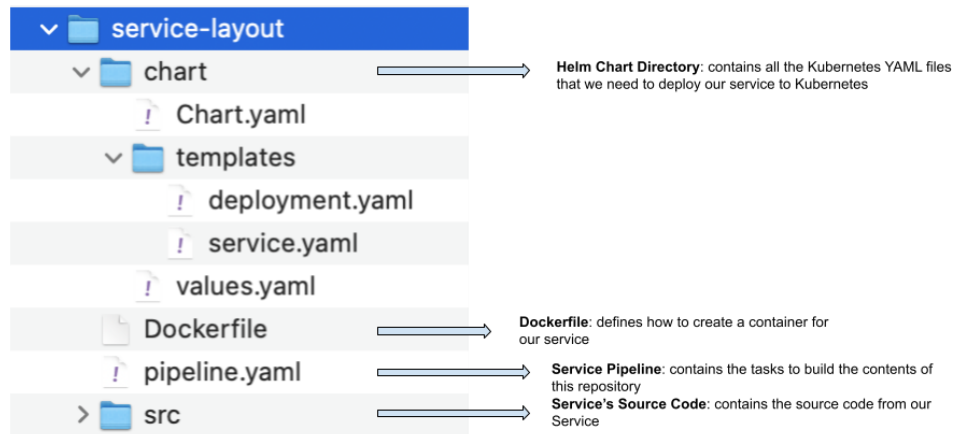


Figure 3.X The Service source code repository needs to have all the configurations for the Service pipeline to work

Figure 3.x shows a possible directory layout of our Service source code repository, which includes the source (`src`) directory which will contain all the files that will be compiled into binary format. The `Dockerfile` used to build our container image for the service and the Helm chart directory containing all the files to create a Helm chart that can be distributed to install the service into a Kubernetes Cluster.

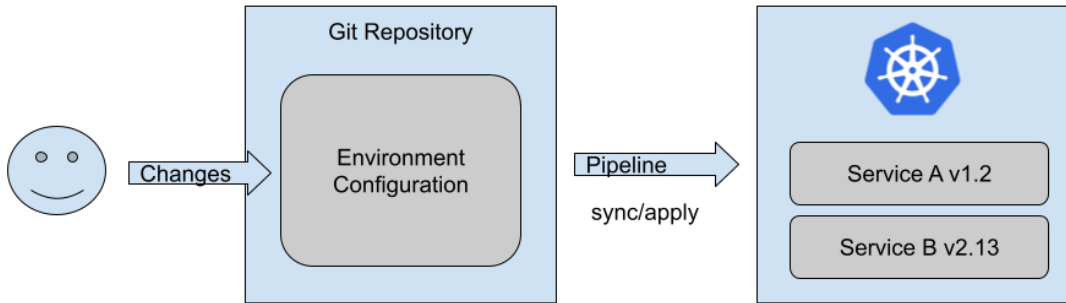
If we include everything that is needed to build, package and run our service into a Kubernetes Cluster, the Service pipeline just needs to run after every change in the main branch to create a new release of the service.

A more advanced but very useful setup might include running the pipeline for Pull Requests (Change Requests) which can include deploying the artifact to a "Preview" environment where developers and other stakeholders can validate the changes before merging them to the main branch. Projects like Jenkins X provide this feature out-of-the-box.

In summary, Service Pipelines are in charge of building our source and related artifacts so they can be deployed into an environment. As mentioned before, Service Pipelines are not responsible for deploying the produced Service into a live environment, that is the responsibility of the Environment Pipeline that is covered in the next sections.

3.2.2 Environment Pipeline

Environment Pipelines are in charge of applying changes to our environments when new services or new versions of existing services are available. Environment Pipelines will run for specific configuration repositories which host the desired configuration for such an environment. A component then will monitor the changes in this repository and for each change apply/sync the desired state into a running Kubernetes Cluster. The whole approach is commonly known as GitOps, as it uses a Git repository as the source of truth for our environments and removes the need to manually interact with the Kubernetes clusters to avoid configuration and security issues.



Environment Pipelines have the goal of monitor configurations changes from a Git Repository and apply those changes to the infrastructure, whenever a new change is detected. Following this approach allows us to rollback changes in the infrastructure by reverting commits and it also allow us to replicate the exact environment configuration by just running the same pipeline against another cluster.

Figure 3.X Defining the state of the cluster using configuration in Git (GitOps)

When you start using Environment Pipelines, you aim to stop interacting, changing or modifying the environment's configuration manually and all interactions are done exclusively by these pipelines. To give a very concrete example, instead of executing `kubectl apply -f` or `helm install` into our Kubernetes Cluster, a pipeline will be in charge of running these commands based on the contents of a Git repository that has the definitions and configurations of what needs to be installed in the cluster.

STEPS INVOLVED WITH AN ENVIRONMENT PIPELINE

An environment pipeline will usually include the following steps:

- **Webhooks for configuration changes notifications:** if there are any new changes into the repository which contains the configurations for the environment a new instance of the Environment Pipeline is triggered
- **Clone the source code from the repository which contains the desired state for our environment:** this step will clone the configurations that had changed to be able to apply them to the cluster. This usually includes doing a ``kubectl apply -f`` or a ``helm install`` command to install new versions of the artifacts. Notice that with both, `kubectl` or `helm`, Kubernetes is smart enough to recognize where the changes are and only apply the differences.
- **Apply the desired state to a live environment:** once the pipeline has all the configurations locally accessible it will use a set of credentials to apply these changes to a Kubernetes Cluster. Notice that we can fine-tune the access rights that the pipelines have to the cluster to make sure that they are not exploited from a security point of view. This also allows you to remove access from individual team members to the clusters where the services are deployed.
- **Verify that the changes are applied and that the state is matching what is described inside the git repository:** once the changes are applied to the live cluster, checking that the new versions of services are up and running is needed to identify if we need to revert back to a previous version. In the case that we need to revert back, it is quite simple as all the history is stored in git, applying the previous version is just looking at the previous commit in the repository.

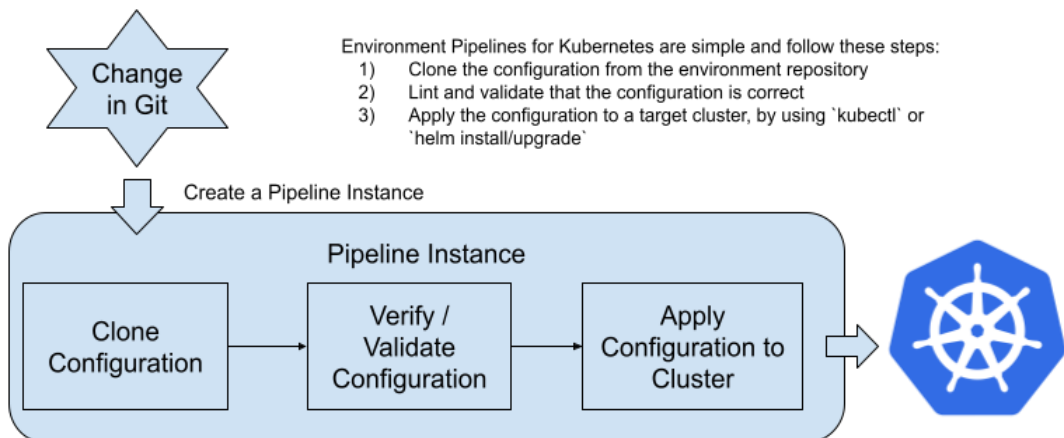


Figure 3.X Environment pipeline for a Kubernetes environment

For the Environment Pipeline to work, a component that can apply the changes to the environment is needed and it needs to be configured accordingly with the right access credentials.

The main idea behind this component is to make sure that nobody will change the environment configuration by manually interacting with the cluster. This component is the only one allowed to change the environment configuration, deploy new services, upgrade services versions or remove services from the environment.

For an Environment Pipeline to work, the following two considerations need to be met:

- The repository containing the desired state for the environment needs to have all the necessary configurations to create and configure the environment successfully.
- The Kubernetes Cluster where the environment will run needs to be configured with the correct credentials for allowing the state to be changed by the pipelines.

The term Environment Pipeline makes reference to the fact that each Environment will have a pipeline associated with it. As having multiple environments is usually required (development, staging, production) for delivering applications, each will have a pipeline in charge of deploying and upgrading the components that are running in them. By using this approach, promoting services between different environments is achieved by sending Pull Requests/Change Requests to the environment's repository and the pipeline will take care of reflecting the changes into the target cluster.

ENVIRONMENT PIPELINE REQUIREMENTS AND DIFFERENT APPROACHES

So what are the contents of these Environment's repositories? In the Kubernetes world, an environment can be a namespace inside a Kubernetes cluster or a Kubernetes cluster itself. Let's start with the most straightforward option, a "Kubernetes namespace". As you will see in the figure that follows, the contents of the Environment Repository is just the definition of which services need to be present in the environment, the pipeline then can just apply these Kubernetes manifests to the target namespace.

The following figure shows 3 different approaches that you can use to apply configurations files to a Kubernetes Cluster. Notice that the three options all include an `environment-pipeline.yaml` file with the definition of the tasks that needs to be executed.

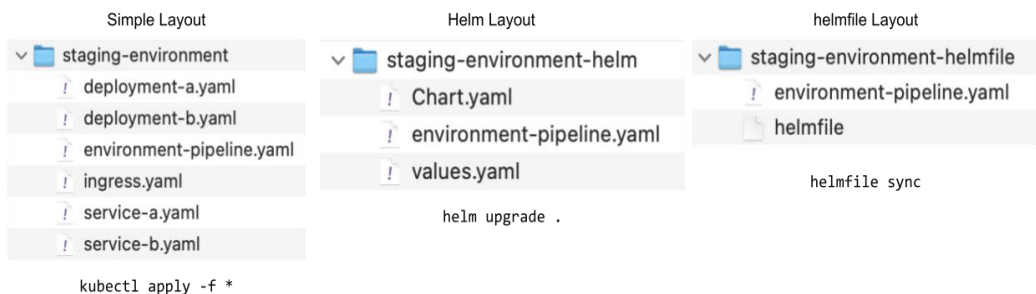


Figure 3.X Three different approaches for defining environments' configurations

The first option (Simple layout) is just to store all the Kubernetes YAML files into a Git repository and then the Environment Pipeline will just use `kubect1 apply -f *` against the configured cluster. While this approach is simple, there is one big drawback, if you have your

Kubernetes YAML files for each service in the service repository, then the environment repository will have these files duplicated and they can go out of sync. Imagine if you have several environments you will need to maintain all the copies in sync and it might become really challenging.

The second option (Helm layout) is a bit more elaborate, now we are using Helm to define the state of the cluster. You can use Helm dependencies to create a parent chart that will include as dependencies all the services that should be present in the environment. If you do so, the environment pipeline can use ``helm update .`` to apply the chart into a cluster. Something that I don't like about this approach is that you create one Helm release per change and there are no separate releases for each service. The prerequisite for this approach is to have every service package as a Helm chart available for the environment to fetch.

The third option is to use a project called ``helmfile`` (<https://github.com/roboll/helmfile>) which was designed for this very specific purpose, to define environment configurations. A ``helmfile`` allows you to declaratively define what Helm releases need to be present in our cluster. This Helm releases will be created when we run ``helmfile sync`` having defined a ``helmfile`` containing the helm releases that we want to have in the cluster.

No matter if you use any of these approaches or other tools to do this, the expectation is clear. You have a repository with the configuration (usually one repository per environment) and a pipeline will be in charge of pickup the configuration and using a tool to apply it to a cluster.

It is common to have several environments (staging, qa, production) , even allowing teams to create their own environments on-demand for running tests or day to day development tasks.

If you use the "one environment per namespace" approach, it is common to have a separate git repository for each environment, as it helps to keep access to environments isolated and secure. This approach is simple, but it doesn't provide enough isolation on the Kubernetes Cluster, as Kubernetes Namespaces were designed for logical partitioning of the cluster and in this case, the Staging Environment will be sharing with the Production environment the cluster resources.

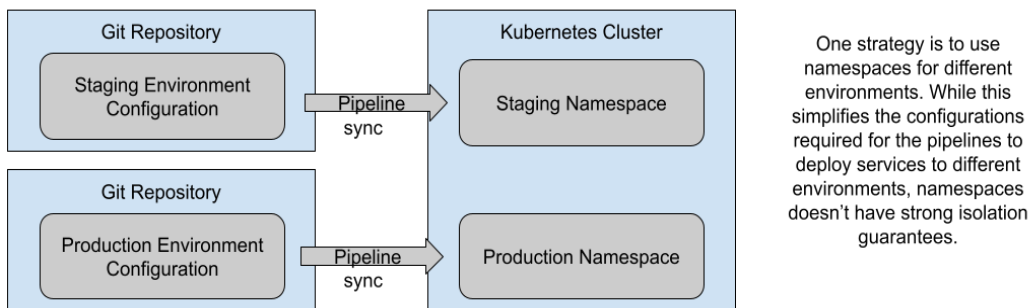
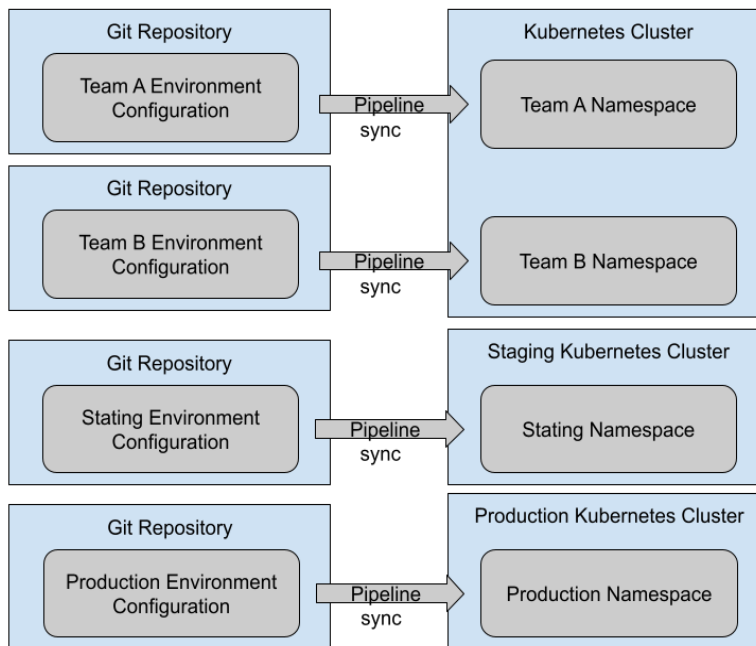


Figure 3.X One Environment per Kubernetes Namespace

An alternative approach can be to use an entirely new cluster for each environment. The main difference is isolation and access control. By having a cluster per environment you can be more strict in defining who and which components can deploy and upgrade things in these environments and have different hardware configurations for each cluster, such as multi-region setups and other

scalability concerns that might not make sense to have in your Staging and Testing environments. By using different clusters you can also aim for a multi cloud setup, where different environments can be hosted in different cloud providers.

Figure 3.x shows how you can use the namespace approach for Development Environments which will be created by different teams and then having separated clusters for Staging and Production. The idea here is to have the Staging and Production cluster configured as similarly as possible, so applications deployed behave in a similar way.



A more realistic approach can be to use the same cluster for different teams day to day work and have separate clusters for Staging and Production Environments.

New Teams can be added by just copying another's team repository and creating a new namespace.

For a service to be promoted to the Staging or Production environment, Pull Requests/Merge

Requests can be sent to the specific environment git repository.

Figure 3.X One Environment per Kubernetes Cluster

3.2.3 Service Pipelines + Environment Pipelines

Finally, let's take a look at how Service Pipelines and Environment Pipeline connect. The connection between these two pipelines happens via Pull Request to repositories, as the pipelines will be triggered when changes are submitted and merged:

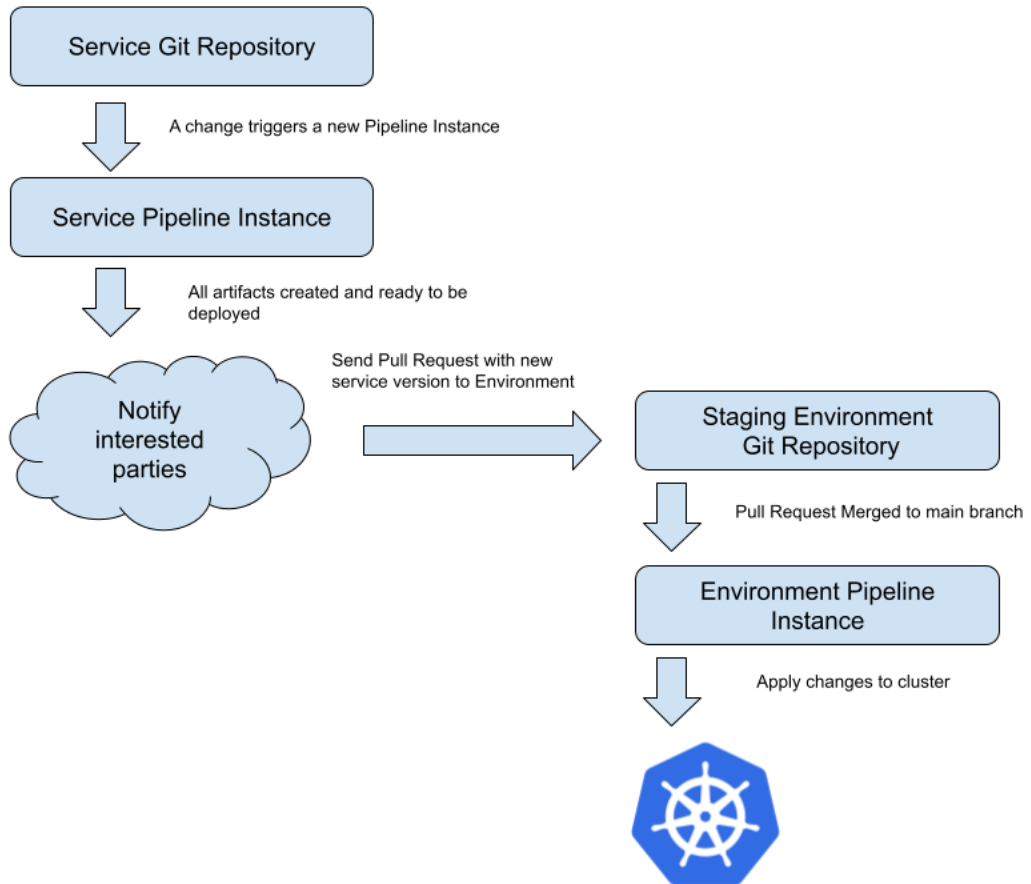


Figure 3.X A Service Pipeline can trigger an Environment Pipeline via a Pull Request

The Service Pipeline, after all the artifacts were released and published, can send an automatic Pull Request to the Environment repository where the service needs to be updated or deployed triggering the Environment Pipeline. This simple but effective mechanism allows automated or a manual Pull Request to be sent every time that we want to upgrade or deploy something into our environments. For certain environments, such as staging or development environments, you can automate the merging of the Pull Request containing new services version enabling changes on the service repositories to be propagated automatically to these low-risk environments.

In the next section, we look at implementations of these pipelines with a Cloud-Native approach.

3.3 Implementing Cloud-Native Pipelines

As we discussed before, if you want to implement these (Service and Environment) or other pipelines you will need a Pipeline Engine that can deal with automating all these tasks so you don't actually need to implement any code to, for example, clone your source code from your version control system or publish your artifacts to a centralized repository. So far, we have talked about Pipelines at a conceptual level, but now it is time to look at a real implementation.

There have been several Pipeline Engines built in the past such as Hudson/Jenkins, but in this book, we are looking into Cloud-Native solutions, which were built specifically to leverage Kubernetes. If we are talking specifically about Kubernetes, we will be looking at a solution with the following characteristics:

- The pipeline engine can scale based on the amount of work that it has, so if we have more pipelines to be executed, the pipeline engine should be able to just request more hardware and dynamically scale
- The pipeline engine should use Kubernetes mechanisms to run the pipeline instances
- The pipeline engine should be integrated with the Kubernetes ecosystem

Most of the tasks related to Service and Environment pipelines are quite common and have been already implemented and shared as open-source code by teams around the world. If you have particular needs of integrating with private systems you might need to implement some integrations, for most of the general steps in the pipelines, there are solutions already available for most technology stacks.

Because this book is using Kubernetes as the target platform, the next section is an overview of a popular Cloud-Native Pipeline Engine called Tekton.

3.3.1 Tekton Cloud-Native Pipelines

Tekton was originally created as part of the Knative project (<http://knative.dev>) from Google, initially called Knative Build and later separated from Knative to be an independent project. You can visit the project site at: <http://tekton.dev>. Tekton's main characteristic is that it is a Cloud-Native Pipeline Engine designed for Kubernetes. In this section, we will look into how to use Tekton to define Service and Environment Pipelines.

TEKTON IN ACTION

In Tekton you have two main concepts: Tasks and Pipelines. Tekton, the Pipeline Engine is composed of a set of components that will understand how to execute Tasks and Pipelines that we define. Tekton, as most of the Kubernetes projects covered in this book, can be installed into your Kubernetes cluster by running `kubectrl` or using Helm Charts.

When you install Tekton you are installing a set of Custom Resource Definitions, which are extensions to the Kubernetes APIs which in the case of Tekton defines what Tasks and Pipelines are. Tekton also installs the Pipeline Engine itself that knows how to deal with Tasks and Pipelines resources when we create them.

You can install Tekton using Helm thanks to a collaboration with the Continuous Delivery foundation: <https://github.com/cdfoundation/tekton-helm-chart>

As with every Helm Chart you will need to first add the Helm Chart repository and then install the chart:

```
helm repo add cdf https://cdfoundation.github.io/tekton-helm-chart/
helm install tekton cdf/tekton-pipeline
```

Once you install the Tekton Helm chart you will see that a new namespace was created. This new namespace called `tekton-pipelines` contains the pipeline controller (which is the pipeline engine) and the pipeline webhook listener, which is used to listen for events coming from external sources, such as git repositories.

You can also install the `tkn` command-line tool, which helps a lot if you are working with multiple tasks and complex pipelines. You can follow the instructions installations here: <https://github.com/tektoncd/cli>

If you now list all the CustomResourceDefinitions that are installed in the cluster and belongs to tekton you will see that there is a new set of resources that you can use:

```
> kubectl get crds | grep tekton #1
clustertasks.tekton.dev
conditions.tekton.dev
pipelineresources.tekton.dev
pipelineruns.tekton.dev #2
pipelines.tekton.dev #3
runs.tekton.dev
taskruns.tekton.dev #4
tasks.tekton.dev #5
```

#1 We can get all the Custom Resource Definitions (crd) using `kubectl` and filtering only the ones related to tekton

#2 #3 #4 #5 Are the resource types that we will be creating for our pipelines, notice that you can get these resources by also using ``kubectl get`` and the resource type

Once you have Tekton installed you can start by creating a simple task definition in YAML. A Task in Tekton will look like a normal Kubernetes resource:

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: echo-hello-world
spec:
  steps:
    - name: echo
      image: ubuntu #1
      command:
        - echo #2
      args:
        - "Hello World" #3
```

#1 The docker image called `ubuntu` is going to be used for this task

#2 The main command used by this task is `echo`, notice that is an list of commands

#3 The command arguments (`args`) in this case is just a "Hello World" string, notice that you can send a list of arguments for more complex commands

Derived from this example, you can create a task for whatever you want, as you have the flexibility to define which container to use and which commands to run. Once you have the task definition, you need to make that available to Tekton by applying this file to the cluster with ``kubectl apply -f task.yaml``. By applying the file into Kubernetes we are only making the definition available to the Tekton components in the cluster, but the task will not run.

If you want to run this task, a task can be executed multiple times, Tekton requires you to create a TaskRun resource like the following:

```
apiVersion: tekton.dev/v1beta1
kind: TaskRun
metadata:
  generateName: hello-run-
spec:
  taskRef:
    name: echo-hello-world
```

Alternatively, you can use ``tkn`` to start a task definition:

```
salaboy> tkn task start echo-hello-world
TaskRun started: echo-hello-world-run-q7vgw
```

In order to track the TaskRun progress run:
`tkn taskrun logs echo-hello-world-run-q7vgw -f -n default`

And then get the logs by running the suggested command:

```
salaboy> tkn taskrun logs echo-hello-world-run-q7vgw -f -n default
[echo] Hello World
```

Whether you apply this TaskRun to the cluster (``kubectl apply -f taskrun.yaml`` or using ``tkn task start``), the Pipeline Engine will execute this task. On the YAML file, you can see this resource doesn't have a ``metadata.name``, instead it has a ``metadata.generateName`` field. When Tekton runs this task, it will generate a unique name for the resource to track that specific execution. You can keep applying the same resource and for each time you apply it to the cluster, a new execution will be scheduled. The TaskRun resources are used to keep all the information of the execution of the Task definitions and the outputs for these executions.

PIPELINES IN TEKTON

A Task in itself can be useful but Tekton becomes really interesting when you create sequences of these tasks by using Pipelines.

A pipeline is a collection of these tasks in a concrete sequence, let's take a look at a simple Service Pipeline defined in Tekton (`service-pipeline.yaml`):

```

apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: service-pipeline
spec:
  resources:
  ...
  tasks:
  - name: clone-repository
    taskRef:
      name: git-clone
      bundle: gcr.io/tekton-releases/catalog/upstream/git-clone:0.4
    params:
    ...
  - name: maven-build
    runAfter: [clone-repository]
    taskRef:
      name: maven
      bundle: gcr.io/tekton-releases/catalog/upstream/maven:0.2
    params:
    ...
  - name: docker-image-build-and-publish
    runAfter: [maven-build]
    taskRef:
    name: kaniko
    bundle: gcr.io/tekton-releases/catalog/upstream/kaniko:0.4
    params:
    ...

```

You can find the full pipeline definition here: <https://github.com/salaboy/from-monolith-to-k8s/blob/master/tekton/service-pipeline.yaml>

Once again you will need to apply this pipeline resource to your cluster for Tekton to know about: ``kubectl apply -f service-pipeline.yaml``.

As you can see in the pipeline definition the ``spec.tasks`` field contains an array of task references. These tasks need to be already deployed into the cluster and the Pipeline definition is in charge of defining the sequence in which these tasks will be executed. These Task references can be your own tasks, or as in the example, this can come from the Tekton Catalog, which is a repository that contains community-maintained task definitions that you can reuse.

In the same way, as Tasks need TaskRuns for the executions, you will need to create a PipelineRun for every time that you want to execute your Pipeline.

```

apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  generateName: service-pipeline-
spec:
  pipelineRef:
    name: service-pipeline

```

Now when you apply this file to the cluster ``kubectl apply -f pipelinerun.yaml`` Tekton will execute the pipeline by running all the tasks defined in the pipeline definition. Alternatively you can also use ``tkn``:

```
salaboy> tkn pipeline start service-pipeline
```

Following a similar approach to the service pipeline defined here, we can create also an Environment Pipeline which will be in charge of syncing the state of a git repository containing the services that need to be deployed in the environment to a running Kubernetes Cluster.

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: staging-environment-pipeline
spec:
  tasks:
    - name: clone-repository
      taskRef:
        name: git-clone
        bundle: gcr.io/tekton-releases/catalog/upstream/git-clone:0.4
      params:
        - name: url
          value: ${params.gitRepository}
        - name: revision
          value: ${params.gitRevision}
    - name: helm-update
      runAfter: [clone-repository]
      taskSpec:
        steps:
          - name: update
            image: quay.io/roboll/helmfile:helm3-v0.135.0
            script: |
              #!/bin/sh
              set -ex
              helmfile sync
```

Once again, this pipeline is quite simple and you can find the full pipeline definition here: <https://github.com/salaboy/from-monolith-to-k8s/blob/master/tekton/environment-pipeline.yaml>

This pipeline contains two tasks, one clone the sources from the Environment Repository, which in this case contains a helmfile that defines what is installed in the environment and the second task just applies this file against a configured cluster. The repository used for running this pipeline can be found here: <https://github.com/salaboy/fmtok8s-staging-env>

You can run this pipeline with:

```
salaboy> tkn pipeline start staging-environment-pipeline -s runner -w
name=sources,volumeClaimTemplateFile=workspace-template.yaml
```

This pipeline, because it is going to execute requests against the Kubernetes APIs, requires specific RBAC (role-based access control configurations) hence the `-s` parameters indicating the `runner` service account is needed for this pipeline to work.

If required, you can find a step by step tutorial on how to install Tekton in your Kubernetes Cluster and how to run Service and Environment pipelines at the following repository: <https://github.com/salaboy/from-monolith-to-k8s/tree/master/tekton>

TEKTON ADVANTAGES AND EXTRAS

As we have seen Tekton is super flexible and allows you to create pretty advanced pipelines, and it includes other features such as:

- Input and Output mappings to share data between tasks
- Event triggers that allow you to listen for events that will trigger Pipelines or Tasks
- A Command-Line tool to easily interact with Tasks and Pipelines from your terminal
- A Simple Dashboard to monitor your pipelines and tasks executions.

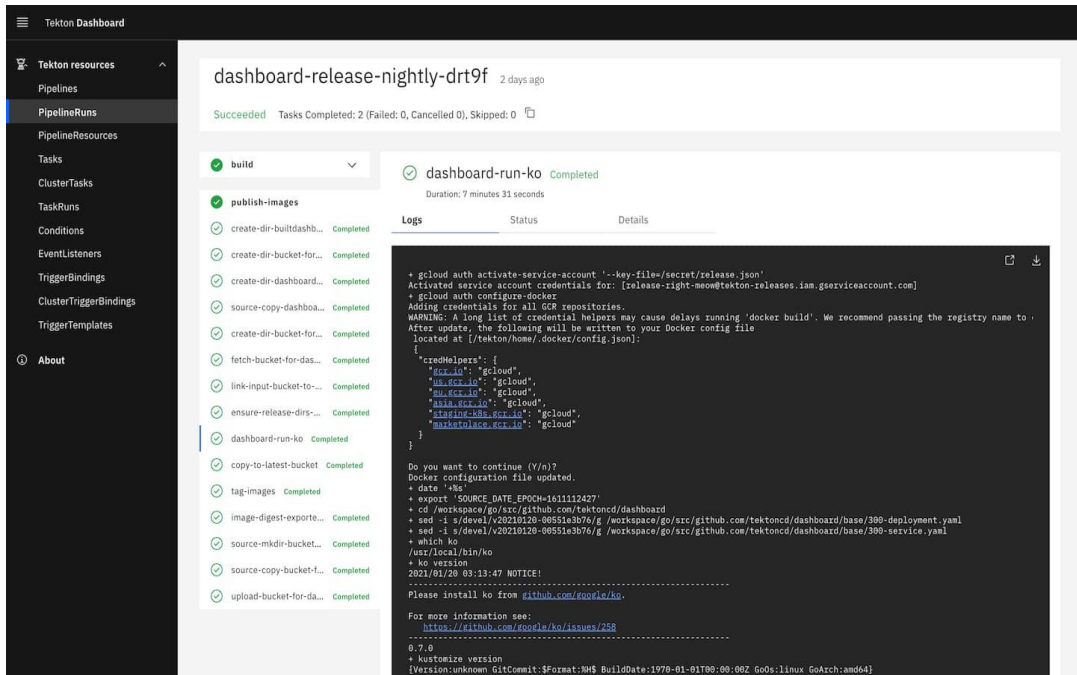


Figure 3.X Tekton Dashboard a user interface to monitor your pipelines

Figure 3.x shows the community-driven Tekton Dashboard which you can use to visualize the execution of your pipelines. Remember that because Tekton was built to work on top of Kubernetes, you can monitor your pipelines using `kubectl` as with any other Kubernetes resource, but nothing beats a User Interface for less technical users.

But now, if you want to implement a Service Pipeline with Tekton, you will spend quite a bit of time defining Tasks, the Pipeline, how to map inputs and outputs, defining the right events listener for your Git repositories and then going more low-level into defining which docker images you will use for each Task. Creating and maintaining these pipelines and their associated resources can become a full-time job and for that Tekton launched an initiative to define a Catalog where Tasks (Pipelines and Resources are planned for future releases) can be shared, the Tekton Catalog: <https://github.com/tektoncd/catalog>

With the help of the Tekton Catalog, we can create pipelines that reference Tasks that have been defined in the catalog, hence we don't need to worry about defining them. You can also visit <https://hub.tekton.dev> which allows you to search for Task definitions and provides you with detailed documentation about how to install and use these tasks in your pipelines.

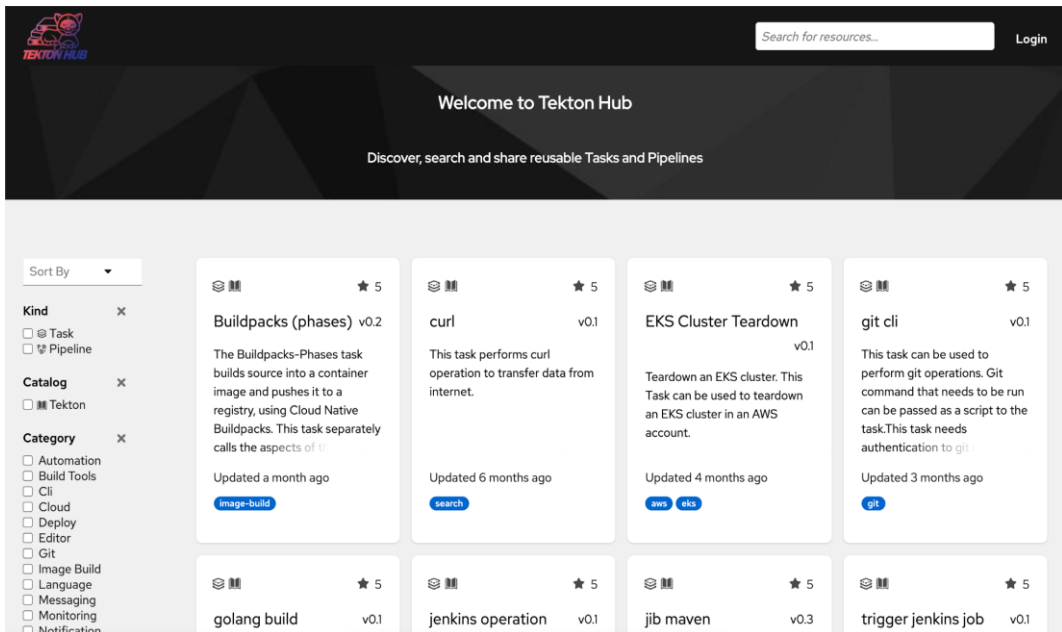


Figure 3.X Tekton Hub a portal to share and reuse Tasks and Pipeline definitions

One good example to look at is the Helm Upgrade from Source Task which allows you to upgrade a Helm release based on the Helm Chart source. You can take a look at the task documentation here: <https://hub.tekton.dev/tekton/task/helm-upgrade-from-source>. A Task like this one, can be really helpful to implement an environment Pipeline, where we define the content of the Environment with a Helm chart and then every time that we change the content of the chart we just upgrade the Helm Release with Tekton.

Tekton Hub and the Tekton Catalog allows you to reuse Tasks and Pipelines that had been created by a large community of users and companies. But you quickly realize that Tekton is not very opinionated on what you do with the pipelines that you build, hence there are loads of different options and choices that you will need to make when creating a Service or an Environment pipeline. Wouldn't it be nice if all the defaults, conventions and best practices that are being used by tons of other people would be shared in a single tool? In the next section, we will take a look at Jenkins X and how it takes a holistic approach to solve CI/CD for Kubernetes.

3.3.2 Jenkins X: A one-stop-shop for CI/CD in Kubernetes

Jenkins X (<http://jenkins-x.io>) offers a whole package for doing CI/CD on top of Kubernetes. This goes from installing all the components that you need, for example, the Pipeline Engine (Tekton in this case) to the user interface to monitor the pipelines, artifact repositories and the best practices applied to convention-based pipelines. In theory, with Jenkins X, after you installed it you just need to import your services with a single command line and both Service and Environment Pipelines will be set up for your project. Jenkins X takes a holistic approach to solve CI/CD hence it is a complex

project to get your head around at first. This section aims to explain how Jenkins X solves and simplify the whole CI/CD process for Kubernetes projects, my recommendation is always to try the project on your own clusters, but it will require a whole book to explain how all the pieces fit together.

JENKINS X ARCHITECTURE

If you are interested in trying out Jenkins X, you need to understand that Jenkins X is divided into two main components:

- **A command-line tool called `jx`**: You need to download the `jx` command-line from the <http://jenkins-x.io> site or use tools such as brew for Mac OSX or a package manager to install the command line locally.
- **A set of services and components installed into a Kubernetes Cluster**: By using the `jx` command-line tool you can install Jenkins X in an existing Kubernetes Cluster. You will probably install Jenkins X in a separate cluster where your applications are deployed.

The following figure shows the components that are installed by default when you install Jenkins X, no matter in which Cloud Provider you decide to install it.

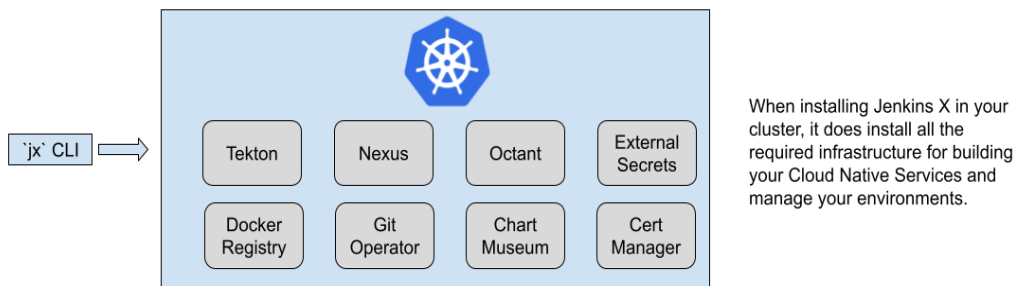


Figure 3.X Jenkins X Components installed on a Kubernetes Cluster

With the command-line tool called `jx` you can install Jenkins X into your cluster. In order to run Jenkins X you need a Kubernetes Cluster, to begin with, Jenkins X can create one for you. Jenkins X uses a tool called Terraform (<https://www.terraform.io>) to create the cluster and install the required components into different Cloud Providers. It also uses Kubernetes and Cloud Provider-specific services to make sure that your service and environment pipelines have everything they need to work. For example, if you are installing Jenkins X in Google Cloud Platform, it configures GCR (Google Container Registry) to be used to store the docker images produced by your service pipelines. In this section, we will look at what gets installed when you install Jenkins X in your cluster and the basic commands to go from zero to full CI/CD for your Services.

Jenkins X installs and configures all the components that are needed for your Service and Environment pipelines, making sure that components can register to git repositories for changes and the pipelines are automatically triggered when a change is pushed. Jenkins X also installed the pipeline engine (Tekton) and the repositories needed for the artifacts that are going to be created as part of the pipelines. Remember that you will need a place to store your binary artifacts (Nexus

is installed), your docker images (depending on the Cloud provider where you are installing Jenkins X a different container registry will be used, if you are in Google Cloud, GCR.io will be used) and also your Kubernetes manifest. Jenkins X relies on Helm for packaging and distributing Kubernetes manifests (YAML files) hence it installs Chart Museum into the cluster.

When you install Jenkins X in your Kubernetes Cluster, besides installing all the components that it needs, it will automatically create two environments: Staging and Production with their respective Environment Pipelines and the Git repositories containing the default configurations. As you can see in the following figure, your applications' services will start in the `dev` environment, automatically promoted to the `staging` environment and finally reaching to the `production` environment, where users consume our services.

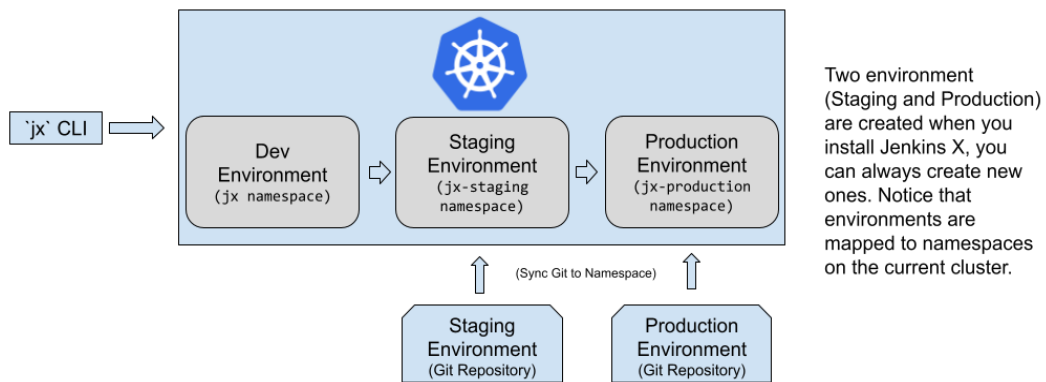


Figure 3.X Jenkins X pre-defined environments

Both pipelines will be associated with two separate namespaces, where the workloads for each environment will run. By default, no service pipeline is created, because there is no application Service defined yet.

As part of the installation process, Jenkins X creates two default environments called "staging" and "production", where automatic promotions are configured for the staging environment, meaning that whenever there is a new version of our services, the Environment Pipeline will be in charge of "promoting" (adding or upgrading the service) into the environment. The production environment requires manual promotion, this means that a user will be in charge of deciding when new versions of the services can be installed in the production environment.

So, far after just installing Jenkins X, we have two environments and default environment pipelines ready to go. You are now ready to start adding Services.

YOUR PROJECTS AND JENKINS X

Once all the components needed by the pipelines are installed and configured to work together you can start "importing" your projects to Jenkins X.

Let's take a look at how this will look for our Conference Platform application which was composed of four services. If we have four services, we need four service pipelines. Each of these pipelines will release each service independently. In Jenkins X, we can create these pipelines by

just importing our project to our Jenkins X installation and we do that by running ``jx import`` inside the directory where our Service code is.

Because the Conference Platform Services were built with Java, let's see how the process of importing a project to Jenkins X works.

Let's assume that we want to create and run a Service Pipeline for our Agenda Service and all we have is the source code of the service. When you run ``jx import`` inside the project source code the following steps are executed locally inside the ``jx`` CLI tool:

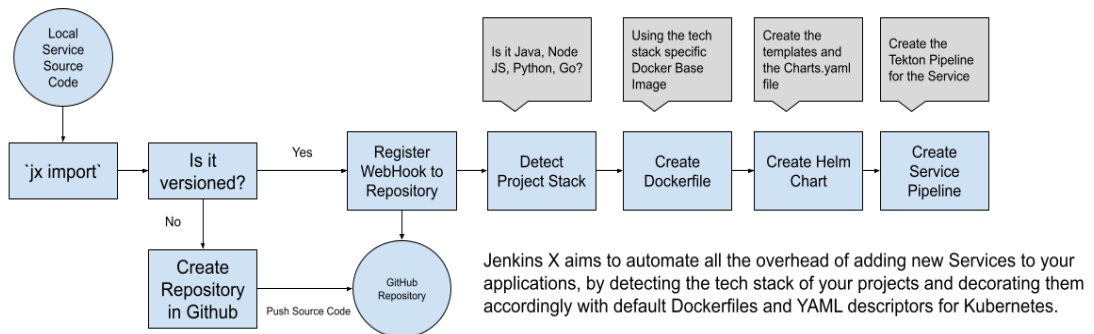


Figure 3.X Jenkins X import process

The first check that is done is about if the project is already in version control or not. If it is not, it will suggest you create a new repository (in Github or your preferred source version control system) to be able to monitor the changes and trigger releases. The next step is to register a Webhook, so Jenkins X components get notified whenever there is a new change or a Pull Request / Change Request. Then the decoration steps happen, in other words, Jenkins X will detect what kind of project do we have, if it is a Java, Node JS, Python or Go project and based on that it will decide to create a Dockerfile, a Helm Chart including the Kubernetes manifests and finally the Service Pipeline using Tekton is created.

A set of simple checks are executed here by Jenkins X, for example, because we have a Java project which is being built by Maven Jenkins X detects the ``pom.xml`` file which describes the project and its dependencies and knows what tools and steps are needed to build this kind of project. Once the project type is detected a Dockerfile is provided for that kind of project if none is detected to be present. The same with creating a Helm chart, if Jenkins X detects that there is no helm chart defined inside the project directory it will automatically create one based on a default chart for the detected project. Finally, Jenkins X will create a default service pipeline for the project which is strictly related to the kind of the detected project as it contains Tasks that have the right tools to build the project source code into a binary, a docker image and a helm chart.

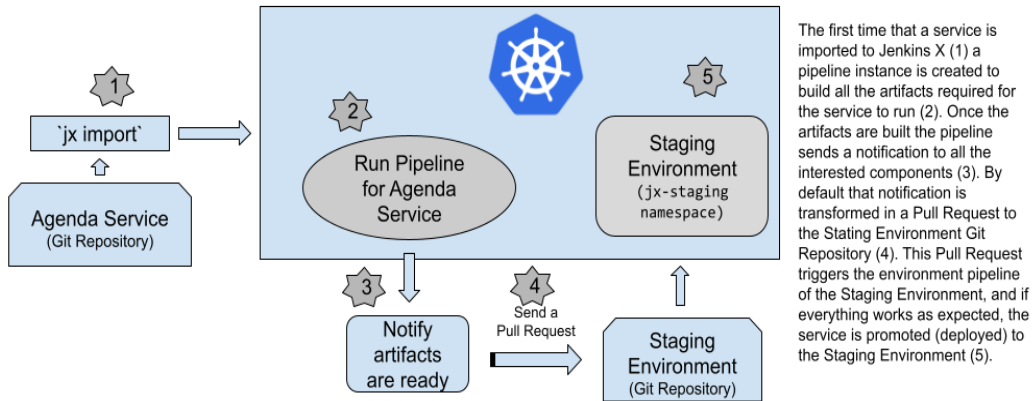


Figure 3.X Agenda Service imported to Jenkins X

After importing your project, you have a Service Pipeline configured and ready to go. Your project now contains all the necessary configurations and definitions to be built, packaged and run into a Kubernetes Cluster. Jenkins X took care of creating the Docker Image, the Helm Chart and the Service Pipeline for you. Because Jenkins X is creating a Tekton pipeline under the hood you can change and modify this pipeline if the one provided by default doesn't meet all your needs.

FROM SOURCE TO SERVICE RUNNING

Once the Service Pipeline is created you can start the pipeline by running ``jx pipeline start`` and then select your service from the list. You will notice that when the pipeline runs it will create a new tag and a new release for your artifacts, which will be automatically promoted to the Staging Environment.

When changes are merged in each service repository, new releases are created by their corresponding pipelines and new versions of the services are promoted to the Staging Environment. This constant stream of releases allows teams to independently release their services in a continuous fashion.

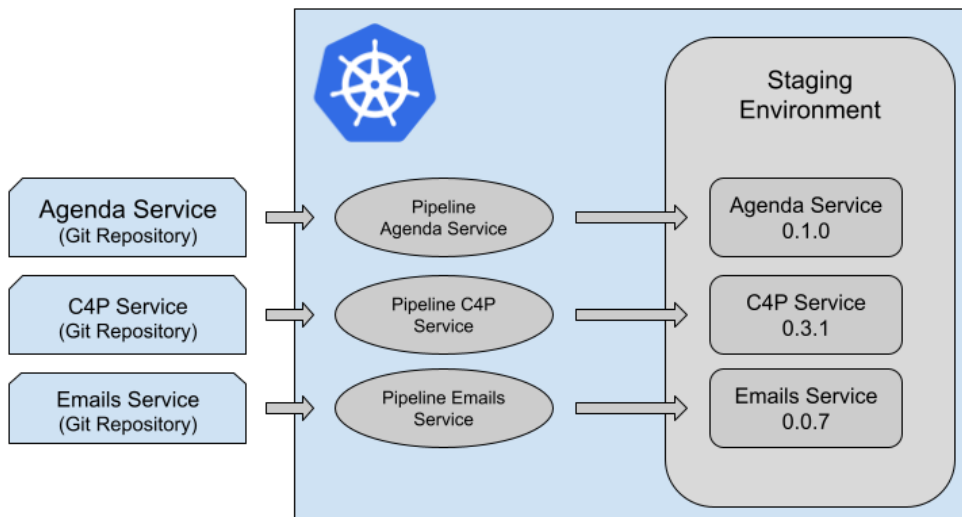


Figure 3.X Multiple services can be imported and deployed

For each service, a pipeline will be in charge of building and promoting the resulting Service artifacts to the Staging environment. But as we have seen before, the Service Pipeline itself doesn't deploy the new version of the services, the promotion of the service happens by sending a Pull Request with the new version that was just released to the Staging Environment. When this PR is received in the Staging Environment Git Repository, an Environment Pipeline configured for Pull Requests will run to validate this service upgrade and automatically merge the change into the main branch. Once the change is merged in the main branch the Environment Pipeline for the main branch will be in charge of syncing the state described in the repository, with the new version of the service, into the cluster that is running the Staging Environment.

Once the services are running in the Staging environment, we can run integration tests and end to end tests which will check that different versions of the services are working together as expected. When we consider that everything is working, the Operations team or the person in charge can send a Pull Request to the Production environment for the promotion to start.

The Staging Environment is configured to automatically run pipelines and promote services. The Production Environment needs manual intervention from the Operations Team to validate which services are stable enough to be promoted to Production. The promotion process only involves sending and merging a Pull Request with the updated versions of the services that we want to run in Production.

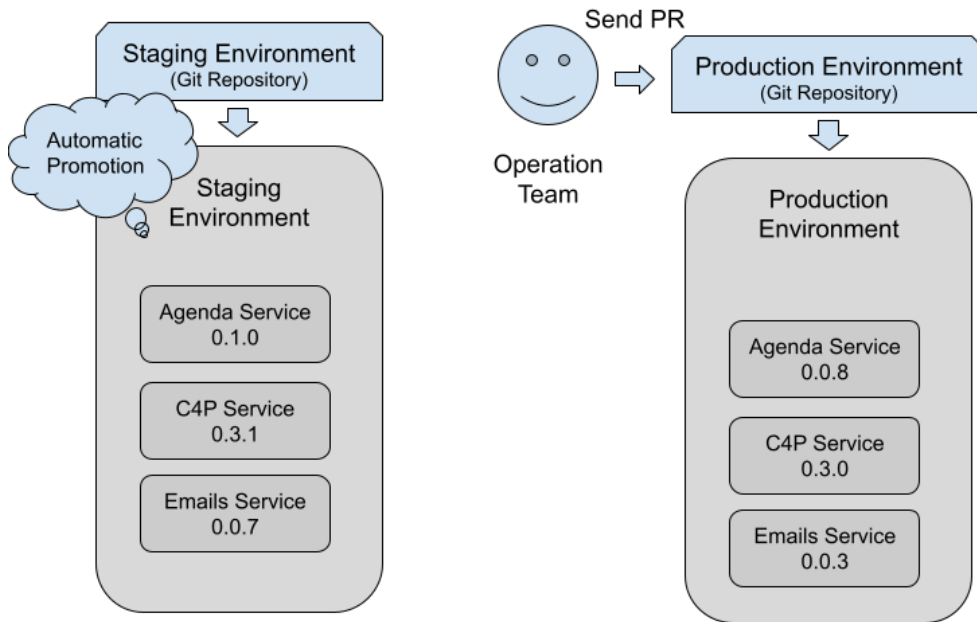


Figure 3.X Promoting services between different environments

It is quite common to require manual intervention to promote new services and new versions to more restricted environments such as the Production environment. A manual Pull Request merge might be required to promote the new services and versions forward to the production environment. By following the approach described in this section you limit the cluster permissions to the pipeline in charge of syncing the state of the Production environment git repository to the cluster state, limiting the accounts which have access to business-critical services and resources.

3.3.3 Other alternatives

Jenkins X provides a one-stop-shop for CI/CD for Kubernetes, but sometimes you don't need all the tools and all the conventions applied by default in Jenkins X. Sometimes you are looking for a more specific tool and here are a few examples of more focused tools that can help to implement Service and Environment pipelines:

- Flux CD
- Argo CD
- Spinnaker

While these tools are more generic and less opinionated, you will need to define with your team what conventions and best practices work for you. I strongly recommend you to check this and other tools that might be more focused to solve specific problems and then with the big picture of CI/CD for Kubernetes in mind, decide how to combine them to achieve Continuous Delivery.

3.4 Summary

- Service pipelines define how to go from source code to artifacts that can be deployed in multiple environments. In general, following Trunk-Based Development and One Service = One repository practices help your teams to standardize how to build and release software artifacts in a more efficient way.
- Environment Pipelines are in charge of deploying software artifact to live environments. Environment Pipelines avoid teams interacting directly with the cluster where the applications run, reducing errors and misconfigurations. Environment Pipelines
- Tekton is a Pipeline Engine designed for Kubernetes, you can use Tekton to design your custom Pipelines and leverage all the shared Tasks and Pipelines openly available in the Tekton Catalog. You can now install Tekton in your cluster and start creating Pipelines.
- If you want to learn about best practices, conventions and how a large Open Source project deals with the complexities of building and delivering cloud-native software you should look at Jenkins X, which provides a one-stop-shop for CI/CD on top of Kubernetes, by using Tekton as the Pipeline Engine.
- If you are interested in smaller projects that tackle more specific challenges around Environment pipelines you can look at Flux CD or Argo CD.

4

Multi-Cloud Infrastructure

This chapter covers

- Define and manage the infrastructure required by your applications
- The challenges of managing your own application infrastructure components
- The Kubernetes way to deal with infrastructure (Crossplane)

In the previous chapters, we installed a walking skeleton, we understood how to build each separate component using Service Pipelines and how to deploy them into different environments using Environment pipelines. We are now faced with a big challenge: dealing with infrastructure, meaning creating environments where our applications will run. It is normal and expected that these applications require other components to be there to work correctly, such as Databases, Message Brokers, Identity Management solutions, Email Servers, etc. While there are several tools out there focused on how to automate the installation or provisioning of these components for On-Premises setups and in different cloud providers, in this chapter, we will focus on just one that does it in a Kubernetes way. This chapter is divided into three main sections:

- The challenges of dealing with infrastructure
- How to deal with infrastructure leveraging Kubernetes constructs
- How to provision infrastructure for our walking skeleton using Crossplane.io

After covering these sections we will talk about building platforms on top of Kubernetes, a challenge that sooner or later will come knocking on your door.

4.1 The challenges of managing with Infrastructure in Kubernetes

When you design applications like the walking skeleton introduced in Chapter 1, you are faced with specific challenges that are not core to achieving your business goals. Installing, configuring and maintaining Application Infrastructure components that support our

application's services is a big task that needs to be planned carefully by the right teams with the right expertise.

These components can be classified as Application Infrastructure, which usually involve third-party components that are not going to be developed in house, such as Databases, Message Brokers, Identity Management solutions, etc. A big reason behind the success of modern Cloud Providers is that they are great at taking care of provisioning and maintaining these components and allowing your development teams to focus on building the core features of our applications, the ones that bring value to the business.

It is important to distinguish between Application Infrastructure and Hardware Infrastructure, which is also needed. In general, I am assuming that for Public Clouds offerings, all hardware related topics are solved by the provider itself. For On-Prem scenarios, it is most likely that you have a specialized team taking care of the Hardware (removing, adding and maintaining hardware as needed).

It is quite common to rely on Cloud Provider services to provision Application Infrastructure, there are a lot of advantages in doing so, such as pay as you use the services, easy provisioning at scale and automated maintenance. But at that point, you heavily rely on provider-specific ways of doing things and their tools. The moment you create a database or a message broker in a cloud provider you are jumping outside of the realms of Kubernetes. At this point you are depending on their tools, their automation mechanisms and you are creating a strong dependency of your business with the specific Cloud Provider.

Let's take a look at the challenges associated with provisioning and maintaining application infrastructure, so your teams can plan ahead and choose the right tool for the job:

- **Configuring components to scale:** each component will require different expertise to be configured (Database Administrators for databases and Message Broker experts) and a deep understanding of how our application's services will use it, as well as the hardware available. These configurations need to be versioned and monitored closely, so new environments can be created quickly to reproduce issues or just to test new versions of our application.
- **Maintaining components in the long run:** components such as databases and message brokers are constantly released and patched to improve performance and security, this pushes the operations teams to make sure that they can upgrade to newer versions, keep all the data safe without bringing down the entire application. This requires a lot of coordination and impact analysis between the teams involved with these components and services.
- **Cloud Provider services affect our multi-cloud strategy:** if we rely on cloud-specific application infrastructure and tools, we need to find a way to enable developers to create and provision their components for developing and testing their services. We need a way to abstract how infrastructure is provisioned to enable applications to define what pieces of infrastructure they need without relying directly on cloud-specific tools.

It is interesting to see that we had these challenges even before having distributed applications, and configuration and provisioning architectural components have always been

hard and usually far away from developers. Cloud Providers are doing a fantastic job by bringing these topics closer to developers so they can be more autonomous and iterate faster. Unfortunately, when working with Kubernetes we have more options that we need to consider carefully to make sure that we understand the trade-offs. The next section covers how we can manage our Application Infrastructure inside Kubernetes; while this is usually not recommended, it can be practical for some scenarios.

4.1.1 Managing your own Application Infrastructure

Application infrastructure has become an exciting arena. With the rise of containers, every developer can bootstrap a database or a message broker with a couple of commands, and for development purposes, this is usually enough. In the Kubernetes world, this translates to Helm Charts, which uses containers to configure and provision Databases (relational and NoSQL), message brokers, identity management solutions, etc. As we saw in chapter 2 you installed the walking skeleton application containing 4 services with a single command. The same can be done for application infrastructure components.

For example, you can run `helm install postgresql bitnami/postgresql` to install an instance of PostgreSQL in your cluster. Same with Redis, Kafka and RabbitMQ. The amount of Helm charts available today is amazing, and it is quite easy to think that installing a Helm Chart will be the way to go.

As discussed in Chapter 2 (Dealing with application state), if we want to scale our services that keep state we will need to provision specialized components such as databases. Application developers will define which kind of database will suit best depending on the data that they need to store and how that data will be structured. For our walking skeleton, we will be provisioning a Redis NoSQL database for the Agenda Service and a PostgreSQL database for the Call for Proposals (C4P) Service.

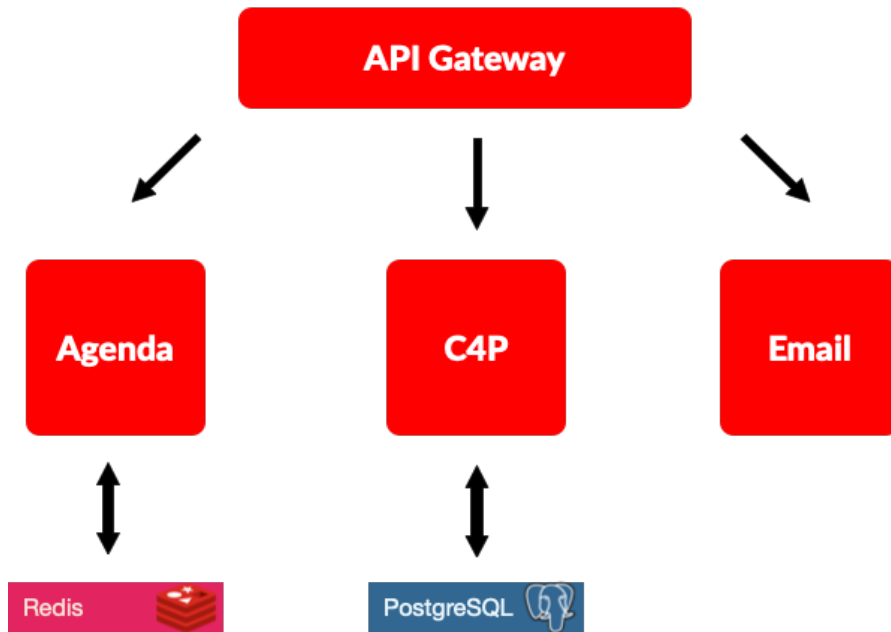


Figure 4.1 Keeping service state using application infrastructure

The process of setting up these components inside your Kubernetes Cluster involves the following steps:

- Finding or creating the right Helm Chart for the component you want to bootstrap: for this case, PostgreSQL (<https://github.com/bitnami/charts/tree/master/bitnami/postgresql>) and Redis (<https://github.com/bitnami/charts/tree/master/bitnami/redis>) can be found in the Bitnami Helm Chart repository. If you cannot find a Helm Chart but you have a Docker Container for the component that you want to provision you can create your own chart after you define the basic Kubernetes constructs needed for the deployment.
- Research the Chart configurations and parameters that you will need to set up to accommodate your requirements. Each chart exposes a set of parameters that can be tuned for different use cases, check the chart website to understand what is available. Here you might want to include your Operations teams and for example, DBAs to check how the Databases needs to be configured. This will also require Kubernetes expertise to make sure that the components can work in HA (High Availability) mode inside Kubernetes.
- Install the chart into your Kubernetes Cluster, using ``helm install``. By running ``helm install`` you are downloading a set of Kubernetes manifest (YAML files) that describe how your applications/services need to be deployed. Helm then will proceed to apply these YAML files into your cluster.

- Maintain these components in the long run, doing backups and making sure that the fail-over mechanisms are working as expected.

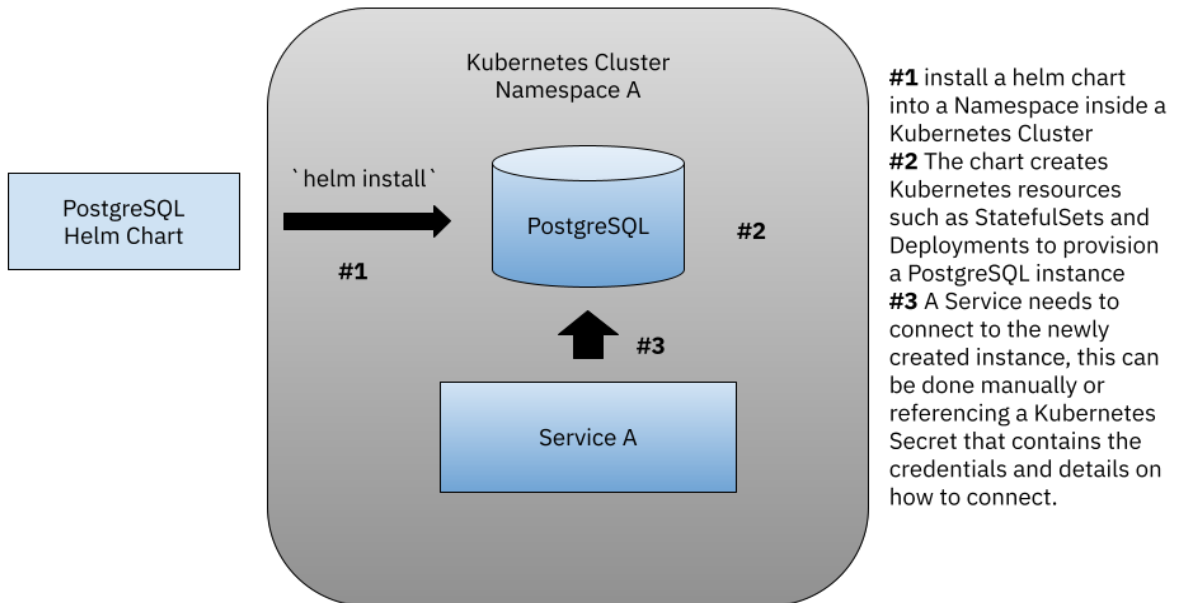


Figure 4.2 Provisioning a new PostgreSQL instance using the PostgreSQL Helm Chart

If you are working with Helm Charts, there are a couple of caveats and tricks that you need to be aware of:

- If the chart doesn't allow you to configure a parameter that you are interested in changing you can always use ``helm template``, then modify the output to add or change the parameters that you need to finally install the components using ``kubectl apply -f``. Alternatively, you can submit a Pull Request to the chart repository. It is a common practice to not expose all possible parameters and wait for community members to suggest more parameters to be exposed by the chart. Don't be shy and get in touch with the maintainers if that is the case. Whatever modification that you do the chart content will need to be maintained and documented. By using ``helm template`` you lose the Helm release management features, which allows you to upgrade a chart when a new chart version is available.

- Most charts come with a default configuration that was designed to scale, meaning that the default deployment will be targeting high-availability scenarios. This results in Charts that when installed consume quite a lot of resources that might not be available if you are using Kubernetes KIND or Minikube on your laptop. Once again, charts documentation usually include special configurations for development and resource-constrained environments.
- If you are installing a Database inside your Kubernetes Cluster, each Database container (pod) will need to have access to storage from the underlying Kubernetes Node. For databases, you might need to have a special kind of storage to enable the database to elastically scale, which might require advanced configurations outside of Kubernetes.

If you have the Conference Platform up and running in your Kubernetes Cluster you can proceed to install the previously mentioned components by using their respective charts. First, you will need to add the Bitnami Helm Chart repository:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo update
```

And then install the charts with the following two lines:

```
helm install postgresql bitnami/postgresql

helm install redis bitnami/redis
```

The output of these two commands will give you instructions on how to access the password to connect to the instance, which is stored inside a Kubernetes Secret. You will also see the name of the Kubernetes Service that you need to use to connect your applications. This is a common pattern in Kubernetes when we are provisioning new components that other applications will be connecting to. Finding the right URL for the service, user and password will be fundamental for your applications to connect.

If you list the pods running with `kubectl` you should get something along the lines:

```
> kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
app-fmtok8s-agenda-rest-68bd9c8bcb-dxbdt	1/1	Running	0	12m
app-fmtok8s-api-gateway-58d49588b4-vw7pc	1/1	Running	0	12m
app-fmtok8s-c4p-rest-7cb8bc4485-scj7r	1/1	Running	0	12m
app-fmtok8s-email-rest-8f954fbbd-p8hld	1/1	Running	0	12m
postgresql-postgresql-0	1/1	Running	0	6m30s
redis-master-0	1/1	Running	0	3m47s
redis-replicas-0	1/1	Running	0	3m47s
redis-replicas-1	1/1	Running	0	3m13s
redis-replicas-2	1/1	Running	0	2m36s

We can see that the PostgreSQL chart by default creates a single replica (pod) and Redis on the other hand creates a master and 3 replicas. It is common for different databases to have different scalability mechanisms and default configurations.

CONNECTING OUR SERVICES TO THE NEWLY PROVISIONED INFRASTRUCTURE

Note: Both services, the Agenda Service and Call for Proposals (C4P) Service were specially prepared to support both in-memory storage and external persistence. In your projects, you shouldn't rely on an in-memory approach unless you are building a Proof of Concept.

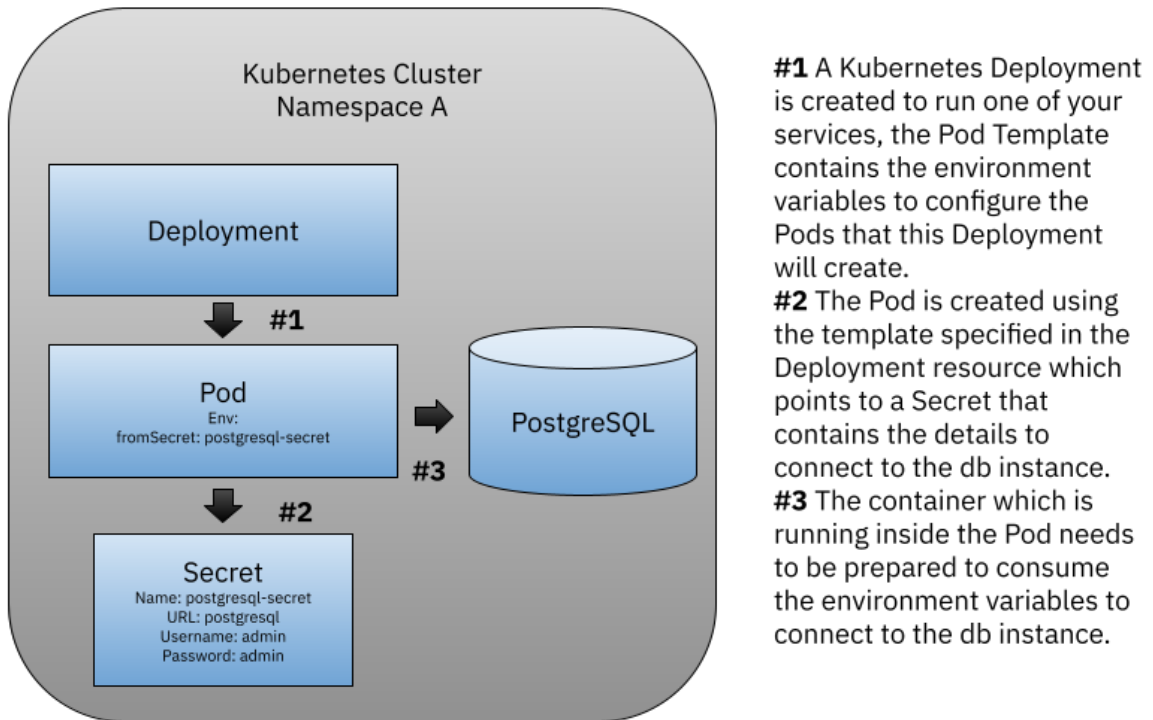


Figure 4.3 Connecting a Service to a provisioned resource using Secrets

Let's connect the Call for Proposals (C4P) Service first, you can do this by editing the Kubernetes Deployment called `app-fmtok8s-c4p-rest` and adding some environment variables to specify the database connection properties. Look for the `spec.template.spec.containers[0].env` section and add the following new variables:

```

- name: SPRING_DATASOURCE_DRIVERCLASSNAME
  value: org.postgresql.Driver
- name: SPRING_DATASOURCE_PLATFORM
  value: org.hibernate.dialect.PostgreSQLDialect
- name: SPRING_DATASOURCE_URL
  value: jdbc:postgresql://${DB_ENDPOINT}:${DB_PORT}/postgres
- name: SPRING_DATASOURCE_USERNAME
  value: postgres
- name: SPRING_DATASOURCE_PASSWORD
  valueFrom:
    secretKeyRef:
      key: postgresql-password
      name: postgresql
- name: DB_ENDPOINT
  value: postgresql
- name: DB_PORT
  value: "5432"

```

In bold are highlighted how we can consume the password that was dynamically generated when we installed the chart and the DB endpoint URL which in this case is the PostgreSQL Kubernetes Service, also installed by the chart. If you used a different chart release name, the DB Endpoint will be different. As soon as you save this deployment configuration, a new Pod will be started and if the Pod manages to connect to the PostgreSQL instance the old version will be removed automatically by Kubernetes.

You are now able to increase the number of replicas of the Call for Proposal service, as the service is now Stateless. All replicas of the Service will connect to the same database instance to fetch and store the data.

The same can be done with the Agenda Service and Redis.

```

- name: SPRING_REDIS_IN_MEMORY
  value: "false"
- name: SPRING_REDIS_HOST
  value: redis-master
- name: SPRING_REDIS_PASSWORD
  valueFrom:
    secretKeyRef:
      key: redis-password
      name: redis

```

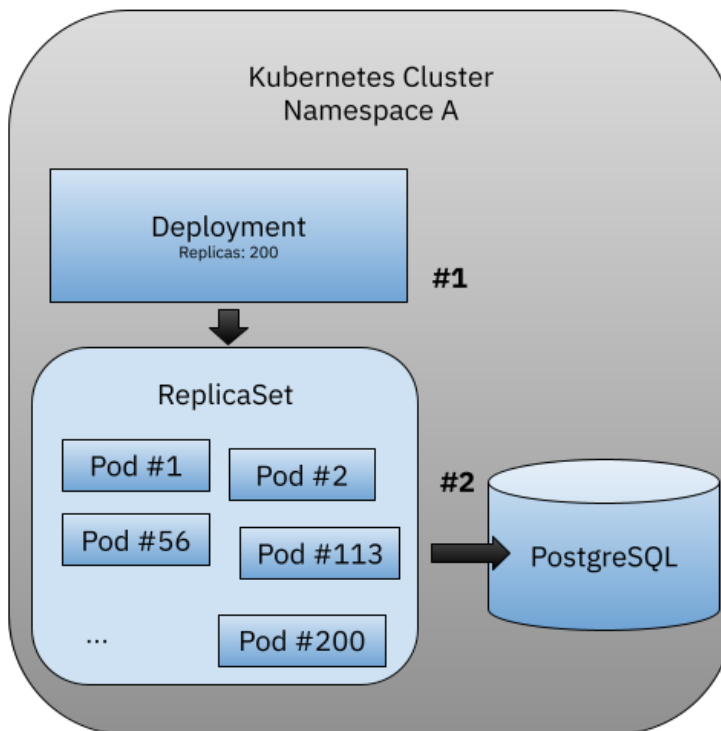
Same as before, we extract the password from a secret called `redis`. The REDIS_HOST is obtained from the name of the Kubernetes Service that is created by the chart, this depends on your `helm release` name that you have used. In the same way, as with the C4P Service, you are now able to add more replicas to your Agenda Service to support more traffic. Remember that also your services can crash now, and as soon as they are back they are going to reconnect with the Database and fetch the available data.

If you want to install PostgreSQL and Redis in your Kubernetes Cluster using Helm you can follow a step-by-step tutorial located here: <https://github.com/salaboy/from-monolith-to-k8s/tree/master/helm-app-infra>

I strongly recommend going over the tutorial to have a first-hand experience with Helm Charts and Kubernetes Deployments.

4.1.2 Different approaches to install and monitor Application Infrastructure components

Depending on how these databases are going to be used by the services, you might encounter that the databases are the bottleneck. Imagine if you have too many requests on the Agenda Service, so you decide to scale up the number of replicas of the Agenda Deployment to 200. At that point, Redis needs to have enough resources to deal with 200 pods connecting to the Redis Cluster. The advantage of using Redis for this scenario, where we might be getting a lot of reads while the conference is ongoing, is that the Redis Cluster allows us to read data from the replicas so the load can be distributed.



#1 If you noticed a surge in demand for one of your services, you might be tempted to increase the number of Replicas, and the Deployment using the ReplicaSet will not complain about it. If the cluster have enough resources, the replicas will be created.

#2 If the application infrastructure is not correctly configured you might encounter a lot of issues, such as exhausting the database connection pool, or overloading the Database Pods, as this are not scaled when you scale up your Deployments.

Figure 4.4 Application infrastructure needs to be configured according how our services will be scaled

If you are installing Application Infrastructure with Helm, notice that Helm is not going to check for the health of these components, it is just doing the installation. It is quite common nowadays to find another alternative to install components in a Kubernetes cluster, called Operators. Usually associated with Application Infrastructure, you can find more active components that will install and monitor the components that were installed. One example of these Operators is the Zalando PostgreSQL Operator that you can find here: <https://github.com/zalando/postgres-operator>.

In general, Kubernetes Operators try to encapsulate the operational tasks associated with a specific component, in this case, PostgreSQL. We will cover the Operator Pattern in more detail in Chapter X. While using Operators might add more features on top of installing a given component, you still need to maintain the component and the operator itself now.

Regarding the Application Infrastructure that you and your teams decide to use if you are planning to run these components inside your cluster, plan accordingly to have the right expertise in house to manage, maintain and scale these extra components.

In the following section, we will look at how we can tackle these challenges by looking at an Open Source project that aims to simplify the provisioning of Cloud resources and application infrastructure by using a declarative approach.

4.2 Defining Infrastructure in a declarative way using Crossplane

Using Helm to install application infrastructure components inside Kubernetes is far from ideal for large applications and user-facing environments, as the complexity of maintaining these components and their requirements such as advanced storage configurations might become too complex to handle for your teams.

Cloud Providers do a fantastic job at allowing us to provision infrastructure, but they all rely on cloud provider-specific tools which are outside of the realms of Kubernetes.

In this section, we are going to look at an alternative tool; a CNCF project called Crossplane (<https://crossplane.io>), which uses the Kubernetes APIs and extension points to enable users to provision real infrastructure in a declarative way, using the Kubernetes APIs. Crossplane relies on the Kubernetes APIs to support multiple Cloud Providers, this also means that it integrates nicely with all the existing Kubernetes tooling.

By understanding how Crossplane works and how it can be extended you can build a multi-cloud approach to build and deploy your Cloud Native applications into different providers without worrying about getting locked-in. Because Crossplane uses the same declarative approach as Kubernetes, you will be able to create your own high level abstractions about the applications that you are trying to deploy and maintain.

4.2.1 Crossplane Providers

Crossplane extends Kubernetes by installing a set of components called “*Providers*” which are in charge of understanding and interacting with cloud provider-specific services to provision these components for us.

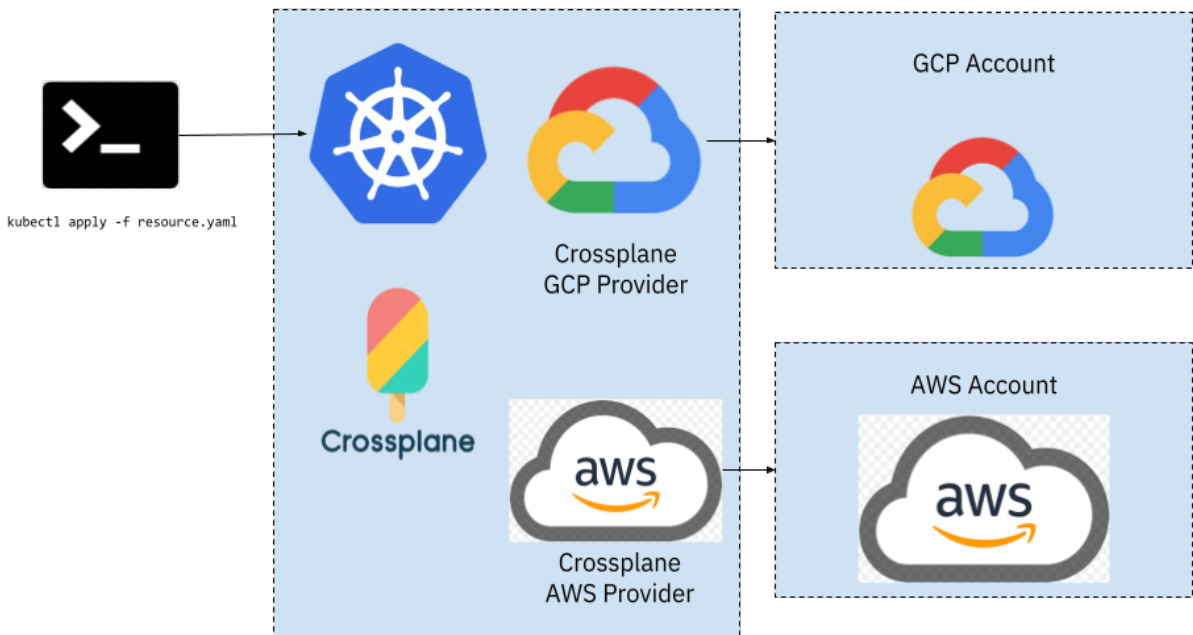


Figure 4.5 Crossplane installed with GCP and AWS Providers

By installing Crossplane providers you are extending the Kubernetes APIs functionality to provision external resources such as Databases, Message Brokers, Buckets and other Cloud Resources that will live outside of your Kubernetes cluster, but inside the Cloud Provider realm. There are several Crossplane providers covering the major cloud providers such as GCP, AWS and Azure. You can find these Crossplane providers in the Crossplane Github's organization: <https://github.com/crossplane/>

Once a Crossplane Provider is installed you can create provider-specific resources in a declarative way, which means that you can create a Kubernetes Resource, apply it with ``kubectl apply -f``, package these definitions in Helm Charts or use Environment Pipelines storing these resources in a Git repository.

Provisioning cloud-specific resources relying on the Kubernetes APIs is a big step forward but Crossplane doesn't stop there. If you go to look at the details of what it takes to provision a database in any major cloud provider you will realize that provisioning the component is just one of the tasks involved in getting the component ready to be used. For connecting to these provisioned components you will need network and security configurations, as well as user credentials.

4.2.2 Crossplane Compositions

Crossplane aims to serve two different Personas: "*Platform teams*" and "*Application teams*". While *Platform* teams are Cloud Providers experts that understand how to provision cloud

provider-specific components, *Application* teams know the application requirements and understand what is required from the Application Infrastructure perspective. The interesting thing about this approach is that when using Crossplane, Platform teams can define these complex configurations for a specific Cloud Provider and expose simplified interfaces for Application teams.

In real-life scenarios, it is rare to just create a single component, for example, if we want to provision a Database instance application teams will also require the correct network and security configurations to be able to access the newly created instance. Being able to compose and wire up together several components is a very convenient feature and for achieving these abstractions and simplified interfaces Crossplane introduced two concepts “*Composite Resource Definitions*” and “*Composite Resources*”.

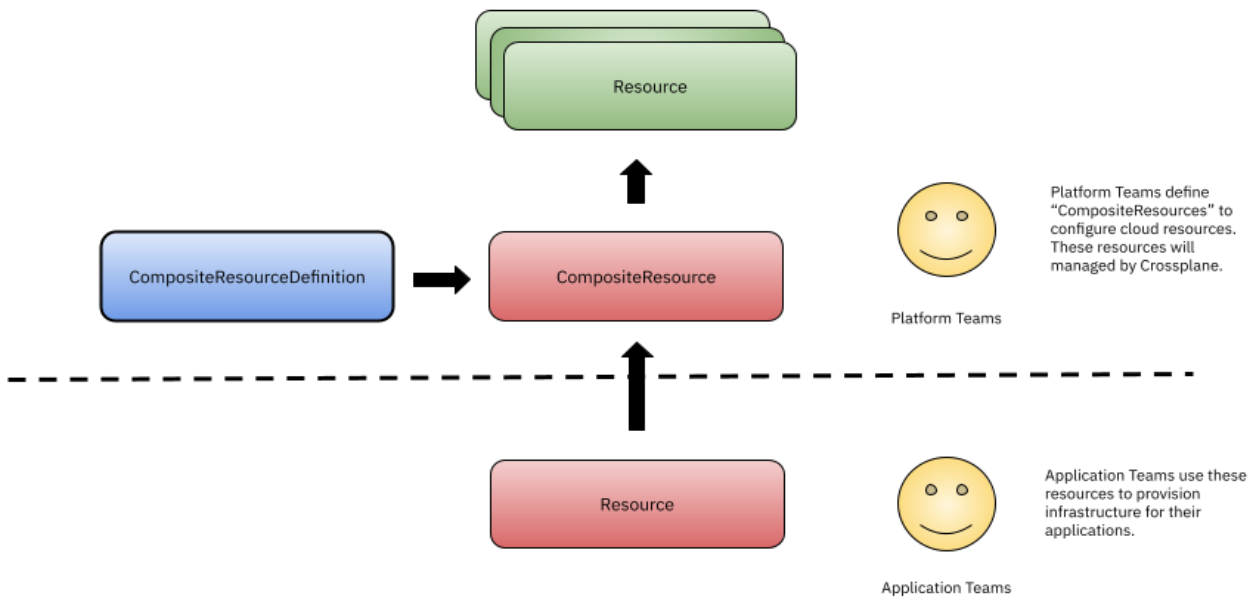


Figure 4.6 Resource Composition abstraction by Crossplane Composite Resources

The previous diagram shows how you can use Crossplane *Composite Resources* to define abstractions for different Cloud Providers. The *Platform team* might be very knowledgeable in Google Cloud or Azure so they will be in charge of defining which Resources they want to wire up together for a specific application. The *Application team* have a simple Resource interface to request the Resource that they are interested in. But as usual, abstractions are complicated and good to show who is responsible for what, but let’s look at a concrete example to understand the power of Crossplane Compositions.

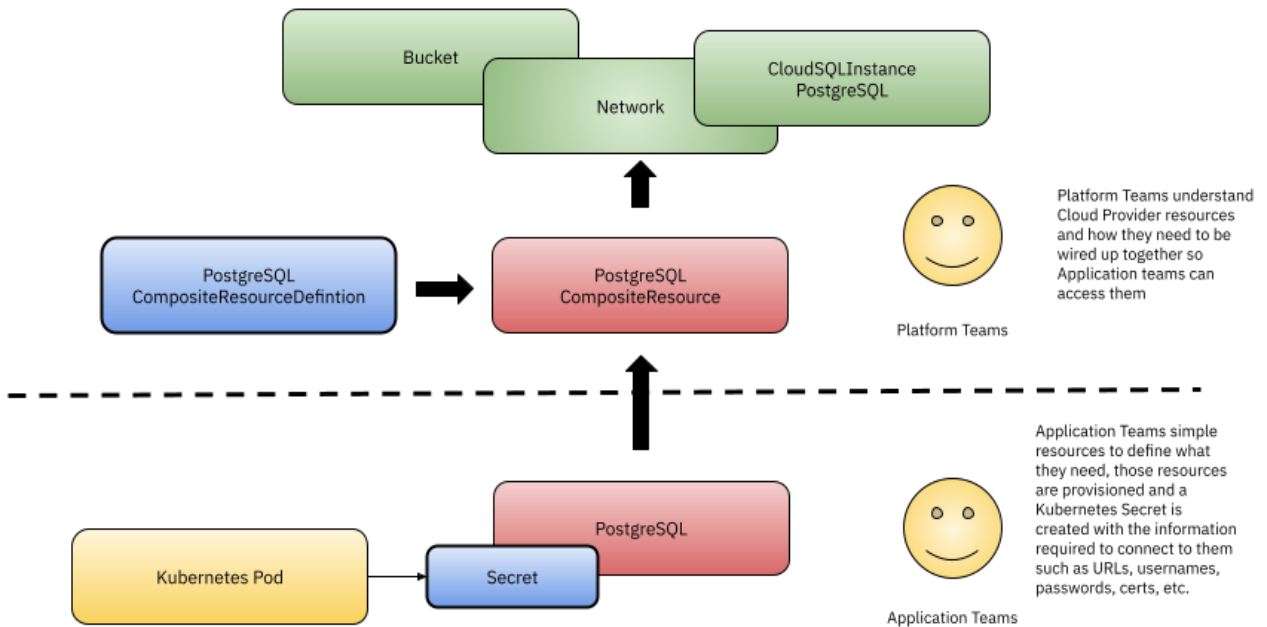


Figure 4.7 Provisioning a PostgreSQL instance in Google Cloud with Crossplane Compositions

Figure 4.x shows how the application team can create a simple PostgreSQL resource to provision in Google Cloud a `CloudSQLInstance` plus a `Network` configuration and a `Bucket`. The application team is not really interested in what resources are created or even in which Cloud Provider they were created, they are only interested in having a PostgreSQL instance to connect their applications to.

This takes us to the “Secret” box in the figure, which represents a Kubernetes Secret that Crossplane will create for our application/services Pods to connect to the provisioned resources. Crossplane creates this Kubernetes Secret with all the details required by our applications to connect to the newly created resources (or just with the one relevant for the application). This Secret typically contains URLs, usernames, passwords, certificates or anything required for your applications to connect. Platform teams define what is going to be included in the secret when defining the `CompositeResources`. In the following sections, when we add real infrastructure to our Conference Platform we will explore how these `CompositeResources` look and how they can be applied to create all the components that our applications need.

Another really important feature of Crossplane is that you can package these compositions into an OCI image (a container) and share that as any other docker image. Once you install these Crossplane packages with configurations, the *Platform* team can create these simplified resources without the need of understanding or knowing which provider is available to them. This clear separation of concerns (requesting application

infrastructure and defining all the configuration required to provision these components) makes Crossplane really strong.

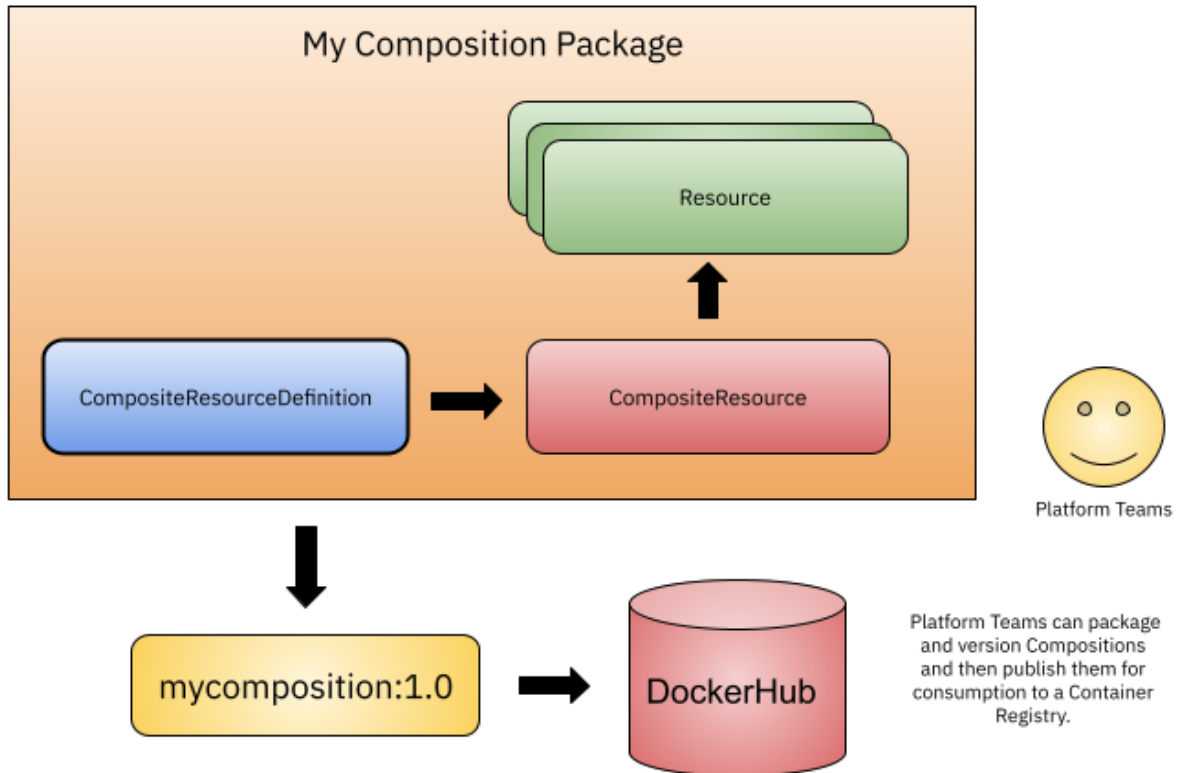


Figure 4.8 Packaging a Composition as an OCI Image for distribution

Figure 4.x shows how we can package and version these compositions as a Container Image (Open Container Image) that then we push to a container registry so other teams can consume it from different Kubernetes Clusters. In the following sections we will be looking and how does this look in practice. But before jumping into the specifics let's look in more detail how Crossplane works and what it requires for us to configure before using all these mechanisms and resources.

4.2.3 Crossplane Components and Requirements

To work with Crossplane Providers and CompositeResources we need to understand how Crossplane components will work together to provision and manage these components inside different Cloud Providers.

This section covers what Crossplane needs in order to work and how Crossplane components will manage our CompositeResources.

First of all, it is important to understand that you will need to install Crossplane in a Kubernetes cluster. This can be the Cluster where your applications are running or a separate cluster where Crossplane will run. This cluster will have some Crossplane components that will understand our CompositeManagedResources and have enough permissions on the cloud platform to provision resources on our behalf. You might be wondering if you can install Crossplane into a KIND Cluster, and the answer is yes, but Crossplane really shines if you have access to a Cloud Provider. For that reason, we will use Google Cloud Platform in the next sections.

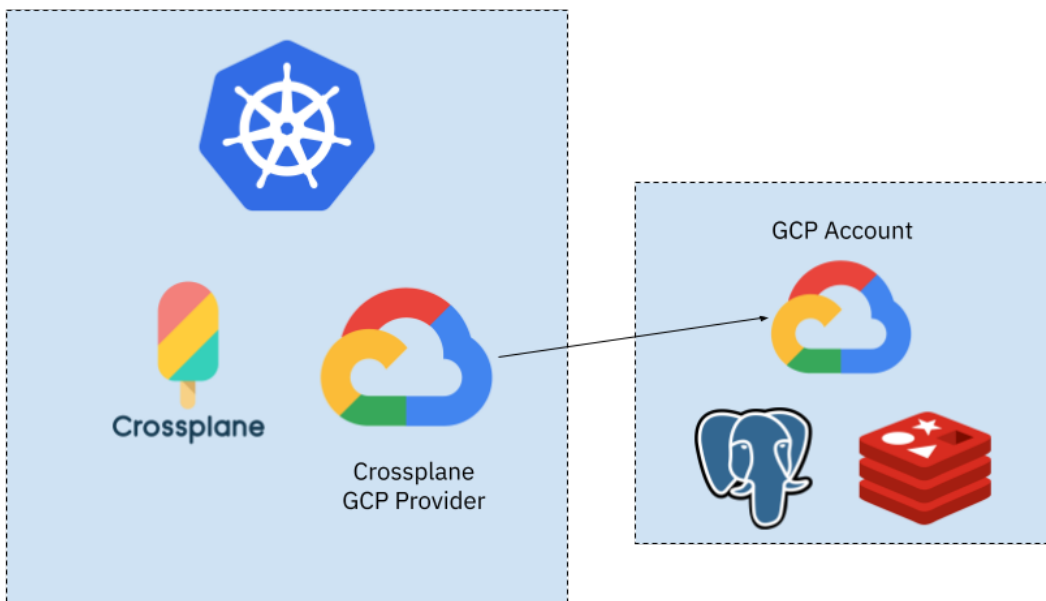


Figure 4.9 Crossplane in Google Cloud Platform

The previous figure shows Crossplane installed inside a Kubernetes Cluster, with the Crossplane GCP provider installed and configured to use a Google Cloud Platform account with enough rights to provision PostgreSQL and Redis instances.

Each of the Crossplane Providers available requires a specific security configuration to work and an account inside the Cloud Provider where we want to create resources.

Once a Crossplane Provider is installed and configured, in this case, the GCP provider we can start creating resources that are managed by this provider. You can find the resources offered by each provider in the following documentation site: <https://doc.crd.dev/github.com/crossplane/provider-gcp>



[crossplane/provider-gcp@v0.17.1](#)

github.com/crossplane/provider-gcp/tree/v0.17.1

v0.17.1 ▾

CRDs discovered: 20

e.g. CacheCluster, acm.aws

Kind ↕	Group ↕	Version ↕
CloudMemorystoreInstance	cache.gcp.crossplane.io	v1beta1
GlobalAddress	compute.gcp.crossplane.io	v1beta1
Network	compute.gcp.crossplane.io	v1beta1
Subnetwork	compute.gcp.crossplane.io	v1beta1
NodePool	container.gcp.crossplane.io	v1alpha1
GKECluster	container.gcp.crossplane.io	v1beta1
CloudSQLInstance	database.gcp.crossplane.io	v1beta1

Figure 4.10 Crossplane GCP supported resources

As you can see in the previous figure, the GCP Provider version 0.17.1 supports 20 different CRDs (Custom Resource Definitions) for creating resources in Google Cloud Platform. Crossplane defines each of these resources as Managed Resources. Each of these individual managed resources will need to be enabled for the Crossplane Provider to have the right access to list, create and modify these resources.

In section 4.3, we will be provisioning new instances of PostgreSQL and Redis for our walking skeleton. We will be creating *CloudMemorystoreInstances* and *CloudSQLInstances* in GCP, but these details are going to be abstracted away from our App Ops teams which will be using cloud-agnostic resources with a simplified interface.

4.2.4 Crossplane Behaviours

In contrast with installing components with Helm in our Kubernetes Clusters, we are using Crossplane to interact with the cloud provider-specific APIs to provision resources inside the Cloud infrastructure. This should simplify the maintenance tasks and costs related to these resources. Another important difference is that the Crossplane provider (GCP provider in this case) will monitor the created Managed Resources for us. These Managed Resources offer some advantages compared with just installed resources using Helm. Managed Resources

have very well defined behaviours, here is a summary of what to expect from a Crossplane Managed Resource:

- **Visible as any other Kubernetes Resource:** Crossplane Managed Resources are just Kubernetes resources, this means that we can use any Kubernetes tool to monitor and query the state of these resources.
- **Continuous Reconciliation:** when a managed resource is created the Provider will continuously monitor the resource to make sure that it exists and is working and report back the status to the Kubernetes resource. The parameters defined inside the managed resource are considered the desired state (source of truth) and providers will work to apply these configurations into the Cloud Provider resources. Once again, we can use standard Kubernetes tools to monitor change in state and trigger remediation flows.
- **Immutable Properties:** providers are in charge of reporting back if a user manually changed properties in the cloud provider. The idea here is to avoid configuration drifts from what was defined to what is actually running in the cloud provider. If so, the state is reported back to the managed resource. Crossplane will not delete the cloud provider resource, but it will notify back so actions can be taken. Other tools like terraform (<https://www.terraform.io>) will automatically delete the remote resources in order to recreate them.
- **Late initialization:** some properties in the Managed Resources can be optional, meaning that each provider will select the default values for these properties. When this happens Crossplane creates the resource with the default values and then sets the selected values into the Managed Resource. This simplifies the amount of configuration needed to create resources and reuse the sensible defaults defined by cloud providers usually in their user interfaces.
- **Deletion:** when deleting a Managed Resource, the action is immediately triggered in the cloud provider, but the Managed Resource is kept until the resource is fully removed from the Cloud Provider. Errors that might happen during deletion on the cloud provider will be added to the Managed resource status field.
- **Importing existing resources:** Crossplane doesn't necessarily need to create the resources to be able to manage them. You can create Managed Resources that start monitoring components that were created previously before Crossplane was installed. You can achieve this by using a specific Crossplane annotation on the Managed Resource: `crossplane.io/external-name`

To summarize the interactions between Crossplane, the Crossplane GCP Provider, and our Managed Resources, let's look at the following diagram:

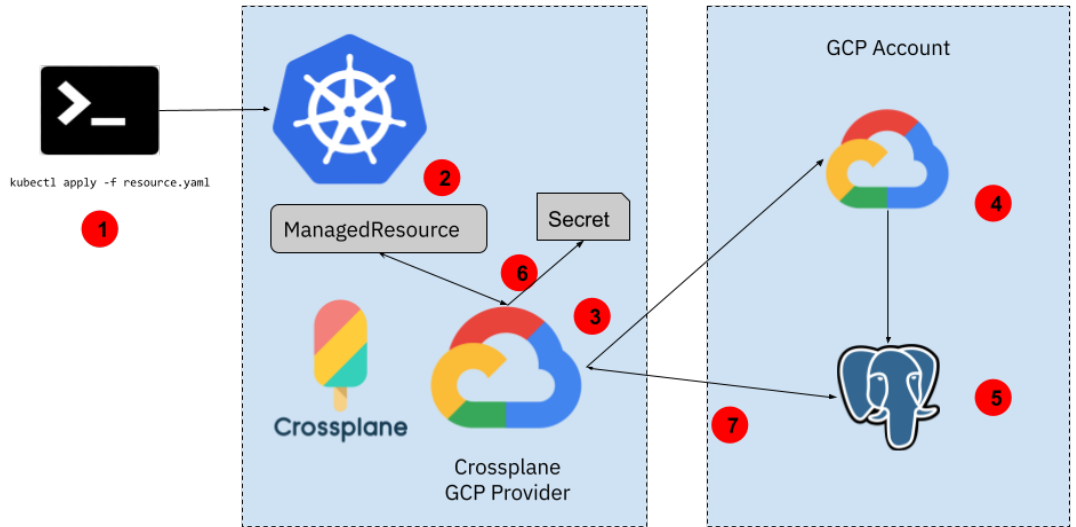


Figure 4.11 Lifecycle of Managed Resources with Crossplane

The following points indicate the sequence observed in Figure 4.X:

1. First, we need to create a resource, we can use any tool to create Kubernetes resources, `kubectl` here is just an example.
2. If the resource that we created is a Crossplane Managed Resource, let's imagine a CloudSQLInstance resource, the specific Crossplane Provider will pick it up and manage it.
3. The first step to execute when managing a resource will be checking if it exists in the infrastructure (that is in the configured GCP account). If it doesn't exist the Provider will send a request for the resource to be created in the infrastructure. Depending on the properties set on the resource, for example, which kind of SQL database is required, the appropriate SQL database will be provisioned, imagine for the sake of the example that we have chosen a PostgreSQL database.
4. The Cloud Provider after receiving the request, if the resources are enabled, will proceed to create a new PostgreSQL instance with the configured parameters in the Managed Resource.
5. The status of the PostgreSQL will be reported back to the Managed Resource, which means that we can use `kubectl` or any other tool to monitor the status of the provisioned resources. Crossplane provider's will keep these in sync.
6. When the database is up and running the Crossplane Provider will create a secret to store the credentials and properties that our applications will need to connect to the newly created instance
7. Crossplane will regularly check the status of the PostgreSQL instance and update the managed resource.

4.2.5 Crossplane Configuration Packages

If we create a CloudSQLInstances resource from the GCP provider we are still making a strong reference to the provider. These GCP resources are still pretty detailed as for example, allow us to set up properties such as DiskEncryptionConfiguration, GceZone, OnPremisesConfiguration among others (<https://doc.crds.dev/github.com/crossplane/provider-gcp/database.gcp.crossplane.io/CloudSQLInstance/v1beta1@v0.17.1>) that we might want to standardize across teams or have a policy defined on how to set them.

In order to abstract away these provider-specific and low-level details on the resources, Crossplane allows us to define Composite Managed Resources. These compositions allow us to compose several provider-specific resources, configure them together and expose a simple interface for the App Ops teams.

You can create your packages with your own Managed Resources based on these compositions. These packages are easy to create as soon as you know who will be consuming them and what kind of resources they want to create. These packages also allow you to create the Managed Resources that can have different cloud providers implementations but expose the same simplified type for the users.

At the end of the day, a package is just a set of configuration files that can be packaged as an OCI container and published to Docker Hub or any other container registry. In the last section of this book, we will cover packages in more detail, let's first see them in action.

The following section will look at how we can use Crossplane with our walking skeleton application.

4.3 Real Infrastructure for our walking skeleton

In this section, we will look at how to use Crossplane in Google Cloud Platform to provision resources for our walking skeleton application.

You can go ahead and create a new Kubernetes Cluster in GCP and install the application using Helm, as you did for KIND. As recommended in Chapter 2, you can get free credits in most major cloud providers, so we created a list for you: <https://github.com/learnk8s/free-kubernetes>. While the examples in this section were designed for Google Cloud, they are pretty easy to port to other Cloud Providers.

Once you have the application up and running you can install Crossplane by following the "Self-Hosted Crossplane" and GCP instructions listed in their documentation: <https://crossplane.io/docs/v1.5/getting-started/install-configure.html>

Notice that Crossplane also provides a `kubectrl` plugin to install custom providers and configuration packages. Make sure you install this plugin as well: <https://crossplane.io/docs/v1.5/getting-started/install-configure.html#install-crossplane-cli>

The installation procedure installs the GCP provider with the proper access to provision components on our behalf. It is common to install Crossplane into a separate cluster that is not where our applications are running; for simplicity here, we will use the same Kubernetes Cluster for our application and Crossplane. Notice that Crossplane examples also use PostgreSQL, hence they enable the CloudSQL API. You will need also to enable the Redis

APIs and you can do this by adding to the installation steps after enabling the Cloud SQL APIs and before creating the Service Account Key file:

```
export SERVICE="redis.googleapis.com"
gcloud services enable $SERVICE --project $PROJECT_ID

export ROLE="roles/redis.admin"
gcloud projects add-iam-policy-binding --role="$ROLE" $PROJECT_ID --member
"serviceAccount:$SA"
```

For a detailed tutorial on how to install Crossplane for the Conference Platform application you can look into the following repository: <https://github.com/salaboy/from-monolith-to-k8s/tree/master/crossplane>

4.3.1 Provisioning our Application Infrastructure

Once Crossplane and the GCP providers are installed you will be able to provision GCP resources, but as mentioned in the previous section, if you still have to define cloud provider-specific resources moving to a different cloud provider, later on, might be tricky, hence creating your abstractions for these resources makes a lot of sense. These abstractions should be easier for developers to use and parameterise. These abstractions shouldn't leak provider-specific configurations or parameters, at the end of the day, what we need for our walking skeleton is a PostgreSQL instance and a Redis Instance.

You can go ahead and install a custom configuration package that was created for the walking skeleton by running:

```
kubectl crossplane install configuration salaboy/crossplane-fmtok8s-gcp:0.0.8
```

This Crossplane configuration package includes the PostgreSQL and Redis abstraction for our walking skeleton, enabling us to provision PostgreSQL and Redis instances. Once the configuration package is installed, you can use `kubectl`, Helm or any other tool to create the newly registered resources.

For example for PostgreSQL you can use the following resource:

```
apiVersion: db.fmtok8s.salaboy.com/v1alpha1
kind: PostgreSQLInstance
metadata:
  name: my-postgresql-db
  namespace: default
spec:
  parameters:
    storageGB: 20
  compositionSelector:
    matchLabels:
      provider: gcp
  writeConnectionSecretToRef:
    name: db-conn
```

Or for Redis:

```
apiVersion: cache.fmtok8s.salaboy.com/v1alpha1
kind: RedisInstance
metadata:
  name: my-redis-db
  namespace: default
spec:
  parameters:
    memorySizeGb: 1
  compositionSelector:
    matchLabels:
      provider: gcp
  writeConnectionSecretToRef:
    name: redis-conn
```

You can apply these resources running for PostgreSQL:

```
kubectl apply -f postgresql.yaml
```

Or for Redis:

```
kubectl apply -f redis.yaml
```

This will instruct Crossplane, who is managing these resource types to create a new PostgreSQL and Redis instance inside GCP. From the previous code snippets it is important to highlight the following values:

- **ApiVersion (*.fmtok8s.salaboy.com) + Kind:** represent our custom resources, these resources are not GCP specific, meaning that we can provide configuration packages to Crossplane to be able to provision the same resources into different Cloud Providers.
- **Name:** the name of the resource that will be created, this will be applied to the resource created in the configured Cloud Provider. By creating different resources with different names you can provision multiple instances.
- **Spec.Parameters.*:** these are the parameters exposed by the abstraction of these resources. You can see that it is different depending on the resource (storageGb and memorySizeGb). By just exposing relevant parameters to the Application teams you simplify the configuration of the resources and abstract away all the complexity required for provisioning these databases.
- **WriteConnectionSecretToRef:** defines the name of the secret where the connection parameters will be written. This is important for application's services to know where to fetch the connection string, username and passwords required to connect to the newly provisioned resources.

4.3.2 Connecting our services with the new provisioned infrastructure

Crossplane will monitor the status of these resources against the status of the provisioned components inside the specific Cloud Provider, keeping them in sync and making sure that the desired configurations are applied. This loop will allow us to query the state of the

components by using `kubectl`. You can run the following command to check the status of your provisioned resources:

```
kubectl get postgresqlinstances.fmtok8s.salaboy.com
```

Check for the READY column, if the status is READY you can start connecting to your newly created databases by using the information in the secrets that Crossplane creates for you. For example, if you want to connect the Call For Proposals (C4P) Service with your newly created PostgreSQL instance you can update the Environment Variables in the C4P deployment. You can do that by editing the running deployment with:

```
kubectl edit deploy app-fmtok8s-c4p
```

And then update the environment variables in the `env` section:

```
env:
- name: SPRING_DATASOURCE_DRIVERCLASSNAME
  value: org.postgresql.Driver
- name: SPRING_DATASOURCE_PLATFORM
  value: postgres
- name: SPRING_DATASOURCE_URL
  value: jdbc:postgresql://${DB_ENDPOINT}:${DB_PORT}/postgres
- name: SPRING_DATASOURCE_USERNAME
  valueFrom:
    secretKeyRef:
      name: db-conn
      key: username
- name: SPRING_DATASOURCE_PASSWORD
  valueFrom:
    secretKeyRef:
      name: db-conn
      key: password
- name: DB_ENDPOINT
  valueFrom:
    secretKeyRef:
      name: db-conn
      key: endpoint
- name: DB_PORT
  valueFrom:
    secretKeyRef:
      name: db-conn
      key: port
```

With these environment variables, we are instructing the service to connect using the PostgreSQL JDBC Driver and using the `db-conn` secret internal keys called: `username`, `password`, `endpoint` and `port`.

As soon as you save these changes, the Pod will be restarted to apply the configuration changes. The new instance of the pod will connect to our newly created PostgreSQL database using the parameters provided. From that point onwards you can scale the number of replicas and will all connect to the same database in GCP, using the same parameters.

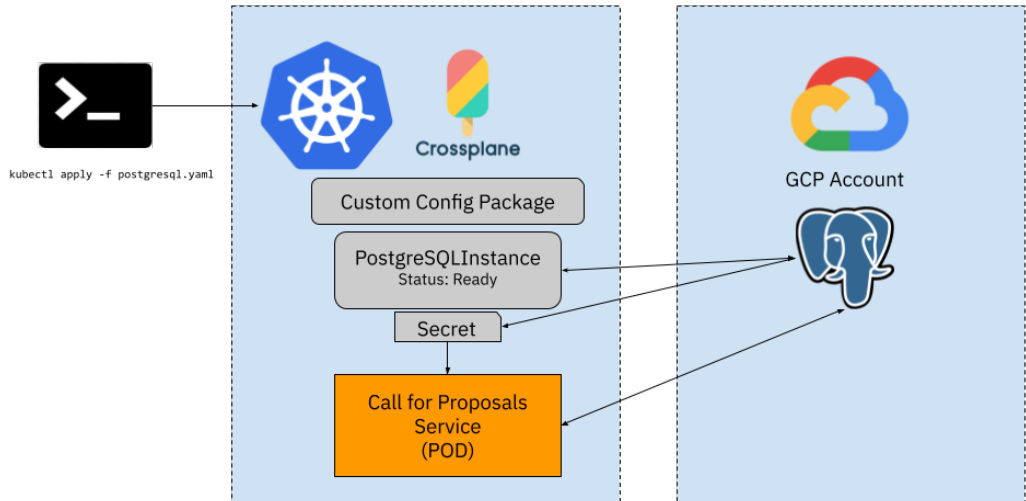


Figure 4.12 Call for Proposals Service using PostgreSQL provisioned by Crossplane

To summarize what we have achieved so far we can say that:

- We abstracted how to provision a Cloud specific component such as a PostgreSQL database and all the configurations needed to access the new instances,
- We have exposed a simplified interface for the App Ops team which is Cloud Provider independent.
- Finally, we have connected our application service to the newly provisioned instance by relying on a Kubernetes Secret that is created by Crossplane containing all the details required to connect to the newly created instance.

The same can be achieved for the Agenda Service and Redis. You can follow along a step by step tutorial that covers all the steps described in this section here: <https://github.com/salaboy/from-monolith-to-k8s/tree/master/crossplane>

In this section, we have used a configuration package that I've created to provision a PostgreSQL and a Redis database on top of Google Cloud without requiring developers to know where the resources are going to be provisioned or all the cloud provider-specific configurations required to create these resources.

If you create more complex composite resources and configurations, what you are defining is basically a higher-level abstraction on how you want to expose the resources that matter for your applications. In one way or another, you will end up building a domain-specific platform on top of Kubernetes and using Crossplane to define and manage these higher-level constructs. This approach not only simplifies the creation and management of such platforms but it will also allow these constructs to be shared across different cloud providers and split the responsibility of defining and using these abstractions to specialized teams.

The following section quickly reviews the Crossplane Composite Resources used for the walking skeleton application, focusing on the definition of the configuration package that we installed previously.

4.4 Building Cloud-Native platforms on top of Kubernetes

Most of the time, the configurations installed in our package will be in charge of the Infra Ops team, who understand more about the Cloud Provider capabilities and configurations.

The source code of the configuration package used can be found here: <https://github.com/salaboy/from-monolith-to-k8s/tree/master/crossplane/pkg>

To define provider or configuration packages you need to first create a new configuration type describing what your package is going to be about and its dependencies. The package that we installed for the Conference Platform had defined as dependency the “provider-gcp” package as it depends on GCP specific types. You can consider this package as an extension on top of GCP types, which is exactly how we used it in the example.

4.4.1 Creating our own Crossplane Configuration Package

As with every component that we install in our Kubernetes Cluster, we will need to maintain it and upgrade it if we need to update the configurations or if Crossplane component changes, hence defining which Crossplane version we are targeting is important.

```
crossplane.yaml:
apiVersion: meta.pkg.crossplane.io/v1
kind: Configuration
metadata:
  name: fmtok8s-gcp
  annotations:
    app: fmtok8s-app
    provider: gcp
spec:
  crossplane:
    version: ">=v1.0.0-0"
  dependsOn:
    - provider: crossplane/provider-gcp
      version: ">=v0.16.0"
```

The next step is to create a Composition and for the example, I have kept it as simple as possible, but here is where the power of Crossplane really shines, as you can create complex compositions implementing common patterns and best practices that the users of your platform will need. The following yaml file defines the CompositePostgreSQLInstance type, which for the sake of simplicity just involves a `CloudSQLInstance` type coming from the Crossplane GCP provider. It is important to notice that a configuration package can contain as many compositions as you want.

```

composition.yaml1:

apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: compositepostgresinstances.gcp.db.fmtok8s.salaboy.com
  labels:
    provider: gcp
    app: fmtok8s-app
spec:
  writeConnectionSecretsToNamespace: crossplane-system
  compositeTypeRef:
    apiVersion: db.fmtok8s.salaboy.com/v1alpha1
    kind: CompositePostgreSQLInstance
  resources:
    - name: cloudsqlinstance
      base:
        apiVersion: database.gcp.crossplane.io/v1beta1
        kind: CloudSQLInstance
        spec:
          forProvider:
            databaseVersion: POSTGRES_9_6
            region: europe-west2
            settings:
              tier: db-custom-1-3840
              dataDiskType: PD_SSD
              ipConfiguration:
                ipv4Enabled: true
                authorizedNetworks:
                  - value: "0.0.0.0/0"
            writeConnectionSecretToRef:
              namespace: crossplane-system
          patches:
            - fromFieldPath: "metadata.uid"
              toFieldPath: "spec.writeConnectionSecretToRef.name"
              transforms:
                - type: string
                  string:
                    fmt: "%s-postgresql"
            - fromFieldPath: "spec.parameters.storageGB"
              toFieldPath: "spec.forProvider.settings.dataDiskSizeGb"
          connectionDetails:
            - fromConnectionSecretKey: username
            - fromConnectionSecretKey: password
            - fromConnectionSecretKey: endpoint
            - type: FromValue
              name: port
              value: "5432"

```

Notice that inside the composition we can set sensible defaults, so our users (App Ops teams) don't need to worry about which version of PostgreSQL they need to select, in which region the database needs to be provisioned or even which kind of storage the DB will need.

Your users will not consume the composition straight away, they will rely on a simplified interface that you can define by creating `CompositeResourceDefinition` which defines the

parameters that a user can set (the shape of the interface) to be propagated to the composition that we defined previously.

With the following `definition.yaml` file we create a schema for our PostgreSQLInstance type which defines which parameters can be set and which parameters are required. For this simple example, the only parameter that you can set and is required is `storageGB`.

```
definition.yaml:

apiVersion: apiextensions.crossplane.io/v1
kind: CompositeResourceDefinition
metadata:
  name: compositepostgresqlinstances.db.fmtok8s.salaboy.com
spec:
  group: db.fmtok8s.salaboy.com
  names:
    kind: CompositePostgreSQLInstance
    plural: compositepostgresqlinstances
  claimNames:
    kind: PostgreSQLInstance
    plural: postgresqlinstances
  connectionSecretKeys:
    - username
    - password
    - endpoint
    - port
  versions:
    - name: v1alpha1
      served: true
      referenceable: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                parameters:
                  type: object
                  properties:
                    storageGB:
                      type: integer
                      required:
                        - storageGB
                  required:
                    - parameters
```

Notice that these simplified interfaces that you are going to expose to the App Ops team can and should be versioned as with any other resource type.

4.4.2 Building and distributing our Configuration Package

Once you have these 3 files defined you can package your configuration package using the Crossplane `kubectrl` plugin, by running:

```
kubectrl crossplane build configuration
```

This will generate an OCI container that you can push to Docker Hub or any other container registry. The name of the OCI container will be defined by the name of the Configuration resource that you have defined in the `crossplane.yaml` file. To push the configuration package to the registry you can also use the plugin:

```
kubect1 crossplane push configuration salaboy/crossplane-fmtok8s-gcp:0.0.8
```

Notice that you need to have the right credentials to push to the target registry. If not specified it will use Docker Hub, where "salaboy" is my Docker Hub username.

If you really want to create a self-service platform, as the Google Cloud user interface but on top of Kubernetes you will need to build a User Interface that reads from the installed Configuration Packages the **CompositeResourceDefinitions** type Schema definitions. You can build parameterizable user screens where users can select the types of resources they want, enter the required parameters and create resources without the need of having `kubect1` installed in their environments.

4.4.3 Extending Crossplane with our custom Providers

Most of the examples and scenarios discussed in this chapter are targeting Cloud Providers, but Crossplane can be also used for On-Premises setups and extended to support other managed services besides the ones provided by AWS, Azure and Google. The abstractions and mechanisms provided by Crossplane are valuable in a wide range of scenarios where the resources that we want to provision and monitor are outside of the Kubernetes Cluster where our services are running.

If you want to extend Crossplane with your own Providers I recommend you to check the Provider Template (<https://github.com/crossplane/provider-template>) project that shows a simple example of how a Provider can be created and distributed. A Crossplane Provider consists of a Kubernetes Controller which will use credentials to connect to the remote Managed Service and then expose the resources that can be provisioned remotely as Kubernetes Resources.

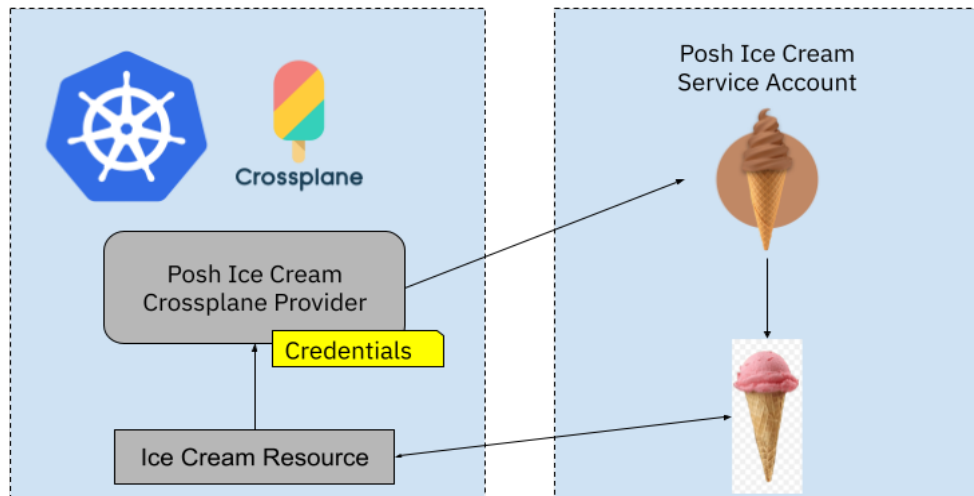


Figure 4.13 Custom Crossplane Provider for your Managed Service

Imagine that you have a service hosted somewhere else that can produce new Posh Ice Creams. For creating new Posh Ice Cream instances all you need is an account to this service. You are not interested in creating Posh Ice Cream instances inside your Kubernetes Clusters, but you are interested in allowing your users to create Posh Ice Creams, no matter where they are created. This is a perfect scenario for creating a Crossplane Provider which understands how to connect to the Posh Ice Cream Service using your account credentials.

The implementation of the Posh Ice Cream Crossplane Provider will include a new Kubernetes Custom Resource Definition called IceCream resource which will be in charge of defining the properties for the Ice Cream that you want to create on the remote service. As with Compositions, the Custom Crossplane Provider can manage and expose as many new resources as the external platform has to offer. As an example, if the Posh Ice Cream Service also offers coffee, the Provider can add an extra resource to enable users to create remote coffee cups.

If you feel that this provider is going to be useful for the entire Crossplane community you can donate your provider to the Crossplane Contrib organization: <https://github.com/crossplane-contrib>.

4.5 Summary

- Cloud-Native applications depend on more application infrastructure, as each service might require different persistent storages, a message broker to send messages and other components to work.
- Creating application infrastructure inside Cloud Providers is easy and can save us a lot of time, but then we rely on their tools and ecosystem.

- Provisioning infrastructure in a cloud-agnostic way can be achieved by relying on the Kubernetes API and tools like Crossplane, which abstract away the underlying Cloud Provider.
- Crossplane provides support for major Cloud Providers and it can be extended for other service providers.
- By creating our simplified abstractions for Application Infrastructure, we can enable developers to provision, using a self-service approach, components without knowing all the cloud provider-specific configuration required. We can share these abstractions by creating Crossplane Configuration Packages (OCI images) that can be distributed and shared with other teams.

5

Release Strategies

This chapter covers

- **Kubernetes release strategies using Deployments, Services and Ingress**
- **How to implement rolling updates, canary releases, blue/green deployments and A/B testing**
- **The limitations and challenges of using Kubernetes built-in resources for releasing software efficiently**
- **Knative Serving advanced traffic routing capabilities for reducing releases risk and speeding up release cadence**

If you and your teams are worried about deploying a new version of your service, you are doing something wrong. You are slowing down your release cadence due to the risk of making or deploying changes to your running applications.

Reducing risk and having the proper mechanisms in place to keep deploying new versions drastically improves the confidence in the system. It also reduces the time since a change is requested until the change is live in front of your users. New releases with fixes and new features directly correlate to business value, as software is not valuable unless it serves our company's users.

While Kubernetes built-in resources such as Deployments, Services and Ingresses provide us with the basic building blocks to deploy and expose our services to our users, there is a lot of manual and error-prone work that needs to happen to implement well-known release strategies.

This chapter is divided into two main sections:

- Release strategies using Kubernetes built-in mechanisms
 - Rolling Updates
 - Canary Releases
 - Blue/Green Deployments
 - A/B Testing
 - Limitations and complexities of using Kubernetes built-in mechanisms
- Reducing releases risk to improve delivery speed
 - Introduction to Knative Serving
 - Advanced traffic-splitting features
 - Knative Serving applied to our Conference Platform application

In this chapter, you will learn about Kubernetes built-in mechanisms and Knative Serving to implement release strategies such as rolling updates, canary releases, blue-green deployments and A/B testing. This chapter starts by covering these patterns, understanding the implementation implications and their limitations.

The second half of the chapter introduces Knative Serving, which allows us to implement more fine-grain control on how traffic is routed to our application's services. We explore how Knative Serving traffic-splitting mechanisms can be used to improve how we release new versions of our Cloud-Native application.

5.1 Kubernetes built-in mechanisms for releasing new versions

Kubernetes comes with built-in mechanisms ready to deploy and upgrade your services. Both Deployment and StatefulSets resources orchestrate ReplicaSets to run a built-in rolling update mechanism when the resource's configurations change. Both Deployments and StatefulSets keep track of the changes between one version and the next, allowing rollbacks to previously recorded versions.

Both Deployments and StatefulSets are like cookie cutters; they contain the definition of how the container(s) for our service(s) needs to be configured and, based on the replicas specified, will create that amount of containers. To route traffic to these containers, we will need to create a Service, as explained in Chapter 2.

Using a Service for each Deployment is standard practice, and it is enough to enable different services to talk to each other by using a well-known Service name. But if we want to allow external users (from outside our Kubernetes Cluster) to interact with our Services, we will need an Ingress resource (plus an ingress controller). The Ingress resource will be in charge of configuring the networking infrastructure to enable external traffic into our Kubernetes cluster; this is usually done for a handful of services. In general, not every service is exposed to external users.

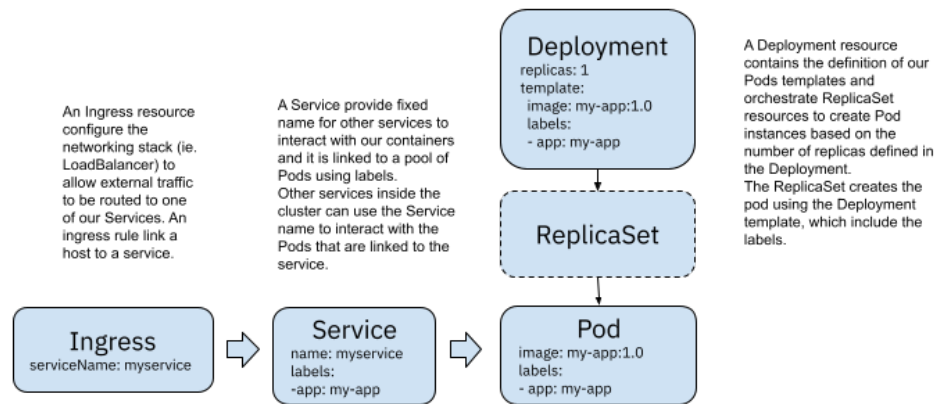


Figure 5.1 Kubernetes built-in resources for routing traffic

Now, imagine what happens when you want to update the version of one of these external-facing services. We can agree that it is pretty common that these services are user interfaces. And we can also agree that it will be pretty good if we can upgrade the service without having downtime. The good news is that Kubernetes was designed with zero-downtime upgrades in mind and for that reason, both Deployments and StatefulSets come equipped with a rolling update mechanism.

If you have multiple replicas of your application Pods, the Kubernetes Service resource acts as a load balancer. This allows other services inside your cluster to use the Service name without caring about which replica they are interacting with (or which IP address each replica has).

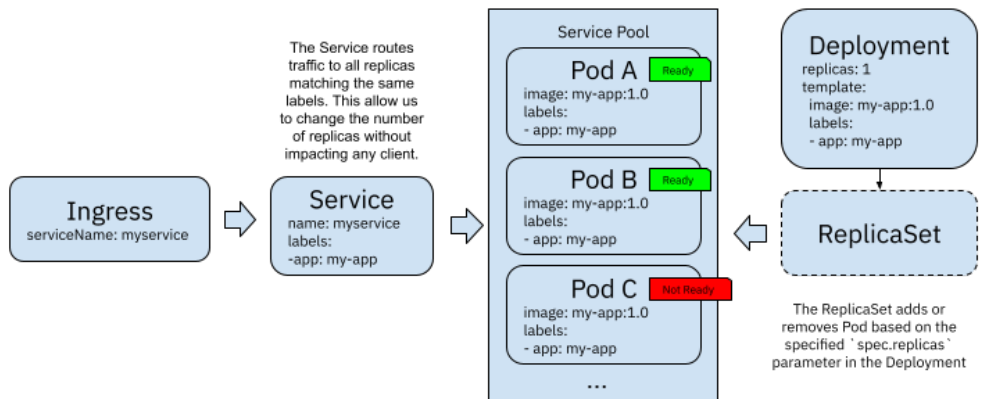


Figure 5.2 Kubernetes Service acting as a load balancer

To attach each of these replicas to the load balancer (Kubernetes Service) pool, Kubernetes uses a probe called “Readiness Probe” (<https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/>) to make sure that the container running inside the Pod is ready to accept incoming requests, in other words, it has finished bootstrapping. In figure 5.2, Pod C is not ready yet, hence it is not attached to the Service pool, and no request has been forwarded to this instance yet.

Now, if we want to upgrade to using the `my-app:1.1` container image, we need to perform some very detailed orchestration of these containers. Suppose we want to make sure that we don’t lose any incoming traffic while doing the update. We need to start a new replica using the `my-app:1.1` image and make sure that this new instance is up and running and ready to receive traffic before we remove the old version. If we have multiple replicas, we probably don’t want to start all the new replicas simultaneously, as this will cause doubling up all the resources required to run this service.

We also don’t want to stop the old `my-app:1.0` replicas in one go. We need to guarantee that the new version is working and handling the load correctly before we shut down the previous version that was working fine. Luckily for us, Kubernetes automates all the starting and stopping of containers using a rolling update strategy (<https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/>).

5.1.1 Rolling updates

Deployments and StatefulSets come with these mechanisms built-in, and we need to understand how these mechanisms work to know when to rely on them and their limitations and challenges.

A rolling update consists of well-defined checks and actions to upgrade any number of replicas managed by a Deployment. Deployment resources orchestrate ReplicaSets to achieve rolling updates, and the following figure shows how this mechanism works:

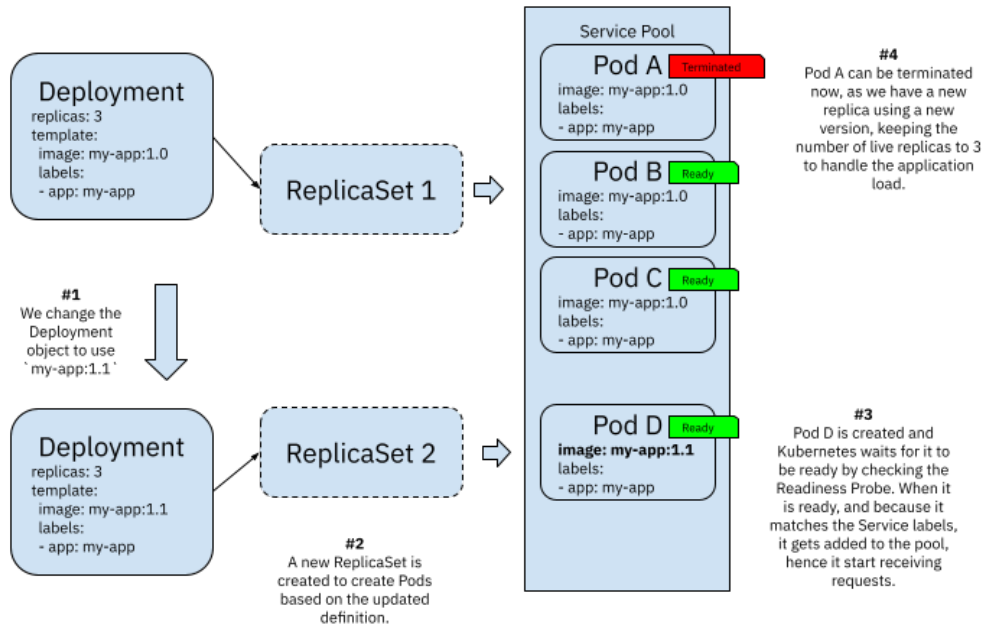


Figure 5.3 Kubernetes Deployments rolling updates

Whenever you update a Deployment resource, the rolling update mechanism kicks off by default. A new ReplicaSet is created to handle the creation of Pods using the newly updated configuration defined in `spec.template`. This new ReplicaSet will not start all three replicas straight away, as this will cause a surge in resource consumption. Hence it will create a single replica, validate that it is ready, attach it to the service pool and then terminate a replica with the old configuration.

By doing this, the Deployment object guarantees three replicas active at all times handling clients' requests. Once Pod D is up and running, and Pod A is terminated, ReplicaSet 2 can create Pod E, wait for it to be ready and then terminate Pod B. This process repeats until all Pods in ReplicaSet 1 are drained and replaced with new versions managed by ReplicaSet 2. If you change the Deployment resource again, a new ReplicaSet (ReplicaSet 3) will be created, and the process will repeat in the same way.

An extra benefit of using rolling updates is that ReplicaSets contain all the information needed to create Pods for a specific Deployment configuration. If something goes wrong with the new container image (in this example `my-app:1.1`) we can easily revert (rollback) to a previous version. You can configure Kubernetes to keep a certain number of revisions (changes in the Deployment configuration), so changes can be rolled back or rolled forward.

Changing a Deployment object will trigger the rolling update mechanism, and you can check some of the parameters that you can configure to the default behaviour [here](#)

(<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>). StatefulSets have a different behavior as the responsibility for each replica is related to the state that is handled. The default rolling update mechanism works a bit differently. You can find the differences and a detailed explanation about how this work here (<https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/>).

Check out the following commands to review the Deployment revisions history and doing rollbacks to previous versions (<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#checking-rollout-history-of-a-deployment>):

```
kubectl rollout history deployment/frontend
kubectl rollout undo deployment/frontend --to-revision=2
```

If you haven't played around rolling updates with Kubernetes I strongly recommend you to create a simple example and try these mechanisms out. There are loads of examples online and interactive tutorials where you can see this in action.

5.1.2 Canary Releases

Rolling updates kick in automatically, and they are performed as soon as possible if we are using Deployments; what happens when we want to have more control over when and how we roll out new versions of our services?

Canary Releases (<https://martinfowler.com/bliki/CanaryRelease.html>) is a technique used to test if a new version is behaving as expected before pushing it live in front of all our live traffic.

While rolling updates will check that the new replicas of the service are ready to receive traffic, Kubernetes will not check that the new version is not failing to do what it is supposed to do. Kubernetes will not check that the latest versions perform the same or better than the previous. Hence we can be introducing issues to our applications. More control on how these updates are doing is needed.

If we start with a Deployment configured to use a Docker image called `my-app:1.0``, have two replicas, and we label it with `app: myapp`` a Service will route traffic as soon as we use the selector `app:myapp``. Kubernetes will be in charge of matching the Service selectors to our Pods. In this scenario, 100% of the traffic will be routed to both replicas using the same Docker image (`my-app:1.0``).

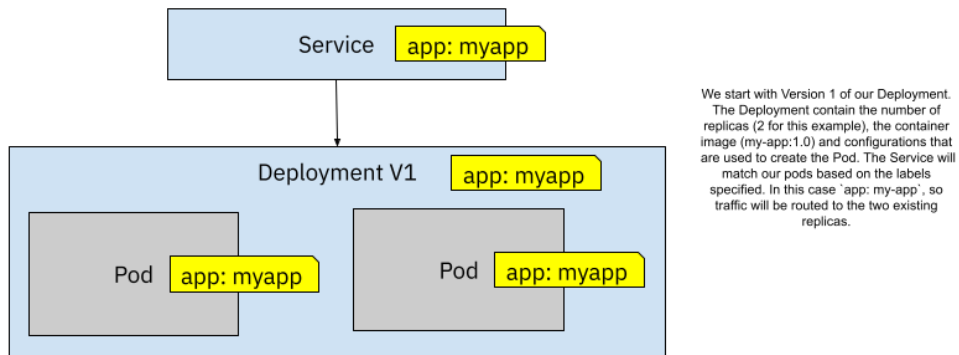


Figure 5.4 Kubernetes Service routes traffic to two replicas by matching labels

Now imagine changing the configuration or having a new Docker image version (maybe `my-app:1.1`). We don't want to automatically migrate our Deployment V1 to the new version of our Docker image. Alternatively, we can create a second Deployment (V2) resource and leverage the service `selector` to route traffic to the new version.

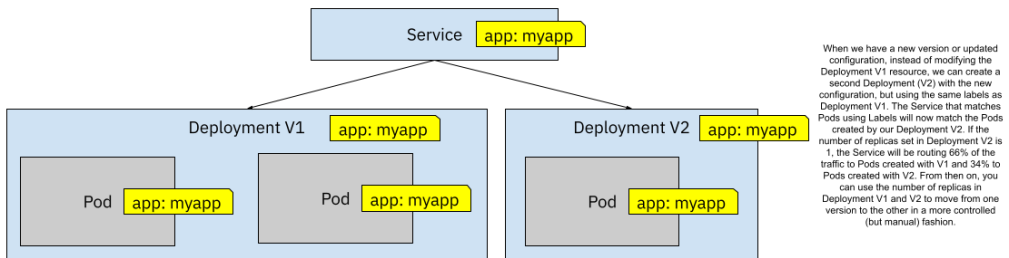


Figure 5.5 One Service and two Deployments sharing the same labels

By creating a second Deployment (using the same labels), we are routing traffic to both versions simultaneously, and how much traffic goes to each version is defined by the number of replicas configured for each Deployment. By starting more replicas on *Deployment V1* than *Deployment V2* you can control what percentage of the traffic will be routed to each version. Figure 5.5 shows a 66%/34% (2 to 1 pods) traffic split between V1 and V2. Then you can decrease the number of replicas for Deployment V1 and increase the replicas for V2 to slowly move towards V2. Notice that you don't have a fine grain control about which requests go to which version—the Service forward traffic in a round-robin fashion to all the matched pods.

Because we have replicas ready to receive traffic at all times, there shouldn't be any downtime of our services when we do Canary Releases.

A significant point to make about rolling updates and Canary Releases is that they depend on our services supporting traffic to be forwarded to different versions simultaneously without breaking. This usually means that we cannot have breaking changes from version 1.0 to version 1.1 that will cause the application (or the service consumers) to crash when switching from one version to the other. Teams making the changes need to be aware of this restriction when using rolling updates and Canary Releases, as traffic will be forwarded to both versions simultaneously. For cases when two different versions can not be used simultaneously, and we need to have a hard switch between version 1.0 and version 1.1 we can look at Blue/Green Deployments.

5.1.3 Blue/Green Deployments

Whenever you face a situation where you can just not upgrade from one version to the next and have users/clients consuming both versions simultaneously, you need a different approach. Canary Deployments or Rolling Updates, as explained in the previous section, will just not work. If you have breaking changes, you might want to try Blue/Green deployments (<https://martinfowler.com/bliki/BlueGreenDeployment.html>).

Blue/Green deployments help us move from version 1.0 to version 1.1 at a fixed point in time, changing how the traffic is routed to the new version without allowing the two versions to receive requests simultaneously and without downtime.

Blue/Green Deployments can be implemented using built-in Kubernetes Resources by creating two different Deployments, as shown in figure 5.6.

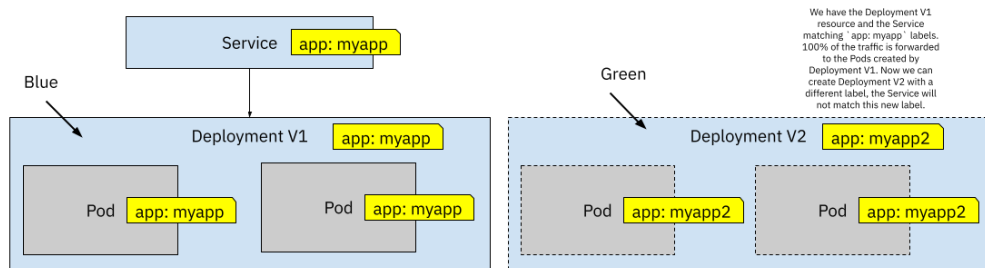


Figure 5.6 Two Deployments using different Labels

In the same way, as we did with Canary Releases, we start with a Service and a Deployment. The first version of these resources is what we call "Blue". For Blue/Green Deployments, we can create a separate Deployment (V2) resource to start and get ready for the new version of our service when we have a new version. This new Deployment needs to have different labels for the Pods that it will create, so the Service doesn't match these Pods just yet. We can connect to Deployment V2 pods by using `kubectl port-forward` or running other in-cluster tests until we are satisfied that this new version is working. When we are happy with our testing, we can switch from Blue to Green by updating the `selector` labels defined in the Service resource.

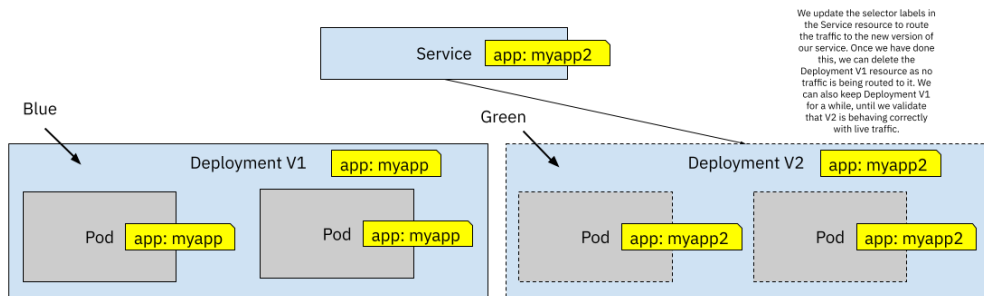


Figure 5.7 When the new version is ready we switch the label from the Service selector to match version V2 labels

Blue/Green Deployments makes a lot of sense when we cannot send traffic to both versions simultaneously, but it has the drawback of requiring both versions to be up at the same time to switch traffic from one to the other. When we do Blue/Green deployments it is recommended to have the same amount of replicas running for the new version. We need to make sure that when traffic is redirected to the new version, this version is ready to handle the same load as the old version.

The moment we switch the label selector in the Service resource, 100% of the traffic is routed to the new version, Deployment V1 pods' stop receiving traffic. This is quite an important detail as if some state was being kept in V1 Pods you will need to drain state from the Pods, migrate and make this state available for V2 Pods. In other words, if your application is holding state in-memory, you should write that state to a persistent storage so V2 Pods can access it and resume whatever work was being done with that data. Remember that most of these mechanisms were designed for stateless workloads, so you need to make sure that you follow these principles to make sure that things work as smoothly as possible.

For Blue/Green deployments, we are interested in moving from one version to the next at a given point in time, but what about scenarios when we want to actively test two or more versions with our users at the same time, to then decide which one performs better. Let's take a look at A/B testing next.

5.1.4 A/B testing

It is a quite common requirement to have two versions of your services running at the same time and we want to test these two versions to see which one performs better. (<https://blog.christianposta.com/deploy/blue-green-deployments-a-b-testing-and-canary-releases/>). Sometimes you want to try a new user interface theme, place some UI elements in different positions or new features added into your app and gather feedback from users to decide which one works best.

To implement this kind of scenario in Kubernetes, you need to have two different Services pointing to two different Deployments. Each Deployment handles just one version of the application/service. If you want to expose these two different versions outside the cluster

you can define an Ingress resource with two rules. Each rule will be in charge of defining the external path or subdomain to access each service.

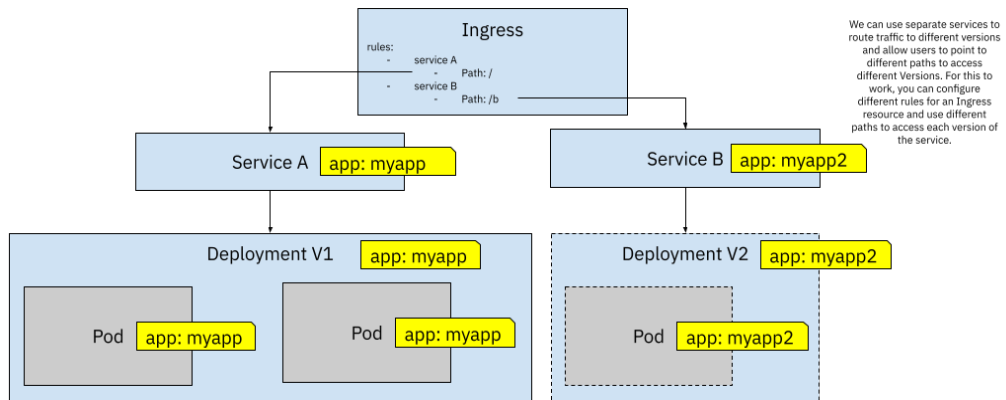


Figure 5.8 Using two Ingress rules for pointing to A and B versions

If you have your application hosted under the `www.example.com` domain, the Ingress resource defined in figure 5.8 will direct traffic to Service A, allowing users to point their browsers to `www.example.com/b` to access Service B. Alternatively, and depending on how you have configured your Ingress Controller, you can also use subdomains instead of path-based routing meaning that to access the default version you can keep using `www.example.com` but to access Service B you can use a subdomain such as `test.example.com`

I can hear you saying, what a pain, look at all the Kubernetes resources that I need to define and maintain to just achieve something that feels basic and needed for everyday operations. Let's quickly summarize the limitations and challenges that we found so far, so we can take a look at Knative Serving and how can it help us to implement these strategies in a more streamlined way.

5.1.5 Limitations and complexities of using Kubernetes built-in building blocks

Canary Releases, Blue/Green deployments and A/B testing can be implemented using built-in Kubernetes resources. But as you have seen in the previous sections, creating different deployments, changing labels and calculating the number of replicas needed to achieve percentage-based distribution of the requests is quite a major task and very error-prone. Even if you use a GitOps approach as shown with Jenkins X or with other tools in Chapter 3, creating the required resources with the right configurations is quite hard and it takes a lot of effort.

We can summarise the drawbacks of implementing these patterns using Kubernetes building blocks as follows:

- Manual creation of more Kubernetes resources, such as Deployments, Services and Ingress Rules to implement these different strategies can be error-prone and cumbersome. The team in charge of implementing the release strategies need to understand deeply how Kubernetes behaves to achieve the desired configuration.
- No automated mechanisms provided out-of-the-box to coordinate and implement the resources required by each release strategy.
- Error-prone, as multiple changes need to be applied at the same time in different resources for everything to work as expected
- If we notice a demand increase or decrease in our services, we need to manually change the number of replicas for our Deployments or install and configure a custom autoscaler (more on this at the end of this chapter). Unfortunately, if you set the number of replicas to 0, there will not be any instance to answer requests, requiring you to have at least one replica running all the time.

Out of the box, Kubernetes doesn't include any mechanism to automate or facilitate these release strategies, and that becomes a problem quite quickly if you are dealing with a large number of services that depend on each other.

One thing is clear, your teams need to be aware of the implicit contracts imposed by Kubernetes regarding 12-factor apps and how their services APIs evolve to avoid downtime. Your developers need to know how Kubernetes built-in mechanisms work in order to have more control over how your applications are upgraded.

If we want to reduce the risk of releasing new versions, we want to empower our developers to have these release strategies available for their daily experimentation. In the next sections, we will look at Knative Serving, a set of tools and mechanisms built on top of Kubernetes to simplify all the manual work described in the previous sections.

5.2 Reducing releases risk to improve delivery speed

We want to ensure that our operations teams have the right tools to implement the introduced strategies for running our applications. This section covers Knative Serving (<http://knative.dev>), a set of tools and mechanisms that facilitate a progressive delivery (<https://redmonk.com/jgovernor/2018/08/06/towards-progressive-delivery/>) approach.

While Knative Serving has been associated with Serverless Platforms, we will see in the following sections that a Serverless approach can be achieved but by no means is the only feature that Knative Serving provides. In the following sections, we will see how to implement the previously introduced release strategies with Knative Serving applied to our Conference Platform application.

5.2.1 Introduction to Knative Serving

Knative is one of these technologies that it is hard to go back and not use when you learn about what it can do for you. After working with the project for almost 3 years and observing the evolution of some of its components, I can confidently say that every Kubernetes Cluster should have Knative installed; your developers will appreciate it. Knative provides higher-

level abstractions on top of Kubernetes built-in resources to implement good practices and common patterns that enable your teams to go faster, have more control over their services and facilitate the implementation of Event-Driven applications.

As the title of this section specifies, the following sections focus on a subset of functionality provided by Knative, called Knative Serving. Knative Serving allows you to define "*Knative Services*", which dramatically simplifies implementing the release strategies exemplified in the previous sections. Knative Services will take care of creating Kubernetes built-in resources for you and keep track of their changes and versions, enabling scenarios that require multiple versions to be present at the same time. Knative Services also provides advanced traffic handling and autoscaling to scale down to zero replicas for a serverless approach.

To install Knative Serving on your cluster, I highly recommend you to check the official Knative documentation that you can find here: <https://knative.dev/docs/admin/install/serving/install-serving-with-yaml/>

If you want to install Knative locally, you should the getting started guide for local development(<https://knative.dev/docs/getting-started/#install-the-knative-quickstart-environment>). Optionally, you can install `kn`, a small binary that allows you to interact with Knative in a more natural way. You can always use `kubectl` to interact with Knative resources as well.

When installing Knative Serving, you are installing a set of components in the `knative-serving` namespace that understand how to work Knative Serving custom resource definitions.

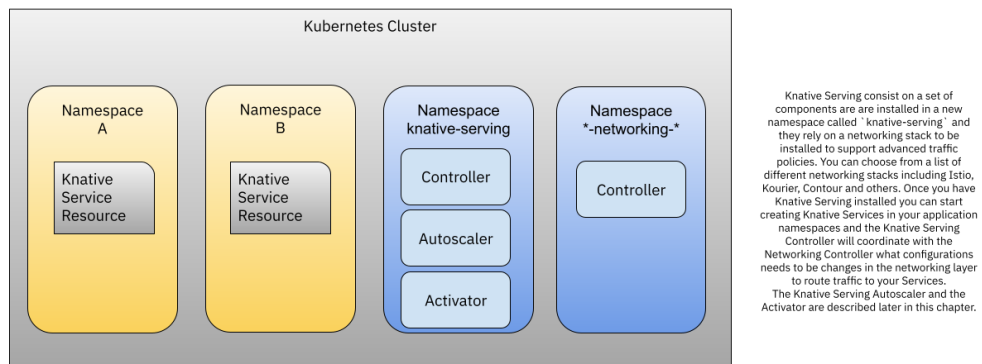


Figure 5.9 Knative Serving components in your Kubernetes Cluster

It is outside of the scope of this book to explain how Knative Serving components and resources work; my recommendation is that if I manage to get your attention with the examples in the following sections, you should check out the Knative In Action book by Jacques Chester (<https://www.manning.com/books/knative-in-action>).

5.2.2 Knative Services

Once you have Knative Serving installed, you can create *Knative Services*. I can hear you thinking: “But we already have Kubernetes Services, why do we need Knative Services?” believe me, I had the same feeling when I saw the same name but follow along, it does make sense.

When you were deploying each Service in the Conference Application before, you needed to create at least two Kubernetes resources, a Deployment and a Service. We have covered in a previous section that using ReplicaSets internally, a Deployment can perform rolling updates by keeping track of the configuration changes in the Deployment resources and creating new ReplicaSets for each change that we made. We also discussed in Chapter 2 the need for creating an Ingress resource, if we want to route traffic from outside the cluster. Usually, you will only create an Ingress resource to map the publicly available services, such as the User Interface of the Conference Platform.

Knative Services build on top of these resources and simplify how we define these resources that you need to create for every application service. While it simplifies the task and reduces the amount of YAML that we need to maintain, it also adds some exciting features, but before jumping into the features, let’s take a look at how a Knative Service looks in action. Let’s start simple and use the Email Service from the Conference Platform to demonstrate how Knative Services work:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: email-service #A
spec:
  template:
    spec:
      containers:
        - image: salaboy/fmtok8s-email-rest:0.1.0 #B
```

#A You need to specify a name for the resource, as with any other Kubernetes resource

#B You need to specify which container image you want to run

In the same way, as a Deployment will pick the `spec.template.spec` field to cookie-cut Pods, a Knative Service defines the configuration for creating other resources using the same field.

Nothing too strange so far, but how is this different from a Kubernetes Service?

If you create this resource using `kubectl apply -f` you can start exploring the differences.

You can also list all Knative Services using `kubectl get ksvc` (ksvc stand for Knative Service) and you should see your newly created Knative Service in there:

NAME	URL	LATEST CREATED	READY
email-service	http://email-service.default.X.X.X.X.sslip.io	email-service-00001	True

There are a couple of details to notice right here; first of all, there is a URL that you can copy into your browser and access the service. If you were running in a Cloud Provider and configured DNS while installing Knative, this URL should be accessible immediately. The `LASTCREATED` column shows the name of the latest Knative Revision of the Service. Knative Revisions are pointers to the specific configuration of our Service, meaning that we can route traffic to them.

You can go ahead and test the Knative Service URL by using ``curl`` or by pointing your browser to `http://email-service.default.X.X.X.X.sslip.io/info`

You should see the following output:

```
{ "name": "Email Service
  (REST)", "version": "v0.1.0", "source": "https://github.com/salaboy/fmtok8s-email-
  rest/releases/tag/v0.1.0", "podId": "", "podNamepsace": "", "podNodeName": "" }
```

As with any other Kubernetes resource, you can also use ``kubectl describe ksvc email-service`` to get a more detailed description of the resource.

If you list other well-known resources such as Deployment, Services and Pods you will find out that Knative Serving is creating them for you and managing them. Because these are managed resources now, it is usually not recommended to change them manually. If you want to change your application configurations, you should edit the Knative Service resource.

A Knative Service, as we applied it before to our cluster, by default behaves differently from creating a Service, a Deployment and an Ingress manually.

A Knative Service by default:

- **It is accessible:** Expose itself under a public URL so you can access it from outside the cluster. It doesn't create an Ingress resource, as it uses the available Knative Networking stack that you installed previously. Because Knative has more control over the network stack and manages Deployments and Services, it knows when the service is ready to serve requests, reducing configuration errors between Services and Deployments.
- **Manages Kubernetes resources:** It creates two Services and a Deployment: Knative Serving allows us to run multiple versions of the same service simultaneously. Hence it will create a new Kubernetes Service for each version (which in Knative Serving are called Revisions)
- **Collects Service usage:** It creates a Pod with the specified ``user-container`` and a sidecar container called ``queue-proxy``.
- **Scale-up and down based on demand:** Automatically downscale itself to zero if no requests are hitting the service (by default after 90 seconds)
 - It achieves this by downscaling the Deployment replicas to 0 using the data collected by the ``queue-proxy``
 - If a request arrives and there is no replica available, it scales up while queuing the request, so it doesn't get lost.

- Configuration changes history are managed by Knative Serving: If you change the Knative Service configuration, a new Revision will be created. By default, all traffic will be routed to the latest revision.

Of course, these are the defaults, but you can fine-tune each of your Knative Services to serve your purpose and, for example, implement the previously described release strategies.

In the next section, we will look at how using Knative Serving advanced traffic-handling features can be used to implement Canary Releases, Blue/Green Deployments, A/B testing and Header based routing.

5.2.3 Advanced traffic-splitting features

Let's start by looking at how you can implement a Canary Release for one of our application's services with a Knative Service.

This section starts by looking into doing Canary Releases using percentage-based traffic splitting. Then it goes into A/B testing by using tag-based and header-based traffic splitting.

CANARY RELEASES USING PERCENTAGE-BASED TRAFFIC SPLITTING

If you get the Knative Service resource (with `kubectl get ksvc email-service -o yaml`), you will notice that the `spec` section now also contain a `spec.traffic` section that was created by default, as we didn't specify anything.

```
traffic:
- latestRevision: true
  percent: 100
```

By default, 100% of the traffic is being routed to the latest Knative Revision of the service.

Now imagine that you made a change in your service to improve how emails are sent, but your team is not sure how well it will be received by people and we want to avoid having any backlash from people not wanting to sign in to our conference because of the website. Hence we can run both versions side by side and control how much of the traffic is being routed to each version (Revision in Knative terms).

Let's edit (`kubectl edit ksvc email-service`) the Knative Service and apply the changes:

```

apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: email-service
spec:
  template:
    spec:
      containers:
        - image: salaboy/fmtok8s-email-rest:0.1.0-improved #A
  traffic: #B
    - percent: 80
      revisionName: email-service-00001
    - latestRevision: true
      percent: 20

```

#A You have updated the docker image that the service will use to ``fmtok8s-email-rest:0.1.0-improved``

#B You have created an 80% / 20% traffic split where 80% of the traffic will keep going to your stable version and 20% to the newest version that you just updated

If you try now with ``curl`` you should be able to see the traffic split in action:

```

salaboy> curl http://email-service.default.x.x.x.x.sslip.io/info
{"name":"Email Service (REST)","version":"v0.1.0",
"source":"https://github.com/salaboy/fmtok8s-email-rest/releases/tag/v0.1.0"}
salaboy> curl http://email-service.default.x.x.x.x.sslip.io/info
{"name":"Email Service (REST)","version":"v0.1.0",
"source":"https://github.com/salaboy/fmtok8s-email-rest/releases/tag/v0.1.0"}
salaboy> curl http://email-service.default.x.x.x.x.sslip.io/info
{"name":"Email Service (REST)","version":"v0.1.0",
"source":"https://github.com/salaboy/fmtok8s-email-rest/releases/tag/v0.1.0"}
salaboy> curl http://email-service.default.x.x.x.x.sslip.io/info
{"name":"Email Service (REST)","version":"v0.1.0",
"source":"https://github.com/salaboy/fmtok8s-email-rest/releases/tag/v0.1.0"}
salaboy> curl http://email-service.default.x.x.x.x.sslip.io/info
{"name":"Email Service (REST) - IMPROVED!!","version":"v0.1.0", #A
"source":"https://github.com/salaboy/fmtok8s-email-rest/releases/tag/v0.1.0"}

```

#A One in five requests will go to the new "IMPROVED!!" version. Notice that this can take a while until the new Knative Revision is up and running.

Once you have validated that the new version of your service is working correctly, you can start sending more traffic until you feel confident to move 100% of the traffic to it. If things go wrong, you just revert the traffic split to the stable version.

Notice that you are not limited to just two service revisions; you can create as many as you want as long as the traffic percentage sum to 100%. Knative will make sure to follow these rules and scale up the required revisions of your services to serve requests. You didn't need to create any new Kubernetes resource, as Knative will create those for you, reducing the likelihood of errors that come with modifying multiple resources at the same time, wishing that everything is consistent.

By using percentages, you don't have control over where subsequent requests will land. Knative will just make sure to maintain a fair distribution based on the percentages that you have specified. This can become a problem if, for example, you have a User Interface instead of a simple REST endpoint.

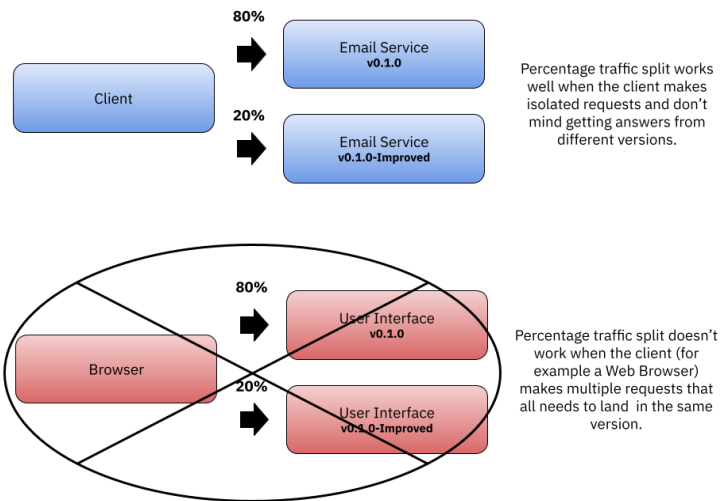


Figure 5.10 Percentage based traffic split scenarios and challenges

User Interfaces are complex because a browser will perform several GET requests to render the page HTML, CSS and images, and so forth. You can quickly end up in a situation where each request hits a different version of your application. Let's look at a different approach that might be better suited for testing User Interfaces or scenarios when we need to make sure that several requests end up in the correct version of our application.

A/B TESTING WITH TAG-BASED ROUTING

If you want to perform A/B testing for a User Interface, you will need to give Knative some way to differentiate where to send the requests. You have two options. First, you can point to a special URL for the service you want to try out, and the second is to use a request header to differentiate where to send the request. Let's look at these two alternatives in action.

But let's now use the User Interface for the Conference Platform:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: ui-service #A
spec:
  template:
    spec:
      containers:
        - image: salaboy/fmtok8s-api-gateway:0.1.0 #B
```

#A You need to specify a name for this Service

#B You now define the API Gateway / User Interface container image, as we are going to test multiple requests going to the same version

Once again, we have just created a Knative Service, but we cannot specify percentage-based routing rules because this container image contains a Web Application. Knative will not stop you from doing so. Still, you will notice requests going to different versions and errors popping up because a given image is not in one of the versions, or you end up with the wrong stylesheet (CSS) coming from the wrong version of the application.

Let's start by defining a Tag that you can use to test a new stylesheet. You can do that by modifying the Knative Service resource as we did before, first change the image to ``salaboy/fmtok8s-api-gateway:0.1.0-color`` and then let's create some new traffic rules using tags:

```
traffic:
- percent: 100 #A
  revisionName: ui-service-00001
- latestRevision: true
  tag: color #B
```

#A 100% of the traffic will go to our stable version, no request will be sent to our newly updated revision

#B You have created a new tag called ``color``, you can find the URL for this new tag by describing the Knative Service resource.

If you describe the Knative Service (``kubectl describe ksvc ui-service``) you will find the URL for the Tag that we just created:

```
Traffic:
  Latest Revision: false
  Percent: 100
  Revision Name: ui-service-00001
  Latest Revision: true
  Percent: 0
  Revision Name: ui-service-00001
  Tag: color
  URL: http://color-ui-service.default.x.x.x.x.sslip.io #A
```

#A Knative Serving had created a new URL by prepending the tag name to the service name. Check using the browser that can consistently access the modified version and the original one as well.

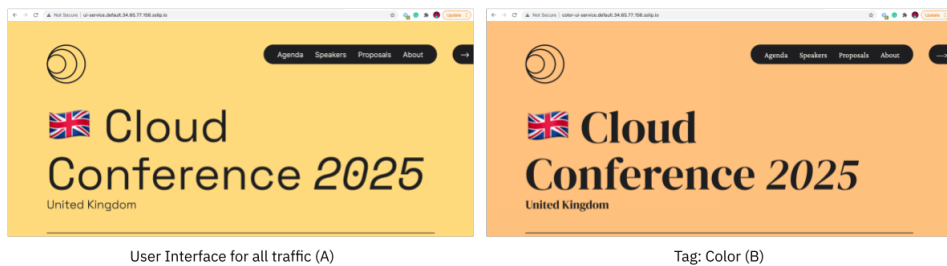


Figure 5.11 A/B testing with Tag-based routing

Using tags guarantees that all requests are hitting the URL to the correct version of your service. One more option avoids you pointing to a different URL for doing A/B testing, and it might be useful for debugging purposes. The next section looks at tag-based routing using HTTP Headers instead of different URLs.

A/B TESTING WITH HEADER-BASED ROUTING

Finally, let's take a look at an experimental feature (at the time of writing) that allows you to use HTTP Headers to route requests. This experimental (at the time of writing) feature also uses tags to know where to route traffic, but instead of using a different URL to access a specific revision, you can add an HTTP Header that will do the trick.

Imagine that you want to enable developers to access a debugging version of the application. Application developers can set a special header in their browsers and then access a specific revision.

To enable this experimental feature, you or the administrator that installs Knative needs to patch a ConfigMap inside the `knative-serving` namespace:

```
kubectl patch cm config-features -n knative-serving -p '{"data":{"tag-header-based-routing":"Enabled"}}'
```

Once the feature is enabled you can test this by changing again the Knative Service that we created in the previous section as follows:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: ui-service
spec:
  template:
    spec:
      containers:
        - image: salaboy/fmtok8s-api-gateway:0.1.0-debug #A
  traffic:
    - percent: 100
      revisionName: ui-service-00001
    - revisionName: ui-service-00002 #B
      tag: color
    - latestRevision: true #C
      tag: debug
```

#A You have another version for the API Gateway / User Interface that gives more debug information

#B Now you pin down to revision 00002 to the tag color

#C With this latest changes (using the -debug image) you create a new tag called `debug`

If you point your browser to the Knative Service URL (`kubectl get ksvc`) you will see the same application as always as shown in figure 5.12, but if you use a tool like ModHeader extension

(<https://chrome.google.com/webstore/detail/modheader/idqpnmonknjnojddfkpgkljpfnnfcklj?hl=en>) for Chrome you can set your custom HTTP Headers that will be included in every request that the browser produce. For this example, and because the tag that you created is called `debug` you need to set the following Http Header: `Knative-Serving-Tag: debug`.

As soon as the HTTP Header is present Knative Serving will route the incoming request to the debug tag.

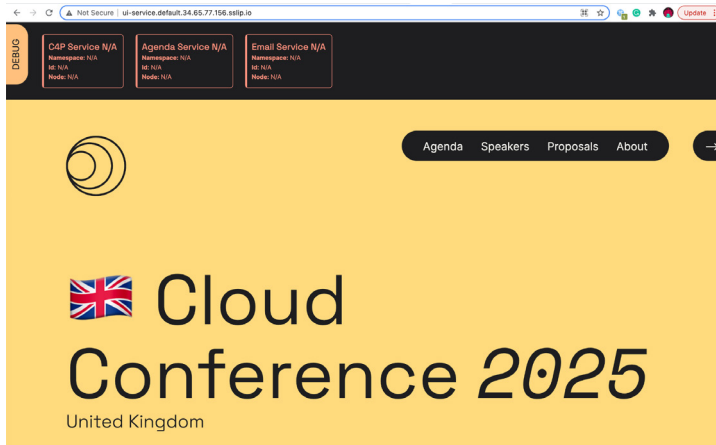


Figure 5.12 A/B testing with Http Header-based routing to Debug version

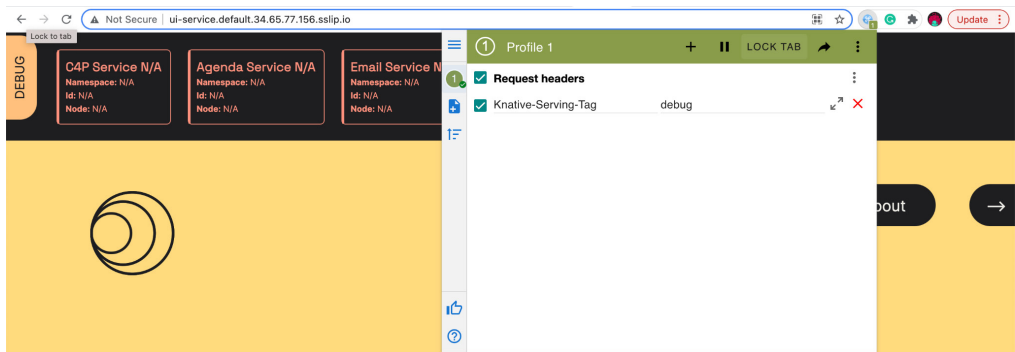


Figure 5.13 Using ModHeader Chrome extension to set custom HTTP Headers

Because the Header-based routing experimental feature also uses tags to route traffic. If you still have the `color` tag created, you can test traffic is being routed by changing the `Knative-Serving-Tag` header value to `color` and see if the correct page comes back.

Both tag and header-based routing are designed to ensure that all requests will be routed to the same revision if you hit a specific URL (created for the tag) or if one particular Header is present. Finally, let's take a look at how to do Blue/Green Deployments with Knative Serving.

BLUE/GREEN DEPLOYMENTS

For situations where we need to change from one version to the next at a very specific point in time, because there is no backward compatibility we can still use tag based routing with percentages. Instead of going gradually from one version to the next, we use percentages as a switch going from 0 to 100 on the new version and from 100 to 0 on the old version.

Most Blue/Green deployment scenarios require coordination between different teams and services to make sure that both the service and the clients are updated at the same time. Knative Serving allows you declaratively define when to do the switching from one version to the next in a controller way.

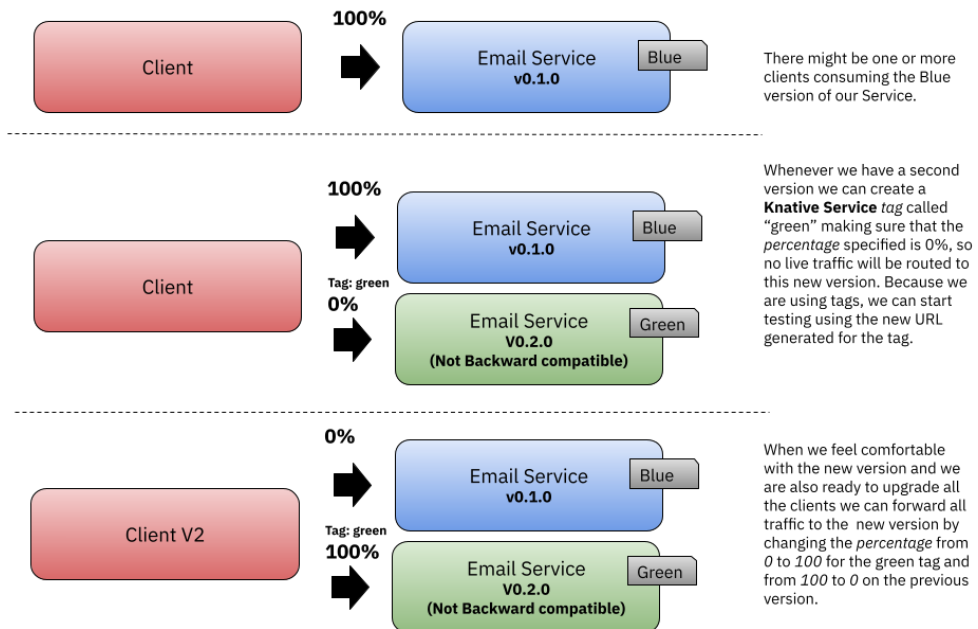


Figure 5.14 Blue/Green Deployments using Knative Serving tag-based routing

To achieve this scenario described in figure 5.14, we can create the "green" tag for the new version inside our Knative Service as follows:

```
...
traffic:
- revisionName: <first-revision-name>
  percent: 100 # All traffic is still being routed to the first revision
- revisionName: <second-revision-name>
  percent: 0 # 0% of traffic routed to the second revision
  tag: green # A named route
```

By creating a new tag (called “green”) we will have now available a new URL to access the new version for testing. This is particularly useful to test new versions of the clients, as if the Service API is changing with a non-backward compatible change, clients might need to be updated as well.

Once all tests are performed we can safely switch all traffic to the “green” version of our service.

```
...
traffic:
  - revisionName: <first-revision-name>
    percent: 0 # All traffic is still being routed to the first revision
  - revisionName: <second-revision-name>
    percent: 100 # 100% of traffic routed to the second revision
  tag: green # A named route
```

Generally, we cannot progressively move traffic from one version to the next, as the client consuming the service will need to understand that requests might land into different (and non-compatible) versions of the service.

In the previous sections, we have been looking into how Knative Serving simplifies the implementation of different Release Strategies for your teams to continuously deliver features and new versions of your services. Knative Serving reduces the need to create several Kubernetes built-in resources to manually implement the release strategies described in this chapter. It does so by providing high-level abstractions such as Knative Services which create and manage Kubernetes built-in resources and a network stack for advanced traffic management.

In the final section of this chapter, we will take a look at the Knative Autoscaler, which adds to the benefits of using Knative.

5.2.4 The Knative Serving Autoscaler

An important part of Knative Serving is the Autoscaler, which allows your Knative Services by default to scale down to zero and handle the logic to queue requests if no replica is available to answer incoming requests until at least one replica is up.

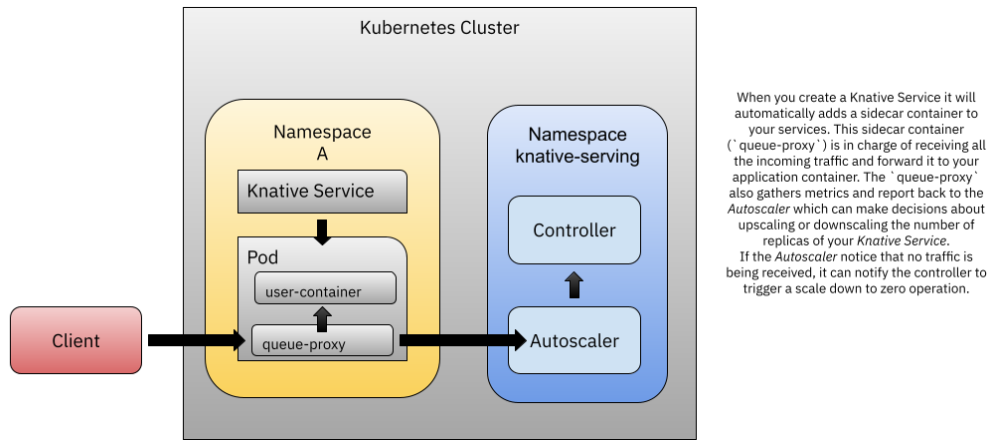


Figure 5.15 Knative Serving Autoscaler and `queue-proxy`

The Knative Autoscaler is installed by default when you install Knative Serving and it continuously monitor your Knative Service requests. As figure 5.15 shows, in order to monitor all inbound traffic a sidecar container is attached to your application container and all traffic is routed via the “queue-proxy” container. The “queue-proxy” container collects metrics and inform the Autoscaler about how much traffic the application container (“user-container”) is receiving.

Based on this information, the Autoscaler can notify the Knative Service controller to scale up or down the replicas of your service. If no requests are being handled by the “user-container” for a period of time, the Autoscaler will notify the Knative Service controller to scale down to zero the numbers of replicas.

Scaling up and down works out of the box, but you can always fine-tune the behaviour based on your application capabilities. In general, in order to configure the Autoscaler you can annotate your Knative Services with the amount of concurrent requests that your application can handle per replica. This information is going to be used by the Autoscaler to decide when new replicas are needed.

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld-go
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/target: "100" # 100 concurrent request by default
```

You can also change this for all Knative Services by changing a global parameter inside the `config-autoscaler` ConfigMap:

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: config-autoscaler
  namespace: knative-serving
data:
  container-concurrency-target-default: "100" # 100 concurrent request by default

```

Knative Serving also allows you to configure how to deal with sudden burst of traffic doing buffering and making sure that no request is lost. I recommended you to check the official documentation for more details about how these mechanisms can be configured: <https://knative.dev/docs/serving/autoscaling/concurrency/>

Finally, you might be wondering how Knative Serving deals with traffic when there is no replica available, as both, the `user-container` and the `queue-proxy` are downscaled to zero if there are no requests going to the service for a period of time. Here is where the Activator component jumps in to queue and buffer the requests until a replica is spawned up by the Knative Service Controller and the requests can be forwarded to that replica.

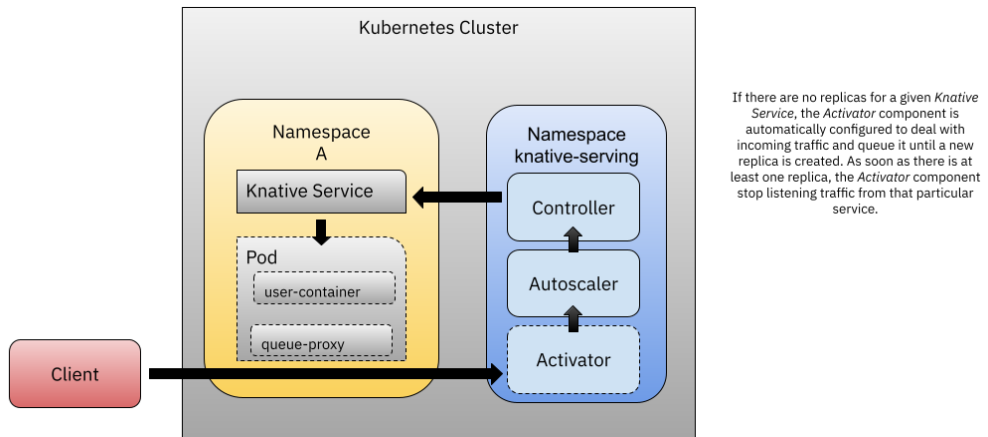


Figure 5.16 Knative Serving Activator queues traffic for services with no replicas

On top of helping us to implement different release strategies without the burden of creating tons of different Kubernetes resources, Knative Serving also allows us to scale up and down based on the concurrent requests that our services are handling at a given time. These mechanisms work out of the box and based on live traffic, so there is no need to manually change the number of replicas of a given service, Knative Serving will adapt to the demand, making sure that the amount of replicas alive can handle the current traffic and saving resources when they are not needed.

5.3 Summary

- Kubernetes built-in Deployments and Services can be used to implement rolling updates, canary release, blue-green deployments and A/B testing patterns. This allows us to manually move from one version of our service to the next without causing Service disruption.
- Canary Releases can be implemented by having a Kubernetes Service, two Deployments (one for each version) and a shared label that routes traffic to both versions at the same time
- Blue-Green deployments can be implemented by having a Kubernetes a single Kubernetes Service, two independent deployments and switching labels when the new version has been tested and it is ready to be promoted
- A/B Testing can be implemented by having two Services and two Deployments and two Ingress resources
- Knative Serving introduces an advanced networking layer that allows us to have fine-grain control about how traffic is routed to different versions of our services, that can be deployed at the same time. This feature is implemented on top of Knative Services and reduces the manual work of creating several Kubernetes resources for implementing canary releases, blue-green deployments and A/B testing release strategies. Knative Serving simplify the operational burden of moving traffic to new versions and with the help of the autoscaler can scale up and down based on demand.

6

Events for Cloud-Native Integrations

This chapter covers

- The power of Event-Driven architectures when building distributed systems
- How to produce and consume asynchronous events in a Kubernetes environment
- How events can help us to externalise what is happening in our services
- How events can be a perfect mechanism to implement integrations between disparate systems
- What tools can save you time when building Event-Driven applications on top of Kubernetes

So far, we have covered very basic service to service interactions using HTTP requests. HTTP requests are synchronous in nature, meaning that the receiving end must be up and running to be able to process the request correctly. While there are some mechanisms to retry HTTP requests if they fail, in real-life projects, this is not enough. Sometimes, you might want to understand what is going on inside your services using asynchronous mechanisms, or you might want to share information between multiple services without them knowing about each other or being up at the same time. A common way to decouple your services and implement asynchronous interactions is to use message queues. Where one service can produce messages and multiple services can be interested in consuming it in an asynchronous way.

If you start digging into message queues, sooner or later you will find out about Event-Driven architectures and how these tools can be used to build Event-based and reactive systems. In general, these architectures rely on the concept of “events.” An event can be thought of as something that has happened, which something else may be interested in. In these architectures, each service will be in charge of externalising what has happened internally by emitting events that other services can consume or react to them.

In this chapter, we will dig into how Event-Driven applications can be built on top of Kubernetes and what kind of tools can help us to ripe the benefits of more reactive

approaches. This chapter builds upon the previous chapters by adding producers and consumers of events to implement a new flow inside the Conference Platform application. The content of this chapter is divided into three main sections:

- Producers and Consumers of Events
 - Why would you use a Message Queue?
 - CloudEvents, what are they?
 - How do we work with Producers and Consumers in Kubernetes?
- Building Event-Driven applications using Knative Eventing
 - Decoupling components and relying on CloudEvents for system-to-system integrations
- Selling Tickets for a very popular event
 - Building an Event-Driven flow using CloudEvents

We will start simple, by looking at Producers and consumers of events, the basic building blocks for Event-Driven architectures, so then we can look at how these concepts can be implemented with real-life tools and frameworks on top of Kubernetes.

6.1 Producers and Consumers of events

In this section we are going to cover the underlying principles behind Event-Driven architectures, how they relate with message queues and message brokers, the need for a standard to encode our events and how all these topics map to the Kubernetes world.

If you were ever tasked with integrating multiple systems together you are probably very familiar with the concepts of Producers and Consumers of data, messages or events. If you are not, I would recommend you check the Enterprise Integration Pattern website and book for more details (<https://www.enterpriseintegrationpatterns.com/>). While producers and consumers can be used for integration purposes, they can be also used to architect more reactive applications. For this reason, I will not spend much time covering the basics, I will focus on the benefits so then we can look at how we can rely on these concepts when architecting and building our Cloud Native applications.

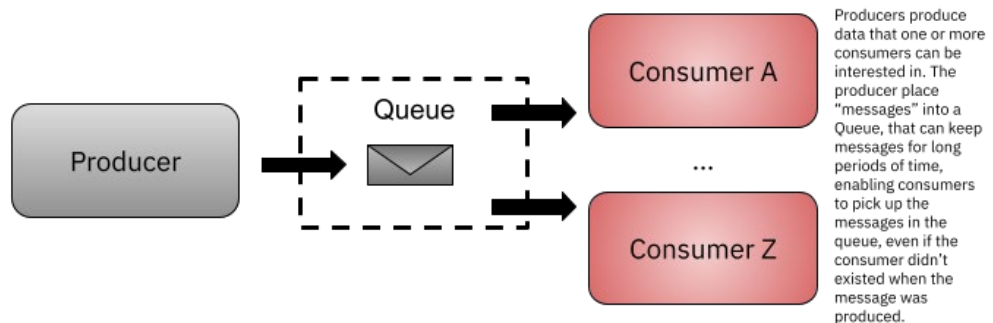


Figure 6.1 Producers and Consumers

In a typical producer/consumer scenario you will find producers producing data and placing this data into a queue of sorts. A very good analogy is how post is delivered, if you want to mail a letter to someone else, you write your letter, put it in an envelope and go to the post office or a mailbox. The mailboxes in software are the queues. The envelopes in software are called messages and usually, they represent a standard format to share information that both producers and consumers will use to exchange information. Queues in software come with guarantees, for example, queues can guarantee that your messages are delivered, as when you send a letter and expect the recipient to sign for it upon arrival. In contrast with sending letters via post, messages are usually not sent with a particular recipient in mind. We don't send messages with a destination address, we let consumers pick up the letters that they are interested in. Messages in these queues can have metadata, with more details about the information that the message is carrying. This information can be used to route these messages to consumers that have explicitly registered interest in some specific messages. Consumers on their end will need to open the envelope to consume the information contained inside the message and this will require consumers to understand the format in which the information was encoded inside the envelope.

You can have multiple producers, multiple completely isolated queues, and multiple consumers consuming messages in different ways. For example, we might be interested in making sure that every consumer consumes every message in the queue or the opposite where just only one consumer can consume a message.

Notice that I am using the term "message", what you put in the queue is completely up to you, but the queue will expect a certain structure to wrap your data. Different implementations of this queue will have different wrapper structures, which both the producers and consumers will need to understand to put messages in the queue and consume messages from it. But what makes producers and consumers really powerful is that neither the producer needs to know about the consumers (or how many they are) and the consumers don't care who is producing messages. Both producers and consumers only need to know where to go to produce or consume messages.

Let's take a look at some of the features provided by these queues:

- **Persistent messages:** once the producers place a message in the queue, the message will be persisted. The queue will acknowledge the message, so the producer can be sure that the message was persisted.
- **Guarantees of delivery and Retrying mechanism:** every time that a message wants to be consumed by a consumer, the consumer needs to acknowledge that the message was processed correctly so the queue can mark the message as consumed. If for some reason, the consumer failed to acknowledge the message, the queue allows other consumers to pick the same message for processing.
- **Consumers can asynchronously consume messages:** distributed systems tends to fail and to be able to consume messages in an asynchronous way make our system more tolerable to failure. We can rely on the queue persistence mechanism to make sure that the message is going to be there even if all consumers are failing. As soon as a consumer come back to life, it can go back to the queue and consume its messages.
- **Enable scaling up and down depending on the load of the queue:** if there is a lot of data being produced into a queue and this data needs urgent processing, we can bring up several consumers to parallelise the processing of all these messages. When the amount of messages is not a lot or when the messages don't have any urgency to be processed, we can scale down the consumers without affecting the producers or the queues.

You might be wondering now, if queues are so great, why we don't replace all the service-to-service interactions with messages and queues?

REPLACING HTTP REQUEST WITH MESSAGES

Imagine that if you are trying to replace an HTTP request with a Message, this might be because you want to decouple the producer and consumer, because they might not be up and running at the same time or because there is no need or you can't afford a synchronous call. You will be tempted to include the data transmitted in the HTTP request body into the message body, so a consumer can pick up the message, check the type of the message and then process it in the same way as a REST HTTP endpoint will process an HTTP request. When the message is processed, then you will be tempted to reply back with another message for the producer with whatever answer the REST HTTP endpoint used to provide.

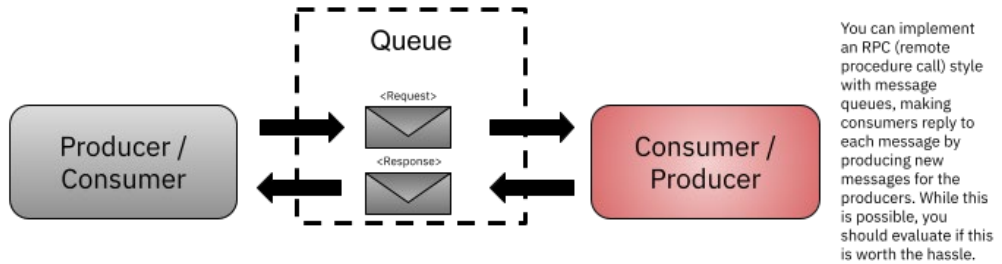


Figure 6.2 Implementing RPC-style interactions with message queues

While replicating HTTP requests with a message queue is doable, there are several drawbacks in doing so, starting with the obvious which is now every interaction will be asynchronous and debugging what is going wrong might be much more difficult. You also will be introducing a message queue to your infrastructure adding extra hops for your data to arrive at the desired service. It might be tempting to justify this change for the sake of guarantee delivery and the retry mechanism provided by message queues, but I encourage you to look into how similar mechanisms had been implemented for HTTP and other protocols like GRPC, much more suited for synchronous bidirectional interactions.

Trying to move a synchronous HTTP request-based distributed application to use message queues had proven to be a really large endeavour, hence I recommend using producers and consumers for scenarios that can benefit the most from the advantages previously described.

6.1.1 Events and Event-Driven architectures

An alternative approach that avoids implementing an RPC-style interaction with messages that has become quite popular is to encode *Events* into these messages. An Event can be defined as:

"An Event records data about something that has happened."

Instead of encoding command-like data inside our messages, we encode events and enable consumers to react to these events. This approach empowers us to build "Event-Driven architectures" as we move away from services calling each other to emitting events about what is happening inside a service, so other services can react to these events. These architectures map well to message queues as we want to make sure that those interested in the events that we are emitting know where to go and fetch them.

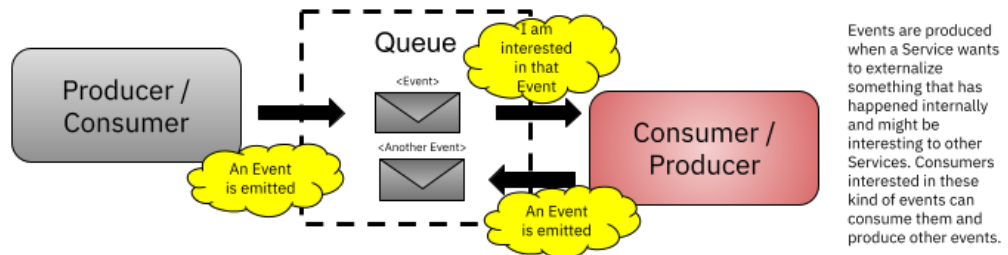


Figure 6.3 An Event-Driven approach

Services can externalize relevant changes of state of what is going on internally by emitting events. The service needs to know where to send the event to, it doesn't need to know which service will consume it, or if there is a service consuming it.

For this approach to work, it is important that we are not coupling services together when we are designing the events. In other words, we need to make sure that we are not explicitly sending an event to another Service which must reply with a specific kind of event to the producer. If we do so, we are just masking RPC-style calls into events.

At this point, you might be wondering about the technicalities of implementing an Event-Driven approach like the one described in figure 6.3, so it is time for us to go into the practical details of implementing this approach for our Cloud Native applications.

6.1.2 CloudEvents and Message Queue

Let's start this section with some high-level details about what you need to implement a scalable solution that follows an Event-Driven approach. First, you will need to choose a Message Queue system based on your requirements. Several mature implementations follow the AMQP (Advanced Message Queuing Protocol) standard, such as RabbitMQ, ActiveMQ, etc. You also have Cloud-Provider specific tools such as Google Pub/Sub, AmazonMQ (managed RabbitMQ) and Azure Message Hub.

Each implementation will have unique features, but if your implementation implements the AMQP standard, you have a base of guaranteed interfaces that you can rely on.

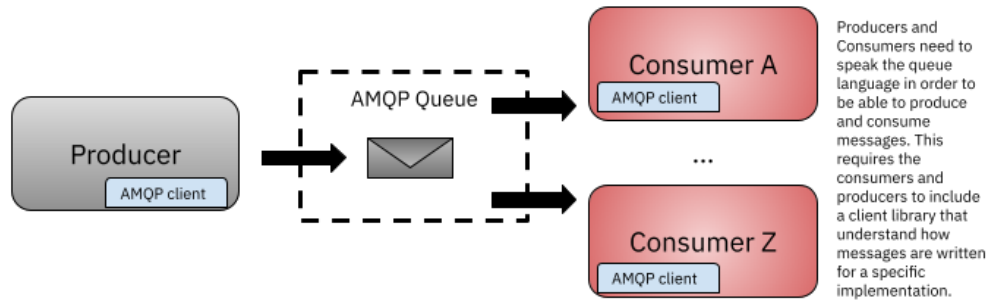


Figure 6.4 AMPQ Producers and Consumers

It is worth considering that if you chose the AMQP standard and then want to use Google Pub/Sub, for example, you are out of luck, as their managed service doesn't implement the AMQP protocol. On the other hand, if you want to use AmazonMQ you can easily move from a hosted RabbitMQ instance to the AmazonMQ managed service. Here you are faced with a trade-off between self-hosted or managed services, as if you choose Google Pub/Sub, for example, you don't need to manage it or make it scale, Google Cloud will do that for you.

Another important detail that needs to be considered early on when designing a solution like this is the Event format inside the messages. Producers and consumers read and write messages in the queue using a client library and then parse the content to understand what the event means. If you have chosen to use an AMQP based message queue you can use a generic interface to interact with the queue, but realistically, your producers and consumers will require a client library for the queue you are using. If you have chosen RabbitMQ, you will most likely need the RabbitMQ client library for your language of choice, so you can include it in both producers and consumers as shown in figure 6.4.

NOTE: Take note of these specific decisions. When selecting a message queue provider, you tie your producers and consumers to the provider by adding the client libraries into your application services. Once this decision is made, changing to another provider, for example, that doesn't support the same interfaces or standards, will require you to change every producer and consumer, probably at the same time, something that your developers will not be happy about.

When you have selected your message queue implementation, you are now faced with another question: How do I encode my events? Remember that a Message can contain information in any shape or form. You need to define the structure or shape of your events. How do producers write an event, and how do consumers read events? It is pretty common to include an "Event Type" and "Version" to encapsulate the information associated with a particular event in a different structure. The information about the source which produced it might be helpful to have as well. Because we are exchanging data between services over the network, having a unique identifier for each event might be beneficial to avoid processing the same event twice and deduplicate in case multiple delivery attempts happen.

To avoid defining what an Event is and how you can encode these events when using different protocols (HTTP, AMQP, AVRO, etc), the CloudEvents specification (<http://cloudevents.io>) was created.

WHAT ARE CLOUDEVENTS? AND WHY DO I NEED THEM?

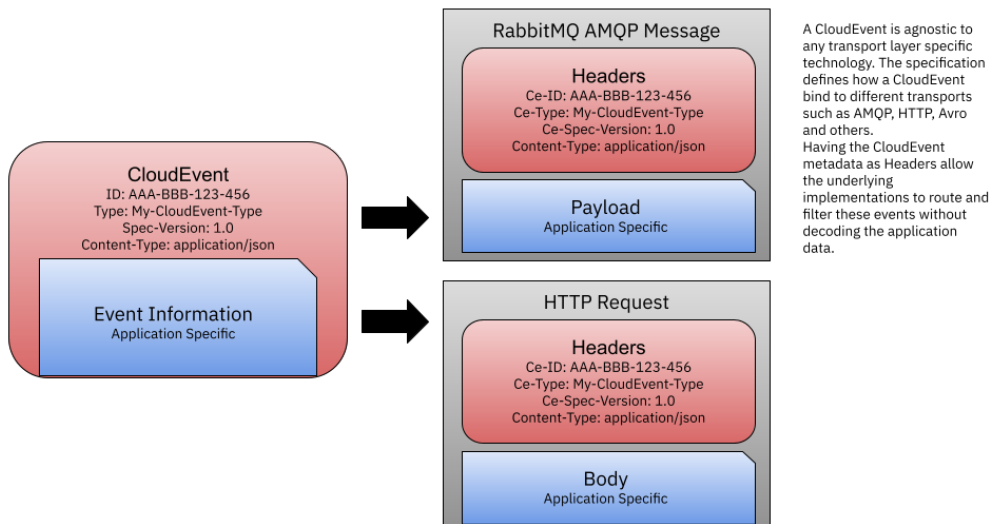


Figure 6.5 CloudEvent bindings to AMQP and HTTP

As shown in figure 6.5, CloudEvents are just a thin wrapper adding metadata about our application-specific events. But this wrapper becomes essential for us to build robust solutions that are transport agnostic. Later in this chapter, we will see CloudEvents in action. Let's now switch to our Realm: Kubernetes.

6.1.3 Producers, Consumers and Events on Kubernetes

Unfortunately, none of the concepts (abstractions) introduced in Chapter 2, such as Services, Deployments, Pods, Ingresses, are designed to work with Events. There is no concept of Deployment or Pods producing or consuming events, neither event routers nor event filters.

Services and Deployments in Kubernetes are just abstractions that we can use to compose our Cloud-Native applications. If our applications or some features follow the Event-

Driven approach, part of these interactions will be hidden away from the Kubernetes land. In other words, by using `kubectl` or a Kubernetes Dashboard, we wouldn't understand who is producing events, where the events are sent to be routed, or which component is consuming what kind of events.

Luckily for us, that is precisely the problem that Knative Eventing is solving. Chapter 5 looked at how Knative Serving built on top of native Kubernetes resources to simplify the implementation of different release strategies and provided advanced traffic routing. Knative Serving provided higher-level abstractions to manage Services and how they can be automatically scaled up and down based on demand.

In the following section, we will take a look at Knative Eventing which provides abstractions for Event Producers (Sources), Event Consumers (Sinks), Event Routers (Brokers) and Event Filters (Triggers).

6.2 Building Event-Driven Applications with Knative Eventing

Knative Eventing is independent of Knative Serving. You can install Knative Eventing even if you don't have Knative Serving installed in your cluster, but of course, your Event-Driven applications can benefit from autoscaling and advanced traffic management. As we discussed previously, Producers and Consumers can be regular Knative or Kubernetes Services, and they usually are.

Knative Eventing, as Knative Serving, builds on top of the Kubernetes API to provide abstractions for Event Consumers (sources), Event Producers (sinks), Event Routers (brokers) and Event Filters (triggers). While the natural starting points would be Consumers and Producers, we will start with Event Routers and Filters.

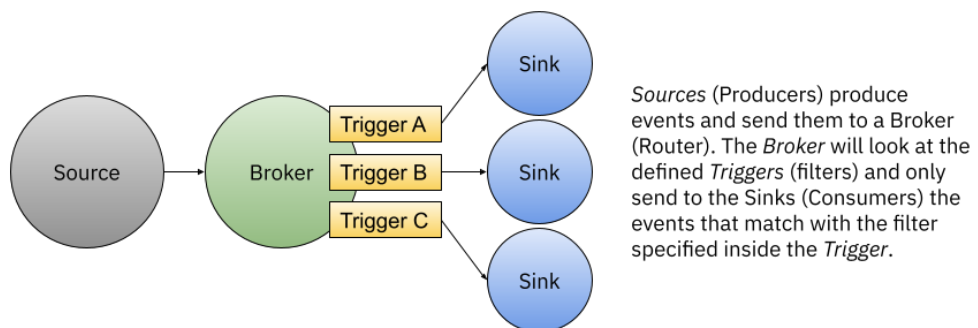


Figure 6.6 Knative Eventing Sources, Brokers, Triggers and Sinks

Both Sources and Sinks are just our applications that we have been building so far, but from the Knative Eventing perspective, they are Event Producers and Consumers.

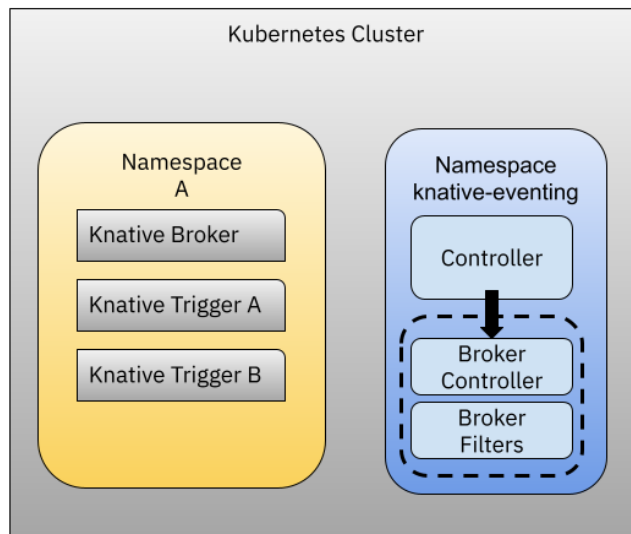
Brokers and Triggers, on the other hand, are something that we haven't had before. The primary responsibility of the Broker is to receive events from Producers and forward them to Consumers based on Trigger definitions.

Because Knative Eventing builds on top of Kubernetes you can browse these resources using `kubectl`. When you install Knative Eventing, a set of Custom Resource Definitions are installed in your cluster, as well as some controllers to manage these resources:

```
brokers.eventing.knative.dev
triggers.eventing.knative.dev
```

For Sources, there is no requirement, as Sources publish events to the Broker by just knowing the Broker URL. On the other hand, Triggers can specify Knative Services or Kubernetes Services as their target Sink, allowing Knative to check if the Consumer exists and report back.

At this point, you might be wondering, ok, but what are these Brokers and Triggers? Brokers and Triggers are just abstractions for the Kubernetes world. As with any other Kubernetes resource, you will be able to interact, add and remove these resources to wire your event-driven applications together.



When you install Knative Eventing into your Kubernetes Cluster, a new namespace is created to host the components to manage Brokers and Triggers resources that you can create in your application namespaces. The Knative Eventing Controller will be in charge of managing one or more Broker and Filter implementations to be able to route Events from Producers to Consumers.

Figure 6.7 Knative Eventing Components and Resources

As we learnt in chapter 5 with Knative Serving, when we install Knative Eventing, we are creating a new Kubernetes namespace that will host Knative Eventing components. Inside

the ``knative-eventing`` namespace we will find a set of controllers which are responsible for dealing with Brokers and Trigger resources and their respective implementations.

This approach is followed for Brokers and their implementations. Figure 6.7 shows 3 popular Knative Eventing Brokers implementations. While it is quite common to start working with the *"In-Memory Broker Implementation"* the Kafka and RabbitMQ implementations are good examples of production-grade implementations that conform with the Broker specification.

As shown in the diagram, both the Kafka Broker Implementation and the RabbitMQ Broker Implementation are just delegating the routing of Events to their respective provider, a Kafka or RabbitMQ installation.

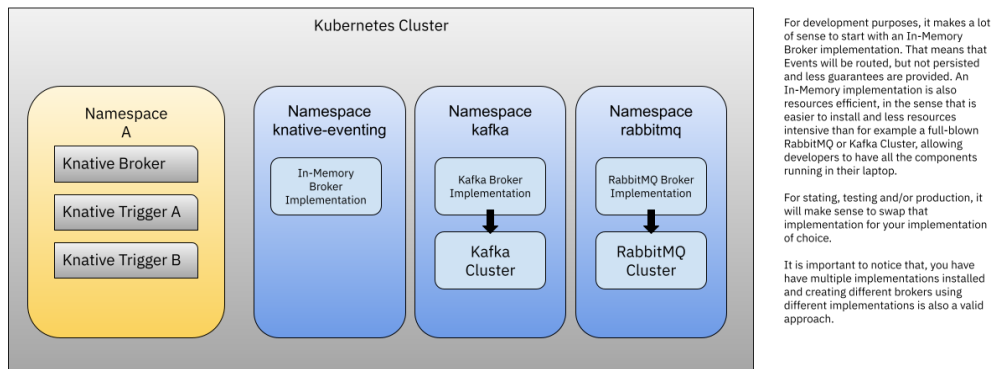


Figure 6.8 Knative Eventing Implementations installed in Kubernetes

The *"In-Memory Broker Implementation"* was created to facilitate development in local environments, allowing developers to quickly start building Event-Driven applications without having a full-blown message queue installation in their local environments. Because all these implementations are built following the Knative Eventing Specification (<https://github.com/knative/specs/tree/main/specs/eventing>), swapping the In-Memory Broker to Kafka or RabbitMQ should be transparent for all the producers and consumers.

If you already have a Kafka or RabbitMQ cluster living outside your Kubernetes Cluster or if you want to connect to a cloud provider managed service such as AmazonMQ or Google Cloud Pub/Sub the Broker Implementation will be in charge to connect with these external systems, once again not requiring any changes to be made on Producers and Consumers.

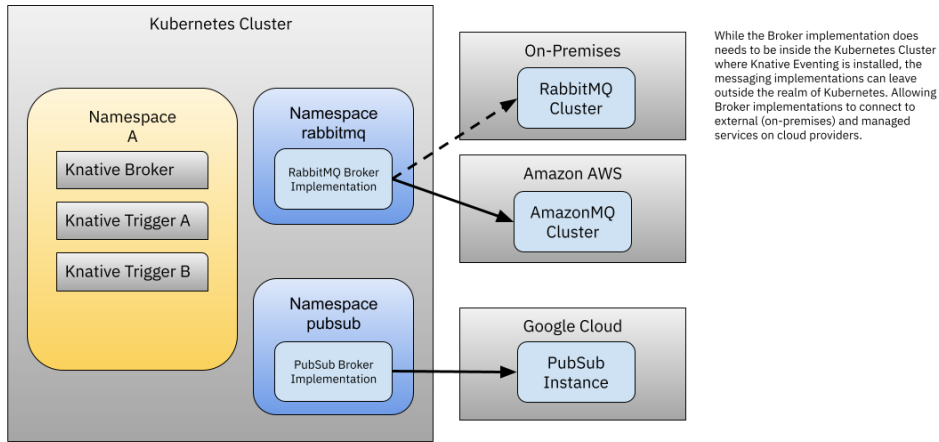


Figure 6.9 Knative Eventing externally managed Broker implementations

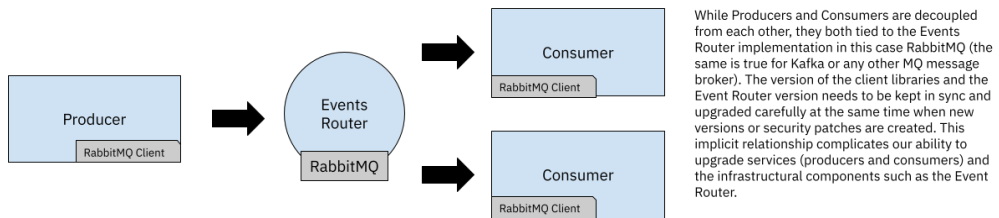
Being able to connect to managed services all implementing the same abstraction (Broker) enables our applications to be ported across Cloud Providers without the need of changing the services which consume or emit events.

As a developer, you shouldn't worry about what's behind the Broker or where the implementation is located. You just need to know that you can send an event to a Broker and based on the configured Triggers, your Events can be routed to one or multiple consumers.

PRODUCING AND CONSUMING EVENTS

As an application developer, no matter which language you are using you should be able to produce and consume events. Section 6.1.3 covers what CloudEvents are and in this section we will quickly look at how the pieces fit together to build Event-Driven applications using CloudEvents and Knative Eventing, so we can go to the final section of this chapter where the Conference Platform is extended to implement new functionality using this approach.

Because CloudEvents are transport agnostic, we can produce a CloudEvent and send it via HTTP or as a Kafka or RabbitMQ message. To do this from inside an application service we will most likely need a way to write HTTP requests, which is available in all popular programming languages or we will need a library to connect to a Kafka or RabbitMQ cluster and then write the CloudEvent using their specific protocol. The moment you add a library that is dependent on your implementation to your application services, it becomes much more painful to swap or even upgrade the implementation, because you are forced to change or upgrade the infrastructure components such as the RabbitMQ or Kafka version and all their client libraries in consumers and producers.



Wouldn't it be great to avoid adding client libraries specific to the infrastructure components that will route our events? Wouldn't it be great to leverage the fact that HTTP clients are provided in every major programming language? With Knative Eventing Brokers we are relying on an HTTP endpoint that abstracts the infrastructure for our Producers and Consumers without losing the advantages of using robust and battle-tested message brokers like Kafka, RabbitMQ or as we discussed in previous sections Cloud-Provider specific implementations.

With Knative Eventing, as soon as we know the Broker URL (HTTP endpoint) we can produce events by just sending CloudEvents via HTTP. The Knative Eventing Broker implementation will acknowledge the received event and forward it to the underlying message broker to route the event to the interested consumers. The consumer only needs to expose an HTTP endpoint that accepts a CloudEvent.

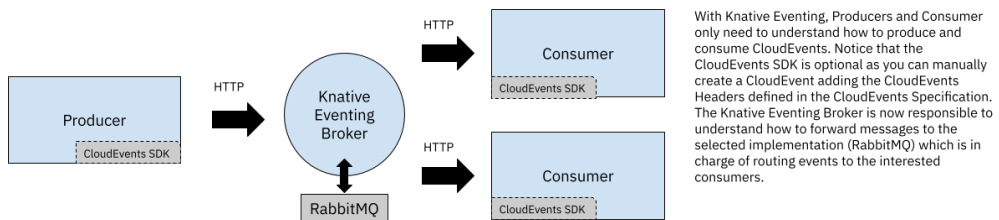


Figure 6.11 Knative Eventing Brokers and CloudEvents help us to keep our consumers and producers decoupled from the infrastructure

Now we can update Producers and Consumers without worrying about which RabbitMQ version has been configured or even knowing that RabbitMQ is being used behind the covers. While the CloudEvents SDK shown in figure 6.11 is completely optional, practically it will save you a lot of time as it provides the helpers to read and write CloudEvents in most popular programming languages, saving you time on silly mistakes around how to set up the right headers or which content type is used to serialize the body of the request.

Check the CloudEvents website to see if there is a CloudEvents SDK available for your language of choice (<http://cloudevents.io>), the following languages are currently supported: *Go, JavaScript, Java, C#, Ruby, PHP, Python, Rust and Powershell.*

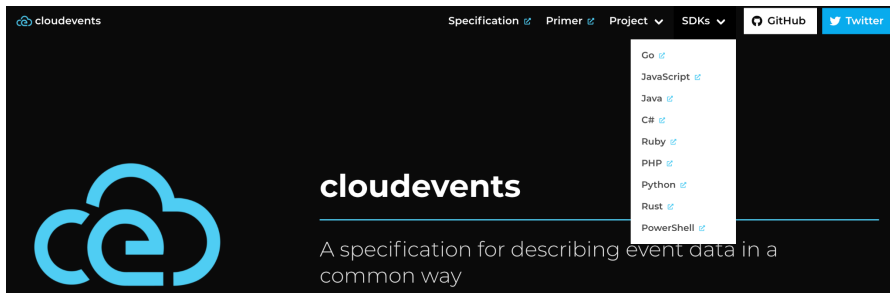
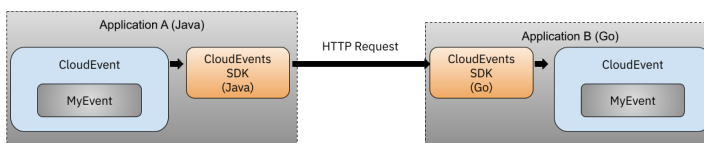


Figure 6.12 CloudEvents supported SDKs

With CloudEvents and Knative Eventing Brokers we promote a common and standard format to exchange events relying on HTTP requests which simplify the cognitive load from developers to produce and consume events. In the next section, we will quickly look at how this looks in practice before jumping into a full-blown example on the Conference Platform application.

If you are using Kubernetes, you should fully leverage the possibility of building applications using different tech stacks and different programming languages. By adopting CloudEvents as the way of describing events and HTTP as the way of serializing these events over the wire until they reach Knative Eventing heavily simplify and enable developers to build both, consumers and producers of events.

The following example shows a producer written Java using the Spring Boot Reactive framework and a consumer in Go. Both examples rely on the CloudEvents SDK (<https://cloudevents.io/>) which facilitates the parsing and serialization of CloudEvents. Using the CloudEvents SDKs reduce the time and the overhead of creating CloudEvents that conforms to the CloudEvents specification and guarantees that the CloudEvents can be read and written from applications using different programming languages.



CloudEvents are used to encapsulate your domain specific events. Think about CloudEvents as an envelope for your event data types. This diagram shows a Java Application creating an object called **MyEvent** which will be wrapped as a **CloudEvent** and sent using an HTTP request to **Application B**. Because **Application A** is written in Java, it can use the **CloudEvents Java SDK** to encode the **CloudEvent** as an HTTP Request. The **MyEvent** object in Java needs to be serialized to be sent over HTTP, you can choose to use **JSON** as a way to encode it. **Application B** will receive an HTTP request and it can use the **Go CloudEvent SDK** to parse the **CloudEvent** and read the contents, probably into a Go struct.

Figure 6.13 Produce in Java and consume in Go

Both applications share a set of characteristics:

- Both expose HTTP endpoints on a given port that can be configured setting an environment variable (`SERVER_PORT`)

- Both can consume and produce a CloudEvent
 - To produce a CloudEvent a POST request can be sent to `/produce``
 - To consume a CloudEvent a POST request can be sent to ``/``
- Both applications can configure the sink where the CloudEvent produced will be sent by exporting the `SINK` environment variable
- The CloudEvent payload produced by both application is a JSON payload
- Both have a data structure to deal with the CloudEvent data payload (`MyCloudEventData` java class and `MyCloudEventData` struct in Go)
- Both applications are also provided as Docker containers (`salaboy/fmtok8s-java-cloudevents` and `salaboy/fmtok8s-go-cloudevents`)

You can follow a step by step tutorial located in <https://github.com/salaboy/from-monolith-to-k8s/tree/master/cloudevents> to get this applications up and running in your local environment or with Knative Eventing.

The step-by-step tutorial guide you to start both applications using Docker and a Docker Network to make sure that both containers can talk to each other. I do strongly recommend you to go through this tutorial and experiment with these two applications. There is also a blog post linked in the tutorial which goes over some internal details of both applications, explaining in detail how different programming languages and frameworks compare between each other: <https://salaboy.com/2022/01/29/event-driven-applications-with-cloudevents-on-kubernetes/>

In the following section we will focus on how this simple example can start leveraging Knative Eventing without making any changes into the example applications source code.

So far, we have two services, both producing and consuming CloudEvents. But having services sending events directly to each other doesn't fit very well with the Event-Driven approach. We want our services to emit events so other services can react to them. To achieve this kind of interaction we need event routers and a way for services to register their interest in certain kinds of events.

Once your Producers and Consumers are ready to be shipped, they will need to be containerized to be deployed on Kubernetes. As we learnt in chapter 2, you will probably need a Deployment and a Service to deploy these services in Kubernetes or you can use a Knative Service as we learned in Chapter 5.

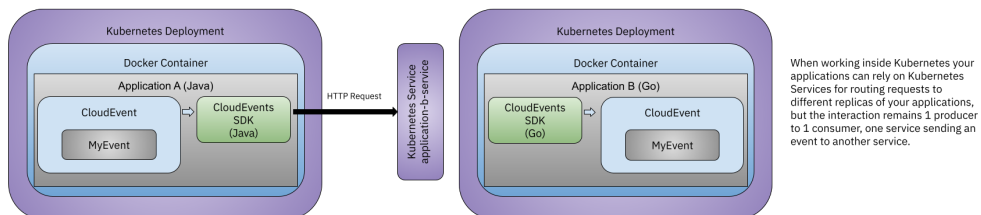


Figure 6.14 Applications running inside Kubernetes

No matter which option you use, having producers and consumers sending events directly to each other is not the way to go as this create tight coupling between these services and doesn't allow other Services to react to what is happening. What we want to achieve is to be able to produce events without knowing how is going to consume them and then have any numbers of consumers registering interest in these events. As covered in section 6.1, we can achieve this kind of patterns using a Message Queue like for example RabbitMQ.

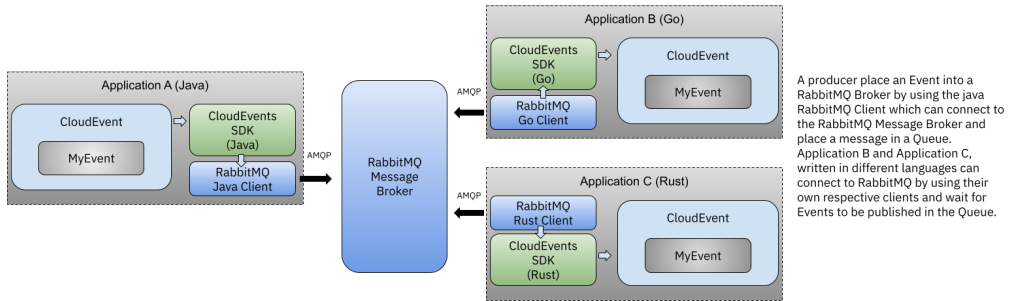


Figure 6.15 Using a message broker to implement an Event-Driven approach

Wouldn't be great to achieve the same without changing our applications code or dependencies? Let's take a look at how Knative Eventing can help us.

With Knative Eventing we keep using HTTP Requests and we don't need to change our producers or consumers.

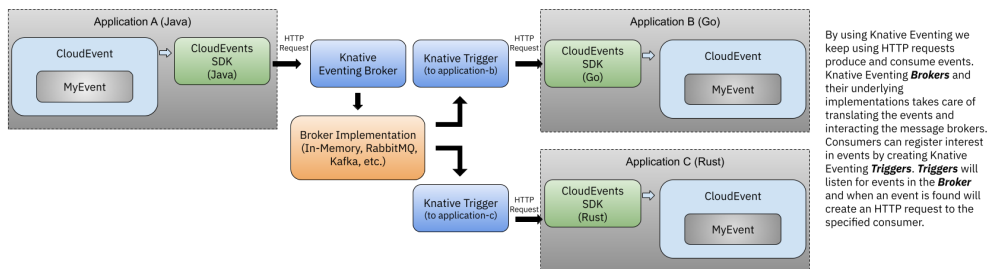


Figure 6.16 Knative Eventing Brokers and triggers

First, you will need to create a Broker, which when started will provide you with an URL where you need to point your producer, and then you need to create a Trigger resource that will define which events will be forwarded to which consumers.

Both Knative Eventing Brokers and Triggers are Kubernetes resources that you can create using `kubectl` or any tool that you want to use to sync Kubernetes resources into your clusters.

```
A Broker resource will look like this:
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  name: default
  namespace: default
```

You can create as many brokers as you want for your application to use. As soon as you create a Broker you can check that the Broker is ready to accept events and that it provides an URL for producers to send events to. Use `kubectl get brokers` to check the available Brokers:

```
salaboy> kubectl get broker
NAME      URL
AGE      READY  REASON
default   http://broker-ingress.knative-eventing.svc.cluster.local/default/default  2s
True
```

If we now configure our producer (application-a) to send events to the Broker URL, the events will be ready to be consumed from the Broker and we can wire consumers to the broker by creating Knative Eventing Triggers which will look as follow:

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: app-b-trigger
  namespace: default
spec:
  broker: default
  subscriber:
    ref:
      apiVersion: v1
      kind: Service
      name: application-b-service
```

A Trigger responsibility is to register interest in events that are published in a Broker. This example trigger is registering interest in all events that are published by producers to the Broker's URL. All the events that are published to the broker by producers will be forward to the `application-b-service` Kubernetes Service as specified in the `subscriber.ref` field.

If we are not interested in all events that are going to the Broker we can specify filters, for example:

```

apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: app-b-trigger
  namespace: default
spec:
  broker: default
  filter:
    attributes:
      type: app-a.MyCloudEvent
  subscriber:
    ref:
      apiVersion: v1
      kind: Service
      name: application-b-service

```

This trigger will only forward CloudEvents with the type matching `app-a.MyCloudEvents` to the `application-b-service` Kubernetes Service.

As shown in Figure 6.15, we can create more Triggers to have the broker forwarding to multiple consumers the CloudEvents that it receives.

By using Brokers and Triggers we have moved from a “1 producer to 1 consumer” communication to 1 producer and multiple consumers approach. We can wire consumers and producers in a declarative way using Kubernetes resources allowing developers to use HTTP request to produce and consume events.

You can get these services, broker and triggers installed in your Kubernetes cluster by following the last section of the step-by-step tutorial that you can find here: <https://github.com/salaboy/from-monolith-to-k8s/blob/master/cloudevents/README.md#with-knative-eventing>

In the final section of this chapter we will use Knative Eventing to implement a more complex example on top of the Conference Platform application. The example is way more complex, but it relies on the same patterns and using the same approach as this simple example has covered.

6.3 Selling Tickets for a trendy event

In this section, we will use Knative Eventing concepts to build a Ticket Selling feature inside our Conference Platform. Instead of changing the services that we have covered in previous chapters, we will introduce a new set of services designed using an “Event-Driven” approach.

Let’s look at the scenario we will be implementing to see how Knative Eventing and CloudEvents can help us build it.

DEALING WITH HIGH DEMAND

For this example, we will be simulating a ticket selling queue mechanism that needs to be implemented to deal with the demand of potential customers who want to get hold of a limited number of tickets. This is quite a typical scenario for web portals in charge of selling concerts, where the sale begins at a particular point in time, and the number of tickets available is limited.

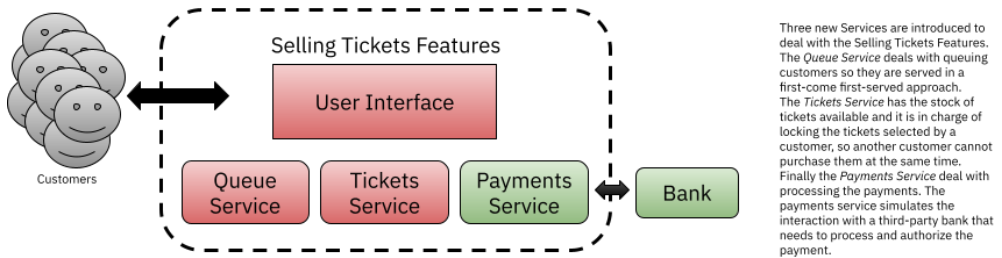


Figure 6.17 Selling limited tickets under high demand

The normal flow to buy tickets requires the customers to queue so they are on a first-come-first-served basis. When there is too much demand, this will allow other services to handle a manageable amount of concurrent users. Once the customer exits the queue, tickets need to be selected and the platform will lock these tickets for the customer to wait for the payment to be processed. If the payment is processed correctly, the tickets are removed from the Tickets Service. If the payment doesn't go through or a timeout is reached, the tickets are released for another customer to buy. For this scenario, the Payments Service is simulating a 3rd party system, which is not under our control. The intention behind this decision is to demonstrate how a system that wasn't designed to operate in an Event-Driven approach can be wrapped around using events for integration purposes.

Each application's service is designed to emit events to externalize what is going on. Each service also is designed to consume the events that it is interested in, allowing the service to react when a meaningful event pops up.

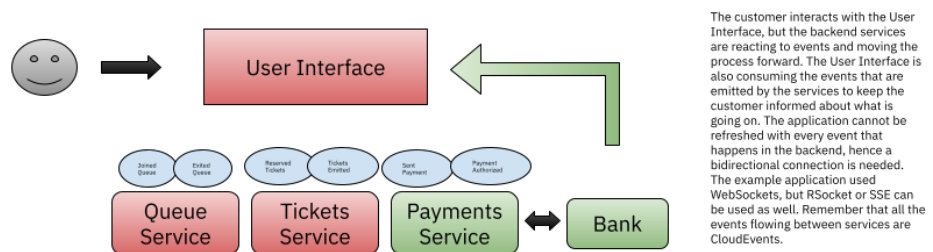


Figure 6.18 Services reacting to different events and keeping the User Interface up to date

This is more complex example than the ones that we covered in section 6.2, but it uses the same building blocks as before. For this reason, the following sections explore the particular challenges of building Event-Driven applications on top of Kubernetes and how by following an Event-Driven approach you can integrate with external and legacy systems while

maintaining services independent to each other, allowing teams to keep evolving them independently.

You can run this scenario in your own Kubernetes Cluster with Knative Eventing, a set by step tutorial on how to achieve this can be found here: <https://github.com/salaboy/from-monolith-to-k8s/blob/master/knative/knative-eventing-example.md>

Notice that the example can be installed with an in-memory Knative Broker implementation, Kafka or RabbitMQ Brokers, which allows you to test different tech stacks depending on your requirements.

USE CASE STEP BY STEP

If you follow the tutorial, the Conference Platform will be extended with new capabilities that allow the users to buy tickets for the Conference events when the platform administrator set the tickets on sale. The following figure shows the new behaviour implemented in the application for selling tickets.

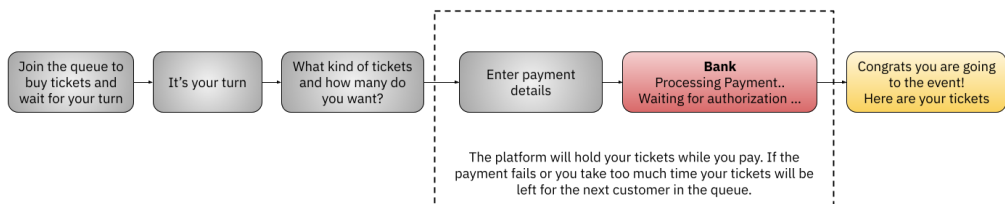


Figure 6.19 Buying tickets steps for every customer

As you can see, this is a very simple flow which follows a very well-known and familiar process for buying tickets online, where there might be a large number of people trying to buy them at the same time. Hence the first step that you are faced with is a queue.

The following screenshots show the same process but implemented into a sequence of pages that each customer will need to go through to buy tickets.

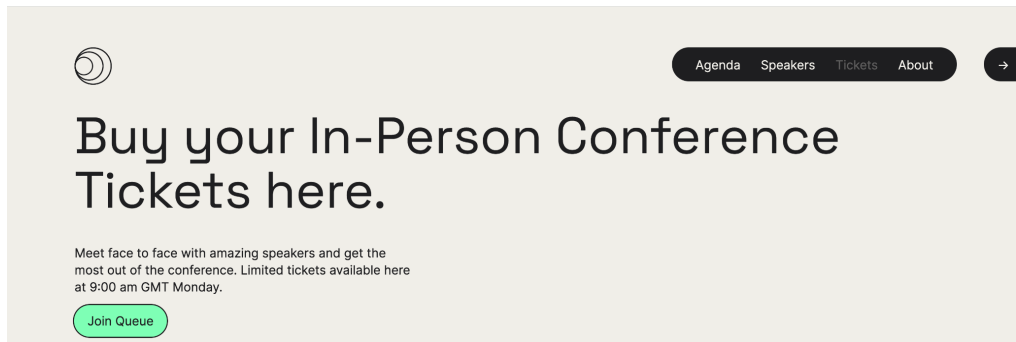


Figure 6.20 Conference Platform Tickets Page

All the features for selling tickets are under the new Tickets section that can be found in the top menu of the Conference Platform.

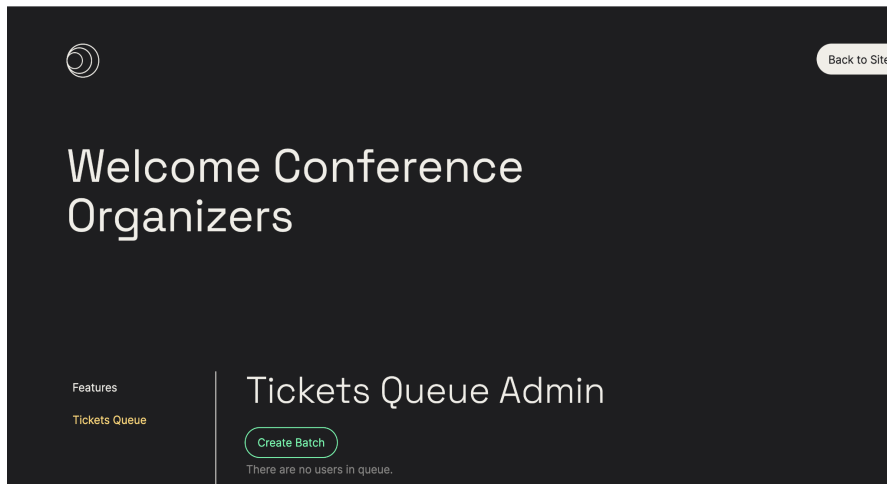


Figure 6.21 Conference Platform Tickets Backoffice options

There is also a backoffice section to simulate other people queuing up and making the tickets available for customers.

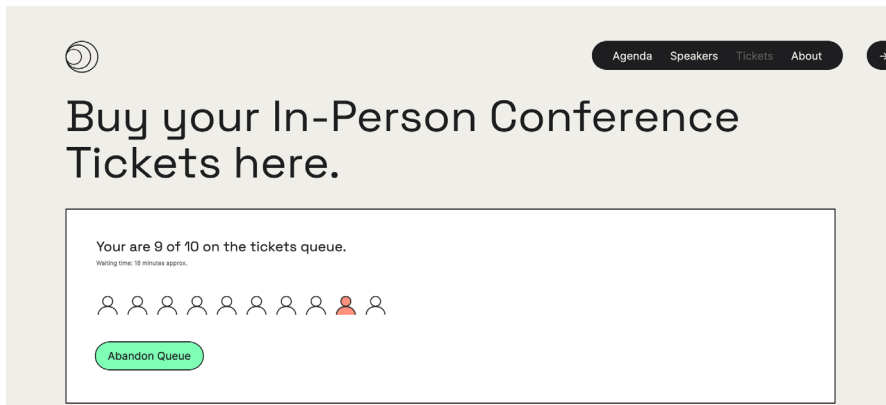


Figure 6.22 Conference Platform queuing for tickets

As a Customer, you will need to wait for your turn to be able to proceed. If you refresh the page or go to a different section, you will need to join the queue again, leaving your place for the next person in the queue.

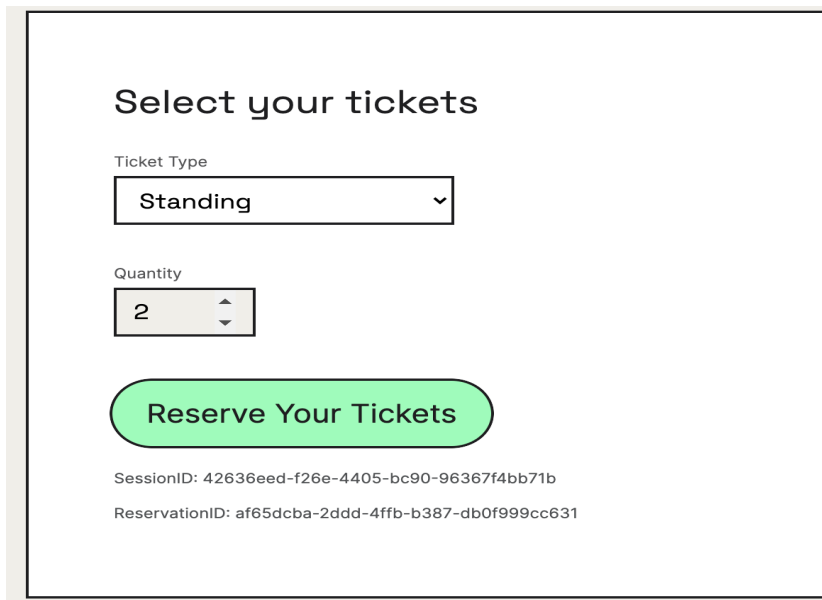


Figure 6.23 Reserving tickets until you complete the payment

Once you are out of the queue, you will be able to reserve the kind of tickets that you are interested in and the amount. At this point, the platform will hold the tickets for you until you

finished paying for them. If you take too much time or if you abandon the page, the tickets that you reserved will be automatically released allowing other customers to buy them.

Enter your payment information

Card Number

1111

2222

3333

4444

Expiration Date




12

2023

Security Code

210

Pay

SessionID: 42636eed-f26e-4405-bc90-96367f4bb71b

ReservationID: af65dcba-2ddd-4ffb-b387-dbof999cc631

Figure 6.24 Using a Credit Card to pay for your tickets

After you reserve your tickets, you will need to follow a very standard procedure to pay for them. In this case, a Credit Card payment is simulated and a 3rd party/external interaction with a Bank is performed to authorize the payment.

Validate your payment

[YOUR BANK](#)

...

Processing your payment

Please wait a minute. Don't leave this page

Session ID: 42636eed-f26e-4405-bc90-96367f4bb71b

Reservation ID: af65dcba-2ddd-4ffb-b387-dbof999cc631

Figure 6.25 The payment will be processed by an external entity, for example your bank

Once your payment is authorized the platform will provide you with your tickets for the conference.

The following section covers some of the challenges related to implementing this functionality plus the Knative Eventing resources that you need to get all the services reacting to the correct events.

CHALLENGES & SOLUTIONS WHEN BUILDING EVENT-DRIVEN APPLICATIONS

- User Interface interactions and routing CloudEvents via WebSockets, this needs to be done in the API Gateway, how.. Expose endpoint and open a websocket connection
- The wiring happens in Knative Eventing Triggers, filters are needed to make sure that each service receives the events that they are interested in
- Dealing with demand, scaling up event processors
- Integrating with 3rd party or legacy systems that don't produce events, building a bridge
- Using Events to monitor or augment existing systems/services

6.4 Summary

- Kubernetes built-in mechanisms doesn't take into account event consumers, producers or routers.
- Event-Driven architectures allow you to build more resilient and decoupled services that can rely on message delivery guarantees provided by message queues
- Building event-driven applications using different technology stacks is possible thanks to the CloudEvents specification (<http://cloudevents.io>) which provide us with a standard format to share events across different systems. The CloudEvents SDKs allow developers to read and write CloudEvents using different programming languages
- Knative Eventing brings to Kubernetes the concepts of event producers, consumers and routers. It allows developers to build their event-driven services and applications without the need to define which underlying technology will be used to route events. Popular broker (event routers) implementations are supported such as In-Memory for development, RabbitMQ and Kafka for production workloads and Cloud-Provider specific managed services such as Google Pub/Sub is also supported.

7

Functions for Kubernetes

This chapter covers

- How to improve your developer's productivity and experience by using a function as a service (faas) approach
- How functions can speed up your development teams while providing a simple programming model. We will be looking at a project called Knative Functions (``func``).
- How to leverage the polyglot nature of Kubernetes to write functions in different languages reusing the same tools, concepts and standards to share information
- How to connect multiple functions together to build complex scenarios for our walking skeleton

We have been talking about Kubernetes and tools that can be used to solve specific challenges when building an application on top of Kubernetes, but in this chapter, we will be focusing on Developers. To enable developers to focus on writing the applications that the business needs, we need to make sure that they have a programming model that doesn't push them to learn how things are built, containerized and deployed on top of Kubernetes.

In this chapter we will be looking at a Knative tool that is called ``func`` (<http://github.com/knative-sandbox/kn-plugin-func>) which uses all that we have learnt in the book so far to provide a simplified Kubernetes-less and Dockerfile-less polyglot experience. ``func`` works by providing a developer experience that enables developers to focus on their application/functions code instead of writing YAMLS or worrying about building containers.

This chapter is divided into three main sections

- Function or Service? What's the difference?
- Getting closer to developers using the ``func`` CLI or IDE plugins
 - Let's create our first function, what to expect and function lifecycle

- Building a highly scalable game for our Conference Event Keynotes

Let's begin with that first question: Functions or Services?

7.1 Functions or Services? What's the difference?

So far, we have been talking about Services to describe the applications that we are running inside Kubernetes. Each of these services usually maps to a single container that can be scaled up by having multiple replicas of the same container running at the same time. We have also seen that Kubernetes uses the same nomenclature, as the Kubernetes Service resource helps us to provide a single entry point for any number of replicas of our containers that are running.

We have also seen the number of steps required to get a Kubernetes Ingress, a Kubernetes Service, a Kubernetes Deployment and a container image built and ready in a container registry to finally have the service up and running in an environment where a user or other services can interact with it.

How does Kubernetes compare with popular Serverless offerings? Do you need to choose between Kubernetes and a Serverless platform such as AWS Lambdas?

In this section, we will talk about Function as a Service (FaaS) platforms and how they compare with the approaches that we have covered so far. We will compare their advantages and drawbacks with the Kubernetes approach and in section 7.2 we will see how we can provide a FaaS experience on top of Kubernetes to have the best of both worlds.

7.1.1 Functions, and Functions as a Service

Functions as a Service (FaaS) platforms, also known as Serverless platforms, aim to remove all the hassle of dealing with infrastructure or even building containers. In this chapter, the term Serverless is going to be used interchangeably with FaaS. Functions-as-a-Service platforms (FaaS) are built with the idea that functions are simple constructs that perform a single task, have one input and produce an output. Let's start with a quick introduction to what functions are to understand how platforms based on functions are supposed to work.

WHAT IS A FUNCTION?

Functions are very scoped and usually small applications that are in charge of performing a single task. To perform this task, the function receives an Input and creates an Output that can be used by another function. Inputs and outputs come usually in the shape of events. Meaning that functions react to events that happen and produce events when they are done.

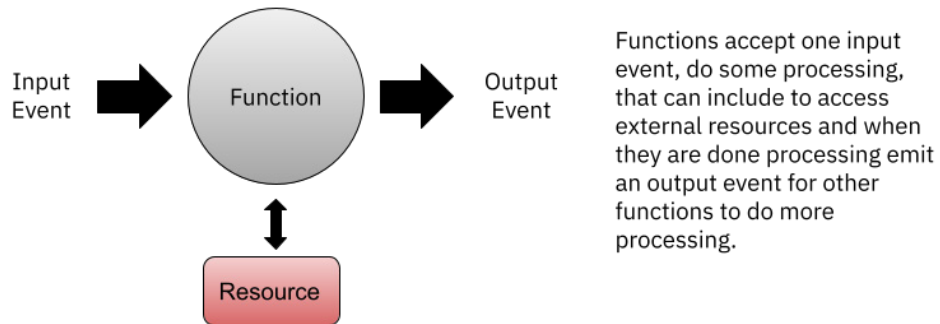
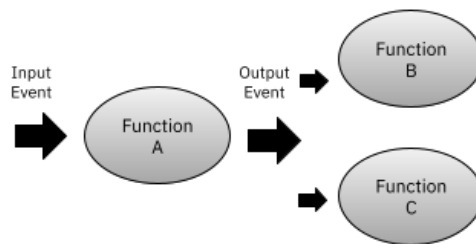


Figure 7.1 Function's inputs and outputs

Functions can be chained together by just tapping into previous function output events. The idea here is that developers shouldn't worry about where the events are coming from or where the output event is placed, this is all handled by the platform as well. This reduces the cognitive load from developers and enables the platform to provide a simplified user experience.



Multiple functions can be interested in a specific event, and the platform is responsible of routing these events to the right functions. This allows application developers to just write their functions and not worry about how to consume an event or where to place the output events.

Figure 7.2 Channing multiple functions based on events

Functions by themselves are really useful constructs, but they cannot work on their own. To be able to work with Functions you need to have a platform that takes care of running these functions, scaling them and routing events to the right functions so they can perform their tasks.

THE NEED FOR FUNCTION-BASED PLATFORMS

If you are a developer tasked to write a set of functions to perform some very domain-specific tasks, you don't want to worry about how these functions will be built and deployed, how they will be scaled up or down, or how to route events from one function to the next.

Functions can be elastically scaled up and down based on demand, but the scaling will be performed automatically by the platform without any user intervention, reducing the amount of maintenance of your runtime environments. Less maintenance also means a more complex platform that will take care of things like scaling for us. It also means that the behaviour that we can include inside our functions is in some way limited to respect the function boundaries.

In FaaS platforms it is quite common to talk about cold starts, as functions can be downscaled to zero when a new event arrives, the platform and the end-user will need to wait until the function bootstrap to be able to process the incoming event. If you have dozens or hundreds of functions, this latency can add up. Also because we are dealing with events, we are in a very asynchronous environment.

But as a function developer, you shouldn't worry about:

- (How to build the function code into a runnable function)
- How to consume an event, the platform will give you a way to register interest in a certain type of event.
- How to produce an event, you just produce it and the platform will be in charge of routing the event to the interested parties
- Being restricted by a single programming language, you can write functions in any language you want as soon as the platform supports it
- Knowing which other functions are running at a given time, since each function is independent of others
- How many replicas you need to support ever-changing demands, as the platform will take care of it

A very prominent FaaS (Serverless) and a very good example of such a platform is AWS Lambda, which provides a polyglot environment to write functions in different languages. But this obviously doesn't run inside Kubernetes and is vendor-specific.

There are some preconceptions as to how functions are different from Services the following table goes into these differences, but as we will see later on we can blur these differences in the Kubernetes world

Functions	Services
New replicas are created when they are needed, based on demand	A Service is started and it runs until it is manually downscaled. If you need to scale up the Service you need to do it manually or configure an autoscaler to perform this scaling when certain criteria are met.
Functions need to start fast, as there will be requests waiting for a new instance to start	A Service can take a bit longer to start, because it is manually started by an operator. The operator can decide when the Service is up and ready to be used.
Functions downscale to zero if there is nobody using them. The FaaS platform will have a way to cache requests until a new instance of the target function is scaled up and ready to process the request	Services needs to be manually downscaled if there is no demand. By default, a Service cannot be downscaled to zero, as incoming requests will fail if there is no instance to process them.
Functions are focused on solving a very specific problem, usually receiving an event as input and producing an event as output	Services can host multiple endpoints to perform different tasks. Services are considered larger in scope

So how can we get closer to Kubernetes and also rip all the benefits from a FaaS Platform?

Let's take a quick look at a new breed of Platforms called Containers as a Service (CaaS).

7.1.2 Containers as a Service Platforms

With the rise of containers, it makes a lot of sense to build platforms that can take any container as input and run it, manage it and scale it for you. These platforms aim to leverage the FaaS platform approach to enable developers to worry about their code and not how to run or scale their containers.

A prominent example of this platform is Google Cloud Run, which will take care of building and deploying your containers for you:

- Java: <https://cloud.google.com/run/docs/quickstarts/build-and-deploy/java>
- Go: <https://cloud.google.com/run/docs/quickstarts/build-and-deploy/go>

AWS Lambda is also supporting Containers. Both Google Cloud Run and AWS Lambdas are vendor-specific tools, but at least both offerings accept pre-built containers.

If we are talking about containers, then running and orchestrating them with Kubernetes makes a lot of sense. With Knative we can build a Container as a Service Platform for our developers which leverage autoscaling, scaling to zero and the Knative Eventing model.

The main missing bits, even with Knative, we still need to create a Knative Service resource using YAML files and we need to make sure that Knative can access a container

image that someone needs to build, hence developers will need to create some kind of Dockerfile to containerize their applications.

Knative is providing all the building blocks to build these developers' experiences and that's why the Knative community has been working on a project called ``func`` which focuses on enabling a FaaS and CaaS experience on top of your Kubernetes Clusters. We'll consider this in more detail in the next section.

7.2 Getting Closer to Developers with Knative ``func``

In this section, we will be looking at Knative Functions (``func``) a project that was created to enable developers to use a function-based approach to build applications on top of Knative. One of the main design objectives behind Knative Functions is to simplify the developer experience by enabling developers to write functions without the need of thinking about containers or even Kubernetes.

The main idea behind looking at how ``func`` was designed and implemented is to highlight the need for building specialized developer experiences that hide the complexity of the technology stack used to run the software that they are writing. ``func`` is focused on allowing developers to choose their own technology stack to write functions and use CloudEvents as the input and output for these functions.

By using functions as the main building block, ``func`` allows you to quickly scaffold a project with an empty function template that consumes a CloudEvent and produce a CloudEvent as result.

In the following sections, we will be:

- Creating a project using the ``func`` CLI
- Building the project
- Running the project locally
- Deploying the function into a Kubernetes Cluster

Let's get started.

7.2.1 Creating a Project with ``func``

Once you created a ``func`` project, ``func`` takes care of the lifecycle of the project by providing commands to *build*, *run* and *deploy* the project into a Kubernetes cluster that needs to have Knative installed.

You can install ``func`` and create a project with the following command in an empty directory, let's call it ``fmtok8s-java-function``:

```
func create fmtok8s-java-function -l springboot -t cloudevents
```

The ``-l`` parameter specifies the tech stack that you want to use, in this case, is Java using the Spring Boot framework and Maven as the build tool. The ``-t`` parameters specifies which function template we want to use, in this case, a function that works with CloudEvents.

I encourage you to try ``-l go`` to create a similar function using Golang.

This command creates a very simple function that echos the CloudEvent that receives as input. If you take a look at the files generated inside the ``fmtok8s-java-function`` directory you will find a very typical Spring Boot Project:

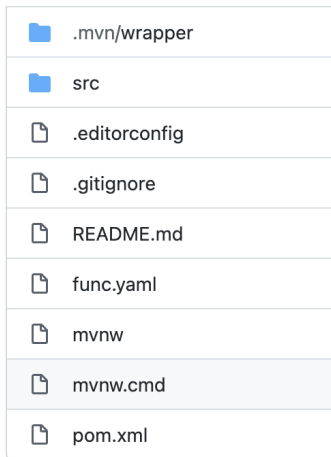


Figure 7.3 ``func`` scaffolded project

Next, let's build our function

7.2.2 Building our function

Besides the ``func.yaml`` file there are no other YAML files or Dockerfiles in the project. Disregarding the lack of YAML files or Dockerfiles, we can still run the following command to create a container for this ``func`` project:

```
func build
```

You should see the following output:

```
A registry for Function images is required. For example, 'docker.io/tigerteam'.
? Registry for Function images: salaboy #1
Note: building a Function the first time will take longer than subsequent builds #2
[+] Function image built: docker.io/salaboy/fmtok8s-java-function:latest #3
```

#1 ``func`` needs to know where to push the resulting image, if you have a Docker Hub account you can enter your user name here, and if you have already used ``docker login`` to authenticate with your account, it will use those credentials to upload the function container to your Docker Hub account.

#2 As ``func`` needs to fetch all the project dependencies, the first build will take a while, but subsequent builds will be faster, as ``func`` will catch these dependencies for future builds. The next time that you run ``func build`` it will check for changes and only download new dependencies if the ``pom.xml`` file has changed and recompile the source code, if there were any changes. The amount of time that this first build takes usually depends on your internet connection.

#3 Finally, the function is built, tagged and ready to use

You might be wondering, how is it possible to create a container without having a Dockerfile? This is the first step to make developers' life easier. `func` uses CNCF Buildpacks (<http://buildpacks.io>) to build and containerize your `func` projects. Buildpacks detect the type of project that we are trying to build and choose predefined templates to build and create a container for our application without pushing developers to write a Dockerfile.

If you have written a Dockerfile before you probably know how easy is to copy one from the internet to get you going, but you will be making some decisions that are not so obvious such as base image, tools included in the final container, versions of libraries that can have security issues or outdated, etc. By using buildpacks, we decouple these decisions that might impact the resultant container from developers who are interested in writing the logic inside the functions. We can then choose sensible and secure buildpacks for our `func` projects to use.

7.2.3 Running our function locally

As the next step, you can use `func run` to run the container that was created using `func build`. This will start a local instance of the container for you to interact with, this is usually faster than running the container in a remote cluster and can help developers to troubleshoot issues locally.

```
func run
```

Should show the function starting in port 8080, which allow you to send CloudEvents to it and see the responses locally. You can send a CloudEvent using `curl` or `http` a more friendly command-line too for creating requests. You can also use `func invoke` which was designed to submit CloudEvents without the hassle of creating an HTTP from scratch.

```
func invoke <LOCAL URL>
```

7.2.4 Deploying our function to a Kubernetes Cluster

Finally if you want to deploy your `func` project to your Kubernetes cluster you can run `func deploy`, this will actually push the container image to the configured registry, in this case Docker Hub. As you might notice, there wasn't a single YAML file written to do this deployment:

```
func deploy
  ☐ Function image built: docker.io/salaboy/fmtok8s-java-function:latest
  ☐ Pushing function image to the registry
```

Now that the `func` project is deployed and we have a public URL we can interact with the function that is running in our Kubernetes Cluster.

```
func invoke <REMOTE URL>
```

So far, we have gone from creating a ``func`` project, building it, running it locally where we tested that it worked as expected by sending a CloudEvent and receiving one in response and finally, we have deployed our ``func`` into a Kubernetes Cluster that has Knative in it, allowing us to interact with our function using a public URL which was automatically exposed without developers writing YAML files.

Finally, you can run ``func delete`` if you want to undeploy the function from the remote cluster.

7.2.5 Developer tooling is a must

While using the command-line tool shouldn't be something new if you are already used to use ``kubectl`` to interact with your Kubernetes clusters having some specialized tools integrated with your day to day IDEs definitely help. Luckily, if you are using VSCode, IntelliJ or GoLand you can get access to the Knative and Knative Functions plugin. This allows you to build, run, deploy and undeploy ``func`` projects right from your IDE. You can also see which other functions are deployed.

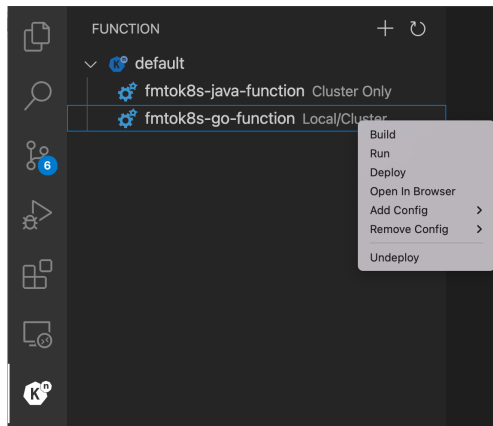


Figure 7.4 Knative ``func`` plugin for VS Code

With the VSCode Knative Plugin you can access to functions and run them locally or deploying them in a Cluster. If you open a ``func`` project, the plugin will automatically recognize which function correspond to the local ``func.yaml`` file and enable the options for you. You can code and run your functions from the same place. VSCode was designed to be extended with different plugins for different languages make it perfect for writing NodeJS or Python functions.

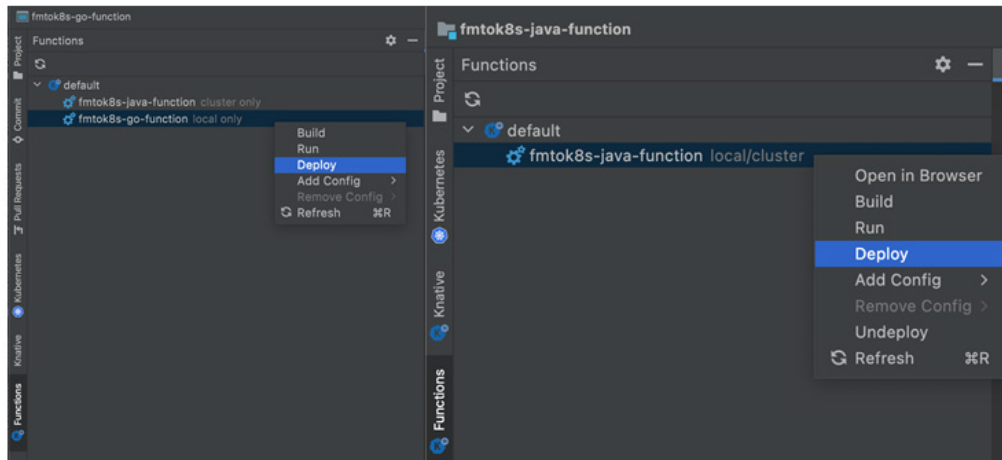


Figure 7.5 Knative `func` plugin for GoLand (left) and IntelliJ (right)

GoLand and IntelliJ are more specific distributions targeting Golang and Java, hence you don't need to install plugins from different providers which cannot work and by default you have a very advanced native experience with your language of choice. From the Knative perspective, the Knative Plugin provides the same functionality as in VSCode and in the same way, it is a plugin that you need to install in both GoLand and IntelliJ.

So far we have seen, how the `func` CLI provides an experience free from Dockerfiles and YAML files that allow developers to manage the lifecycle of a `func` project all the way from creation to running in a remote cluster. The `func` CLI provides users 6 main commands to manage their `func` projects lifecycle: `create`, `build`, `run`, `deploy`, `emit` and `delete`.

By looking at plugins that help developers to manage their `func` projects from their day to day IDEs, the experience is simplified even further, but so far we have focused on a simple function inside the `func` project. What happens when we want to build real-life applications that might contain several functions and how do we connect them together?

7.2.6 Connecting functions together in a polyglot ecosystem

Functions, as we have seen before, consume a CloudEvent, do some processing and then return a CloudEvent. Functions don't send events to each other, as they are not supposed to be aware of each other. So, how do you connect them together if you want to compose more complex logic?

You can follow a step-by-step tutorial to create two `func` projects one in Go and the other in Java with 3 functions that will connect to each other by using Knative Eventing: <https://github.com/salaboy/from-monolith-to-k8s/blob/master/knative/func.md>

While you can go really extreme and map one function per container, it does make sense to bundle several functions that are related and we want to scale together into a single container.

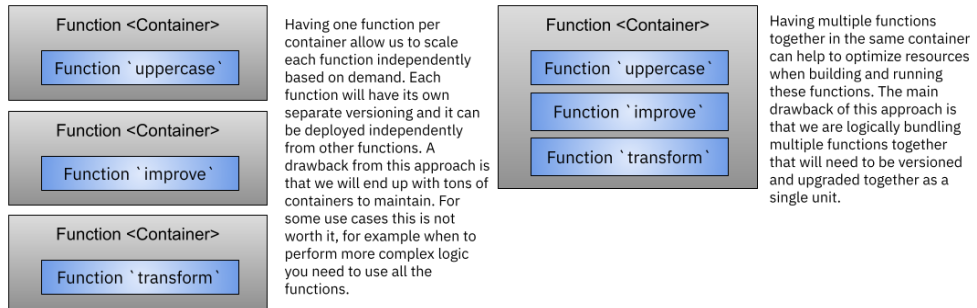


Figure 7.6 One function per project vs multiple functions per project

If you do so, then you will need a way to route events to different functions inside the same `func`` project (all the functions that you bundle together will share the same container). This routing mechanism will be then specific for each language but we can see two very common patterns:

- Routing based on a CloudEvent attribute (HTTP headers)
- Routing based on path, each function can expose a different path where it will receive CloudEvents

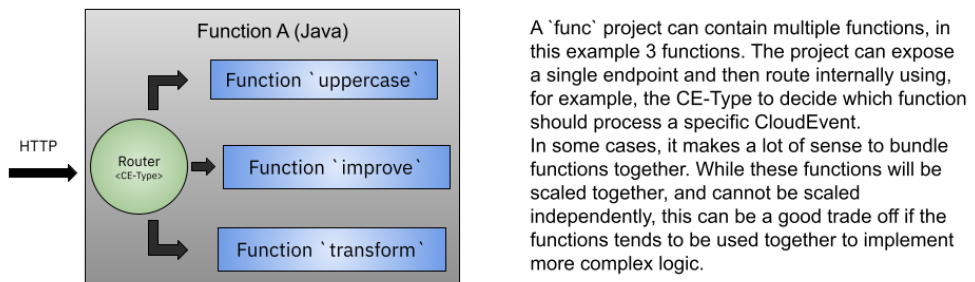
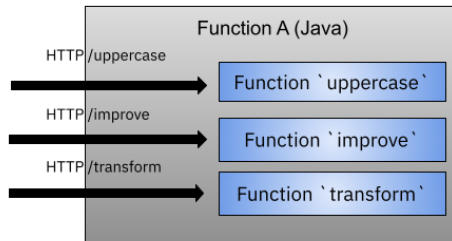


Figure 7.7 Multiple functions per `func`` project using internal routing mechanism



Path-based routing requires the producer or some routing mechanism to know the path for the function that you want to process the incoming CloudEvent.

Figure 7.8 Multiple functions per `func` project using path-based routing

While some frameworks already provide one or both of these functionalities out of the box you can implement these approaches using any programming language.

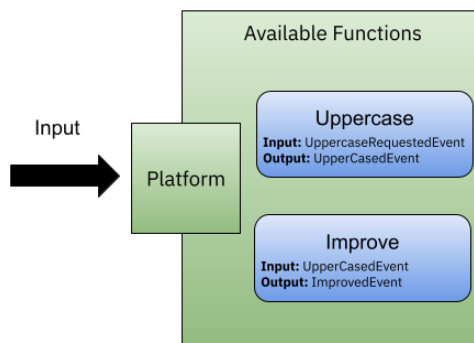
But no matter if we have one or more functions inside the same container, how do we connect one function with another? It will be great to let functions react to the event that they are designed to consume instead of manually sending HTTP requests directly to the function that we want to use. Wouldn't it be great if when a function returns a CloudEvent, that is automatically routed to a function that is consuming that CloudEvent?

ROUTING EVENTS BETWEEN FUNCTIONS USING KNATIVE EVENTING

But wait for a second, we have already seen how to wire Consumers and Producers of events using Knative Eventing, can we do the same for functions? The answer is yes, so let's take a look at what we need to connect two functions together.

As we have seen in the previous diagrams, we need to have different functions performing different operations to test that the right functions are being called at the right time.

Let's introduce our two functions: Uppercase and Improve.



When working with functions, it makes a lot of sense to let the platform to route the events to the functions that can process them. The application producing the events will not know where the functions are to send a direct request to them. Instead, they will know a public address where they can send events too and the platform will take care of the routing.

As you can see, as input you can provide an `UppercaseRequestedEvent` or an `UpperCasedEvent` and the platform will know how to route your events to the right function. If you provide an `UppercaseRequestedEvent`, the output event will be automatically forwarded to the `Improve` function, as the output type matches the input type for the `Improve` function.

Figure 7.9 Uppercase and Improve function examples

To make things more exciting we can create the **Uppercase** function in Java using Spring Boot and Spring Cloud functions and the **Improve** function in Go. Because each function will use a different tech stack and programming language, the option of having both functions colocated in the same container is not viable. Hence we will have two functions each in its own container. For creating these two functions we can use another very handy feature of `func` which is using external templates coming from custom repositories. I've created templates for these two functions so no coding is needed to get a basic version of the Uppercase and Improve functions.

```
func create -l springboot -t uppercase --repository https://github.com/salaboy/func-templates
```

And

```
func create -l go -t improve --repository https://github.com/salaboy/func-templates
```

Once we have both functions, we will need to figure out how to route events together and for that we will use the Broker and Trigger resources that we learnt about in chapter 6.

First we need a Broker, which will act as the main entry point for events coming from our applications. There are no limitations about how many brokers we can have, but for this example we just need one. You can create one by applying the following resource to your cluster:

```
kubectl create -f - <<EOF
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  name: default #1
  namespace: default #2
EOF
```

#1 The name of the Broker is default, you can choose any name that make sense for your application

#2 This Broker is created in the `default` namespace, this is important to understand how to route events to the Broker from outside our Kubernetes Cluster.

Once we have a Broker, the only thing we need is to create Triggers to make sure that our functions receives the right kind of events.

We will be creating one Trigger to forward events to the `uppercase` function and one trigger to forward events to the `improve` function.

```

kubectl create -f - <<EOF
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: uppercase-function-trigger
  namespace: default
spec:
  broker: default #1
  filter:
    attributes:
      type: UppercaseRequestedEvent #2
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: fmtok8s-java-uppercase-function #3
---

apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: improve-function-trigger
  namespace: default
spec:
  broker: default
  filter:
    attributes:
      type: UppercaseRequestedEvent #4
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: fmtok8s-go-improve-function
EOF

```

- #1 Both Triggers** (`improve-function-trigger` and `uppercase-function-trigger``) are created for the `default` Broker`. `default`` it's just the name for the broker that we created, it doesn't have any special meaning. You can have as many Brokers as you want, but triggers will always refer to a single broker.
- #2 For each Trigger**, you can define filters. At the moment of writing this book, these filters are exact matches, which means that only events that match the defined filters will be forwarded to subscribers. You can filter based on multiple attributes, including CloudEvents extensions, but all the attributes must match. There is an ongoing community effort to enable more advanced and dynamic filtering for Triggers.
- #3 As we saw in Chapter 6**, we can make reference to subscribers by specifying an addressable Kubernetes Resource, this allows the Trigger to validate that the Subscriber exists. If your subscriber is not an addressable resource, you can always use the `uri` approach`. Refer to the official Knative documentation for more details about these options.
- #4 By filtering on different event types**, we can chain functions or we can create different topologies to process different events. Notice that creating Triggers is something that doesn't need to be done by developers and how functions are wired together can be done at runtime, supporting different configurations in different environments.

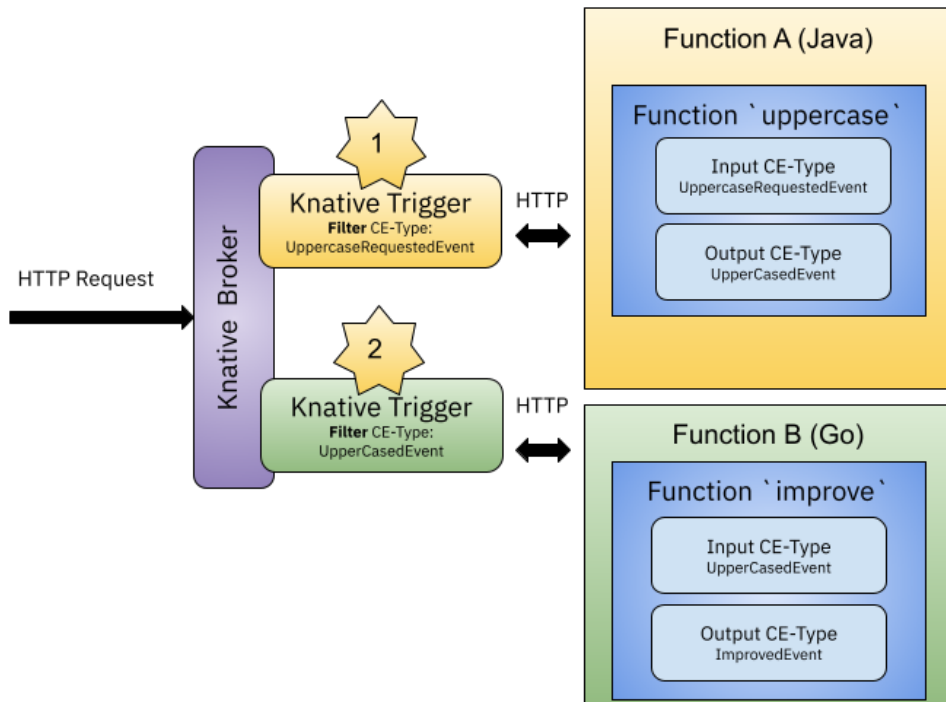


Figure 7.10 Broker and Triggers to wire our functions together

To recap, we have created two functions, each function expects a different kind of event. The `'uppercase'` function expect a `'UppercaseRequestedEvent'` and returns an `'UppercasedEvent'`, while the `'improve'` function expect an `'UppercasedEvent'` as input and returns an `'ImprovedEvent'` event, which no one is consuming right now.

I do expect the `'func'` project to include some tools in the future for automatically registering the triggers into a Broker based on the functions signatures, removing the need for manually defining the Broker and the Triggers.

To test this setup you only need to send an event to the Broker URL (you can find the Broker URL by running `'kubectl get broker'` and then exposing the Broker Ingress Service outside the Cluster using `'kubectl port-forward'` If you are sending events from inside the cluster, using the Broker URL is the way to go.

MONITORING ALL EVENTS GOING TO A BROKER WITH SOCKEYE

Because all events are going to the Broker and each function is returning events to the Broker we can tap and forward all events to another service to see what is going on. For this we can use Sockeye, and we will only need one more Trigger with no filters, in other words, it will send all events to Sockeye.

We can deploy Sockeye to our cluster by running:

```
kubectl apply -f https://github.com/n3wscott/sockeye/releases/download/v0.7.0/release.yaml
```

And create a new trigger for Sockeye with:

```
kubectl create -f - <<EOF
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: sockeye-trigger
  namespace: default
spec:
  broker: default #1
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: sockeye #2
EOF
```

#1 Once again, we target the `default` broker. Notice that there is no `filter` section for this Trigger definition, this means that all events will be forwarded to the subscriber.

#2 For this example, we are using Sockeye which is also deployed as a Knative Service. This simple web application will allow us to monitor all the events that are emitted to the Broker.

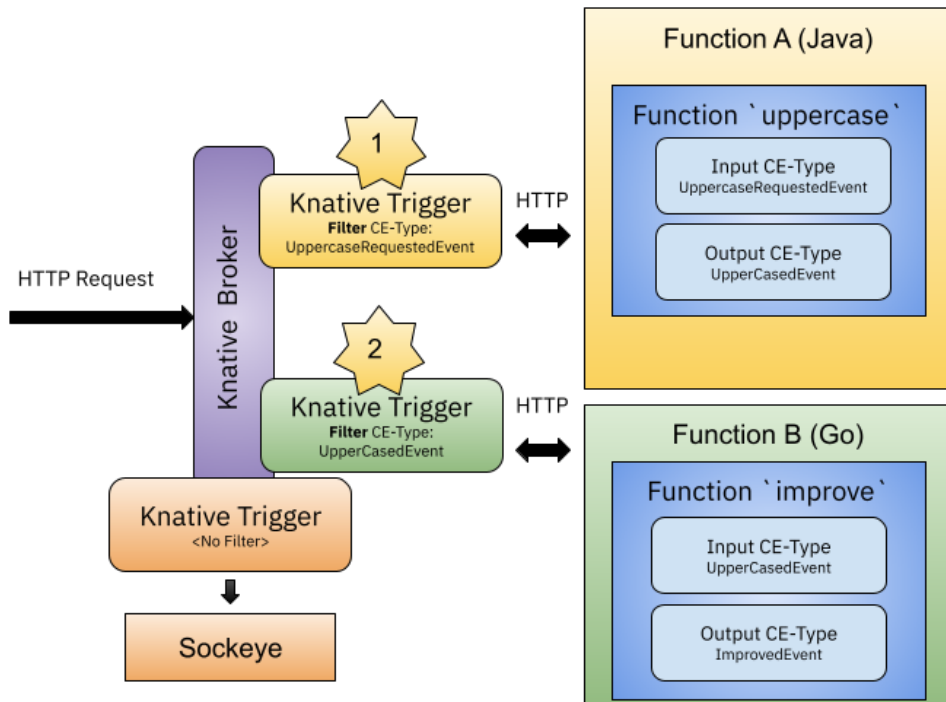


Figure 7.11 Using a Trigger with no filter to send all events to Sockeye

If we now send events to the Broker URL from inside or outside the cluster we should be able to see how our functions pick the right events and produce the expected outputs.

Can you implement a third function called `'transform'`, using your language of choice that consume the `'ImprovedEvent'` event and apply some transformation to the input, for example reversing the order of the input characters?

In this section, we have created two functions using two different technology stacks and then we have used Knative Eventing to route events between these functions. This manual wiring of functions can be automated, but Knative Functions provides all the building blocks to make this happen as you might have different restrictions on where functions will be deployed or which functions can talk to each other. I do personally want to see Knative Functions providing a full experience for connecting things together following the no YAML files approach in the future.

In the next section, using these building blocks we are going to build a much more complex scenario on top of our Conference Platform. We will be using functions to create a real-time game for our conference keynote.

7.3 Building a highly-scalable and polyglot game using functions

Real-time games are quite hard to architect and build, and while the game presented in this section is quite limited in its capabilities, it helps us to test our functions on top of a real cloud infrastructure that will scale based on the demand.

This section's focus is to consider at a high level a more complex scenario so we can focus on the challenges that you will face when building these kinds of applications on top of Kubernetes. We will be looking at how these functions will scale, how events will be routed and what are the most common issues that you will encounter when working with functions in a high-demand scenario.

The game presented here is built as part of our Conference Platform and it aims to be played with a live audience as part of the Conference Keynote presentation. For this game to be fun and engaging, it needs to be interactive and it should enable everyone in the audience to play live and compete against each other.

The game architecture allows level designers to add new levels to the game while it is running. Imagine a "Candy Crush"-like game, which keeps adding levels every day.

To implement new levels, developers just need to implement a function that accepts `KeyPressedEvents` coming from each of the players and have an internal logic to define when the player completed the level which at that point it should emit a `LevelCompletedEvent`. The user interface will also send a `LevelStartedEvent` when the user starts playing a new level, so the game can keep track of how much time it takes a player to complete each level.

The architecture for the game looks as follows:

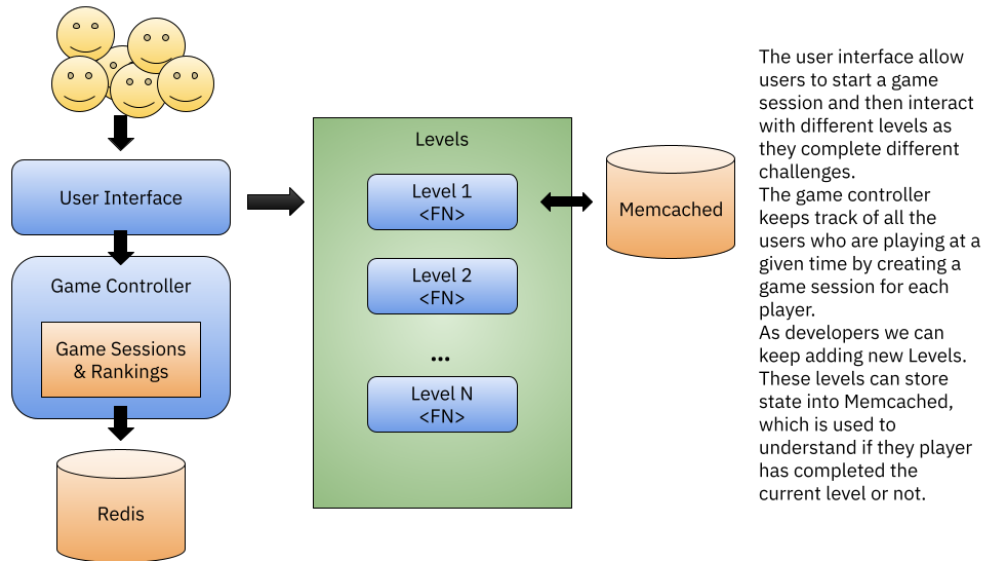


Figure 7.12 Function-based game architecture

Because we are using functions as our main building blocks for levels of the game, if nobody is playing the game, no functions will be running and only when a new player wants to play a specific level, the `StartLevelEvent` will cause the level function to be instantiated by Kubernetes.

This architecture presents different challenges from supporting multiple users playing at the same time, handling the correlation between different events and dealing with each player state.

In the following sections, we will look into how the pieces work together and how they scale up when the number of concurrent players is sending loads of events, hence we will be covering the following aspects:

- Routing a large number of events concurrently
- Dealing with functions state
- Monitoring all players and creating a real-time dashboard

Let's start with routing.

7.3.1 Routing a large number of events concurrently

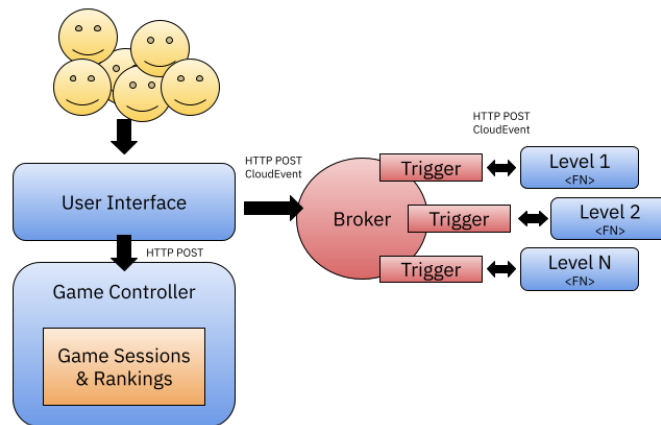
Not everything in our application are events, for example, to create a new game session, there is no specific need for encoding that request as an event. Creating a session is something synchronous by nature, we want a session and we want it now. To achieve this functionality the Game Controller Service exposes a REST endpoint that accepts a POST

request, as a reply the User Interface receives a SessionId and some more information about the session that was created.

Now when the user starts playing a specific level, all player interactions are sent as events into the function which is implementing that level. Each key that the player presses on their keyboard while playing the game will be transformed into a CloudEvent and sent to the function that is dealing with the current level. For routing these events to the right functions we will use a Knative Broker and Triggers which will filter the correct events for the level that the user is playing.

At the moment of writing this book, Knative `func` doesn't automatically create Triggers for your functions, nor the Broker(s). This is something that might come up as a community effort in following releases or as part of commercial offerings built on top of `func` or plain Knative.

For our example, let's imagine that we want to separate the functions from how we wire them together, as we can chain them together in different ways depending on what kind of game experience we want to provide.



The User Interface will produce Events for each player interaction and dispatch them directly to the Broker which will forward those events based on the filters defined in the Triggers. This allows us to add new levels and wire them without changing any application or level code. Level functions developers doesn't need to know which level comes before or after or how levels are going to be wired together.

Figure 7.13 Adding Knative Eventing to wire functions together

By separating level interactions into functions, we enable the application to scale up by creating new replicas for each level only when there is demand for them. If no one is playing level X then the function is automatically downscaled to zero by the Knative Serving autoscaler.

There is nothing specific about the Broker, and because we can choose between different implementations of the Broker we can use RabbitMQ or Kafka backed Broker implementations. Each of these implementations will come with a set of tools to monitor all the events that are flowing between the application and the level functions.

Triggers on the other hand will be filtering by level and type of event:

```

apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: level-1-trigger
  namespace: default
spec:
  broker: default #1
  filter:
    attributes:
      level: level-1 #2
      type: KeyPressedEvent #3
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: level-1 #4
      uri: /KeyPressedEvent #5

```

#1 We have a single Broker called ``default``, to keep this example simple we are just using a single Broker, but it makes sense to create different brokers to make sure that if we have too many events the Broker itself doesn't become the bottleneck.

#2 Filters can include CloudEvents extensions, in this case, we are defining a new extension called ``level`` that is being set by the User Interface when sending the event to the Broker, this allows the Trigger to only forward events to the level that the player is currently playing

#3 For this example, we are focusing only on ``KeyPressedEvents`` but there is nothing stopping us to create more triggers for different events. It is important to note that, if we are dealing with different types of events and we want to correlate them, we will probably need to have a way to group these different events together. A common way to do that is to include another extension usually called ``correlation-key`` which is shared by all the events which belong to the same logical group.

#4 Because our function is a Knative Service we can use the ``subscriber.ref`` reference for the subscriber

#5 Because the endpoint exposed in our function is not ``/`` we can also specify the contextual ``uri`` where the events are expected, in this case ``/KeyPressedEvent``

For each new level that we want to add to our game, we will need to define one of these triggers to make sure that all the events produced by the player are routed to the right function.

So far we have been dealing with Events being sent to the functions implementing the levels, but as we have seen before, each function will also produce an event as a result of processing each incoming event. For this example, an Event with the type ``LevelCompletedEvent`` will be produced when the current level has been successfully completed. If the level is not completed an event with the type ``LevelFailedEvent`` will be produced. Both events will end up in our Broker, so other triggers can be created to deal with these events.

For this example, we are ignoring the ``LevelFailedEvent`` but we are registering a Trigger for ``LevelCompletedEvent`` to forward these events to the Game Controller which has the logic to move the game forward to the next level.

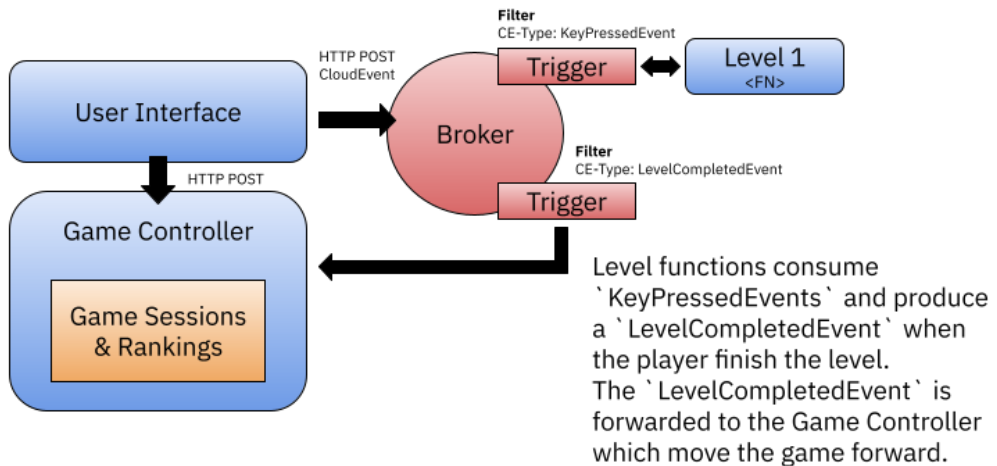


Figure 7.14 Linking level functions and the Game Controller using Triggers

In this context, the Game Controller is orchestrating the sequence of levels that the player is going to go through. This orchestration is done without knowing how many levels are available or which level comes next. While this approach is dynamic and can work for simple scenarios, having a level registry can help us to provide a better experience and reduce the complexity for checking which levels are available.

It is important to notice that under high demand, the Broker itself might become the bottleneck as each player will be hitting the same Broker.

7.3.2 Dealing with function's State

The game scenario has two main pieces that require us to store state to make sure that the application is resilient to crashes. The Game Controller needs to keep game sessions stored externally to make sure that we can scale this component by having multiple replicas. The information that the Game Controller need to store is not much, but we need to make sure this information is available at all times for the game experience to work as expected. As the Game Controller is keeping where each player is and receiving events when each player completes a level, we want to only keep track of the level statistics as part of moving each session forward. For this reason, the Game Controller will keep track of how much time takes since a player starts a level and the LevelCompletedEvent is received by the level function.

The second place where we will need a way to persist our events is for each function logic, as each level will need to collect events to validate if the player has completed the level or not. Because we have tons of events arriving, we need to make sure that different instances of the levels can access the same received events.

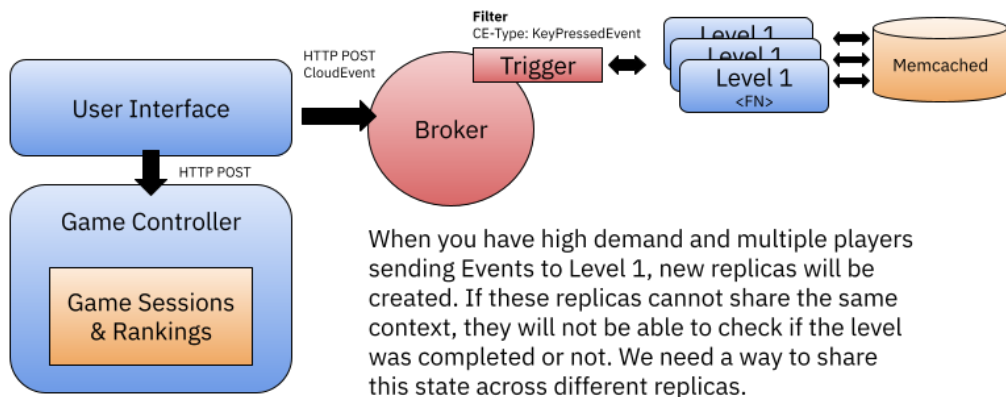


Figure 7.15 Storing events for aggregation to enable scaling up of our functions

This can be tricky, as latency will play a major role in how reactive our levels are, depending on how much time it takes for us to load all previous events when a new event arrives. We can use a tool like Memcached to store the events when they arrive and then fetch and process all the previous events for a specific player. We can even store snapshots aggregating previous events, but it will all come down to latency and how efficient will be to access external storage for each event that arrives.

Another approach, if you are using a Broker like Kafka, is to use the message broker implementation as a database and aggregate the data using any tool that allow you to consume messages and aggregate them together at that layer. By following this approach you avoid storing the events and aggregating them in different places.

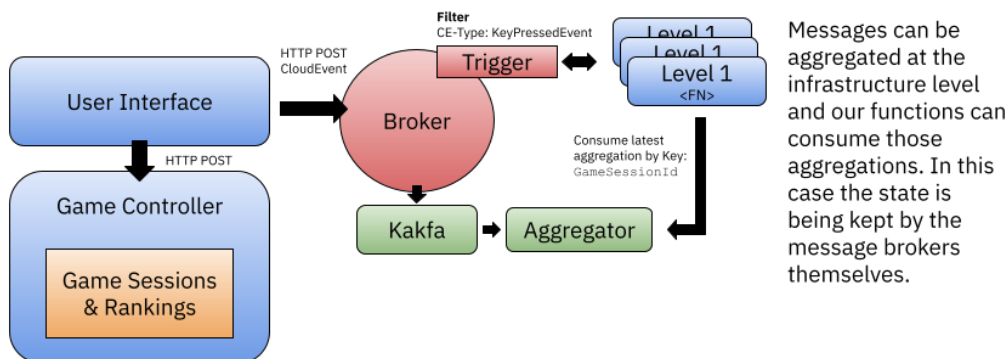


Figure 7.16 Using Broker implementation's tools to do aggregation

While this option is more performant, we are now defining application logic (how to aggregate our events) in a Broker's implementation-specific way.

The last approach that might be worth noting here, is to make functions sticky based on for example the `GameSessionId`, if we know that the same replica of our level function will receive all the events associated with a `GameSessionId` we can implement a fast local storage mechanism for holding the events as close to the function as possible. Unfortunately, Knative Serving doesn't support this scenario yet (<https://github.com/knative/serving/issues/8160>)

7.3.3 Monitoring all players and creating a real-time dashboard

As we discussed in the previous two sections, the Game experience is divided into the Game Controller and the events that are being sent to each level function.

We can tap into the Broker events to build a dashboard that shows real-time data about each player interaction. Depending on the game demand, the number of players that we are expecting to be playing concurrently, we can decide to create a Broker per level and by that split the load into different Brokers. From the monitoring/dashboard perspective, we will need to tap into these different brokers to have a full picture of what is going on.

In this chapter we have looked into how, by using a function-based approach, we can provide a simplified experience for our developers building more scoped applications. We have seen how these functions will scale up based on demand and how we can use Knative Eventing to route events to wire functions together to build more complex functionality. We have also seen the challenges that we will face when building this kind of application hence there is always a trade-off. My intention behind discussing a FaaS approach on top of Kubernetes is twofold:

- Promote a polyglot approach that leverage the Kubernetes ecosystem and offers a simplified developer experience by leveraging functions.
- Demonstrate how a developer experience can be built to reduce the cognitive load required to be productive. In this case, ``func`` aims to remove the complexity of Kubernetes and building containers, but if you are building your own platform you might want to abstract away more aspects to make your developers more productive.

In the next chapter, we will be reusing all we have learned so far in this book to describe all the pieces required to build a platform designed with Continuous Delivery in mind. We will be looking at how such a platform can be built, which tools can be used and what kind of developer experience we should be aiming for if we want to enable our teams to deliver more software more efficiently on top of Kubernetes.

7.4 Summary

- A function consumes an event and produce an event as a result. These events in the context of Kubernetes and Cloud-Native applications can be expressed using the CNCF CloudEvents specification.
- You can use Knative `func` to create, build, run locally and deploy a function project into a Kubernetes cluster without writing a Dockerfile or a single line of YAML. These functions can use your preferred programming language, so you can leverage the polyglot nature of Kubernetes.
- You can have one or more functions per container. You can have group functions together if they are all related to each other, for example, they are usually used together. When you group functions together in the same container you will need to define how events arriving to the container are routed to the right function. Two approach discussed in this chapter are `path-based` routing and `attribute-based` routing. Spring Cloud Functions for Spring Boot applications supports both.
- There are a number of Knative and Knative `func` plugins that you can use in VSCode, IntelliJ Idea and GoLand to manage your functions lifecycle.
- You can use Knative eventing resources such as Brokers and Triggers to chain functions together to build more complex interactions.