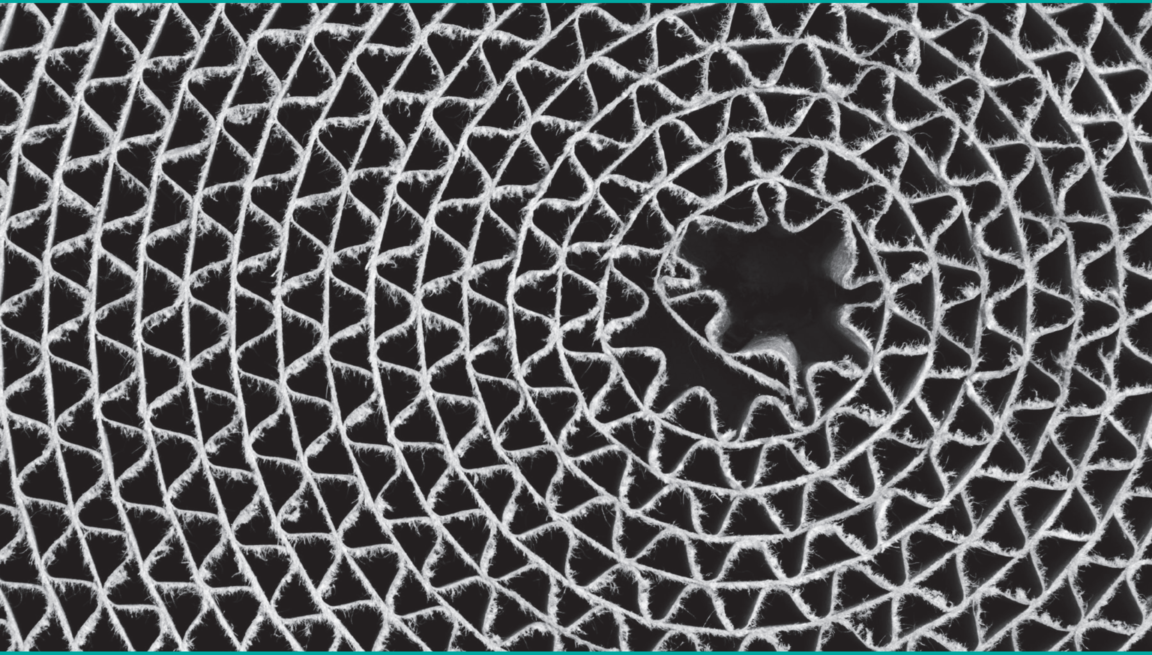
O'REILLY®

Compliments of Pivotal®

Getting Started with Knative

Building Modern Serverless Workloads on Kubernetes



Brian McClain & Bryan Friedman

Pivotal
Cloud Foundry®

The best way to run Kubernetes meets the best way to run Knative.

- Pivotal Container Service is the enterprise solution to Kubernetes, built with Day 2 operations in mind
- Knative-powered Pivotal Function Service lets your developers quickly and easily deploy their event-driven code
- All your workloads on one secure, multi-cloud platform

Learn more at pivotal.io/platform

Pivotal®

Getting Started with Knative

Building Modern Serverless Workloads on Kubernetes

Brian McClain and Bryan Friedman

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Getting Started with Knative

by Brian McClain and Bryan Friedman

Copyright © 2019 O'Reilly Media Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors:
Virginia Wilson and Nikki McDonald
Production Editor: Nan Barber
Copyeditor: Kim Cofer
Proofreader: Nan Barber
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

March 2019: First Edition

Revision History for the First Edition

2019-02-13: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Getting Started with Knative*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Pivotal. See our statement of editorial independence.

978-1-492-04699-8

[LSI]

Table of Contents

Preface

Kubernetes has won. Not the boldest statement ever made, but true nonetheless. Container-based deployments have been rising in popularity, and Kubernetes has risen as the de facto way to run them. By its own admission, though, Kubernetes is a platform for *containers* rather than *code*. It's a great platform to run and manage containers, but how those containers are built and how they run, scale, and are routed to is largely left up to the user. These are the missing pieces that Knative looks to fill.

Maybe you're running Kubernetes in production today, or maybe you're a starry-eyed enthusiast dreaming to modernize your OS/2-running organization. Either way, this report doesn't make many assumptions and only really requires that you know what a container is, have some working knowledge of Kubernetes, and have access to a Kubernetes installation. If you don't, Minikube is a great option to get started.

We'll be using a lot of code samples and prebuilt container images that we've made available and open source to all readers. You can find all code samples at *http://github.com/gswk* and all container images at *http://hub.docker.com/u/gswk*. You can also find handy links to both of these repositories as well as other great reference material at *http://gswkbook.com*.

We're extremely excited for what Knative aspires to become. While we are colleagues at Pivotal—one of the largest contributors to Knative—this report comes simply from us, the authors, who are very passionate about Knative and the evolving landscape of developing and running functions. Some of this report consists of our opinions, which some readers will inevitably disagree with and will enthusiastically let us know why we're wrong. That's ok! This area of computing is very new and is constantly redefining itself. At the very least, this report will have you thinking about serverless architecture and get you feeling just as excited for Knative as we are.

Who This Report Is For

We are developers by nature, so this report is written primarily with a developer audience in mind. Throughout the report, we explore serverless architecture patterns and show examples of self-service use cases for developers (such as building and deploying code). However, Knative appeals to technologists playing many different roles. In particular, operators and platform builders will be intrigued by the idea of using Knative components as part of a larger platform or integrated with their systems. This report will be useful for these audiences as they explore using Knative to serve their specific purposes.

What You Will Learn

While this report isn't intended to be a comprehensive, bit-by-bit look at the complete laundry list of features in Knative, it is still a fairly deep dive that will take you from zero knowledge of what Knative is to a very solid understanding of how to use it and how it works. After exploring the goals of Knative, we'll spend some time looking at how to use each of its major components. Then, we'll move to a few advanced use cases, and finally we'll end by building a real-world example application that will leverage much of what you learn in this report.

Acknowledgments

We would like to thank Pivotal. We are both first time authors, and I don't think either of us would have been able to say that without the support of our team at Pivotal. Dan Baskette, Director of Technical Marketing (and our boss) and Richard Seroter, VP of Product Marketing, have been a huge part in our growth at Pivotal and wonderful leaders. We'd like to thank Jared Ruckle, Derrick Harris, and Jeff Kelly, whose help to our growth as writers cannot be overstated. We'd also like to thank Tevin Rawls who has been a great intern on our team at Pivotal and helped us build the frontend for our demo in Chapter 7. Of course, we'd like to thank the O'Reilly team for all their support and guidance. A huge thank you to the entire Knative community, especially those at Pivotal who have helped us out any time we had a question, no matter how big or small it might be. Last but certainly not least, we'd like to thank Virginia Wilson, Dr. Nic Williams, Mark Fisher, Nate Schutta, Michael Kehoe, and Andrew Martin for taking the time to review our work in progress and offer guidance to shape the final product.

Brian McClain: I'd like to thank my wonderful wife Sarah for her constant support and motivation through the writing process. I'd also like to thank our two dogs, Tony and Brutus, for keeping me company nearly the entire time spent working on this report. Also thanks to our three cats Tyson, Marty, and Doc, who actively made writing harder by wanting to sleep on my laptop, but I still appreciated their company. Finally, a thank you to my awesome coauthor Bryan Friedman, without whom this report would not be possible. Pivotal has taught me that pairing often yields multiplicative results rather than additive, and this has been no different.

Bryan Friedman: Thank you to my amazing wife Alison, who is certainly the more talented writer in the family but is always so supportive of *my* writing. I should also thank my two beautiful daughters, Madelyn and Arielle, who inspire me to be better every day. I also have a loyal office mate, my dog Princeton, who mostly just enjoys the couch but occasionally would look at me with a face that implied he was proud of my work on this report. And of course, there's no way I could have done this alone, so I have to thank my coauthor, Brian McClain, whose technical prowess and contagious passion helped me immensely throughout. It's been an honor to pair with him.

CHAPTER 1 Knative Overview

A belief of ours is that having a platform as a place for your software is one of the best choices you can make. A standardized development and deployment process has continually been shown to reduce both time and money spent writing code by allowing developers to focus on delivering new features. Not only that, ensured consistency across applications means that they're easier to patch, update, and monitor, allowing operators to be more efficient. Knative aims to be this modern platform.

What Is Knative?

Let's get to the meat of Knative. If Knative does indeed aim to bookend the development cycle on top of Kubernetes, not only does it need to help you run and scale your applications, but to help you architect and package them, too. It should enable you as a developer to write code how you want, in the language you want.

To do this, Knative focuses on three key categories: *building* your application, *serving* traffic to it, and enabling applications to easily consume and produce *events*.

Build
: Flexible, pluggable build system to go from source to container. Already has support for several build systems such as Google's Kaniko, which can build container images on your Kubernetes cluster without the need for a running Docker daemon.

Serving

Automatically scale based on load, including scaling to zero when there is no load. Allows you to create traffic policies for multiple revisions, enabling easy routing to applications via URL.

Events

Makes it easy to produce and consume events. Abstracts away from event sources and allows operators to run their messaging layer of choice.

Knative is installed as a set of Custom Resource Definitions (CRDs) for Kubernetes, so it's as easy to get started with Knative as applying a few YAML files. This also means that, on-premises or with a managed cloud provider, you can run Knative and your code anywhere you can run Kubernetes.

Kubernetes Knowledge

Since Knative is a series of extensions for Kubernetes, having some background on Kubernetes and Docker constructs and terminology is recommended. We will be referring to objects like namespaces, Deployments, ReplicaSets, and Pods. Familiarity with these Kubernetes terms will help you better understand the underlying workings of Knative as you read on. If you're new to either, both Kubernetes and Docker have great in-browser training material!

Serverless?

We've talked about containerizing our applications so far, but it's 2019 and we've gone through half of a chapter without mentioning the word "serverless." Perhaps the most loaded word in technology today, serverless is still looking for a definition that the industry as a whole can agree on. Many agree that one of the major changes in mindset is at the code level, where instead of dealing with large, monolithic applications, you write small, single-purpose *functions* that are invoked via *events*. Those events could be as simple as an HTTP request or a message from a message broker such as Apache Kafka. They could also be events that are less direct, such as uploading an image to Google Cloud Storage, or making an update to a table in Amazon's DynamoDB.

Many also agree that it means your code is using compute resources only while serving requests. For hosted services such as Amazon's Lambda or Google's Cloud Functions, this means that you're only paying for active compute time rather than paying for a virtual machine running 24/7 that may not even be doing anything much of the time. On-premises or in a nonmanaged serverless platform, this might translate to only running your code when it's needed and scaling it down to zero when it's not, leaving your infrastructure free to spend compute cycles elsewhere.

Beyond these fundamentals lies a holy war. Some insist serverless only works in a managed cloud environment and that running such a platform on-premises completely misses the point. Others look at it as more of a design philosophy than anything. Maybe these definitions will eventually merge, maybe they won't. For now, Knative looks to standardize some of these emerging trends as serverless adoption continues to grow.

Why Knative?

Arguments on the definition of serverless aside, the next logical question is "why was Knative built?" As trends have grown toward container-based architectures and the popularity of Kubernetes has exploded, we've started to see some of the same questions arise that previously drove the growth of Platform-as-a-Service (PaaS) solutions. How do we ensure consistency when building containers? Who's responsible for keeping everything patched? How do you scale based on demand? How do you achieve zero-downtime deployment?

While Kubernetes has certainly evolved and begun to address some of these concerns, the concepts we mentioned with respect to the growing serverless space start to raise even more questions. How do you recover infrastructure from sources with no traffic to scale them to zero? How can you consistently manage multiple event types? How do you define event sources and destinations?

A number of serverless or Functions-as-a-Service (FaaS) frameworks have attempted to answer these questions, but not all of them leverage Kubernetes, and they have all gone about solving these problems in different ways. Knative looks to build on Kubernetes and present a consistent, standard pattern for building and deploying serverless and event-driven applications. Knative removes the overhead that often comes with this new approach to software-development, while abstracting away complexity around routing and eventing.

Conclusion

Now that we have a good handle on what Knative is and why it was created, we can start diving in a little further. The next chapters describe the key components of Knative. We will examine all three of them in detail and explain how they work together and how to leverage them to their full potential. After that, we'll look at how you can install Knative on your Kubernetes cluster as well as some more advanced use cases. Finally, we'll walk through a demo that implements much of what you'll learn over the course of the report.

CHAPTER 2
Serving

Even with serverless architectures, the ability to handle and respond to HTTP requests is an important concept. Before you write some code and have events trigger a function, you need a place for the code to run.

This chapter examines Knative's Serving component. You will learn how Knative Serving manages the deployment and serving of applications and functions. Serving lets you easily deploy a prebuilt image to the underlying Kubernetes cluster. (In Chapter 3, you will see that Knative Build can help build your images for you to run in the Serving component.) Knative Serving maintains point-in-time snapshots, provides automatic scaling (both up and down to zero), and handles the necessary routing and network programming.

The Serving module defines a specific set of objects to control all this functionality: *Revision*, *Configuration*, *Route*, and *Service*. Knative implements these objects in Kubernetes as Custom Resource Definitions (CRDs). Figure 2-1 shows the relationship between all the Serving components. The following sections will explore each in detail.

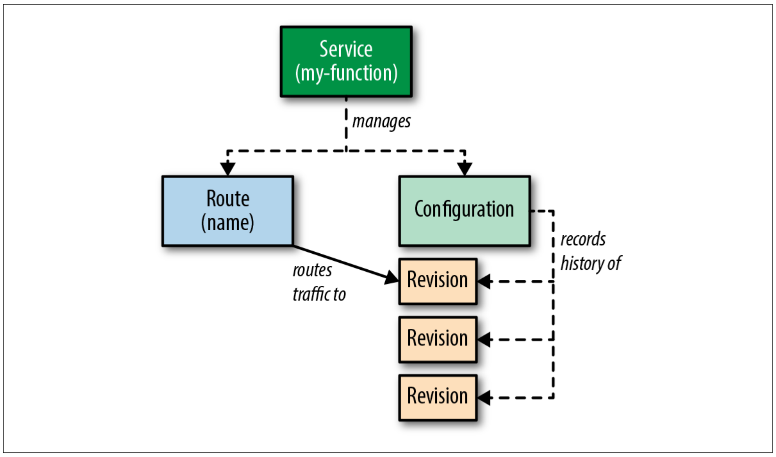


Figure 2-1. The Knative Serving object model

Configurations and Revisions

Configurations are a great place to start when working with Knative Serving. A Configuration is where you define your desired state for a deployment. At a minimum, this includes a Configuration name and a reference to the container image to deploy. In Knative, you define this reference as a Revision.

Revisions represent immutable, point-in-time snapshots of code and configuration. Each Revision references a specific container image to run, along with any specification required to run it (such as environment variables or volumes). You will not explicitly create Revisions, though. Since Revisions are immutable, they are never changed or deleted. Instead, Knative creates a new Revision whenever you modify the Configuration. This allows a Configuration to reflect the present state of a workload while also maintaining a list of its historical Revisions.

Example 2-1 shows a full Configuration definition. It specifies a Revision that refers to a particular image as a container registry URI and specified version tag.

Example 2-1. knative-helloworld/configuration.yml

```
apiVersion: serving.knative.dev/v1alpha1
kind: Configuration
metadata:
  name: knative-helloworld
  namespace: default
spec:
  revisionTemplate:
    spec:
      container:
        image: docker.io/gswk/knative-helloworld:latest
        env:
          - name: MESSAGE
            value: "Knative!"
```

Now you can apply this YAML file with a simple command:

```
$ kubectl apply -f configuration.yaml
```

Defining a Custom Port

By default, Knative will assume that your application listens on port 8080. However, if this is not the case, you can define a custom port via the `containerPort` argument:

```
spec:
  revisionTemplate:
    spec:
      container:
        image: docker.io/gswk/knative-helloworld:latest
        env:
          - name: MESSAGE
            value: "Knative!"
        ports:
          - containerPort: 8081
```

As with any Kubernetes objects, you may view Revisions and Configurations in the system using the command-line interface (CLI). You can use `kubectl get revisions` and `kubectl get configurations` to get a list of them. To get our specific Configuration that we just created from Example 2-1, we'll use `kubectl get configuration knative-helloworld -oyaml`. This will show the full details of this Configuration in YAML form (see Example 2-2).

Example 2-2. Output of `kubectl get configuration knative-helloworld -oyaml`

```
apiVersion: serving.knative.dev/v1alpha1
kind: Configuration
metadata:
  creationTimestamp: YYYY-MM-DDTHH:MM:SSZ
  generation: 1
  labels:
    serving.knative.dev/route: knative-helloworld
    serving.knative.dev/service: knative-helloworld
  name: knative-helloworld
  namespace: default
  ownerReferences:
  - apiVersion: serving.knative.dev/v1alpha1
    blockOwnerDeletion: true
    controller: true
    kind: Service
    name: knative-helloworld
    uid: 9835040f-f29c-11e8-a238-42010a8e0068
  resourceVersion: "374548"
  selfLink: "/apis/serving.knative.dev/v1alpha1/namespaces\
    /default/configurations/knative-helloworld"
  uid: 987101a0-f29c-11e8-a238-42010a8e0068
spec:
  generation: 1
  revisionTemplate:
    metadata:
      creationTimestamp: null
    spec:
      container:
        image: docker.io/gswk/knative-helloworld:latest
        name: ""
        resources: {}
status:
  conditions:
  - lastTransitionTime: YYYY-MM-DDTHH:MM:SSZ
    status: "True"
    type: Ready
  latestCreatedRevisionName: knative-helloworld-00001
  latestReadyRevisionName: knative-helloworld-00001
  observedGeneration: 1
```

Notice under the `status` section in Example 2-2 that the Configuration controller keeps track of the most recently created and most recently ready Revisions. It also contains the condition of the Revision, indicating whether it is ready to receive traffic.

NOTE

The Configuration may specify a preexisting container image, as in Example 2-1. Or, it may instead choose to reference a Build resource to create a container image from source code. Chapter 3 covers the Knative Build module in more detail and offers some examples of this.

So what's really going on inside our Kubernetes cluster? What happens with the container image we specified in the Configuration? Knative is turning the Configuration definition into a number of Kubernetes objects and creating them on the cluster. After applying the Configuration, you can see a corresponding Deployment, ReplicaSet, and Pod. Example 2-3 shows the objects that were created for the Hello World sample from Example 2-1.

Example 2-3. Kubernetes objects created by Knative

```
$ kubectl get deployments -oname
deployment.extensions/knative-helloworld-00001-deployment

$ kubectl get replicasets -oname
replicaset.extensions/knative-helloworld-00001-deployment-5f7b54c768

$ kubectl get pods -oname
pod/knative-helloworld-00001-deployment-5f7b54c768-lrqt5
```

We now have a Pod running our application, but how do we know where to send requests to it? This is where Routes come in.

Routes

A Route in Knative provides a mechanism for routing traffic to your running code. It maps a named, HTTP-addressable endpoint to one or more Revisions. A Configuration alone does not define a Route. Example 2-4 shows the definition for the most basic Route that sends traffic to the latest Revision of a specified Configuration.

Example 2-4. knative-helloworld/route.yml

```
apiVersion: serving.knative.dev/v1alpha1
kind: Route
metadata:
  name: knative-helloworld
  namespace: default
spec:
  traffic:
  - configurationName: knative-helloworld
    percent: 100
```

Just as we did with our Configuration, we can apply this YAML file with a simple command:

```
kubectl apply -f route.yaml
```

This Route sends 100% of traffic to the latestReadyRevisionName of the Configuration specified in configurationName. You can test this Route and Configuration by issuing the following curl command:

```
curl -H "Host: knative-routing-demo.default.example.com"
http://$KNATIVE_INGRESS
```

Instead of using the latestReadyRevisionName, you can instead pin a Route to send traffic to a specific Revision using revisionName. Using the name parameter, you can also access Revisions via addressable subdomain. Example 2-5 shows both of these scenarios together.

Example 2-5. knative-routing-demo/route.yml

```
apiVersion: serving.knative.dev/v1alpha1
kind: Route
metadata:
  name: knative-routing-demo
  namespace: default
spec:
  traffic:
  - revisionName: knative-routing-demo-00001
    name: v1
    percent: 100
```

Again we can apply this YAML file with a simple command:

```
kubectl apply -f route.yaml
```

The specified Revision will be accessible using the v1 subdomain as in the following curl command:

```
curl -H "Host: v1.knative-routing-demo.default.example.com"
http://$KNATIVE_INGRESS
```

NOTE

By default, Knative uses the `example.com` domain, but it is not intended for production use. You'll notice the URL passed as a host header in the `curl` command (`v1.knative-routing-demo.default.example.com`) includes this default as the domain suffix. The format for this URL follows the pattern `{REVISION_NAME}.{SERVICE_NAME}.{NAMESPACE}.{DOMAIN}`.

The `default` portion of the subdomain refers to the namespace being used in this case. You will learn how to change this value and use a custom domain in "Deployment Considerations" on page 42.

Knative also allows for splitting traffic across Revisions on a percentage basis. This supports things like incremental rollouts, blue-green deployments, or other complex routing scenarios. You will see these and other examples in Chapter 6.

Autoscaler and Activator

A key principle of serverless is scaling up to meet demand and down to save resources. Serverless workloads should scale all the way down to zero. That means no container instances are running if there are no incoming requests. Knative uses two key components to achieve this functionality. It implements Autoscaler and Activator as Pods on the cluster. You can see them running alongside other Serving components in the `knative-serving` namespace (see Example 2-6).

Example 2-6. Output of `kubectl get pods -n knative-serving`

```
NAME                            READY     STATUS    RESTARTS    AGE
activator-69dc4755b5-p2m5h      2/2       Running   0           7h
autoscaler-7645479876-4h2ds     2/2       Running   0           7h
controller-545d44d6b5-2s2vt     1/1       Running   0           7h
webhook-68fdc88598-qrt52        1/1       Running   0           7h
```

The Autoscaler gathers information about the number of concurrent requests to a Revision. To do so, it runs a container called the `queue-proxy` inside the Revision's Pod that also runs the user-provided image. You can see it by using the `kubectl describe` command on the Pod that represents the desired Revision (see Example 2-7).

Example 2-7. Snippet from output of `kubectl describe pod knative-helloworld-00001-deployment-id`

```
...
Containers:
  user-container:
    Container ID:   docker://f02dc...
    Image:          index.docker.io/gswk/knative-helloworld...
...
  queue-proxy:
    Container ID:  docker://1afcb...
    Image:         gcr.io/knative-releases/github.com/knative...
...
```

The `queue-proxy` checks the observed concurrency for that Revision. It then sends this data to the Autoscaler *every one second*. The Autoscaler evaluates these metrics *every two seconds*. Based on this evaluation, it increases or decreases the size of the Revision's underlying Deployment.

By default, the Autoscaler tries to maintain an average of 100 requests per Pod per second. This concurrency target and the average concurrency window are both changeable. The Autoscaler can also be configured to leverage the Kubernetes Horizontal Pod Autoscaler (HPA) instead. This will autoscale based on CPU usage but does not support scaling to zero. These settings can all be customized via annotations in the metadata of the Revision. Check the Knative documentation for details on these annotations.

For example, say a Revision is receiving 350 requests per second and each request takes about .5 seconds. Using the default setting of 100 requests per Pod, the Revision will receive 2 Pods:

```
350 * .5 = 175
175 / 100 = 1.75
ceil(1.75) = 2 pods
```

The Autoscaler is also responsible for scaling down to zero. Revisions receiving traffic are in the Active state. When a Revision stops receiving traffic, the Autoscaler moves it to the Reserve state. For this to happen, the average concurrency per Pod must remain at 0.0 for 30 seconds. (This is the default setting, but it is configurable.)

In the Reserve state, a Revision's underlying Deployment scales to zero and all its traffic gets routed to the Activator. The Activator is a shared component that catches all traffic for Reserve Revisions (though it can be scaled horizontally to handle increased load).

When it receives a request for a Reserve Revision, it transitions that Revision to Active. It then proxies the requests to the appropriate Pods.

How Autoscaler Scales

The scaling algorithm used by Autoscaler averages all data points over two separate time intervals. It maintains both a 60-second window and a 6-second window. The Autoscaler then uses this data to operate in two different modes: Stable Mode and Panic Mode. In Stable Mode, it uses the 60-second window average to determine how it should scale the Deployment to meet the desired concurrency.

If the 6-second average concurrency reaches twice the desired target, the Autoscaler transitions into Panic Mode and uses the 6-second window instead. This makes it much more responsive to sudden increases in traffic. It will also only scale *up* during Panic Mode to prevent rapid fluctuations in Pod count. The Autoscaler transitions back to Stable Mode after 60 seconds without scaling up.

Figure 2-2 shows how the Autoscaler and Activator work with Routes and Revisions.

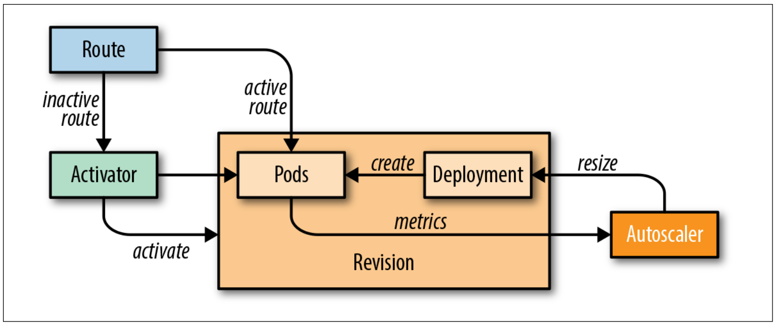


Figure 2-2. How the Autoscaler and Activator interact with Knative Routes and Revisions.

> Both the Autoscaler and Activator are rapidly evolving pieces of Knative. Refer to the latest Knative documentation for any recent changes or enhancements.

Services

A Service in Knative manages the entire life cycle of a workload. This includes deployment, routing, and rollback. (Do not confuse a Knative Service with a Kubernetes Service. They are different resources.) A Knative Service controls the collection of Routes and Configurations that make up your software. A Knative Service can be considered the piece of code—the application or function you are deploying.

A Service takes care to ensure that an app has a Route, a Configuration, and a new Revision for each update of the Service. If you do not specifically define a Route when creating a Service, Knative creates one that sends traffic to the latest Revision. You could instead choose to specify a particular Revision to route traffic to.

You are not required to explicitly create a Service. Routes and Configurations may be separate YAML files (as in Example 2-1 and Example 2-4). In that case, you would apply each one individually to the cluster. However, the recommended approach is to use a Service to orchestrate both the Route and Configuration. The file shown in Example 2-8 replaces the `configuration.yml` and `route.yml` from Example 2-1 and Example 2-4.

Example 2-8. knative-helloworld/service.yml

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: knative-helloworld
  namespace: default
spec:
  runLatest:
    configuration:
      revisionTemplate:
        spec:
          container:
            image: docker.io/gswk/knative-helloworld:latest
```

Notice this `service.yml` file is very similar to the `configuration.yml`. This file defines the Configuration and is the most minimal Service definition. Since there is no Route definition, a default Route points to the latest Revision. The Service's controller collectively tracks the statuses of the Configuration and Route that it owns. It then reflects these statuses in its `ConfigurationsReady` and

`RoutesReady` conditions. These statuses can be seen when requesting information about a Knative Service from the CLI using the `kubectl get ksvc` command.

Example 2-9. Snippet from output of `kubectl get ksvc knative-helloworld -oyaml`

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
...
  name: knative-helloworld
  namespace: default
...
spec:
...
status:
  conditions:
  - lastTransitionTime: YYYY-MM-DDTHH:MM:SSZ
    status: "True"
    type: ConfigurationsReady
  - lastTransitionTime: YYYY-MM-DDTHH:MM:SSZ
    status: "True"
    type: Ready
  - lastTransitionTime: YYYY-MM-DDTHH:MM:SSZ
    status: "True"
    type: RoutesReady
  domain: knative-helloworld.default.example.com
  domainInternal: knative-helloworld.default.svc.cluster.local
  latestCreatedRevisionName: knative-helloworld-00001
  latestReadyRevisionName: knative-helloworld-00001
  observedGeneration: 1
  targetable:
    domainInternal: knative-helloworld.default.svc.cluster.local
  traffic:
  - percent: 100
    revisionName: knative-helloworld-00001
```

Example 2-9 shows the output of this command. You can see the statuses along with the default Route all highlighted in bold.

Conclusion

Now you've been introduced to Services, Routes, Configurations, and Revisions. Revisions are immutable and only created along with changes to Configurations. You can create Configurations and Routes individually, or combine them together and define them as a

Service. Understanding these building blocks of the Serving module is essential to working with Knative. The apps you deploy all require a Service or Configuration in order to run as a container in Knative.

But how do you package your source code into a container image to deploy in this way? Chapter 3 will answer this question and introduce you to the Knative Build module.

CHAPTER 3
Build

Whereas the Serving component of Knative is how you go from container-to-URL, the Build component is how you go from source-to-container. Rather than pointing to a prebuilt container image, the Build resource lets you define how your code is compiled and the container is built. This ensures a consistent way to compile and package your code before shipping it to the container registry of your choice. There are a few new components that we'll introduce in this chapter:

Builds
: The custom Kubernetes resource that drives a build process. When you define a build, you define how to get your source code and how to create the container image that will run it.

Build Templates
: A template that encapsulates a repeatable collection of build steps and allows builds to be parameterized.

Service Accounts
: Allows for authentication to private resources, such as a Git repository or container registry.

NOTE

At the time of writing, there is active work to migrate Builds to Build Pipelines, a restructuring of builds in Knative that more closely resembles CI/CD pipelines. This means builds in Knative, in addition to compiling and packaging your code, can also easily run tests and publish those results. Make sure to keep an eye on future releases of Knative for this change.

Service Accounts

Before we begin to configure our Build we first face an immediate question: How do we reach out to services that require authentication at build-time? How do we pull code from a private Git repository or push container images to Docker Hub? For this, we can leverage a combination of two Kubernetes-native components: *Secrets* and *Service Accounts*. Secrets allow us to securely store the credentials needed for these authenticated requests while Service Accounts allow us the flexibility of providing and maintaining credentials for multiple Builds without manually configuring them each time we build a new application.

In Example 3-1, we first create our Secret named `dockerhub-account` that includes our credentials. We can of course then apply this like we would with any other YAML, as shown in Example 3-2.

Example 3-1. knative-build-demo/secret.yaml

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: dockerhub-account
  annotations:
    build.knative.dev/docker-0: https://index.docker.io/v1/
type: kubernetes.io/basic-auth
data:
  # 'echo -n "username" | base64'
  username: dXNlcm5hbWUK
  # 'echo -n "password" | base64'
  password: cGFzc3dvcmQK
```

Example 3-2. kubectl apply

```
kubectl apply -f knative-build-demo/secret.yaml
```

The first thing to notice is that both the username and password are base64 encoded when passed to Kubernetes. We've also noted that we're using `basic-auth` to authenticate against Docker Hub, meaning that we'll authenticate with a username and password rather than something like an access token. Additionally, Knative also ships with *ssh-auth* out of the box, allowing us to authenticate using an SSH private key if we would like to pull code from a private Git repository, for example.

In addition to giving the Secret the name of `dockerhub-account`, we've also annotated our Secret. Annotations are a way of saying which credentials to use when connecting to a specific host. In our case in Example 3-3, we've defined a basic-auth set of credentials to use when connecting to Docker Hub.

Are My Credentials Secure?

Encoding our credentials using base64 encoding is not done for security, but rather a means to reliably transfer these strings into Kubernetes. On the backend, Kubernetes provides more options on how Secrets are encrypted. For more information on encrypting Secrets, please refer to the Kubernetes documentation.

Once we've created the Secret named `dockerhub-account`, we must then create the Service Account that will run our application, so that it will have access to the credentials in Kubernetes. The configuration is straightforward, which we can see in Example 3-3.

Example 3-3. knative-build-demo/serviceaccount.yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-bot
secrets:
- name: dockerhub-account
```

Here we've seen that we create a `ServiceAccount` named `build-bot`, giving it access to the `dockerhub-account` Secret. In our example Knative uses these credentials to authenticate to Docker Hub when pushing our container image.

The Build Resource

Let's start with our Hello World app to get started. It's a simple Go application that listens on port 8080 and responds to HTTP GET requests with "Hello from Knative!" The entirety of its code can be seen in Example 3-4.

Example 3-4. knative-helloworld/app.go

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func handlePost(rw http.ResponseWriter, req *http.Request) {
    fmt.Fprintf(rw, "%s", "Hello from Knative!")
}

func main() {
    log.Print("Starting server on port 8080...")
    http.HandleFunc("/", handlePost)
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

We'll also write a Dockerfile that will build our code and container, shown in Example 3-5.

Example 3-5. knative-helloworld/Dockerfile

```
FROM golang

ADD . /knative-build-demo
WORKDIR /knative-build-demo

RUN go build

ENTRYPOINT ./knative-build-demo
EXPOSE 8080
```

Previously in Chapter 2, we built the container locally and pushed it to our container registry manually. However, Knative provides a great way to do these steps for us within our Kubernetes cluster using Builds. Like Configurations and Routes, Builds are also implemented as a Kubernetes Custom Resource Definition (CRD) that we define via YAML. Before we start digging down into each of the components, let's take a look at Example 3-6 to see what a Build configuration looks like.

Example 3-6. knative-build-demo/service.yaml

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: knative-build-demo
  namespace: default
spec:
  runLatest:
    configuration:
      build:
        serviceAccountName: build-bot
        source:
          git:
            url: https://github.com/gswk/knative-helloworld.git
            revision: master
        template:
          name: kaniko
          arguments:
          - name: IMAGE
            value: docker.io/gswk/knative-build-demo:latest
      revisionTemplate:
        spec:
          container:
            image: docker.io/gswk/knative-build-demo:latest
```

Prior to the Build's steps, we also see the source section where we define the location of our source code. Today, Knative ships with three options for sources:

git
: A Git repository that can optionally take an argument to define the branch, tag, or commit SHA.

gcs
: An archive located in Google Cloud Storage.

custom
: An arbitrary container image. This allows users to write their own sources so long as it places the source code in the `/workspace` directory.

There's only one additional component we'll need to install, which is the Build Template. We'll cover these more in depth in "Build Templates" on page 22, but for now, we'll go ahead and just install the one that we've defined to use in our YAML, which in this case is the Kaniko Build Template (see Example 3-7).

Example 3-7. Install the Kaniko Build Template

```
kubectl apply -f https://
  raw.githubusercontent.com/knative/build-templates/master
                             /kaniko/kaniko.yaml
```

With our template applied, we can deploy our Service configuration just like we have in our Serving examples (see Example 3-8).

Example 3-8. Deploy our application

```
kubectl apply -f knative-build-demo/service.yaml
```

This build will then run through the following steps:

1. Pull the code from the GitHub repo at *gswk/knative-helloworld*.
2. Build the container using the Dockerfile in the repo using the Kaniko Build Template (described in more detail in the next section).
3. Push the container to Docker Hub at *gswk/knative-build-demo* using the "build-bot" Service Account we set up earlier.
4. Deploy our application using the freshly built container.

Build Templates

In Example 3-6, we used a Build Template without ever actually explaining what a Build Template is or what it does. Simply, Build Templates are a sharable, encapsulated, parameterized collection of build steps. Today, Knative already supports several Build Templates, including:

Kaniko
: Build container images inside a running container without relying on running a Docker daemon.

Jib
: Build container images for Java applications.

Buildpacks
: Automatically detect the application's runtime and build a container image using Cloud Foundry Buildpacks.

While this isn't a comprehensive list of what's available, we can easily integrate new templates developed by the Knative community. Installing a Build Template is as easy as applying a YAML file as we would a Service, Route, or Build configuration:

```
kubectl apply -f https://raw.githubusercontent.com/knative/
build-templates/master/kaniko/kaniko.yaml
```

Then we can apply Example 3-6 as we would any other configuration to deploy our application and start sending requests to it like we did in Chapter 2:

```
kubectl apply -f knative-build-demo/service.yml

$ curl -H "Host: knative-build-demo.default.example.com"
http://$KNATIVE_INGRESS

Hello from Knative!
```

Let's take a closer look at a Build Template, continuing to use Kaniko as a reference in Example 3-9.

Example 3-9. https://github.com/knative/build-templates/blob/master/kaniko/kaniko.yaml

```
apiVersion: build.knative.dev/v1alpha1
kind: BuildTemplate
metadata:
  name: kaniko
spec:
  parameters:
  - name: IMAGE
    description: The name of the image to push
  - name: DOCKERFILE
    description: Path to the Dockerfile to build.
    default: /workspace/Dockerfile

  steps:
  - name: build-and-push
    image: gcr.io/kaniko-project/executor
    args:
    - --dockerfile=${DOCKERFILE}
    - --destination=${IMAGE}
```

The `steps` section of a Build Template has the exact same syntax as a Build does, only templated with named variables. In fact, we'll see that other than having our paths replaced with variables, the `steps` section looks very similar to the template section of Example 3-6. The `parameters` section puts some structure around the arguments that a Build Template expects. The Kaniko Build Template requires an `IMAGE` argument defining where to push the container image, but has an optional `DOCKERFILE` parameter and provides a default value if it's not defined.

Conclusion

We've seen that Builds in Knative remove quite a few manual steps when it comes to deploying your application. Additionally, Build Templates already provide a few great ways to build your code and remove the number of manually managed components. As time goes on, the potential for more and more Build Templates to be built and shared with the Knative community remains possibly one of the most exciting things to keep an eye on.

We've spent a lot of time on how we build and run our applications, but one of the biggest promises of serverless is that it makes it easy to wire your Services to Event Sources. In the next chapter we'll look at the Eventing component of Knative and all of the sources that are provided out of the box.

CHAPTER 4
Eventing

So far we've only sent basic HTTP requests to our applications, and that's a perfectly valid way to consume functions on Knative. However, the loosely coupled nature of serverless fits an event-driven architecture as well. That is to say, perhaps we want to invoke our function when a file is uploaded to an FTP server. Or, maybe any time we make a sale we need to invoke a function to process the payment and update our inventory. Rather than having our applications and functions worry about the logic of watching for these events, instead we can express interest in certain events and let Knative handle letting us know when they occur.

Doing this on your own would be quite a bit of work and implementation-specific coding. Luckily, Knative provides a layer of abstraction that makes it easy to consume events. Instead of writing code specific to your message broker of choice, Knative simply delivers an "event." Your application doesn't have to care where it came from or how it got there, just simply that it happened. To accomplish this, Knative introduces three new concepts: Sources, Channels, and Subscriptions.

Sources

Sources are, as you may have guessed, the source of the events. They're how we define where events are being generated and how they're delivered to those interested in them. For example, the Knative teams have developed a number of Sources that are provided right out of the box. To name just a few:

GCP PubSub
: Subscribe to a topic in Google's PubSub Service and listen for messages.

Kubernetes Events
: A feed of all events happening in the Kubernetes cluster.

GitHub
: Watches for events in a GitHub repository, such as pull requests, pushes, and creation of releases.

Container Source
: In case you need to create your own Event Source, Knative has a further abstraction, a Container Source. This allows you to easily create your own Event Source, packaged as a container. See "Building a Custom Event Source" on page 48.

While this is just a subset of current Event Sources, the list is quickly and constantly growing as well. You can see a current list of Event Sources in the Knative ecosystem in the Knative Eventing documentation.

Let's take a look at a simple demo that will use the Kubernetes Events Source and log them to STDOUT. We'll deploy a function that listens for POST requests on port 8080 and spits them back out, shown in Example 4-1.

Example 4-1. knative-eventing-demo/app.go

```
package main

import (
  "fmt"
  "io/ioutil"
  "log"
  "net/http"
)

func handlePost(rw http.ResponseWriter, req *http.Request) {
  defer req.Body.Close()
  body, _ := ioutil.ReadAll(req.Body)

  fmt.Fprintf(rw, "%s", body)
  log.Printf("%s", body)
}

func main() {
  log.Print("Starting server on port 8080...")
  http.HandleFunc("/", handlePost)
  log.Fatal(http.ListenAndServe(":8080", nil))
}
```

We'll deploy this Service just as we would any other, shown in Example 4-2.

Example 4-2. knative-eventing-demo/service.yaml

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: knative-eventing-demo
spec:
  runLatest:
    configuration:
      revisionTemplate:
        spec:
          container:
            image: docker.io/gswk/knative-eventing-demo:latest
```

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

So far, no surprises. We can even send requests to this Service like we have done in the previous two chapters:

```
    $ curl $SERVICE_IP -H "Host: knative-eventing-demo.default.
    example.com" -XPOST -d "Hello, Eventing"

    > Hello, Eventing
```

Next, we can set up the Kubernetes Event Source. Different Event Sources will have different requirements when it comes to configuration and authentication. The GCP PubSub source, for example, requires information to authenticate to GCP. For the Kubernetes Event Source, we'll need to create a Service Account that has permission to read the events happening inside of our Kubernetes cluster. Like we did in Chapter 3, we define this Service Account in YAML and apply it to our cluster, shown in Example 4-3.

Example 4-3. knative-eventing-demo/serviceaccount.yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: events-sa
  namespace: default
```

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  creationTimestamp: null
  name: event-watcher
rules:
- apiGroups:
  - ""
  resources:
  - events
  verbs:
  - get
  - list
  - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  creationTimestamp: null
  name: k8s-ra-event-watcher
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: event-watcher
subjects:
- kind: ServiceAccount
  name: events-sa
  namespace: default
```

```
    kubectl apply -f serviceaccount.yaml
```

With our "events-sa" Service Account in place, all that's left is to define our actual source, an instance of the Kubernetes Event Source in our case. An instance of an Event Source will run with specific configuration, in our case a predefined Service Account. We can see what our configuration looks like in Example 4-4.

Example 4-4. knative-eventing-demo/source.yaml

```
apiVersion: sources.eventing.knative.dev/v1alpha1
kind: KubernetesEventSource
metadata:
  name: k8sevents
spec:
  namespace: default
  serviceAccountName: events-sa
  sink:
    apiVersion: eventing.knative.dev/v1alpha1
    kind: Channel
    name: knative-eventing-demo-channel
```

Most of this is fairly straightforward. We define the kind of object we're creating as a *KubernetesEventSource*, give it the name *k8sevents,* and pass along some instance-specific configuration such as the namespace we should run in and the Service Account we should use. There is one new thing you may have noticed though, the *sink* configuration.

Sinks are a way of defining where we want to send events to and are a Kubernetes ObjectReference, or more simply, a way of addressing another predefined object in Kubernetes. When working with Event Sources in Knative, this will generally either be a Service (in case we want to send events directly to an application running on Knative), or a yet-to-be-introduced component, a Channel.

Channels

Now that we've defined a source for our events, we need somewhere to send them. While you can send events straight to a Service, this means it's up to you to handle retry logic and queuing. And what happens when an event is sent to your Service and it happens to be down? What if you want to send the same events to multiple Services? To answer all of these questions, Knative introduces the concept of Channels.

Channels handle buffering and persistence, helping ensure that events are delivered to their intended Services, even if that service is down. Additionally, Channels are an abstraction between our code and the underlying messaging solution. This means we could swap this between something like Kafka and RabbitMQ, but in neither case are we writing code specific to either. Continuing through our demo, we'll set up a Channel that we'll send all of our events, as shown in Example 4-5. You'll notice that this Channel matches the sink we defined in our Event Source in Example 4-4.

Example 4-5. knative-eventing-demo/channel.yaml

```
apiVersion: eventing.knative.dev/v1alpha1
kind: Channel
metadata:
  name: knative-eventing-demo-channel
spec:
  provisioner:
    apiVersion: eventing.knative.dev/v1alpha1
    kind: ClusterChannelProvisioner
    name: in-memory-channel

    kubectl apply -f channel.yaml
```

Here we create a Channel named `knative-eventing-demo-channel` and define the type of Channel we'd like to create, in this case an `in-memory-channel`. As mentioned before, a big goal of eventing in Knative is that it's completely abstracted away from the underlying infrastructure, and this means making the messaging service backing our Channels pluggable. This is done by implementations of the `ClusterChannelProvisioner`, a pattern for defining how Knative should communicate with our messaging services. Our demo uses the `in-memory-channel provisioner`, but Knative actually ships with a few options for backing services for our Channels as well:

in-memory-channel
: Handled completely in-memory inside of our Kubernetes cluster and does not rely on a separate running service to deliver events. Great for development but is not recommended to be used in production.

GCP PubSub
: Uses Google's PubSub hosted service to deliver messages, only needs access to a GCP account.

Kafka
: Sends events to a running Apache Kafka cluster, an open source distributed streaming platform with great message queue capabilities.

NATS
: Sends events to a running NATS cluster, an open source message system that can deliver and consume messages in a wide variety of patterns and configurations.

With these pieces in place, one question remains: How do we get our events from our Channel to our Service?

Subscriptions

We have our Event Source sending events to a Channel, and a Service ready to go to start processing them, but currently we don't have a way to get our events from our Channel to our Service. Knative allows us to define a Subscription for just this scenario. Subscriptions are the glue between Channels and Services, instructing Knative how our events should be piped through the entire system. Figure 4-1 shows an example of how Subscriptions can be used to route events to multiple applications.

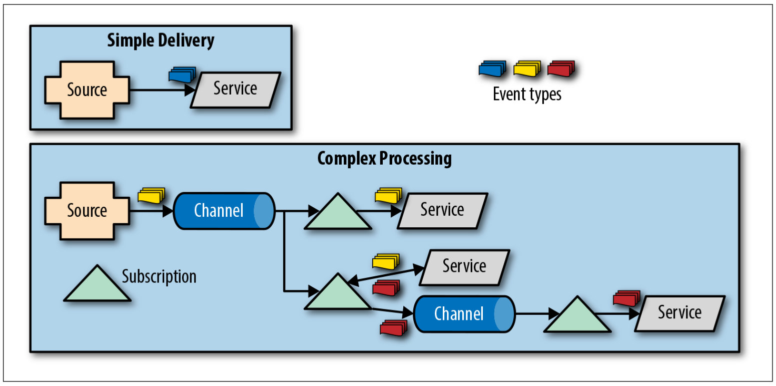


Figure 4-1. Event Sources can send events to a Channel so multiple Services can receive them simultaneously, or they can instead be sent straight to one Service

Services in Knative don't know or care how events and requests are getting to them. It could be an HTTP request from the ingress gateway or it could be an event sent from a Channel. Either way, our Service simply receives an HTTP request. This is an important decoupling in Knative that ensures that we're writing our code to our architecture, not our infrastructure. Let's create the Subscription that will deliver events from our Channel to our Service. As you can see in Example 4-6, the definition takes just two references, one to a Channel and one to a Service.

Example 4-6. knative-eventing-demo/subscription.yaml

```
apiVersion: eventing.knative.dev/v1alpha1
kind: Subscription
metadata:
  name: knative-eventing-demo-subscription
spec:
  channel:
    apiVersion: eventing.knative.dev/v1alpha1
    kind: Channel
    name: knative-eventing-demo-channel
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1alpha1
      kind: Service
      name: knative-eventing-demo
```

With this final piece in place, we now have all the plumbing in place to deliver events to our application. Kubernetes logs events occurring in the cluster, which our Event Source picks up and sends to our Channel and subsequently to our Service thanks to the Subscription that we defined. If we check out the logs in our Service, we'll see these events start coming across right away, as shown in Example 4-7.

Example 4-7. Retrieving the logs from our Service

```
$ kubectl get pods -l app=knative-eventing-demo-00001 -o name
pod/knative-eventing-demo-00001-deployment-764c8ccdf8-8w782

$ kubectl logs knative-eventing-demo-00001-deployment-f4c794667
  -mcrcv -c user-container
2019/01/04 22:46:41 Starting server on port 8080...
2019/01/04 22:46:44 {"metadata":{"name":"knative-eventing-demo-00001.
  15761326c1edda18","namespace":"default"...[Truncated for brevity]
```

Conclusion

These building blocks pave the way to help enable a rich, robust event-driven architecture, but this is just the beginning. We'll look at creating a custom source using the ContainerSource in "Building a Custom Event Source" on page 48. We'll also show Eventing in action in Chapter 7.

CHAPTER 5

Installing Knative

Before you can begin using Knative to build and host workloads, you need to install it. You should also run a few commands to validate that it's up and working as expected. This chapter walks through the necessary steps for installing and validating Knative from a Mac or Linux shell.

Standing Up a Knative Cluster

To start with, you'll need a Kubernetes cluster. Knative requires Kubernetes version 1.11 or newer. You must enable the MutatingAdmissionWebhook admission controller on the cluster. For a simple start, you can use Minikube on your local machine or get a cluster up and running in the cloud.

Why Are We Installing Istio?

We haven't discussed Istio at all so far, but all of a sudden it shows up as part of the install. What is it and why do we need it for Knative?

Istio is a service mesh that provides many useful features on top of Kubernetes including traffic management, network policy enforcement, and observability. We don't consider Istio to be a component of Knative but instead one of its dependencies, just as Kubernetes is. Knative ultimately runs atop a Kubernetes cluster *with Istio*.

The purpose of this report is not to detail the inner workings of Istio. Pretty much everything you need to know about Istio in order to work with Knative is covered in this chapter. If you're looking for more, start with "What is Istio?" and the Istio documentation.

NOTE

There are many options for standing up a Kubernetes cluster. Deciding which one to use depends on your requirements and level of comfort with the provider's specific set of tools. Reference Knative's installation documentation in the GitHub repository for certain provider-specific instructions.

On your local machine, make sure you have installed `kubectl` v1.11 or higher. Set the context to point it at the cluster you've set up for Knative. You will use `kubectl` to apply all the necessary CRDs in the form of YAML files.

Once your cluster is up, set up Istio using the following two commands:

```
kubectl apply --filename https://storage.googleapis.com/
knative-releases/serving/latest/istio.yaml

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

The first command imports all the necessary Istio objects into the cluster. The second part enables Istio injection in the `default` namespace. This ensures that Istio injects each Pod in this namespace with a sidecar at creation time. (You will notice that all Pods will have at least two containers as a result. One will be the `user-container`; one will be the `istio-proxy`.) Knative depends on the Istio components. Validate the Istio install with the following command until all the Pods show as `Running` or `Completed`:

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

Now that you've got your cluster running with Istio, you can begin installing Knative. Use the `kubectl` command to apply the Knative core components starting with Serving and Build. The YAML declarations are available from Google Storage or the Knative GitHub repo. You may target a specific version or reference the `latest` release. The following commands will apply the latest release from Google Storage:

```
kubectl apply --filename https://storage.googleapis.com/
knative-releases/serving/latest/release.yaml
```

```
kubectl apply --filename https://storage.googleapis.com/
knative-releases/build/latest/release.yaml
```

Again, validate that these objects were imported correctly. Monitor them with the following commands until all the Pods show as `Running`:

```
kubectl get pods --namespace knative-serving --watch

kubectl get pods --namespace knative-build --watch
```

TIP

Lightweight Install

If you are installing Knative on your local machine or just starting out, you may wish to install it without some of the built-in monitoring components. In this case, you can use the following command to install Serving instead:

```
kubectl apply --filename https://storage.
googleapis.com/knative-releases/serving/
latest/release-no-mon.yaml
```

This avoids installing any components in the monitoring namespace. For more information about what this provides, see the "Metrics and Logging" on page 63.

Once Serving and Build are up and running, follow similar steps to get the Eventing module going:

```
kubectl apply --filename https://storage.googleapis.com/
knative-releases/eventing/latest/release.yaml

kubectl get pods --namespace knative-eventing --watch
```

Finally, you may choose to install any Build Templates that are required. This step is exactly as you saw in Chapter 3. Here are the commands to install the templates for Kaniko and Buildpacks:

```
kubectl apply -f https://raw.githubusercontent.com/knative/
build-templates/master/kaniko/kaniko.yaml

kubectl apply -f https://raw.githubusercontent.com/knative/
build-templates/master/buildpack/buildpack.yaml
```

NOTE

If you plan to use the Build module to package your source into an image, you need a container registry to push it to. Along with installing Knative, consider setting up access to your container registry of choice. Publicly hosted options like Docker Hub or Google Container Registry work well, or you can set up your own private registry if you prefer.

Refer to Chapter 4 on the Build component for more information on accessing and pushing images to your registry.

You can validate that the Build Templates were installed successfully with the `kubectl get buildtemplates` command. This will return a list of all Build Templates installed in the `default` namespace:

```
$ kubectl get knative
NAME         AGE
buildpack    1m
kaniko       1m
```

Deleting Knative Objects

You may wish to remove certain Knative components after adding them. Any Knative objects can be removed from the cluster using the `kubectl delete` command. Just as you reference the YAML file with `kubectl apply`, you can do the same with `kubectl delete`:

```
kubectl delete -f https://raw.githubusercontent.com/knative/
build-templates/master/kaniko/kaniko.yaml
```

The Kubernetes cluster is up. Istio is installed. You've applied the Serving, Build, and Eventing components. You might have added a Build Template or two. You're almost ready to start using Knative. All that's left is to grab some information about how to reach it on the network.

TIP

Alternative Install Methods

The steps in this chapter show how to install the Knative components individually using the native kubectl apply commands. However, some serverless frameworks built on top of Knative may also include shortcuts to installing the system. For example, Project riff provides a one-liner for getting Knative installed and running on your Kubernetes cluster:

```
riff system install
```

This requires the riff CLI and a Kubernetes cluster already set up with kubectl pointing to the right context.

Accessing Your Knative Cluster

After your Knative cluster is all set up, you are ready to deploy apps to it. Except you'll need to know how to reach them. How do they get exposed on the cluster? Knative uses a LoadBalancer object in the istio-system namespace. Use the following command to get its external IP address listed as EXTERNAL-IP:

```
$ kubectl get svc istio-ingressgateway --namespace istio-system
NAME                    TYPE           CLUSTER-IP     EXTERNAL-IP
istio-ingressgateway   LoadBalancer   10.23.247.74   35.203.155.229
```

As you'll see in Chapter 6, this IP address, along with the correct HTTP Host Header, can be used to make requests to an app on the cluster. For easy reference, you may wish to set an KNATIVE_INGRESS environment variable for this external IP:

```
$ export KNATIVE_INGRESS=$(kubectl get svc istio-ingressgateway
    --namespace istio-system
    --output 'jsonpath={.status.loadBalancer.ingress[0].ip}')
$ echo $KNATIVE_INGRESS
35.203.155.229
```

Now, using a similar curl command to what we saw in Chapter 2, we can use this variable as follows to make a request to a Service hosted in our Knative environment:

```
curl -H "Host: my-knative-service-name.default.example.com"
http://$KNATIVE_INGRESS
```

No External Load Balancer?

If your Kubernetes instance has no external load balancer option (in the case of Minikube or a bare metal cluster), the command will be a little different because the `EXTERNAL-IP` field shows as `<pending>`. Use the following command to return the `NodeIP` and `NodePort` instead:

```
$ export KNATIVE_INGRESS=$(kubectl get node
   --output 'jsonpath={.items[0].status.addresses[0]
                      .address}'):
   $(kubectl get svc istio-ingressgateway
   --namespace istio-system
   --output 'jsonpath={.spec.ports[?(@.port==80)]
                      .nodePort}')

$ echo $KNATIVE_INGRESS
10.10.0.10:32380
```

Conclusion

Now that you've got everything set up, you're ready to deploy your apps to Knative! In Chapter 6, you will see some examples for different methods of doing so. You will also learn about more robust methods for exposing your cluster by setting a static IP and a custom domain, and configuring DNS.

CHAPTER 6

Using Knative

With a solid grasp on what components make up Knative, it's time to start looking at some more advanced topics. Serving offers quite a bit of flexibility in how to route traffic and there are other Build Templates that make building applications easy. It's even easy to make our own Event Source with just a few lines of code. In this chapter we'll take a deeper look at these features to see how we can make running our code on Knative even easier.

Creating and Running Knative Services

The concept of a Knative Service was introduced in Chapter 2. Recall that a Service in Knative is the combination of a single configuration plus a collection of Routes. Under the Knative and Kubernetes covers it is ultimately 0 or more containers within a Pod along with the components to make your application addressable. All of this is supported by a routing layer with robust options for traffic policies.

Whether you think of your workload as an application, container, or process, it runs as a Service in Knative. This offers flexibility to handle many scenarios, depending on which assets make up the software. This section provides another alternative that Knative provides you to build and deploy your software.

Using the Cloud Foundry Buildpack Build Template

You saw in Chapter 3 that the Kaniko Build Template lets you build container images using a Dockerfile. This method tasks you with the responsibility of writing and maintaining that Dockerfile. If you prefer to remove the burden of container-management altogether, you may wish to use a different Build Template. The Buildpack Build Template is responsible for the base image as well as bringing in all the dependencies required to build and run an application.

Cloud Foundry, an open source Platform-as-a-Service (PaaS), leverages buildpacks to ease development and operations alike. In Cloud Foundry, buildpacks will examine your source code to automatically determine what runtime and dependencies to download, build your code, and run your application. For example, with a Ruby application the buildpack will download the Ruby runtime and run a *bundle install* on the Gemfile to download all the required dependencies. The Java buildpack will download the JVM and any required dependencies for your application. This model is also available in Knative through the use of the Buildpack Build Template.

As with the Kaniko Build Template, you must bring the Buildpack Build Template CRD into the environment:

```
kubectl apply -f https://raw.githubusercontent.com/knative/
build-templates/master/buildpack/buildpack.yaml
```

When Should I Use Builds?

Knative Services rely only on the Serving component. The Build module is not required to deploy and run a Service in Knative. So why would you embed a Build in your Service? How do you know if it's a good idea for a given situation?

It's important to consider your software development lifecycle process. Do you have an existing, mature build pipeline for generating container images and pushing them to a registry? If so, you probably don't need Knative Build to do the work for you. If not, defining a Build method within your Knative Service will likely make things easier.

Deciding which Build Template to use also requires examining how you want to package up your code and dependencies. For heavy users of Docker with established processes for managing Dockerfiles, Kaniko is a great option. Users of Cloud Foundry or developers who prefer to write the code only and worry less about infrastructure will choose the Buildpack Build Template. Experienced Java users may already be familiar with Jib for building Java containers, making that the right choice. No matter your process, Knative offers some nice abstractions while allowing you to choose the method that works best for you.

In Knative, the Buildpack Build Template will use the same buildpacks that Cloud Foundry uses, including automatically detecting which buildpack to apply to your code. If you refer to Example 6-1 you'll see that much like the Kaniko Buildpack, you only define the location of the code and where to push the container image. The biggest difference is that there is no need to supply a Dockerfile. Instead, the Build Template knows how to build the container for this application.

Example 6-1. knative-buildpack-demo/service.yaml

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: knative-buildpack-demo
  namespace: default
spec:
  runLatest:
    configuration:
      build:
        serviceAccountName: build-bot
        source:
          git:
            url: https://github.com/gswk/knative-buildpack-demo.git
            revision: master
        template:
          name: buildpack
          arguments:
          - name: IMAGE
            value: docker.io/gswk/knative-buildpack-demo:latest
      revisionTemplate:
        spec:
          container:
            image: docker.io/gswk/knative-buildpack-demo:latest
```

Other than the path to the code and the container registry, the only difference from the Kaniko build in Example 3-6 is that the name of the template has been changed from `kaniko` to `buildpack`. In this example the git repository is a Node.js application `hello.js`, as well as a *package.json* file that defines the dependencies and metadata of the application. In this case the Build Template will download the Node runtime and npm executable, run `npm install`, and finally build the container and push it to Docker Hub.

TIP

Getting All Knative Objects

You may find that you have applied a lot of YAML files and aren't sure all of the Knative objects that have been created. There's a handy command to help you with just that. Simply type `kubectl get knative -n {NAMESPACE}` to return a categorized list of all Knative objects in a given namespace or use `kubectl get knative --all-namespaces` to return all Knative objects that exist on the cluster.

Deployment Considerations

Knative also offers different methods of deployment depending on what scenario best fits your Service. We showed in Chapter 2 how a Knative Route can be used to send traffic to a specific Revision. The following sections will detail how Knative Routes allow for blue-green deployments and incremental rollouts.

Zero Downtime Deployments

In Chapter 2, you saw how to point a single Route to multiple Revisions and how it enables zero downtime deployments. Since Revisions are immutable and multiple versions can be running at once, it's possible to bring up a new Revision while serving traffic to the old version. Then, once you are ready to direct traffic to the new Revision, update the Route to instantly switch over. This is sometimes referred to as a *blue-green deployment*, with blue and green representing the different versions.

Example 6-2 revisits the Route definition from Chapter 2.

Example 6-2. All traffic is routed to Revision 00001

```
apiVersion: serving.knative.dev/v1alpha1
kind: Route
metadata:
  name: knative-helloworld
  namespace: default
spec:
  traffic:
  - revisionName: knative-routing-demo-00001
    name: v1
    percent: 100
```

As mentioned in Chapter 2, you can access Revision `knative-routing-demo-00001` at both `knative-helloworld.default.example.com` and `v1.knative-helloworld.default.example.com`. Let's consider a scenario where you've added a few new features or fixed a few bugs in your code, then built and pushed it up to Knative. This results in a new Revision named `knative-routing-demo-00002`. However, before you start sending live traffic to the application, we want to ensure it's up and running correctly. In Example 6-3, there is a new Route named *v2*, but no live traffic routed to it.

Example 6-3. A new Route named "v2" for our new Revision

```
apiVersion: serving.knative.dev/v1alpha1
kind: Route
metadata:
  name: knative-helloworld
  namespace: default
spec:
  traffic:
  - revisionName: knative-routing-demo-00001
    name: v1
    percent: 100
  - revisionName: knative-routing-demo-00002
    name: v2
    percent: 0
```

These Revisions have been named as *v1* and *v2* (though you may choose any name such as *blue* and *green*). This means you can access the new Revision at `v2.knative-helloworld.default.example.com` but the live Route will still only send traffic to Revision 00001. Before changing the traffic over, access the new Revision and test it to make sure it's ready for production traffic. When the new Revision is ready to receive live traffic, update the Route one more time as in Example 6-4.

Example 6-4. Send all live traffic to our new Revision

```
apiVersion: serving.knative.dev/v1alpha1
kind: Route
metadata:
  name: knative-helloworld
  namespace: default
spec:
  traffic:
  - revisionName: knative-routing-demo-00002
    name: v2
    percent: 100
```

Once applied, the new configuration traffic will instantly start routing to Revision 00002 without any downtime. Discover a new bug in your code and need to roll back? It's just as easy to update the Route configuration again to point back to the original Revision. Since Revisions are immutable but simple, lightweight YAML configurations, Knative will store past Revisions and you may route to them at any time. That includes the reference to a particular container image, configuration, and any build information related to the Revision.

Incremental Rollouts

Another pattern for deployment supported by Knative Routes is to deploy a new version of code incrementally. This could be used for A-B testing or for rolling out features to a subset of users before unleashing it for everyone. In Knative, this is achieved by using percentage-based routing.

Though similar to the blue-green deployment example in Example 6-4, you can see in Example 6-5 that instead of routing 0% of the traffic to v2, we have evenly split the load across both v1 and v2. You may also choose to use a different split like 80-20 or even split across three Revisions. Each Revision is still accessible via the named subdomain, but user traffic will be split as indicated by the percent values.

Example 6-5. Partial load routing

```
apiVersion: serving.knative.dev/v1alpha1
kind: Route
metadata:
  name: knative-helloworld
  namespace: default
```

```
spec:
  traffic:
  - revisionName: knative-routing-demo-00001
    name: v1
    percent: 50
  - revisionName: knative-routing-demo-00002
    name: v2
    percent: 50
```

Custom Domains

Every example covered thus far has used a generic example domain. This is not a great URL to use for production applications. Not only that, but it's not possible to route to *example.com*. Thankfully, Knative provides the option to use a custom domain. Out of the box, Knative uses the `{route}.{namespace}.{domain}` scheme for every Route and a default domain of *example.com*.

Using the `knative-custom-domain` example as a starting place shown in Example 6-6, by default it receives a Route of *knative-custom-domain.default.example.com*.

Example 6-6. knative-custom-domain/configuration.yaml

```
apiVersion: serving.knative.dev/v1alpha1
kind: Configuration
metadata:
  name: knative-custom-domain
  namespace: default
spec:
  revisionTemplate:
    spec:
      container:
        image: docker.io/gswk/knative-helloworld
        imagePullPolicy: Always
```

Since we've defined this as a Configuration rather than a Service, we'll also need to define a Route to our application, as shown in Example 6-7. Having these two configurations separate will afford us a higher level of customization such as those we say when discussing zero downtime deployments, but will also let us update our domain and Route without needing to redeploy the entire application.

Example 6-7. knative-custom-domain/route.yaml

```
apiVersion: serving.knative.dev/v1alpha1
kind: Route
metadata:
  name: knative-custom-domain
  namespace: default
spec:
  traffic:
  - revisionName: knative-custom-domain-00001
    name: v1
    percent: 100
```

As expected, this will create a Service and make a Route at *knative-custom-domain.default.example.com*. Now take a look at how to change the domain in the default URL scheme from *example.com* to something you can actually route to. This example uses a subdomain for this book's website, *dev.gswkbook.com*. This can be easily done by updating the `config-domain` ConfigMap, which is configured by Knative by default, as shown in Example 6-8.

Example 6-8. knative-custom-domain/domain.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-domain
  namespace: knative-serving
data:
  dev.gswkbook.com: ""
```

Knative should eventually reconcile the changes made to the domain. Re-creating the Route will speed up the process:

```
$ kubectl delete -f route.yaml
```

```
$ kubectl apply -f route.yaml
```

Taking a look at the route after the change, you'll see that it is now given the updated URL of *knative-custom-domain.default.dev.gswkbook.com*. If your ingress gateway is publicly accessible (i.e., set up on Google's GKE or a similar managed Kubernetes offering), you can create a wildcard DNS entry for **.dev.gswkbook.com* to route to your Knative install and access your Services and functions directly over the internet, as shown in Example 6-9.

```
$ kubectl get route knative-custom-domain -oyaml
```

Example 6-9. knative-custom-domain Route represented as YAML

```
apiVersion: serving.knative.dev/v1alpha1
kind: Route
metadata:
  ...
status:
 ...
  domain: knative-custom-domain.default.dev.gswkbook.com
  ...
  traffic:
  - name: v1
    percent: 100
    revisionName: knative-custom-domain-00001
```

You may also want to have different domains for different deployments. For example, by default you may want everything to be deployed to your development domain, and then after testing promote it to a production domain. Knative provides a simple mechanism for enabling this, allowing you to define multiple domains and label Routes to determine which domain they're placed on. Let's set up another domain used for production, **.prod.gswkbook.com*, by updating the config-domain ConfigMap one more time, as shown in Example 6-10.

Example 6-10. knative-custom-domain/domain.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-domain
  namespace: knative-serving
data:
  prod.gswkbook.com: |
    selector:
      environment: prod
  dev.gswkbook.com: ""
```

In Example 6-11, we've defined that any Route with the label `environment: prod` will be placed on the *prod.gswkbook.com* domain, otherwise it will be placed on the *dev.gswkbook.com* domain by default. All you need to do then to move your application to this new domain is update your Route with this new label in the `metadata` section of your config.

Example 6-11. knative-custom-domain/route-label.yaml

```
apiVersion: serving.knative.dev/v1alpha1
kind: Route
metadata:
  name: knative-custom-domain
  namespace: default
  labels:
    environment: prod
spec:
  traffic:
  - revisionName: knative-custom-domain-00001
    name: v1
    percent: 100
```

Once applied, your Route is automatically updated and, assuming your DNS is properly configured, immediately accessible at the new domain. You can verify this by ensuring the `domain` entry matches when retrieving the Route CRD, as shown in Example 6-12.

```
    kubectl describe route knative-custom-domain
```

Example 6-12. knative-custom-domain Route description

```
Name:          knative-custom-domain
Namespace:     default
...
Kind:          Route
Metadata:
  ...
Spec:
  ...
Status:
  ...
  Domain:          knative-custom-domain.default.prod.gswkbook.com
  Domain Internal: knative-custom-domain.default.svc.cluster.local
  ...
```

Building a Custom Event Source

Let's say that we want our application to receive events from a source that we don't have an Event Source for. For example, maybe we want to periodically check a fileserver for new files, or make a request to a URL to watch for changes. It's pretty easy to put together some code to do this, but then what's the best way to run it? It doesn't make sense to run this like we would other Knative applications since it needs to run perpetually, but if we run it outside of Knative, then we would have new components that we need to manually manage and configure.

Knative addresses this by making it easy to create your own Event Source using the ContainerSource Event Source. Using this Event Source, we provide Knative a container, and Knative will provide the container a URL to POST events to. We can write our Event Source in any language we like so long as we meet just a couple of requirements:

1. It can be packaged up as a container and has an ENTRYPOINT defined so it knows how to run our code.
2. It expects to take in a `--sink` CLI flag, which Knative will provide along with a URL to the configured destination.

Let's take a look at how this works by building an Event Source that will emit the current time on a given interval, called the time-event-source. First, let's look at the code for our Event Source in Example 6-13.

Example 6-13. time-event-source/app.rb

```
require 'optparse'
require 'net/http'
require 'uri'

# Default CLI options if omitted
options = {
    :sink => "http://localhost:8080",
    :interval => 1
}

# Parse CLI flags
opt_parser = OptionParser.new do |opt|
    opt.on("-s", "--sink SINK", "Location to send events")
        do |sink| options[:sink] = sink
    end

    opt.on("-i", "--interval INTERVAL", "Poll frequency")
        do |interval| options[:interval] = interval.to_i
    end
end
opt_parser.parse!

# Send the current time on the given interval
# to the given sink
uri = URI.parse(options[:sink])
```

```
header = {'Content-Type': 'text/plain'}
loop do
    http = Net::HTTP.new(uri.host, uri.port)
    request = Net::HTTP::Post.new(uri.request_uri, header)
    request.body = Time.now.to_s
    response = http.request(request)
    sleep options[:interval]
end
```

This Ruby app, after parsing the CLI flags, enters an infinite loop that continually POSTs the current time to the URL provided by the `--sink` flag, and then sleeps for the number of seconds provided by the `--interval` flag. Since this needs to be packaged up as a container, we also have a Dockerfile that we use to build it, shown in Example 6-14.

Example 6-14. time-event-source/Dockerfile

```
FROM ruby:2.5.3-alpine3.8

ADD . /time-event-source
WORKDIR /time-event-source

ENTRYPOINT ["ruby", "app.rb"]
```

There's not much surprising here. We use the official Ruby image as a base, add our code, and define how to run our code. We can build our container and send it up to Docker Hub. Before we run our Event Source, we need a place to send events. We've gone ahead and built a very simple Knative app that logs the body of all HTTP POST requests that it receives. We'll apply the YAML shown in Example 6-15 to deploy it, named `logger`.

Example 6-15. time-event-source/service.yaml

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: logger
spec:
  runLatest:
    configuration:
      revisionTemplate:
        spec:
          container:
            image: docker.io/gswk/logger:latest
```

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

All that's left then is to get our Event Source running in Knative. The YAML for this follows the same outline as other Event Sources, which we can see in Example 6-16.

Example 6-16. time-event-source/source.yaml

```
apiVersion: sources.eventing.knative.dev/v1alpha1
kind: ContainerSource
metadata:
  labels:
    controller-tools.k8s.io: "1.0"
  name: time-eventsource
spec:
  image: docker.io/gswk/time-event-source:latest
  args:
   - '--interval=1'
  sink:
    apiVersion: serving.knative.dev/v1alpha1
    kind: Service
    name: logger
```

There's a couple of things to take note of here. First, we've provided Knative the location of our Event Source container in the `image` argument, much like we do when we deploy a Service. Second, we've provided the `--interval` flag in the `args` array, but we've omitted the `--sink` flag. This is because Knative will look at the `sink` that we've provided (in this case, our `logger` Service), look up the URL to that resource, and automatically supply it to our Event Source. This means that, internal to our Event Source container, in the end our code is invoked with a command that will look similar to the following:

```
ruby app.rb
--interval=1
--sink=http://logger.default.example.com
```

We'll deploy it with the usual `kubectl apply` command that we've come to expect:

```
kubectl apply -f source.yaml
```

Soon, we'll see a new Pod spun up, but with the important difference that it will run perpetually rather than scaling up from and down to zero. We can take a look at the logs from our *logger* Service to verify that our events are being sent as expected, shown in Example 6-17.

Example 6-17. Retrieving the logs from our logger Service

```
$ kubectl get pods -l app=logger-00001 -o name pod/logger-00001-
  deployment-57446ffb59-vzg97

$ kubectl logs logger-00001-deployment-57446ffb59-vzg97 \
    -c user-container
2018/12/26 21:12:59 Starting server on port 8080...
[Wed 26 Dec 2018 21:13:00 UTC] 2018-12-26 21:13:00 +0000
[Wed 26 Dec 2018 21:13:01 UTC] 2018-12-26 21:13:01 +0000
[Wed 26 Dec 2018 21:13:02 UTC] 2018-12-26 21:13:02 +0000
[Wed 26 Dec 2018 21:13:03 UTC] 2018-12-26 21:13:03 +0000
[Wed 26 Dec 2018 21:13:04 UTC] 2018-12-26 21:13:04 +0000
[Wed 26 Dec 2018 21:13:05 UTC] 2018-12-26 21:13:05 +0000
```

This simple abstraction allows us to quickly and easily provide our own Event Sources. Not only that, but much like Build Templates, you can envision how these Event Sources can be easily shared with the Knative community since they're so easy to plug into your environment. We'll also look at another custom Event Source in Chapter 7.

Conclusion

We've covered quite a bit of ground so far, from basic fundamentals to advanced use cases, but we've only looked at these concepts on their own, showing off one feature at a time. If you're like us though, what really helps is seeing it all put together in a real-world example and observing how each component interacts with each other. In the next chapter, we'll do just that and build an application that leverages much of what we've learned!

CHAPTER 7

Putting It All Together

Let's put everything we've learned to use and build something! We've put together a demo that leverages much of what you've learned and made a Service to visualize earthquake activity throughout the world by consuming the USGS Earthquake data feed. You can find the code that we'll be walking through in the gswk/earthquake-demo GitHub repository.

The Architecture

Before we dive into the code, let's take a look at the architecture of our application, shown in Figure 7-1. We're building three important things here: An Event Source, a Service, and a frontend.

Each of these components inside of Knative in the diagram represents something we'll build leveraging something we've learned so far, including a Service that will use the Kaniko Build Template and a custom Event Source that will poll for our data:

USGS Event Source
: We'll build a custom ContainerSource Event Source that will poll the data provided by the USGS at a given interval. Packaged up as a prebuilt container image.

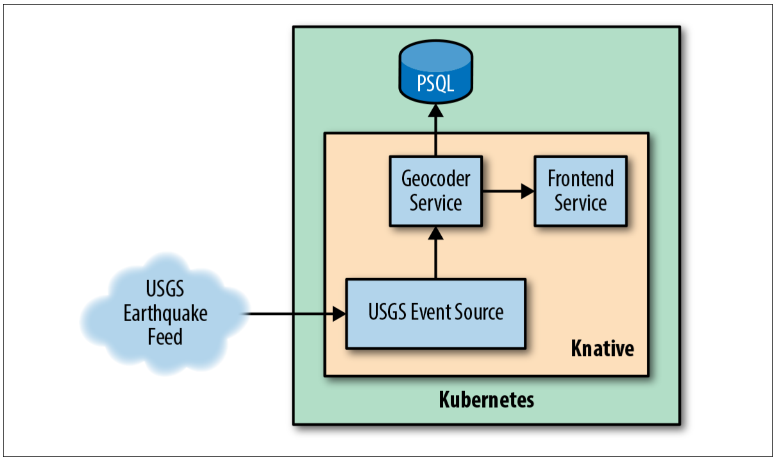


Figure 7-1. Architecture of our application. Events come in from the USGS Earthquake feed into our Event Source, which triggers our Geocoder Service to persist events. Our frontend will also use our Geocoder Service to query for recent events.

Geocoder Service
: This will provide an endpoint for the Event Source to POST events to, and will use the provided coordinates to look up the address. It will also provide an endpoint for the frontend to query and retrieve recent events. We'll use the Build Service to build our container image. Communicates with a Postgres database running on Kubernetes.

Frontend
: A lightweight persistently running frontend to visualize recent earthquake activity.

We can set up a Postgres database on our Kubernetes cluster easily using Helm, a tool that makes it easy to package and share application packages on Kubernetes. Make sure to refer to Helm's documentation for instructions on how to get up and running on your Kubernetes cluster. If you're running on Minikube or a Kubernetes cluster without any specific permissions requirements, you can set up Helm with the simple command:

```
$ helm init
```

For clusters such as Google's GCP with deeper security configurations, refer to the Helm Quickstart Guide. Next we can set up a Postgres database and pass it some configuration parameters to make setup a little easier:

```
$ helm install
--name geocodedb
--set postgresqlPassword=devPass,postgresqlDatabase
                        =geocode stable/postgresql
```

This will create a Postgres database in our Kubernetes cluster, set the user password to `devPass` and create a database named `geocode`. We've named our Postgres server `geocodedb`, which means from inside our Kubernetes cluster we can reach this server at *geocodedb-postgresql.default.svc.cluster.local*. Now let's dive into the code!

Geocoder Service

As shown in the architecture diagram, both our Event Source and our frontend will be sending requests to our Geocoder service, which will be communicating with our Postgres database. This puts our Service right in the heart of our application. HTTP POST requests to our Service will record events to the database and GET requests will retrieve the events that occurred over the last 24 hours. Check out the code for our Service in Example 7-1.

Example 7-1. geocoder/app.rb

```
require 'geocoder'
require 'json'
require 'pg'
require 'sinatra'

set :bind, '0.0.0.0'

# DB connection credentials are passed via environment
# variables
DB_HOST = ENV["DB_HOST"] || 'localhost'
DB_DATABASE = ENV["DB_DATABASE"] || 'geocode'
DB_USER = ENV["DB_USER"] || 'postgres'
DB_PASS = ENV["DB_PASS"] || 'password'

# Connect to the database and create table if it doesn't exist
conn = PG.connect( dbname: DB_DATABASE, host: DB_HOST,
    password: DB_PASS, user: DB_USER)
conn.exec "CREATE TABLE IF NOT EXISTS events (
    id varchar(20) NOT NULL PRIMARY KEY,
    timestamp timestamp,
    lat double precision,
    lon double precision,
    mag real,
    address text
);"

# Store an event
post '/' do
    d = JSON.parse(request.body.read.to_s)
    address = coords_to_address(d["lat"], d["long"])
    id = d["id"]

    conn.prepare("insert_#{id}",
        'INSERT INTO events VALUES ($1, $2, $3, $4, $5, $6)')
    conn.exec_prepared("insert_#{id}", [d["id"], d["time"],
        d["lat"], d["long"], d["mag"], address.to_json])
end

# Get all events from the last 24 hours
get '/' do
    select_statement = "select * from events where
        timestamp > 'now'::timestamp - '24 hours'::interval;"
    results = conn.exec(select_statement)
    jResults = []
    results.each do |row|
        jResults << row
    end

    content_type 'application/json'
    headers 'Access-Control-Allow-Origin' => "*"
    return jResults.to_json
end

# Get the address from a given set of coordinates
def coords_to_address(lat, lon)
    coords = [lat, lon]
    results = Geocoder.search(coords)

    a = results.first
    address = {
        address: a.address,
        house_number: a.house_number,
        street: a.street,
        county: a.county,
        city: a.city,
        state: a.state,
        state_code: a.state_code,
        postal_code: a.postal_code,
        country: a.country,
        country_code: a.country_code,
        coordinates: a.coordinates
    }
    return address
end
```

We'll have Knative build our container image for us, pass it the information needed to connect to our Postgres database, and run our Service. We can see the how this is all set up in Example 7-2.

Example 7-2. earthquake-demo/geocoder-service.yaml

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: geocoder
  namespace: default
spec:
  runLatest:
    configuration:
      build:
        serviceAccountName: build-bot
        source:
          git:
           url: https://github.com/gswk/geocoder.git
           revision: master
        template:
          name: kaniko
          arguments:
          - name: IMAGE
            value: docker.io/gswk/geocoder
      revisionTemplate:
        spec:
          container:
           image: docker.io/gswk/geocoder
           env:
           - name: DB_HOST
             value: "geocodedb-postgresql.default.svc.cluster.local"
           - name: DB_DATABASE
             value: "geocode"
           - name: DB_USER
             value: "postgres"
           - name: DB_PASS
             value: "devPass"
```

```
    kubectl apply -f earthquake-demo/geocoder-service.yaml
```

Since we've passed all of the connection info required to connect to our Postgres database as environment variables, this is all we'll need to get our Service running. Next, we'll get our Event Source up and running so that we can start sending events to our newly deployed Service.

USGS Event Source

Our Event Source will be responsible for polling the feed of USGS earthquake activity on a given interval, parse it, and send it to our defined sink. Since we need to poll our data and don't have the option to have it pushed to us, this makes it a great candidate to write a custom Event Source using the ContainerSource.

Before we set up our Event Source, we'll also need a Channel to send events to. While we could send events from our Event Source straight to our Service, this will give us some flexibility in the future in case we ever want to send events to another Service. We just need a simple Channel, which we'll define in Example 7-3.

Example 7-3. earthquake-demo/channel.yaml

```
apiVersion: eventing.knative.dev/v1alpha1
kind: Channel
metadata:
  name: geocode-channel
spec:
  provisioner:
    apiVersion: eventing.knative.dev/v1alpha1
    kind: ClusterChannelProvisioner
    name: in-memory-channel
```

```
    $ kubectl apply -f earthquake-demo/channel.yaml
```

Just like when we built a custom Event Source in Chapter 6, ours is made up of a script, in this case a Ruby script, that takes in two command-line flags: `--sink` and `--interval`. Let's take a look at this in Example 7-4.

Example 7-4. usgs-event-source/usgs-event-source.rb

```
require 'date'
require "httparty"
require 'json'
require 'logger'
require 'optimist'

$stdout.sync = true
@logger = Logger.new(STDOUT)
```

```
@logger.level = Logger::DEBUG

# Poll the USGS feed for real-time earthquake readings
def pull_hourly_earthquake(lastTime, sink)
    # Get all detected earthquakes in the last hour
    url = "https://earthquake.usgs.gov/earthquakes/feed/v1.0/" \
    + "summary/all_hour.geojson"
    response = HTTParty.get(url)
    j = JSON.parse(response.body)

    # Keep track of latest recorded event, reporting all
    # if none have been tracked so far
    cycleLastTime = lastTime

    # Parse each reading and emit new ones as events
    j["features"].each do |f|
        time = f["properties"]["time"]

        if time > lastTime
            msg = {
                time: DateTime.strptime(time.to_s,'%Q'),
                id: f["id"],
                mag: f["properties"]["mag"],
                lat: f["geometry"]["coordinates"][1],
                long: f["geometry"]["coordinates"][0]
            }

            publish_event(msg, sink)
        end

        # Keep track of latest reading
        if time > cycleLastTime
            cycleLastTime = time
        end
    end

    lastTime = cycleLastTime
    return lastTime
end

# POST event to provided sink
def publish_event(message, sink)
    @logger.info("Sending #{message[:id]} to #{sink}")
    puts message.to_json
    r = HTTParty.post(sink,
        :headers => {'Content-Type'=>'text/plain'},
        :body => message.to_json)

    if r.code != 200
        @logger.error("Error! #{r}")
    end
```

```
end


# Parse CLI flags
opts = Optimist::options do
    banner <<-EOS
Poll USGS Real-Time Earthquake data

Usage:
  ruby usgs-event-source.rb

EOS
    opt :interval, "Poll Frequenvy",
        :default => 10
    opt :sink, "Sink to send events",
        :default => "http://localhost:8080"
end

# Begin polling USGS data
lastTime = 0
@logger.info("Polling every #{opts[:interval]} seconds")
while true do
    @logger.debug("Polling . . .")
    lastTime = pull_hourly_earthquake(lastTime, opts[:sink])
    sleep(opts[:interval])
end
```

As usual, Knative will handle providing the `--sink` flag when run as a ContainerSource Event Source. We've provide an additional flag named `--interval` that we'll define ourselves since we've written our code to allow users to define their own polling interval. The script is packaged as a Docker container and uploaded to Dockerhub at gswk/usgs-event-source. All that's left is to create our source's YAML shown in Example 7-5 and create our subscription to send events from the Channel to our Service shown in Example 7-6.

Example 7-5. earthquake-demo/usgs-event-source.yaml

```yaml
apiVersion: sources.eventing.knative.dev/v1alpha1
kind: ContainerSource
metadata:
  labels:
    controller-tools.k8s.io: "1.0"
  name: usgs-event-source
spec:
  image: docker.io/gswk/usgs-event-source:latest
  args:
    - "--interval=10"
  sink:
    apiVersion: eventing.knative.dev/v1alpha1
    kind: Channel
    name: geocode-channel
```

```
    $ kubectl apply -f earthquake-demo/usgs-event-source.yaml
```

Once we apply this YAML, the Event Source will spin up a persistently running container that will poll for events and send them to the Channel we've created. Additionally, we'll need to hook our Geocoder Service up to the Channel.

Example 7-6. earthquake-demo/subscription.yaml

```
apiVersion: eventing.knative.dev/v1alpha1
kind: Subscription
metadata:
  name: geocode-subscription
spec:
  channel:
    apiVersion: eventing.knative.dev/v1alpha1
    kind: Channel
    name: geocode-channel
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1alpha1
      kind: Service
      name: geocoder
```

```
    $ kubectl apply -f earthquake-demo/subscription.yaml
```

With this subscription created, we've wired everything up to bring our events into our environment with our custom Event Source and then send them to our Service, which will persist them in our Postgres database. We have one final piece to deploy, which is our frontend to visualize everything.

Frontend

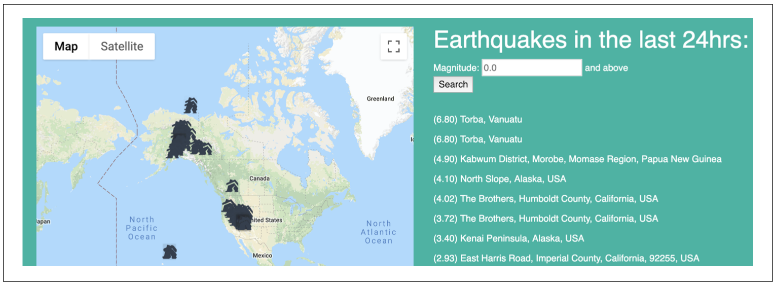
Finally we need to put together our frontend to visualize all the data we've collected. We've put together a simple website and packaged it in a container that will serve it using Nginx. When the page is loaded, it will make a call to our Geocoder Service, return an array of earthquake events including coordinates and magnitude, and visualize them on our map. We'll also set this up as a Knative Service so we get things like easy routing and metrics for free. Again, we'll write up our YAML like other Knative Services and use the Kaniko Build Template, shown in Example 7-7.

Example 7-7. earthquake-demo/frontend/frontend-service.yaml

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: earthquake-demo
  namespace: default
spec:
  runLatest:
    configuration:
      build:
        serviceAccountName: build-bot
        source:
          git:
            url: https://github.com/gswk/
              earthquake-demo-frontend.git
            revision: master
        template:
          name: kaniko
          arguments:
          - name: IMAGE
            value: docker.io/gswk/earthquake-demo-frontend
      revisionTemplate:
        spec:
          container:
            image: docker.io/gswk/earthquake-demo-frontend
            env:
            - name: EVENTS_API
              value: "http://geocoder.default.svc.cluster.local"
```

```
    $ kubectl apply -f earthquake-demo/frontend-service.yaml
```

We define the `EVENTS_API` environment variable, which our front-end will use to know where our Geocoder Service is. With this last piece in place, we have our whole system up and running! Our application is shown in action in Figure 7-2.



Earthquakes in the last 24hrs:

Magnitude: 0.0 and above

Search

(6.80) Torba, Vanuatu
(6.80) Torba, Vanuatu
(4.90) Kabwum District, Morobe, Momase Region, Papua New Guinea
(4.10) North Slope, Alaska, USA
(4.02) The Brothers, Humboldt County, California, USA
(3.72) The Brothers, Humboldt County, California, USA
(3.40) Kenai Peninsula, Alaska, USA
(2.93) East Harris Road, Imperial County, California, 92255, USA

Figure 7-2. Our demo application up and running!

As requests come into our frontend application it will pull events from the Geocoder Service, and as new events come in, they'll be picked up by our custom Event Source. Additionally, Knative provides a few additional tools to help you keep your apps and Services up and running by providing some great insight, with built-in logging, metrics, and tracing.

Metrics and Logging

Anyone who's ever run code in production knows that our story isn't done. Just because our code is written and our application deployed, there's an ongoing responsibility for management and operations. Having proper insight into what your code is doing with logs and metrics is integral to that operational process and luckily Knative ships with a number of tools to provide that information. Even better, much of it is tied into your code automatically without you needed to do anything special.

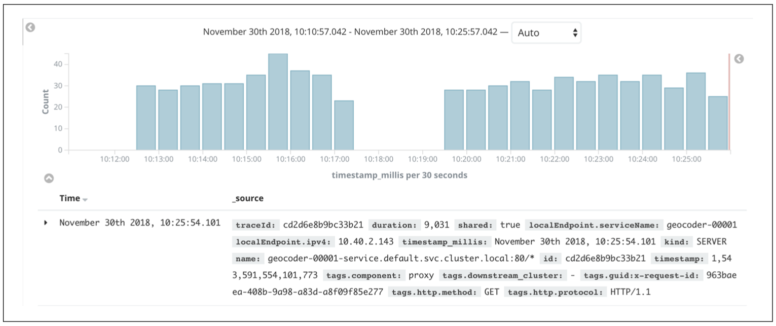
Let's start with digging into the logs of our Geocoder Service, which are provided by Kibana, installed when we set up our Serving components of Knative. Before we can access anything, we need to setup a proxy into our Kubernetes cluster, easily done with a single command:

```
$ kubectl proxy
```

This will open up a proxy into our entire Kubernetes cluster and make it accessible on port 8001 of our machine. This includes Kibana, which we can reach at *http://localhost:8001/api/v1/namespaces/knative-monitoring/services/kibana-logging/proxy/app/kibana*.

We'll need to provide an indexing pattern, which for now we can simply provide `*` and a time filter of `timestamp_millis`. Finally, if we go to the Discover tab in Kibana, we'll see every log going through our system! Let's take a look at the requests going to our Geocoder Service with the following search term and its results, as shown in Figure 7-3.

```
localEndpoint.serviceName = geocoder
```



```
November 30th 2018, 10:25:54.101   traceId: cd2d6e8b9bc33b21 duration: 9,031 shared: true localEndpoint.serviceName: geocoder-00001
                                   localEndpoint.ipv4: 10.40.2.143 timestamp_millis: November 30th 2018, 10:25:54.101 kind: SERVER
                                   name: geocoder-00001-service.default.svc.cluster.local:80/* id: cd2d6e8b9bc33b21 timestamp: 1,54
                                   3,591,554,101,773 tags.component: proxy tags.downstream_cluster: - tags.guid:x-request-id: 963bae
                                   ea-408b-9a98-a83d-a8f09f85e277 tags.http.method: GET tags.http.protocol: HTTP/1.1
```

Figure 7-3. Kibana dashboard with logs from our Geocoder Service

What about at-a-glance metrics though? Seeing how certain metrics like failed requests versus response time can offer clues into solving issues with our applications. Knative also helps us out here by shipping with Grafana and delivering an absolute boatload of metrics—everything from distribution of response codes to how much CPU our Services are using. Knative even includes a dashboard to visualize current cluster usage to help with capacity planning. Before we load up Grafana, we'll need to forward another port into our Kubernetes cluster with the following command:

```
$ kubectl port-forward
    --namespace knative-monitoring $(kubectl get pods
    --namespace knative-monitoring
    --selector=app=grafana
    --output=jsonpath="{.items..metadata.name}") 3000
```

Once forwarded, we can access our dashboards at *http://localhost:3000*. In Figure 7-4, we can see the graphs for requests sent to our Geocoder Service, looking nice and healthy!

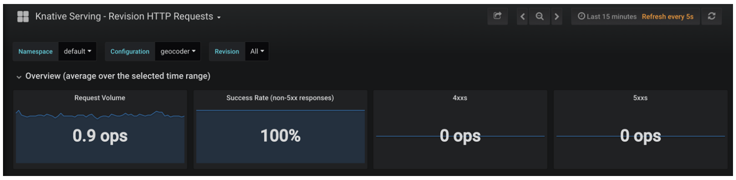
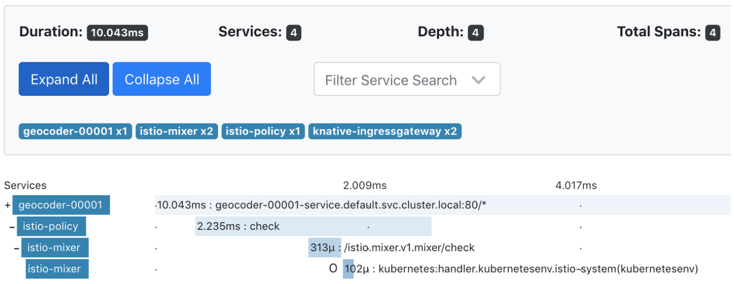

Knative Serving - Revision HTTP Requests
Namespace default
Configuration geocoder
Revision All
Last 15 minutes Refresh every 5s
Overview (average over the selected time range)
Request Volume
0.9 ops
Success Rate (non-5xx responses)
100%
4xxs
0 ops
5xxs
0 ops

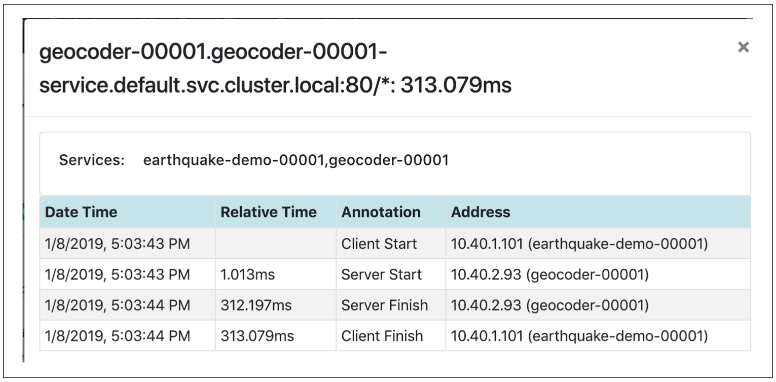
Figure 7-4. Graphs showing successful versus failed requests to our Geocoder Service

Finally, Knative also ships with Zipkin to help trace our requests. As they come in through our ingress gateway and travel all the way down to the database, with some simple instrumentation we can get a great look inside our application. With our proxy from earlier still set up, we can access Zipkin at *http://localhost:8001/api/v1/namespaces/istio-system/services/zipkin:9411/proxy/zipkin*. Once in, we can see how GET requests to our Geocoder service flow through it, shown in Figures 7-5 and 7-6.



```
10.043ms : geocoder-00001-service.default.svc.cluster.local:80/*
2.235ms : check
313μ : /istio.mixer.v1.mixer/check
102μ : kubernetes:handler.kubernetesenv.istio-system(kubernetesenv)
```

Figure 7-5. Simple trace of a request to our Geocoder Service



geocoder-00001.geocoder-00001-service.default.svc.cluster.local:80/*: 313.079ms

Services: earthquake-demo-00001,geocoder-00001

Date Time	Relative Time	Annotation	Address
1/8/2019, 5:03:43 PM		Client Start	10.40.1.101 (earthquake-demo-00001)
1/8/2019, 5:03:43 PM	1.013ms	Server Start	10.40.2.93 (geocoder-00001)
1/8/2019, 5:03:44 PM	312.197ms	Server Finish	10.40.2.93 (geocoder-00001)
1/8/2019, 5:03:44 PM	313.079ms	Client Finish	10.40.1.101 (earthquake-demo-00001)

Figure 7-6. Stack breakdown of how our requests flow through our Services

Conclusion

There we have it! A full-fledged application complete with our own custom-built Event Source. While this largely concludes what we'll learn in this report, there's still more that Knative can offer. Also, Knative is constantly evolving and improving. There's a lot of resources to keep an eye on as you continue your journey so before we wrap up, let's make sure we cover a few other references in Chapter 8.

CHAPTER 8
What's Next?

There's still so much more in the young Knative ecosystem, and more is constantly being added. There is already work being done to bring other existing open source serverless frameworks onto Knative. For example, Kwsk is an effort to replace much of the underlying Apache OpenWhisk server components with Knative instead. Other open source serverless projects have been specifically built with Knative in mind and have even helped contribute upstream to the Knative effort. For example, Project riff already provides a set of tools to help ease building functions and working with Knative. This chapter will take a brief look at what it's like to build and run functions on Knative using some of the work from the Project riff team.

Building Functions with Project riff

The Hello World examples in Chapter 2 showed how easy it is to deploy an existing image from a container registry to Knative. The Kaniko example in Chapter 3 as well as the Buildpack method in Example 6-1 demonstrate how to both build and deploy a simple 12-factor app to Knative. The examples so far have focused on containers or applications as the unit of software. Now think back to Chapter 1 and the mention of *functions*. What does it look like to deploy a function to Knative? The answer is that it looks pretty much the same. Thanks to the Build module, Knative can take your function code and turn it into a container in a similar way as it does with any application code.

What Makes It a Function?

Applications are code. So are functions. So what is so special about a function? Isn't it just an application? An application may be made up of many components from a frontend UI to a backend database and all the processing in between. In contrast, a function is usually a small piece of code with a single purpose that is meant to run quickly and asynchronously. It is also typically triggered by an event as opposed to being called directly by a user in a request/response scenario.

Recall the Cloud Foundry Buildpacks example from Chapter 6. The *service.yaml* file shown in Example 6-1 references a full-fledged Node.js Express app that is explicitly written to listen on a given port for a GET request and then return a Hello World message. Instead of a Hello World app, what if our program was a function that accepted a numerical input and then returned the square of that number as the result? This code might look something like what we see in Example 8-1.

Example 8-1. knative-function-app-demo/square-app.js

```
const express = require('express');
const app = express();

app.post('/', function (req, res) {
    let body = '';
    req.on('data', chunk => {
        body += chunk.toString();
    });
    req.on('end', () => {
        if (isNaN(body))
            res.sendStatus(400);
        else {
            var square = body ** 2;
            res.send(square.toString());
        }
    });
});

var port = 8080;
app.listen(port, function () {
    console.log('Listening on port', port);
});
```

We could use the same Buildpack from Example 6-1 to build this function and deploy it to Knative. Consider instead Example 8-2, which shows a function also written in Node.js. Instead of a full Express application, it consists only of a function and does not include any additional Node.js modules.

Example 8-2. knative-function-demo/square.js

```
module.exports = (x) => x ** 2
```

Knative supports this because of its flexibility as provided by the Build module. To build and deploy code like this to Knative, a custom Build Template is used to turn this simple function-only code into a runnable Node.js application. The code in Example 8-2 uses the programming model specifically supported by the function invokers that are part of Project riff.

Project riff is an open source project from Pivotal built on top of Knative that provides a couple of great things: a CLI to install Knative and manage functions deployed on top of it, as well as invokers that enable us to write code shown in Example 8-2. These invokers are responsible for taking *literal* functions like the Node.js example we've seen, or Spring Cloud Functions, or even Bash scripts. Much like Build Templates, invokers are open source and the list continues to grow as riff matures. Make sure to check out *https://projectriff.io* for more!

Further Reading

There is an absolute plethora of documentation, samples, and demos built around Knative to read and reference as you continue on. The best place to start is of course the Knative Docs GitHub repository. Not only does this contain detailed notes on how every piece of Knative works, but there are also even more demos to read through and links to join the community, such as the Knative Slack channel or the mailing list.

We really appreciate the time you've spent with our report, and hope that it was helpful to start getting up and running with Knative. The best advice that we can leave you with is just to get your hands dirty and start building something, no matter how big or small. Explore and learn by doing, by making mistakes and learning how to fix them. Share what you've learned with others! The community around Knative is very young but growing very fast, and we hope to see you become a part of it.

About the Authors

Brian McClain is a Principal Product Marketing Manager for the Technical Marketing team at Pivotal. Brian has always had a passion for learning new technology and sharing lessons picked up along the way, and comes from a mixed professional background including finance, technology, and entertainment. At Pivotal, he gets to do what he enjoys most: building demos and writing about technology, built both inside and outside of Pivotal. You can find him on Twitter at @BrianMMcClain for a mix of tech discussion and bad jokes.

Bryan Friedman is a Product Marketing Director on the Technical Marketing team at Pivotal. After more than ten years working in a many different information technology capacities, he crossed over into the cloud product space with the desire to help others improve their IT organizations and deliver real value to their business. With a background in computer science and a powerful sense of curiosity, he feels lucky to be working at Pivotal in a role where he's able to combine these passions to write about and work with new technologies. Find Bryan on Twitter at @bryanfriedman.