Compliments of VMware Tanzu

Knative Fundamentals

Selections from *Knative in Action* by Jacques Chester

manning

Knative Fundamentals

Selections from *Knative in Action* by Jacques Chester

Manning Author Picks

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN: 9781617298516

contents

1 Introduction

One of my north stars is the Onsi Haiku Test:

Here is my source code.
Run it on the cloud for me.
I do not care how.

This is a radical notion of how software can best be developed, deployed, upgraded, observed, managed, and improved. It must be, because so often it emerges long after we've tried everything else first. It implies:

- That a fast, reliable path to production is a shared goal for everyone.
- That a crisp contractual boundary exists between folks who provide platforms and folks whose work will consume the platform.
- That building software that handles other software is, for most developers, not the most urgent, most valuable work they could be doing.

Kubernetes, by itself, doesn't pass the Onsi Haiku Test. The boundary between development and operation is unclear. Developers can't walk up to a vanilla Kubernetes cluster, hand it raw source code, and get all the basic amenities of routing, logging, service injection, and so on. Kubernetes gives you a rich toolbox for solving the haiku test in your own particular way. But a toolbox isn't a machine. It's a toolbox.

This book isn't about Kubernetes; it's about Knative. Knative builds on the toolbox Kubernetes provides, but also sets out to achieve a level of consistency, simplicity, and ease of use that brings Kubernetes much closer to meeting the Onsi Haiku Test's high standard. Knative is a machine.

While it has something to offer many different professional specialties, Knative is primarily focused on the needs and pains of developers, to elevate them to the heights of "I do not care how". Kubernetes is amazing, but it never strongly demarcated what's meant to be changed or managed by whom. That's a strength: you can do anything! And a weakness: you could, and did, do anything! Knative provides crisp abstractions that, by design, don't refer to the grungy physical business of nodes and containers and VMs. I'll also focus on developers in this book, referring to or explaining Kubernetes only when necessary to understand Knative.

1.1 What is Knative?

There are several ways to answer this question.

The purpose of Knative is to provide a simple, consistent layer over Kubernetes that solves common problems of deploying software, connecting disparate systems together, upgrading software, observing software, routing traffic, and scaling automatically. This layer creates a firmer boundary between the developer and the platform, allowing the developer to concentrate of the software they are directly responsible for.

The major subprojects of Knative are Serving and Eventing.[1] Serving is responsible for deployment, upgrade, routing, and scaling. Eventing is responsible for connecting disparate systems. Both Serving and Eventing have observability as concerns. Dividing responsibilities this way allows each to be developed more independently and rapidly by the Knative community.

The software artifacts of Knative are a collection of software processes, packaged into containers that run on a Kubernetes cluster. In addition, Knative installs additional customizations into Kubernetes itself to achieve its ends. This is true of both Serving and Eventing, each of which installs its own components and customizations. While this may interest a platform engineer or platform operator, it shouldn't matter to developer. You should only care *that* it's installed, not where or how.

The API or surface area of Knative is primarily YAML documents that declaratively convey your intention as a developer. These are "CRDs", Custom Resource Documents. They are, essentially, plugins or extensions for Kubernetes that look and feel like vanilla Kubernetes does.

You can also work in a more imperative style using the `kn` CLI, which is useful for tinkering and rapid iteration. I'll show both of these approaches throughout the book.

Let's take a quick motivational tour of Knative's capabilities.

[1] If you look at early talks and blog posts about Knative, you'll see references to a third subproject, "Build". Build has since evolved and spun out into Tekton, an independent project. This decision moved Knative away from the Onsi Haiku Test, but it also resolved a number of architectural tensions in Serving. Overall, it was the right decision, but leaves you with the responsibility of deciding how to convert source code into containers. Happily, there are many ways to do this, and I'll introduce several later in the book.

1.1.1 Deploying, upgrading, and routing

Deployment has evolved: what used to be a process of manually promoting software artifacts through environments (with scheduled downtime, 200 people on a bridge call all weekend …) became continuous delivery and blue-green deploys.

But should deployment be all or nothing? Knative enables progressive delivery: instead of requests arriving at a production system, which is entirely one version of the software, they arrive at a system where multiple versions can be running together with traffic being split between them. Deployments can proceed at the granularity of *requests*, rather than *instances*. ”Send 10% of traffic to v2” is different from “10% of instances are v2”.

1.1.2 Autoscaling

Sometimes there *is* no traffic. Sometimes there's *too much* traffic. One of these is wasteful, the other is stressful. Knative is ready with the Knative Pod Autoscaler, a request-centric autoscaler that's deeply integrated with Knative's routing, buffering, and metrics components. The autoscaler can't solve all your problems, but it will solve enough that you can focus on more important problems.

1.1.3 Eventing

Easy management of HTTP requests will take you a long way, but not everything looks like a `POST`. Sometimes we want to react to events instead of responding to requests. Events might come from your software or external services, but they may arrive without anyone *requesting* something. That's where Knative Eventing for events comes into focus. It enables you to compose small pieces of software into flexible processing pipelines, connected through events. You can even prepare to process things that don't even exist yet (really).

1.2 So what?

I know your secret: somewhere in your repo is `deploy.sh`. It's a grungy bash script that does grep-and-sed and calls `kubectl` a bunch of times and has some `sleeps`, and maybe you got ambitious so there's a `wget` floating around in it too. You wrote it in a hurry and of course, of course, of course you're going to do a better job, but right now you're busy working getting this thing done before Q3, and you need to implement floozlebit support and refactor the twizzleflorp and deploy.sh works well enough.

But this is always true for everything; there's never enough *time*. Why, really, didn't you make the change yet?[2] Easy: it's too hard. Too much work when you already have enough.

Kubernetes itself is great, once you set it up. It absolutely shines at its core purpose in life: reconcile the differences between the desired state of the system and the actual state of the system on a continuous basis. If all you ever needed was to deploy your sys-

[2] Those of you in the class who're pointing at their Spinnaker instances can lower your hands.

tem once and let it run forever without changing it, then you're good to go and lucky you. The rest of us, however, are on the hedonic treadmill. We have desired worlds that *change*. We ship bugs that need to be fixed, our users think of new features they want, and our competitors make us scramble to answer new services.

And that's how you wound up with the script. And doing a better job of deployment doesn't seem urgent. After all: it works, right? Yes . . . if and *only* if your goal is to be afraid to upgrade anything or to have umpteen slightly different versions of deploy.sh floating around company repos or to write your own CD system without intending to. Why bother? Let Knative toil for you instead.

I know two of your secrets. Your code knows a lot about all your other code. The login service knows about the user service and the are-you-a-robot? service. It tells them what it wants, and it waits for their answer. This is the imperative style, and with it we as a profession have built incredible monuments to human genius. But we've also built incredible bowls of spaghetti and warm compost.

It would be nice to decouple your services a bit, so that software responds to reports of stuff happening, and, in turn, reports stuff that it did. This isn't a novel concept: the idea of software connected through pipes of events or data has sailed under various flags and in various fleets for decades now. There are deep and important and profound differences between all of these historical schools of thought. I will, in an act of mercy, spare you any meaningful discussion of them. Because before you learn how to chisel apart the monolith, you need a chisel and a hammer.

1.3 Where Knative shines

Knative's focus on event-driven, progressively delivered, autoscaling architectures lends itself to particular sweet spots.

1.3.1 Workloads with unpredictable, latency-insensitive demand

Variability is a fact of life: nothing repeats perfectly. Nothing can be perfectly predicted or optimized. Many workloads face *demand* variability: it isn't always clear, from moment to moment, what demand to expect.

The Law of Variability Buffering says that you can deal with demand variability by buffering it in one of three ways:

1. With inventory: Something you produced earlier and have at hand. For example, caching.
2. With capacity: Unused reserve capacity that can absorb more demand without meaningful effect. For example, idle instances.
3. With time: By making the demand wait longer.

These are all costly. Inventory costs money to hold (RAM and disk space isn't free), capacity costs money to reserve (an idle CPU still uses electricity) and famously, "time is money" and nobody likes to wait.

NOTE Inventory, capacity, and time are the *only* options for buffering variability. It's basic calculus. Inventory is an integral, a sum of previous capacity utilization and demand. Capacity is a derivative, a rate of change of inventory. And time is time. You can rearrange the terms, and you can change their values, but you can't escape the boundaries of mathematics. The only alternative is to reduce variability so that you need less buffering in the first place.

Knative's default strategy for buffering is *time*. If demand shows up but capacity is low or even zero, Knative's autoscaler will react by raising capacity and holding your request until it can be served. That's well and good, but it takes time to bring capacity online. This is the famous "cold start" problem.

Does this matter? It depends on the nature of the demand. If the demand is latency-sensitive, then maybe scaling to zero isn't for you. You can tell Knative to keep a minimum number of instances alive (no more pinging your function). But if it's a batch job or background process that can wait a while to kick off, buffering by time is sensible and efficient. Let that thing drop to zero. Spend the savings on ice cream.

Regardless of sensitivity to latency, the other consideration is: how predictable is the demand? Highly variable demands require larger buffers. Either you hold more inventory, or more reserve capacity, or make folks wait longer. You have no alternatives. If you don't know how you want to trade these off, the autoscaler can relieves you of dealing with common cases (see figure 1.1).

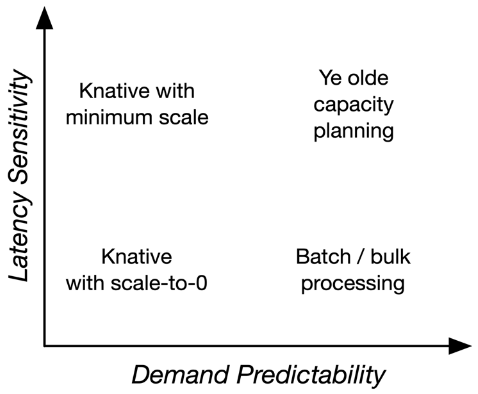


Figure 1.1 Knative's sweet spots in terms of latency sensitivity and demand predictability.

One thing Knative can't do much to save you from is *supply* variability. That is, it can't make variability due to your software vanish, or magic away variability due to upstream systems you rely on. How long your software takes to become live and how responsive it is remains largely in your court. Upstream variability might be in your court, but you'll still be affected by it.

1.3.2 Stitching together events from multiple sources

Sometimes you have a square peg, a round hole, and a deadline. Knative won't shave the peg or hammer it into the hole, but Knative Eventing lets you glue things together so that you can achieve your original purpose. By design, Eventing is meant to receive events from heterogenous sources and convey them to heterogenous consumers. Webhook from GitHub? Yes. Pub/Sub message from Google? Yes. File uploaded? Yes.

A combination of these? Also yes, which is the interesting part. Relatively small, consistent, standards-based interfaces allow many combinations of elements. To this, Knative adds simple abstractions to enable you to go from dabs of glue to relatively sophisticated event flows. As long as an event or message can be expressed as a CloudEvent, which is pretty much anything ever, Knative Eventing can be used to do something smart with it.

The flipside of generality is that it can't be everything to everyone. Should you use it for CI/CD (continuous implementation/continuous deployment)? Maybe. For streaming data analysis? Perhaps. Business workflow processing? Reply hazy, try again.

The key is that for all of these, there are existing, more specialized tools that might be a better fit. For example, you can build a MapReduce pattern using Knative. But realistically, you won't get anywhere near the kind of performance and scale of a dedicated MapReduce system. You can build CI/CD with Knative, but now you have to do homework to implement all the inflows and outflows (see figure 1.2).

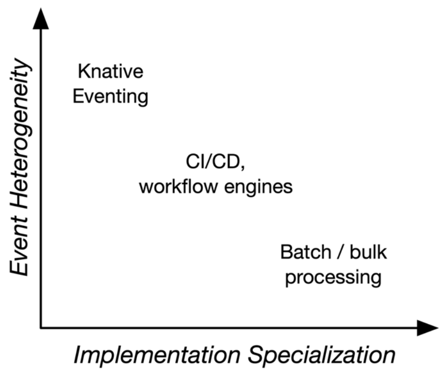


Figure 1.2 Knative's sweet spots in terms of event heterogeneity and implementation specialization.

Where Knative can shine is when you want to connect a variety of tools and systems in simple ways, in small increments. We all do this in our work, but typically it gets jammed into whatever system that happens to have room for boarders. And so our web apps sprout obscure endpoints or our CI/CD accumulates increasingly hairy Bash scripts. Knative lets us pull these out into the open, so that they can be more easily tested, monitored, and reused.

1.3.3 Decomposing monoliths in small increments

Microservices describes a family of powerful architectural patterns. But getting to a microservices architecture isn't easy, because most existing systems aren't designed for it. For better or worse, they're monoliths.

Easy, you say: use the strangler pattern. Add microservices incrementally, route requests to them so that the original code path goes cold, repeat until you're done.

Knative makes this easier in two ways. The first is that it's good at the routing thing. The concept of routing portions of traffic is key to its design. This matters because the strangler pattern tends to falter once you've strangled the less-scary bits (look boss, we broke out the cat GIF subsystem!) and move onto the parts where the big money lives. Suddenly it's a bit scarier, because (1) a cutover is a cutover, (2) a big-bang cutover is a bet-your-job event, and (3) Knative makes it easier to stop believing in (1) and (2) (see figure 1.3).

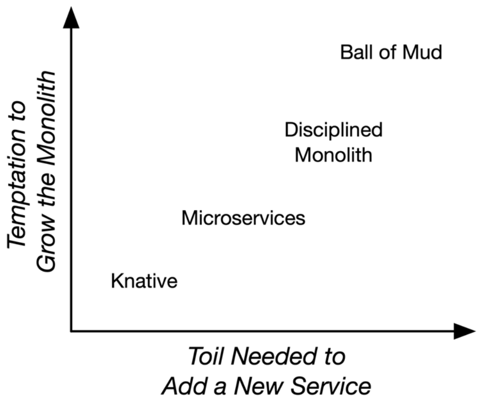


Figure 1.3 Knative's sweet spots in terms of resisting temptation to grow a monolith.

The second way Knative makes strangulation easier is that you can deploy small units easily. Knative has a deep design assumption that you'll have a bunch of little functions that will come and go. A function is less to recreate than a service. The smaller you can start, the easier it is to start.

1.4 It's a hit

Up to now, I've promised a lot: easier deployments, easier event systems, incremental development, Martian unicorns—the usual stuff that everyone promises to developers. But I haven't given you any concrete details. To support my pitch that we can start in small increments, I'll begin with one of the oldest, simplest examples of the dynamic web and show how Knative makes it faster, smarter, and easier.

Remember hit counters (see figure 1.4)?

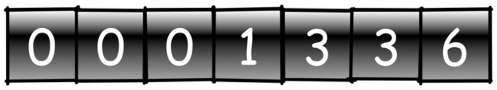


Figure 1.4 The late 1990s were truly a golden era.

I sure do. The first time I saw one, it *blew my mind*. It changed! By itself! Magic!

Not magic, of course, it was a CGI program, probably Perl.[3] CGI is one of the spiritual parents of Knative[4], so in its honor, we're going to make a hit counter for MY AWESOME HOMEPAGE, as shown in the following listing.

Listing 1.1 The awesome homepage HTML

```
<html>
<body>
 <style>body { font-family: "awesomefont" }</style>
 <center>
  <b>MY AWESOME HOMEPAGE</b><br />
  <img src="//hits.png" />
 </center>
</body>
</html>
```

First, let's talk about the basic flow of requests and responses. A visitor to the homepage will GET an HTML document from the web server. The document contains style and, most importantly, the hit counter, as shown in figure 1.5.

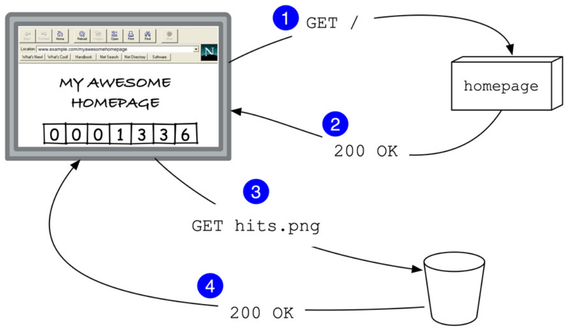


Figure 1.5 The flow of requests and responses.

Specifically:

1. The browser issues a GET request for the homepage.
2. The homepage service returns the HTML of the homepage.
3. The browser finds an img tag for hits.png. It issues a GET for hits.png..
4. A file bucket returns hits.png

In the old world, all of the processing needed to generate the hit counter would block the webserver response. You'd submit your request, the web server would bestir the elder gods of Cämelbuk and then /CGI-BIN/hitctr.pl would render the image. It might take a second or two, but nobody could tell, unless they were using one of those blazing 28.8k modems.

[3] OK, using ImageMagick, but not magic magic.
[4] Two others are inetd and stored procedures.

But now everyone is impatient: spending a few second to render an image that could otherwise be served from a fast data path isn't going to be acceptable. Instead we'll break that responsibility out and do it asynchronously. That way, the web server can immediately respond with HTML and leave the creation of hit counter images to something else.

How will the web server signal that intention? It won't. Instead it will signal that a hit occurred. Remember: the web server wants to serve web pages, not orchestrate image rendering. Instead of blocking, it emits a `new_hit` CloudEvent.

Emits to *where?*

To a `Broker`, a central meeting point for systems in Knative Eventing. The `Broker` has no particular interest in the `new_hit` event, it merely receives and forwards events. The exact details of who gets what is defined with `Triggers`. Each Trigger represents an interest in a set of events and where to forward them to (see figure 1.6). When events arrive at the Broker, it will apply each Trigger's `filter` and, if it matches, forward the event to the `subscriber`:

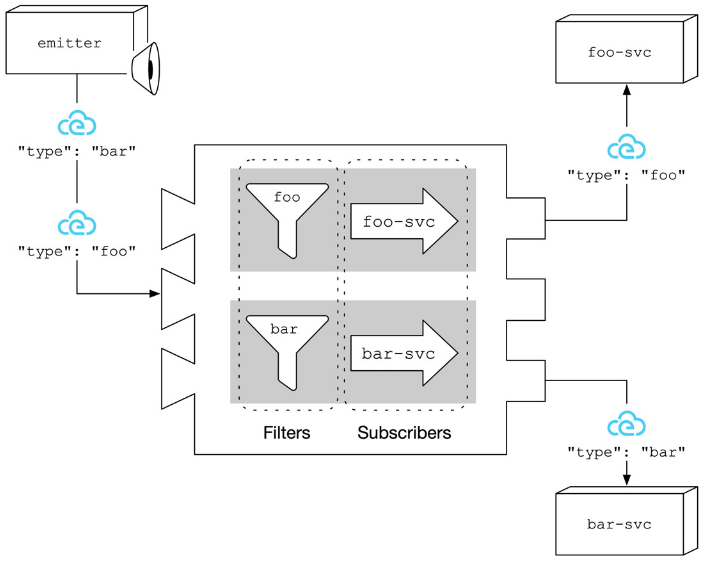


Figure 1.6 Broker applying Triggers to CloudEvents.

It's Triggers that enable the incremental composition of event flows. The web server doesn't know where `new_hit` will wind up and doesn't care. Given our `new_hit`, we can start to tally up the count of hits. Already, we're ahead of the 1999 status quo: we could take our original Perl script and have it react to the `new_hit` event instead of blocking the main web response.

But since we're here, let's go a step further. After all: is rendering images the actual proper concern of a tallying service? When I perform an SQL `UPDATE` I don't get back

JPEG files. Instead I will have the tally service consume the `new_hit` and emit a new count, which can then wing its way to other subscribers.

Putting it together in figure 1.7:

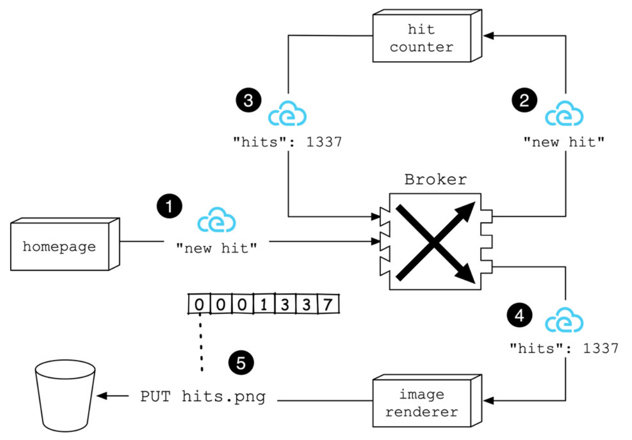


Figure 1.7 The flow of events.

1. The `homepage` service emits a `new_hit` event.
2. A Trigger matches `new_hit`, so the Broker forwards it to `hit counter`.
3. `hit counter` updates its internal counter, then emits a `hits` event with the value of that counter..
4. Another Trigger matches for `hits`, so the Broker forwards it to `image renderer`
5. The `image renderer` renders a new image and replaces `hits.png` in the file bucket.

And now, if the visitor reloads their browser, they'll see that the hit counter has incremented.

1.4.1 Trouble in paradise

Except, maybe, they don't. To see why, let's put the diagrams together in figure 1.8: Note that I'm showing two sets of numbers, one for web request/response and another for the event flow. This illuminates the important point: the web flow is *synchronous*, the event flow is *asynchronous*. You knew that, but I handwaved away the consequences, and now I need to slap my wrist. The distinction matters.

Because the event flow is asynchronous (see figure 1.9), there's no guarantee that `hits.png` will have been updated before the next visitor arrives. I might see 0001336, reload and then see 0001336 again.[5] And that's not all: where one visitor might see no change, another visitor might observe that the hit counter jumps forward, because later renderings can overwrite earlier renderings before they were served. And *that's* not all! An observer might see the count go *backward*, because the rendering that increased the

[5] Assuming that I used cache-disabling headers to force the browser to refetch each time.

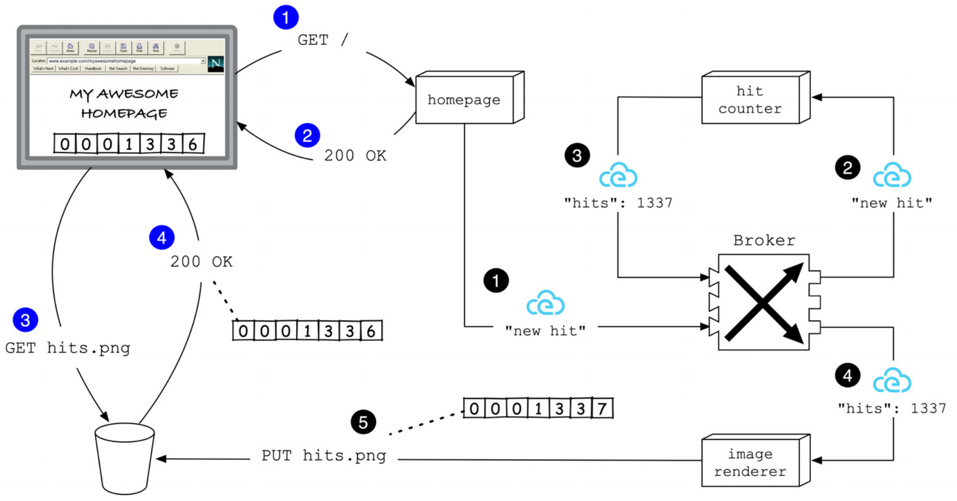


Figure 1.8 Combining the flows in one diagram.

number to 0001338 might have finished before the rendering for 0001337 did. Or it may be that the events arrived out of order. Or some events never even arrived.

I'm not done. Remember how I said that `hit counter` was keeping a tally of hits? I didn't say *where*. If it's just keeping a value in memory, then you have new problems. For example, if Knative's autoscaler decides that things are too quiet lately, it will

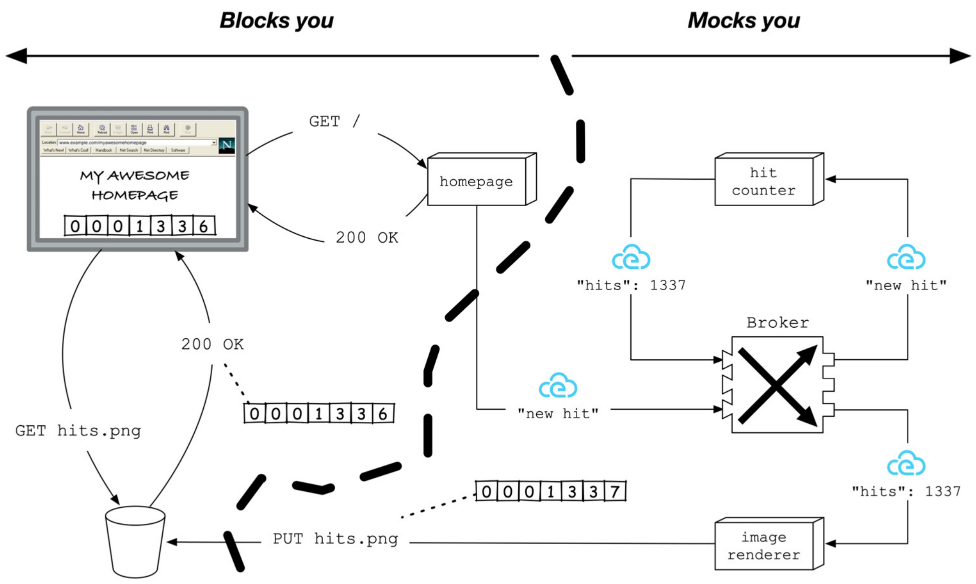


Figure 1.9 Synchronous flows can be inefficient. Asynchronous workloads can be inexplicable.

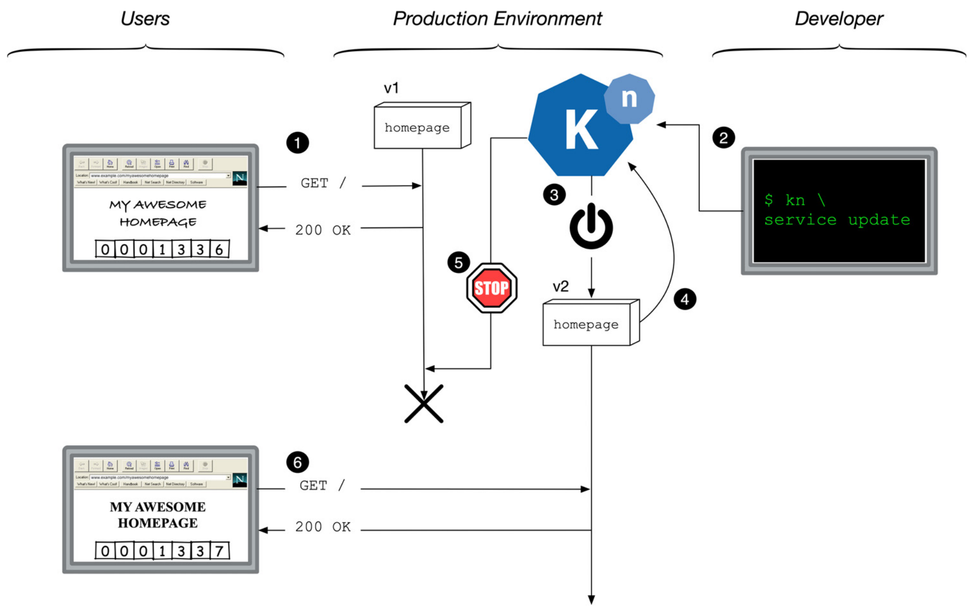
reduce that number of `hit counters` to zero and pow, your tally is gone. Next time it spins up, your hit count will be reset to zero. But if you have more than one `hit counter`, they're keeping separate tallies. The exact hit count image at any moment will depend on traffic, but not in the way you might have expected.

I'm describing stateless systems, of course. The answer is to keep state in a shared location, separately from the logic that operates on it. For example, each `hit counter` might be using Redis to increment a common value. Or you might get super fancy[6] and have each instance listen for `hits` events. If the incoming event is a higher tally, jump to that value and hope you're not participating in an infinite event loop.

You've probably noticed that my focus has been on an already-deployed system. That's the bad news. The good news is that you can fix a key bug I introduced in the previous section. Can you guess what it is?

Correct. The font sucks.

You quickly learn that Knative prizes *immutability*. This has many implications. For now, it means that we can't SSH into `homepage`, open `vi`, and do it live (see figure 1.10).[7] But it does raise the question of how changes get moved from your workstation to the cloud.



```
$ kn \
service update
```

Figure 1.10 **Updating the homepage.**

[6] Please don't.
[7] And I do not, for the purposes of law, recall ever doing so myself.

Knative encapsulates "run the thing" and "change the thing" into `Services`.[8] When the `Service` is changed, Knative acts to bring the world into sync with the change.

1. A user who arrives before the update sees the existing HTML, as served by `homepage` v1.
2. The developer uses `kn` to update the Service.
3. Knative starts `homepage` v2.
4. `homepage` v2 passes its readiness check.
5. Knative stops `homepage` v1.
6. A second user arriving after the update sees a more professional font.

This blue/green deployment behavior is Knative's default. When updating `Services`, it ensures that no traffic is lost and that load is only switched when it's safe to do so.

1.5 What's in the Knative box?

Let's break this down into subprojects: Serving and Eventing.

1.5.1 Serving

Serving is the first and most well-known part of Knative. It encompasses the logic needed to run your software, manage request traffic, keep your software running while you need it, and stop it running when you don't need it.

As a developer, Knative gives you three basic types of document you can use to express your desires: `Configuration`, `Revision`, and `Route`.

Configuration is your statement of what your running system should look like. You provide details about the desired container image, environment variables, and the like. Knative converts this information into lower-level Kubernetes concepts such as deployments. In fact, those of you with Kubernetes familiarity might wonder what Knative adds. After all, you can create and submit a `Deployment` yourself; no need to use another component for that.

Which takes us to Revisions. These are *snapshots* of a Configuration. Each time that you change a Configuration, Knative first creates a Revision and in fact, it's the *Revision* that's converted into lower-level primitives.

But this might still seem like overhead. Why bother with this versioning scheme in Knative, when you have Git? Because blue/green deployment is not the *only* option. In fact, Knative allows you to create nuanced rules about traffic to multiple Revisions.

For example, when I deployed `homepage` v2, the deployment was all or nothing. But suppose I was worried that changing fonts would affect how long people stay on my page (that is, an A/B test). If I perform an all-or-nothing update, I'll get lots of data for the before and after. But there may be a number of confounding factors, such as time-of-day effects. Without running both versions side by side, I can't control for those variables.

[8] Not to be confused with Kubernetes Services. More on that later.

But Knative can divvy up traffic to Revisions by percentage. I might decide to send 10% of my traffic to v2 and 90% of my traffic to v1. If the new font turns out to be worse for users, then I can roll it back easily without much fuss. If instead it was a triumph, I can quickly roll forward, directing 100% of traffic to v2.

It's this ability to selectively target traffic that makes Revisions a necessity. In vanilla Kubernetes, I can roll forward and I can roll back, but I can't do so with *traffic*, I can only do it with *instances of the service*. This has important architectural and operational consequences, which I'll dive into later in the book.

Perhaps you wondered what happened to the `Services` I was talking about in the walkthrough. Well, these are essentially a one-stop shop for all things serving. Each service combines a Configuration and a Route. This compounding makes common cases easier, because everything you need to know is in one place.

But these concepts aren't necessarily what get listed on the marketing flyer. Many of you have come to hear about autoscaling, including scale-to-zero. For many folks, it's the ability for the platform to scale all the way to zero that captures their imagination: no more wasting money on instances that are mostly idle. And similarly, the ability to scale up: no more getting paged at absurd o'clock in the morning in New York because something huge happened in Sydney (or vice versa). Instead you delegate the business of balancing demand and supply to Knative. Sometimes you'll want to understand what the heck it's doing, so I'll spend time delving into the surprisingly difficult world of autoscaling.

1.5.2 Eventing

Eventing is the second, less-well-known part of Knative. It provides ways to express connections between different pieces of software through events. In practical terms, "this is my software" is simpler to describe than "here is how all my software connects together". Eventing consequently has a larger surface area, with more document types, than serving does.

Earlier in the chapter you learned that in the middle of the Eventing world is where `Triggers` and `Brokers` live. The trigger exists to map from an event filter to a target. The Broker exists to manage the flow of events based on Triggers.

But that's the headline description, light on detail. For example, how does a CloudEvent actually get into the Broker? It turns out, there are multiple possibilities. The most powerful and idiomatic of these is a `Source`. These represent configuration information about a kind of emitter of events and a Broker to which they should be sent. A Source can be more or less anything: GitHub webhooks, direct HTTP requests, you name it. As long as it emits CloudEvents to a Broker, it can be a source.

What kinds of events are there? That's where the event registry comes along, providing a catalogue of `EventTypes`. At a command line you can quickly discover what events you can react to.

Great! You're probably already composing event-processing graphs in your head, and it won't be long before you get tired of writing Trigger upon Trigger. It will be handy if you had a simple way to do things in order. This is what `Sequences` can express for you: that A runs before B. Or maybe you want to do more than one thing at a time. That's what `Parallel` does, allowing you to express that A and B can run independently.

Analogous to how Serving provides the convenience of `Service`, Sequence and Parallel constructed from the same concepts that you can use directly. They're a convenience, not a constraint. They'll enable you assemble event flows with much less YAML than hand-wiring equivalent Triggers would.

Beneath these smooth surfaces lies a fair amount of plumbing: `Channels`, `Subscribers`, `Reply`, `Addressable` and `Callable`. Right now, these aren't important. We'll get to them in a due time. Meanwhile you can do most of what you need to do with a mix of `Source`, `Trigger`, `Broker`, `Sequence`, and `Parallel`.

1.5.3 Serving and eventing

By design, you don't need Serving to use Eventing and you don't need Eventing to use Serving. But they do mesh pretty well together. For example, if you have long processing pipelines, it's nice if idle instances don't sit around burning money waiting on upstream work to finish. Or, if there's a bottleneck, it's helpful if that part of the pipeline is scaled up. That's Eventing gaining a superpower from serving.

And it works the other way. Serving's focus is on request/reply designs, the simple, robust but sometimes slow blocking approach. By itself this will favor adding functionality to existing services instead of creating new ones. Blocking is still blocking but blocking on threads is faster than blocking on HTTP. You can easily drift back from microservices to monoliths in costume.

Eventing relieves a bit of that design pressure (see figure 1.11). You can now offload work that doesn't need to block, or which should react to events instead of following commands. Encouraging smaller units of logic and behavior allows Serving to really shine: autoscaling the GigantoServ™ is better than nothing. But it's wasteful to burn 100Gb of RAM on a system with 300 endpoints when only two of them are seeing any kind of traffic surge.

In the hit counter system discussed previously, I put both Serving and Eventing to work. Serving handles the business of `homepage`, `hit counter`, and `image renderer`. Eventing handles the Broker so that Services will receive and emit events without direct coordination. In this book I'll be describing them individually, so that I can go into some depth. But they're intended to work well together. Ultimately, I want you to do that.

1.6 Keeping things under control

"Knative" is a clever name. First, everyone gets to practice pronouncing it a few times ("KAY-naytiv"). From personal experience, I know that if folks are struggling to pronounce your name, they'll *really* concentrate on it.

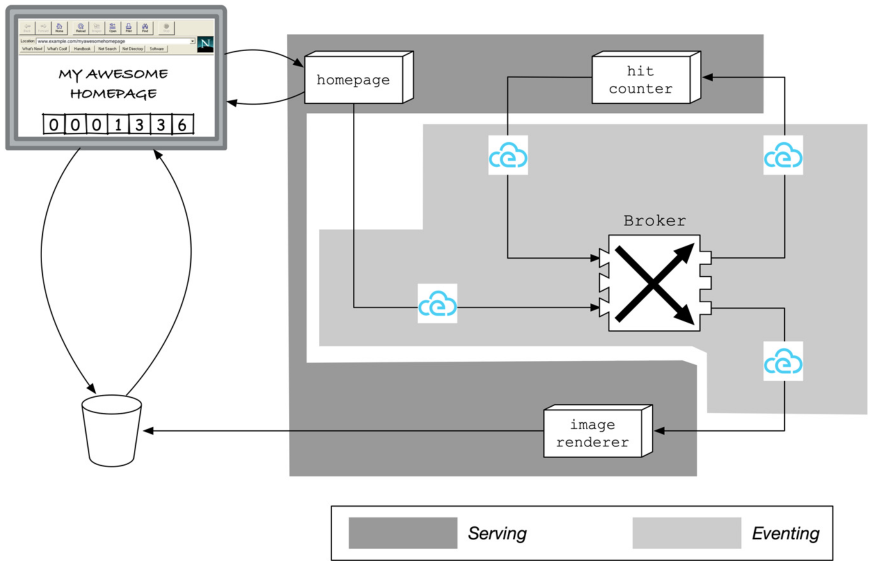


Figure 1.11 Serving runs the services, and Eventing wires them together.

Second, it encompasses some of the design vision. Knative is native to Kubernetes both in spirit and implementation. In a kind of judo throw, it uses the extensibility of Kubernetes to conceal the complexity of Kubernetes. But every throw needs a little leverage to make it work. To give you that leverage, I need to step back a bit from Knative and give you a basic level of familiarity with a core organizing concept in Kubernetes and Knative: the feedback control loop.

1.6.1 Loops

As a profession, we use terms such as “feedback loop” pretty loosely. Strictly, feedback loops are any circular causality that amplifies or dampens itself. (I use the word “strictly” informally).

For example: compound interest is a feedback loop. No humans are involved, only computers multiplying numbers. But the amount of interest paid is a function of the accumulated principal, which is itself a function of previous interest paid. After each period the effect is amplified. Each payment feeds back into the system.

Or consider an avalanche after heavy snow. A small amount of snow slips further down, making the next spot down slightly heavier. More snow slips further down, making the next spot even heavier again. Within seconds, what starts as a few grams of attractive light fluff transforms into thousands of tons of mindless destruction.

The nature of pure feedback loops is that they require no intelligence or logic. They can be composed of pure causality. This is why both compound interest and an avalanche are of the same species. Whether the structure of the system was set by humans or nature is unimportant to how it will behave.

We often *assume* that intelligence is involved in feedback loops, because pure circular causality is rarely apprehended and understood: purely damping loops disappear and purely amplifying loops fly apart. At a human level, the universe *appears* to be composed of linear causality, but beneath most of it lies a seething world of loops shoving and pushing each other around an equilibrium.

Because purely causal circularity is rarely apprehended, we attribute intelligence to those that we do observe, because in our experience humans are necessary to create a special case: control loops.[9]

Control loops are a special case because they add a controller to the loop (see figure 1.12). A controller observes the *actual* world, compares it to some reference of the *desired* world, then acts upon the actual world to make it look more like the desired world. This simple description disguises centuries of work and generations of engineering students being unceremoniously doused with calculus. But, at its heart, the idea of a control loops is simple. Make what we have look more like what we want.

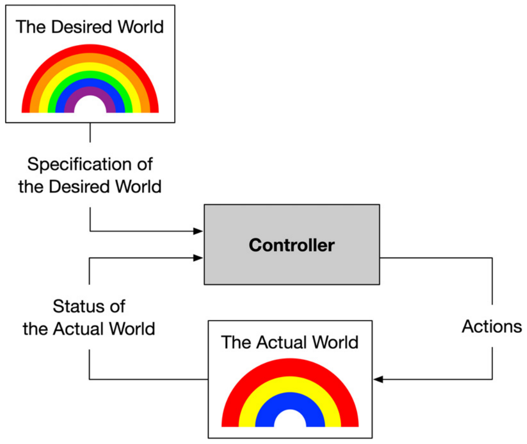

The Desired World
Specification of the Desired World
Controller
Status of the Actual World
The Actual World
Actions

Figure 1.12 **The basic structure of a control loop.**

The key is that the loop runs repeatedly. The Controller regularly takes in information about the desired and actual worlds, comparing them, then deciding whether to take actions in the actual world. The repeated observations of the world are "fed back" to the controller, which is why this is a "feedback controller".[10]

[9] Attributing intelligence to causality is human. Lightning isn't due to angry super-beings, but if you've ever been near a lightning strike you can understand why "static electricity" wasn't the first thing people thought of to explain such a phenomenon.

[10] When you design systems without the loop, the controller is using "feed forward". The designer has often taken advantage of a property of the controlled system to make feedback unnecessary. For example, you don't see many feedback controllers governing the position of concrete slabs, because these will typically stay put on their own. Feed forward control is a useful, legitimate design technique for many kinds of systems. For highly dynamic systems like software, though, feedback control is well suited to maintaining an amount of stability and reliability.

Controllers vs controllers

"Controller", in this context, isn't referring to the Model-View-Controller (MVC) pattern you might recognize from software frameworks. Trygve Reenskaug is typically credited with inventing the MVC pattern, initially using the name "Editor" — in a different universe we'd be talking about the MVE pattern. The name came about because "After long discussions, particularly with Adele Goldberg, we ended with the terms Model-View-Controller".

The Controller, or Editor, was meant to "bridge the gap between the human user's mental model and the digital model that exists in the computer. The ideal MVC solution supports the user illusion of seeing and manipulating the domain information directly."

This isn't what Kubernetes, and by extension Knative, mean by "Controller". Instead the meaning is taken by analogy from control theory, which deals with how dynamic systems can be made to behave more predictably and reliably. It's widely applied by engineers in fields such as electrical and electronic systems, aerodynamics, chemical plant design, manufacturing systems, mining, refineries, and many others.

You'll avoid confusion by pretending you've never heard of MVC.

From my description so far, it's easy to form the impression that control loops are all about counteracting unwanted changes in the actual world. That the desired world is immutable, fixed, so that bursts of activity are only signaled by the controller when the actual world shifts out of alignment with the desired world.[11]

This isn't true. The Controller doesn't "see" a change in the actual world in contrast to an immutable desired world. What it sees is a difference between one input and another input.[12] On each pass around the loop the controller sees the two inputs afresh, as if for the first time.[13] It doesn't need to know that the actual changed since "last time". It doesn't know that the desired changed since "last time". It doesn't care. It knows that they aren't the same.

This leads to a simple conclusion: the controller may act due to changes in *either* of the actual world or the desired world. Because it's reacting to the emergence of the difference, not to the worlds per se. Something or someone outside the control loop can change the desired world in order to prompt activity (see figure 1.13).

[11] This, in turn, leads to the question of "what is the desired world?". This is the kind of annoying open-ended question that led to Socrates being fairly permanently voted off the island by his fellow Athenians. Using Socrates as a sockpuppet, Plato argued that there are perfect ideas, perfect forms, independent of mere matter. He'd be somewhat at home with the mistaken interpretation that control loops are mostly about a controller (he'd call it a Guardian) continuously striving to return to The Good.

[12] That difference isn't commutative, so the order of inputs still matters. What comes through the "desired" door does need to be the desired state and what comes through "actual" needs to be the actual state.

[13] This isn't universally true of controllers in control theory: they can have many kinds of "memory" to carry information forward in time. In the most common approach to control theory, what I'm describing is a purely proportional controller. Adding averaging over previous states would add integral control. Adjusting the forcefulness of actions based on how quickly the two worlds are diverging would add derivative control.

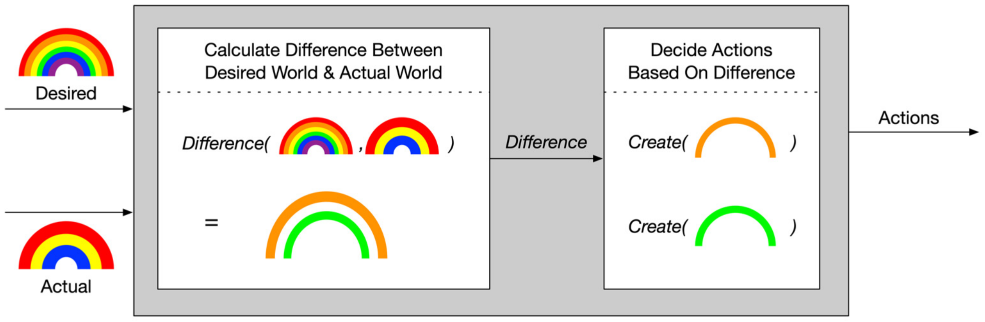

Desired
Actual
Calculate Difference Between Desired World & Actual World
Difference(,)
=
Difference
Decide Actions Based On Difference
Create( )
Create( )
Actions

Figure 1.13 The internal structure of a controller.

1.6.2 Loops within loops

Who changes the desired world? Most of us assume that a human will do it at first. Something like push YAML to update the desired world, knock off, and go home.

This will work, but it has at least one problem. The actual world is complex. Often obnoxiously so. As a profession we've tackled this complexity using abstraction (name things to banish their complexity) and composition (combine things to amplify their power). If I could not use abstraction and composition, if I had to define every detail for every part of my world, then I would (1) send many worlds over the wire and (2) I would need a very complex controller indeed. About as complex as the world itself.[14]

In the industrial world this is dealt with by "hierarchical control". That is, the desired world of one controller is modified by the actions of a supervising controller. For example, an industrial kiln will have controllers for managing individual gas burners, to ensure that they burn the right amount of flammable gas. What's the right amount? That's decided by a supervising controller which is interested in controlling the temperature of the kiln. Instead of a controller which runs all the way from "right temperature" to "right gas flow for hundreds of burners", we have two feedback control loops that are nested (see figure 1.14).

This should be recognizable as architectural layering according to the Single Responsibility Principle. Temperature control is a different concern from gas flow control. And so it is with software systems: the business of shipping photons over fiber optic cable is distinct from the business of forming frames which is different from sending packets, which doesn't at all resemble a `GET` request. Developing optical control algorithms is not a precondition for using JavaScript. People might think that's a pity, but that's beside my point.

[14] Ross Ashby, an early cyberneticist, called this the "Law of Requisite Variety": any perfect controller of a system must be as complex as the system. Of course, "perfect" is impossible in practice and in fact, we don't need it (do you really think the kiln controller should include a weather forecaster and an ability to tell if the site foreperson is angry today?). The tactic of breaking control problems into hierarchies makes each level much more tractable to solve to a satisfactory standard.

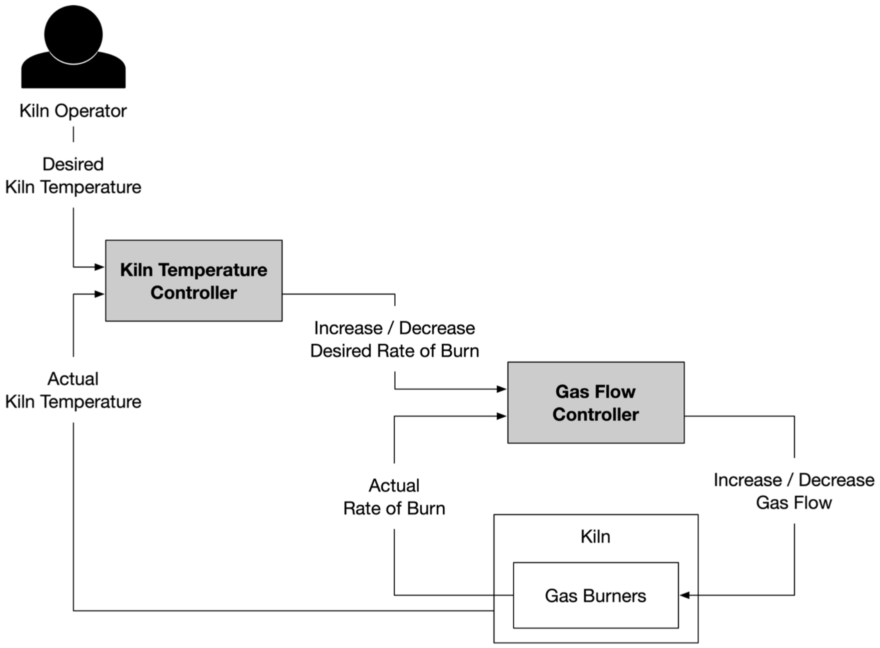


Figure 1.14 A hierarchical kiln controller.

Ultimately this hierarchy of feedback control loops reaches up to you. You have a desired world of "software that achieves such-and-such purpose". Your desired world changes, creating a cascade of other worlds that change. Soon a deployment is setting new targets for lower-level controllers to react to. Most of the time we're focused solely on the actions we're taking, but we (hopefully) don't act like pure noise. We're purposeful.

Kubernetes explicitly models its architecture on feedback control loops and provides infrastructure to enable the easy development of a variety of controllers for different purposes. Kubernetes then uses hierarchical control to layer responsibilities: pods can be supervised by ReplicaSets which are supervised by deployments (see figure 1.15).

Knative Serving builds on this infrastructure and adopts its norms. It presents the surface interface of Services, Configurations, Revisions and Routes. These are handled by first-level controllers, which break them into targets for other controllers, and so on until code lands on a VM you don't care about and runs code that you care about very much. Your role is to be the highest-level controller in the hierarchy. Knative is meant to see to the rest.

1.7 Are you ready?

Before we dive in, let me tell you my assumptions about you. The first is that you've done some programming and can get the gist of examples in Java or Go. The second is that you're comfortable with installing and using CLI tools. Basically, I'm assuming that you're in Knative's primary audience: developers.

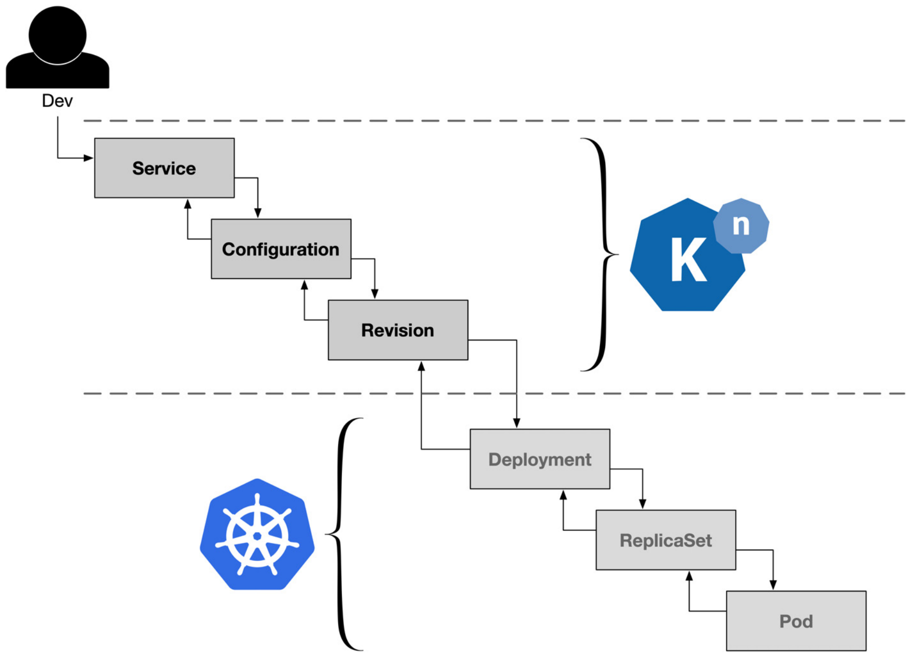


Figure 1.15 **Several of the hierarchy of Knative and Kubernetes controllers involved in running a Service.**

I don't assume that you know anything about Kubernetes or service meshes. I don't assume that you have used a serverless platform before. When I need to introduce necessary information I will, but my goal throughout is that Knative should live up to its vision of enabling you to ignore Kubernetes altogether.

From the next chapter, I'll need you to set up several tools. Most importantly, I'm guessing that you've installed Knative or someone is providing it for you. I'm also assuming that you've installed the `kn` tool, which I'll focus on throughout. See the appendix for an installation guide for Knative and `kn`.

If you want to run the samples, you'll need to have installed Java, Maven and Go.

Take a moment to set up YAML support in your favorite editor. Certain editor YAML extensions also include specialized Kubernetes support, which is nice to have but not essential.

Most of all, I want you to have fun. Grab a drink of your choice, and let's begin.

Summary

- Knative makes it easier to deploy, update, autoscale, and compose event-driven software.
- Knative has two major components: Serving and Eventing. Serving is focused on running software, scaling, and routing. Eventing is focused on event flows.
- The world is filled with feedback loops. Some of these are controlled.

- A controller compares a desired world and an actual world, then decides what's necessary to make the actual world resemble the desired world. This process occurs repeatedly, creating a feedback control loop.
- Controllers can be nested, arranged into hierarchies. Higher controllers adjust the desired world of lower controllers.
- Control loops are a core architectural principle of Knative.

References

- Humble, J & Farley, D. *Continuous Delivery: Reliable Software Releases Through Build, Test and Deployment Automation.*
- Governor, J. "Towards Progressive Delivery". *James Governor's* Monkchips, 6 August 2018. redmonk.com/jgovernor/2018/08/06/towards-progressive-delivery/.
- twitter.com/onsijoe/status/598235841635360768.
- Hopp, W.J. & Spearman, M.L. *Factory Physics, 3rd Edition.* Page 309.
- Fowler, M. "StranglerFigApplication", *MartinFowler.com*, 29 June 2004. martin-fowler.com/bliki/StranglerFigApplication.html
- Reenskaug, T. *MVC Xerox Park 1978-79.* heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html.

2 Introducing Knative Serving

This chapter covers

- Deploying a new Service with Knative Serving
- Updating the Service with Revisions
- Splitting traffic between Revisions
- Understanding the major components of Serving and what they do

Serving is where I'm going to start you off in Knative, and the coming chapters will take you into a deeper dive on the major concepts and mechanisms. To begin with, I'm going to spend this chapter getting you warmed up in two ways.

To begin with, I'm going to *use* Knative. You'll notice that I ducked and weaved around this in chapter 1. I did walk you through an example, and that example *was* realistic. But it was also intended to whet your appetite for the whole book and so necessarily needed to touch on many points. A hypothetical with diagrams and narrative is a quick way to do so.

But now I'm going to put your fingers on a keyboard. We'll use the `kn` CLI tool to deploy software, changing its settings, changing its software, and finally to configure traffic. I won't be doing any YAMLeering. We'll try a purely interactive approach to Knative.

In the second part of the chapter, I'll take a whistlestop tour of Serving's key software components. I'm doing this now because I want to introduce them in one easy-to-find place. The following chapters are all structured around the concepts that Knative exposes to developers. I could introduce components as I go, but it would mean that you might need to hunt through the book to find component information.

This, too, will tie back to chapter 1, where I introduced you to the basic concept of control loops. In this chapter, we'll apply that basic concept to explain the high-level architecture of Serving, which is one based on hierarchical control loops.

By the end of the chapter, my goal is that you can (1) start poking around `kn` with your own example apps, and (2) you'll have a nodding acquaintance with Knative Serving's runtime components. These will set up our progress into following chapters, where we'll go into greater depth on concepts such as Configurations, Routes, and the Autoscaler.

2.1 A walkthrough

In this section, I'm going to use `kn` exclusively to demonstrate several Knative Serving capabilities.

NOTE `kn` is the "official" CLI for Knative, but it wasn't the first. Before it came a number of alternatives, such as `knctl`. These helped to explore different approaches to a CLI experience for Knative. `kn` serves two purposes. The first is as a CLI in itself, specific to `kn`, rather than requiring users to anxiously skitter around `kubectl` pretending that Kubernetes isn't *right there*. The secondary purpose is to drive out and Golang APIs for Knative.

2.1.1 Your first deployment

Let's first use `kn service list` to ensure you're in a clean state. You should see `No services found` as the response.

Now we create a service in the following listing using `kn service create`.

Listing 2.1 Use kn to create our first service

The docker image reference. In this case, we're using a sample app image provided by the Knative project.

The first argument for `kn service create` is the name of the service.

Inject an environment variable, `TARGET`, which will be consumed by the sample app.

```
$ kn service create hello-example \
  --image gcr.io/knative-samples/helloworld-go \
  --env TARGET="First"
```

`kn` monitors the deployment process and emits logs

```
Creating service 'hello-example' in namespace 'default':

  0.084s The Route is still working to reflect the latest desired
     specification.
  0.260s Configuration "hello-example" is waiting for a Revision to become
     ready.
  4.356s ...
  4.762s Ingress has not yet been reconciled.
  6.104s Ready to serve.
```

```
Service 'hello-example' created with latest revision 'hello-example-pjyvr-1'
     and URL:
http://hello-example.default.35.194.0.173.nip.io
```

`kn` gives you the URL for the newly-deployed software

The logs emitted by `kn` refer to concepts I discussed in chapter 1. The Service you provide is split into a Configuration and Route. The Configuration creates a Revision. The Revision needs to be ready before Route can attach Ingress to it and Ingress needs to be ready before traffic can be served at the URL.

This dance illustrates how hierarchical control breaks your high-level intentions into particular software to be configured and run. At the end of the process, Knative has launched the container you nominated and configured routing so that it's listening at the given URL.

What's at the URL? Let's see in the following listing:

Listing 2.2 The first hello

```
$ curl http://hello-example.default.35.194.0.173.nip.io
Hello First!
```

Very cheerful.

2.1.2 Your second deployment

Mind you, perhaps you don't like First. Maybe you like Second better. Easily fixed in the following listing:

Listing 2.3 Updating hello-example

```
$ kn service update hello-example \
  --env TARGET=Second

Updating Service 'hello-example' in namespace 'default':

  3.418s Traffic is not yet migrated to the latest revision.
  3.466s Ingress has not yet been reconciled.
  4.823s Ready to serve.

Service 'hello-example' updated with latest revision 'hello-example-bqbbr-2'
     and URL:
http://hello-example.default.35.194.0.173.nip.io

$ curl http://hello-example.default.35.194.0.173.nip.io
Hello Second!
```

What happened is that I changed the `TARGET` environment variable that the example application interpolates into a simple template, as shown in the following example:

Listing 2.4 How a hello sausage gets made

```
func handler(w http.ResponseWriter, r *http.Request) {
  target := os.Getenv("TARGET")
  fmt.Fprintf(w, "Hello %s!\n", target)
}
```

You may have noticed that the revision name changed. "First" was `hello-example-pjyvr-1` and "Second" was `hello-example-bqbbr-2`. Yours might look slightly different, because part of the name is randomly generated. `hello-example` comes from the name of the Service, and the `1` and `2` suffixes indicate the "generation" of the Service (more on that in a second). But the bit in the middle is randomized to prevent accidental name collisions.

Did Second replace First? The answer is: it depends whom you ask. If you're an end user sending HTTP requests to the URL, yes, it appears as though a total replacement took place. But from the point of view a developer, both Revisions still exist, as shown in the following listing.

Listing 2.5 Both revisions still exist

```
$ kn revision list
NAME                    SERVICE         GENERATION   AGE     CONDITIONS   READY    REASON
hello-example-bqbbr-2   hello-example   2            2m3s    4 OK / 4     True
hello-example-pjyvr-1   hello-example   1            3m15s   3 OK / 4     True
```

I can look more closely at each of these in the following listing with `kn revision describe`.

Listing 2.6 Looking at the first revision

```
$ kn revision describe hello-example-pjyvr-1
Name:        hello-example-pjyvr-1
Namespace:   default
Age:         5m15s
Image:       gcr.io/knative-samples/helloworld-go (pinned to 5ea96b)
Env:         TARGET=First
Service:     hello-example

Conditions:
  OK TYPE                      AGE REASON
  ++ Ready                      3h
  ++ ContainerHealthy           3h
  ++ ResourcesAvailable         3h
   I Active                     3h NoTraffic
```

2.1.3 Conditions

It's worth taking a slightly closer look at the `Conditions` table. Software can be in any number of states, and it can be useful to know what they are. A smoke test or external monitoring service can detect *that* you have a problem, but it may not tell you *why* you have a problem.

What this table gives you is four pieces of information:

1. `OK` gives the quick summary about whether the news is good or bad. The `++` signals that everything is fine. The `I` signals an informational condition—not bad, but not as unambiguous as `++`. If things were going badly, you'd see `!!`. If Knative doesn't know what's happening, you'll see `??`.
2. `TYPE` is the unique condition being described. In this table we can see four being reported. The `Ready` condition, for example, surfaces the result of an underlying Kubernetes readiness probe. Of greater interest to us is the `Active` condition, which tells us whether there's an instance of the Revision running.
3. `AGE` reports on when this Condition was last observed to have changed. In the example, these are all three hours. But they don't have to be.
4. `REASON` allows a Condition to provide a clue as to deeper causes. For example, our `Active` condition shows `NoTraffic` as its reason.

This line:

```
I Active 3h NoTraffic
```

Can be read as: "As of 3 hours ago, the `Active` Condition has an Informational status due to `NoTraffic`".

Suppose we got this line:

```
-- Ready 1h AliensAttackedTooSoon
```

We could read it as: "As of an hour ago, the `Ready` Condition become not-OK, because the `AliensAttackedTooSoon`".

2.1.4 What does `Active` mean?

When the `Active` condition gives `NoTraffic` as a reason, there are no active instances of the Revision running. Suppose we poke it with `curl`:

```
$ kn revision describe hello-example-bqbbr-2
Name:       hello-example-bqbbr-2
Namespace:  default
Age:        7d
Image:      gcr.io/knative-samples/helloworld-go (pinned to 5ea96b)
Env:        TARGET=Second
Service:    hello-example

Conditions:
  OK TYPE                    AGE REASON
  ++ Ready                    4h
  ++ ContainerHealthy         4h
  ++ ResourcesAvailable       4h
   I Active                   4h NoTraffic

$ curl http://hello-example.default.35.194.0.173.nip.io
# ... a pause while the container launches
Hello Second!
```

```
$ kn revision describe hello-example-bqbbr-2
Name:        hello-example-bqbbr-2
Namespace:   default
Age:         7d
Image:       gcr.io/knative-samples/helloworld-go (pinned to 5ea96b)
Env:         TARGET=Second
Service:     hello-example

Conditions:
  OK TYPE                    AGE REASON
  ++ Ready                    4h
  ++ ContainerHealthy         4h
  ++ ResourcesAvailable       4h
  ++ Active                   2s
```

Note that we now see `++ Active`, *without* the `NoTraffic` reason. Knative is saying that a running process was created and is active. If you leave it for a minute, it will be shut down again, and the `Active` Condition will return to complaining about a lack of traffic.

2.1.5 Changing the image

The Go programming language, aka "Golang" to its friends, "erhrhfjahaahh" to its enemies, is the Old Hotness. The New Hotness is Rust, which I have so far been able to evade forming an opinion about. All I know is that it's the New Hotness and that therefore, as a responsible engineer, I know that it's better.

This means `helloworld-go` no longer excites me, and I'd like to use `helloworld-rust`. Easily done, as shown in the following listing.

Listing 2.7 Updating the container image

```
$ kn service update hello-example \
  --image gcr.io/knative-samples/helloworld-rust
Updating Service 'hello-example' in namespace 'default':

 49.523s Traffic is not yet migrated to the latest revision.
 49.648s Ingress has not yet been reconciled.
 49.725s Ready to serve.

Service 'hello-example' updated with latest revision 'hello-example-nfwgx-3'
     and URL:
http://hello-example.default.35.194.0.173.nip.io
```

And then I poke it, as shown in the following listing:

Listing 2.8 The New Hotness says hello

```
curl http://hello-example.default.35.194.0.173.nip.io
Hello world: Second
```

Note that the message is slightly different: "Hello world: Second" instead of "Hello Second!". Not being deeply familiar with Rust, I can only suppose that it forbids excessive informality when greeting people it has never met. But it does at least prove that I didn't cheat and change the `TARGET` environment variable.

You have an important point to remember here: changing the environment variable caused the second Revision to come into being. Changing the image caused a third Revision to be created. And in fact, almost any update I make to a Service will cause a new Revision to be stamped out.

Almost any? What's the exception? It's Routes. Updating these as part of a Service won't create a new Revision.

2.1.6 Splitting traffic

I'm going to prove it by splitting traffic evenly between the last two Revisions, as shown in the following listing.

Listing 2.9 Splitting traffic 50/50

```
$ kn service update hello-example \
  --traffic hello-example-bqbbr-2=50 \
  --traffic hello-example-nfwgx-3=50

Updating Service 'hello-example' in namespace 'default':

  0.057s The Route is still working to reflect the latest desired
     specification.
  0.072s Ingress has not yet been reconciled.
  1.476s Ready to serve.

Service 'hello-example' updated with latest revision 'hello-example-nfwgx-3'
     (unchanged) and URL:
http://hello-example.default.35.194.0.173.nip.io
```

The `--traffic` parameter allows us to assign percentages to each revision. The key is that the percentages must all add up to 100. If I give `50` and `60`, I'm told that `given traffic percents sum to 110, want 100`. Likewise, if I try to cut corners by giving `50` and `40`, I'll get `given traffic percents sum to 90, want 100`. It's my responsibility to ensure that the numbers add up correctly.

Does it work? Let's see in the following listing:

Listing 2.10 Totally not a perfect made-up sequence of events

```
$ curl http://hello-example.default.35.194.0.173.nip.io
Hello Second!

$ curl http://hello-example.default.35.194.0.173.nip.io
Hello world: Second
```

It works. Half your traffic will now be allocated to each Revision.

50/50 is only one split; you may split the traffic however you please. Suppose you had Revisions called `un`, `deux`, `trois`, and `quatre`. You might split it evenly, as shown in the following listing:

Listing 2.11 Even four-way split

```
$ kn service update french-flashbacks-example \
  --traffic un=25 \
  --traffic deux=25 \
  --traffic trois=25 \
  --traffic quatre=25
```

Or you can split it so that `quatre` is getting a tiny sliver to prove itself, as shown in the following listing, while the bulk of the work lands on `trois`:

Listing 2.12 Production and next versions

```
$ kn service update french-flashbacks-example \
  --traffic un=0 \
  --traffic deux=0 \
  --traffic trois=98 \
  --traffic quatre=2
```

You don't explicitly need to set traffic to 0%; you can achieve the same by leaving out Revisions from the list, as shown in the following listing:

Listing 2.13 Implicit zero traffic level

```
$ kn service update french-flashbacks-example \
  --traffic trois=98 \
  --traffic quatre=2
```

Finally, if I'm satisfied that `quatre` is ready, I can switch over all the traffic using `@latest` as my target, as shown in the following listing:

Listing 2.14 Targeting @latest

```
$ kn service update french-flashbacks-example \
  --traffic @latest=100
```

2.2 Serving Components

As promised, I'm going to spend some time looking at some Knative Serving internals. In chapter 1 I explained that Knative and Kubernetes are built on the concept of control loops. A control loop involves a mechanism for comparing a desired world and an actual world, then taking action to close the gap between them.

But that's the boxes-and-lines explanation. The concept of a control loop needs to be embodied as actual software processes. Knative Serving has several of these, falling broadly into four groups:

1. Reconcilers, responsible for acting on both user-facing concepts like Services, Revisions, Configurations, and Routes as well as lower-level housekeeping.
2. The "Webhook", responsible for validating and enriching the Services, Configurations, and Routes that users provide.

3. Networking controllers that configure TLS certificates and HTTP ingress routing.
4. The Autoscaler/Activator/Queue-Proxy triad, which manage the business of comprehending and reacting to changes on traffic.

2.2.1 The Controller and Reconcilers

Let's talk about names for a second.

Knative has a component named `controller`, which is a bundle of individual ”Reconcilers”. Reconcilers are controllers in the sense I discussed in chapter 1: a system that reacts to changes in the difference between desired and actual worlds. Reconcilers are controllers, but the `controller` isn't a controller. Got it?

No? You're wondering why the names are different? The simplest answer is, to avoid confusion about what's what. That may sound silly. Bear with me, I promise it will make sense.

At the top, in terms of actual running processes managed directly by Kubernetes, Knative Serving only has one `controller`. But in terms of logical processes, Knative Serving has several controllers, running in goroutines inside the single physical `controller` process. Moreover, Reconciler is a Golang interface that implementations of the controller pattern are expected to implement.

So that we don't wind up saying “the controller controller” and “the controllers that run on the controller” or other less-than-illuminating naming schemes, we have instead two names: `controller` and Reconciler.

Each Reconciler is responsible for some aspect of Knative Serving's work, which falls into two categories. The first category is simple to understand—it's the reconcilers responsible for managing the developer-facing resources. These reconcilers are called `configuration`, `revision`, `route`, and `service`.

For example, when you use `kn service create`, the first port of call will be for a Service record to be picked up by the `service` controller. When you used `kn service update` to create a traffic split, you sent the `route` controller outside to work for you. I'll touch on several of these controllers in coming chapters.

Reconcilers in the second category work behind the scenes to carry out essential lower-level tasks. These are `labeler`, `serverlessservice`, and `gc`. The `labeler` is part of how networking works; it essentially sets and maintains labels on Kubernetes objects that networking systems can use to target them for traffic. I'll touch on this when we get to routing.

The `serverlessservice` reconciler is part of how the Activator works. It reacts to and updates `ServerlessService` records (say that five times fast!). These are also mostly about networking in Kubernetes-land. I'll go into more depth on this in the routing chapter.

Last, the `gc` reconciler performs garbage-collection duties, and hopefully, you'll never need to think about it again (see figure 2.1).

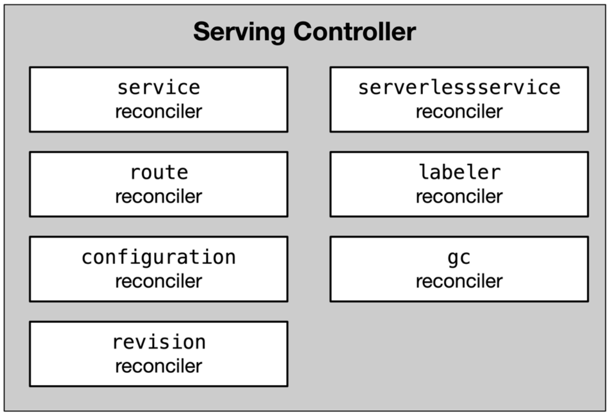


Figure 2.1 The serving controller and its Reconcilers.

2.2.2 The webhook

Things go wrong. A great deal of software engineering is centered on ensuring that when things *do* go wrong, they at least choose to go wrong at the least-painful and/or least-Tweetable moment. Type systems, static analysis, unit test harnesses, linters, fuzzers, the list goes on and on. We submit to their nagging because solving the mysteries of fatal errors in production is less fun than Agatha Christie made it out to be.

At runtime, Serving relies on the completeness and validity of information provided about things you want to manage (for example, Services) and how you want it to behave generally (for example, Autoscaler configuration). This brings us to the webhook, which validates and augments your submissions. Like the controller, it's a group of logical processes that are collected together into a single physical process for ease of deployment.

The name "webhook" is a little deceptive, because it describes the implementation rather than its actual purpose. If you're familiar with webhooks, you might have thought that its purpose was to dial out to an endpoint that you provide. Not so. Or perhaps it was an endpoint that you could ping yourself. Closer, but still incorrect. Instead, the name comes from its role as a Kubernetes "admissions webhook". When processing API submissions, the Knative Webhook is registered as the delegated authority to inspect and modify Knative Serving resources. A better name might be "Validation and Annotation Clearing House" or perhaps the "Ditch It or Fix It Emporium". But "webhook" is what we have.

The webhook's principal roles include:

- **Setting default configurations:** This includes values for timeouts, concurrency limits, container resources limits, and garbage collection timing. You only need to set values you want to override. I'll touch on these as needed.
- **Injecting routing and networking information** *into Kubernetes:* I'll discuss this when I get to routing.

- **Validating** *that users didn't ask for impossible configurations:* For example, the webhook will reject negative concurrency limits. I'll refer to these when needed.
- **Resolving** *partial Docker image references to include the digest:* For example, `example/example:latest` would be resolved to include the digest, so it looks like `example/example@sha256:1a4bccf2. . .`I'm going to revisit this topic a few times, but generally, this is one of the best things Knative can do for you, and the webhook deserves the credit for it.

2.2.3 Networking controllers

Early versions of Knative relied directly on Istio for core networking capabilities. That hasn't entirely changed. In the default installation provided by the Knative project, Istio will be installed as a component and Knative will use part of its capabilities.

However, as it has evolved, more of Knative's networking logic has been abstracted up from Istio. Doing so allows swappability of components. Istio might make sense for your case, but it's featuresome and might be overkill. But you might have Istio provided as part of your standard Kubernetes environment. Knative will extend to either approach.

Knative Serving requires that networking controllers answer for two basic record types: `Certificate` and `Ingress`.

Certificates

TLS is essential to the safety and performance of the modern internet, but the business of storing and shipping TLS certificates has always been inconvenient. The Knative Certificate abstraction provides information about the TLS certificate that is desired, without providing it directly.

For example, TLS certificates are scoped to particular domain names or IP addresses. When creating a Certificate, a list of `DNSNames` is used to indicate what domains the Certificate should be valid for. A conforming controller can then create or obtain certificates that fulfill that need.

I'll have more to say about Certificates when we dive into routing.

Ingress

Routing traffic is always one of those turtles-all-the-way-down affairs. Something, somewhere, is meeting traffic at the boundary of your system. In Knative, that's the Ingress.[15]

Ingress controllers act as a single entrance to the entire Knative installation. They convert Knative's abstract specification into particular configurations for their own routing infrastructure. For example, the default `networking-istio` controller will convert a Knative Ingress into an Istio `Gateway`.

Knative Ingress has several implementations, which I'll revisit later.

[15] This is distinct from a Kubernetes Ingress.

2.2.4 Autoscaler, Activator, and Queue-Proxy

These three work together quite closely, so I've grouped them under the same heading (see figure 2.2).

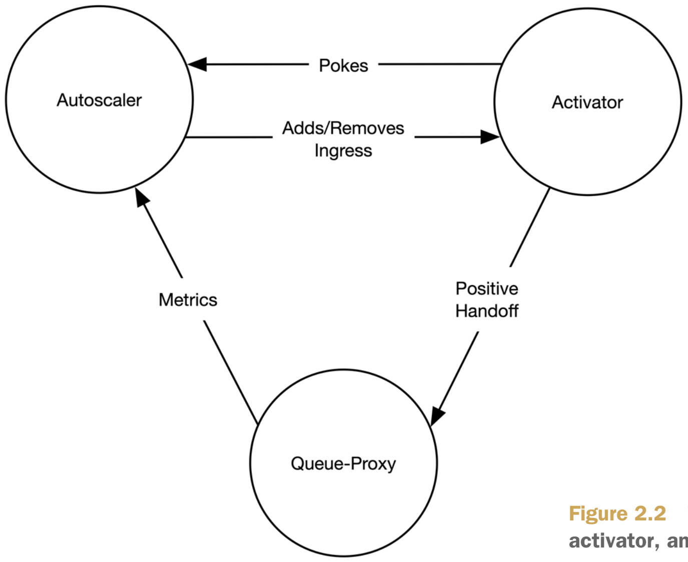


Figure 2.2 The triad of Autoscaler, activator, and queue-proxy.

NOTE When I talk about "the Autoscaler", I'm generally referring to the Knative Pod Autoscaler (KPA). This is the out-of-the-box Autoscaler that ships with Knative Serving. It is possible to configure Knative to use the Kubernetes Horizontal Pod Autoscaler (HPA) instead. In future the KPA might be retired as the HPA becomes better suited to serverless patterns of activity, but at time of writing that seemed to be a fairly distant milestone. In this book, I focus exclusively on the KPA.

The autoscaler is the easiest to give an elevator pitch for: observe demand for a Service, calculate the number of instances needed to serve that demand, then update the Service's scale to reflect the calculation. You've probably recognized that this is a supervisory control loop. Its desired world is "minimal mismatch between demand and instances". Its output is a scale number that becomes the desired world of a Service control loop (see figure 2.3).

It's worth noting that the Knative Pod Autoscaler operates solely through horizontal scaling. That is, launching more copies of your software. "Vertical scaling" means launching it with additional computing resources. In general, vertical scaling is simpler—you pay more for a beefier machine. But the costs are highly nonlinear, and you have an upper limit to what can be achieved. Horizontal scaling typically requires deliberate architectural decisions to make it possible, but once achieved can face higher demands than any one machine could handle. The Knative Pod Autoscaler

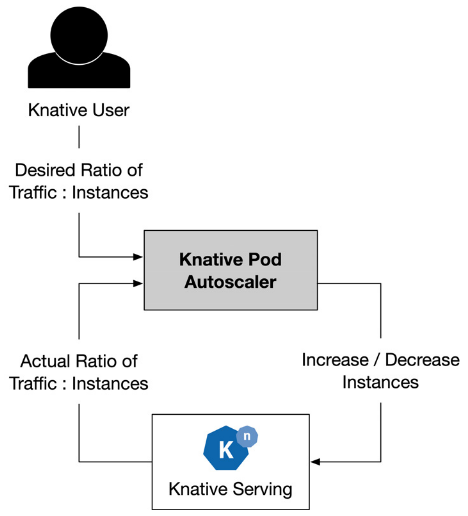

Knative User
Desired Ratio of Traffic : Instances
Knative Pod Autoscaler
Actual Ratio of Traffic : Instances
Increase / Decrease Instances
Knative Serving

Figure 2.3 The Knative Pod Autoscaler is a control loop.

assumes you've done the work to ensure that instances coming and going at a rapid clip won't be overly disruptive.

When no traffic exists, the desired number calculated by the Autoscaler will eventually be set to zero. This is great, right until a new request shows up without anything ready to serve it. We could plausibly bounce the request with an HTTP `503 Service Unavailable` status—perhaps even, in a fit of generosity, providing a `Retry-After` header. The problem is that (1) humans hate this, and (2) vast amounts of upstream software assumes that network requests are magical and perfect and can never fail. They'll either barfing on *their* users or, more likely, ignore your `Retry-After` and hammer the endpoint into paste. Not to mention (3), which is that all of this will be screencapped and mocked on Reddit.

But what to do when no instances are running—the dreaded cold start? In this case, the Activator is a traffic target of last resort; the Ingress will be configured to send traffic for routes with no active instances to the Activator (see figure 2.4).

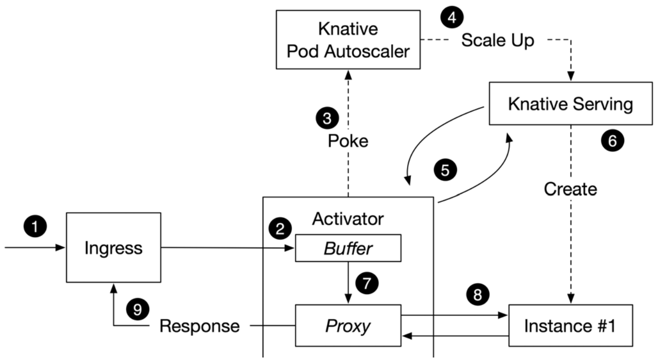


Figure 2.4 The Activator's role in managing cold starts.

1. The Ingress receives a new request. The Ingress sends the request to its configured target, which is the Activator.
2. The Activator places the new request into a buffer.
3. The Activator pokes the Autoscaler. The poke does two things: first, it carries information about requests waiting in the buffer. Second, the arrival of a poke signal prompts the Autoscaler to make an immediate scaling decision, instead of waiting until the next scheduled decision time.
4. After considering that a request is waiting to be served, but there are zero instances available to serve it, the Autoscaler decides that there ought to be one instance running. It sets a new scale target for Serving.
5. While waiting for the Autoscaler and Serving to do their work, the Activator polls Serving to see if any instances are live.
6. Serving's actions cause Kubernetes to launch an instance.
7. The Activator learns from its polling that an instance is now available and moves the request from its buffer to a simple proxy service.
8. The proxy component sends the request to the instance, which responds normally.
9. The proxy component sends the response back to the Ingress, which then sends it back to the requester.

Does this mean all traffic flows through the Activator? No. The Activator remains on the data path during the transition from "no instances" to "enough instances". Once the Autoscaler is satisfied that it has enough capacity to meet current demand, it updates the Ingress, changing the traffic target from the Activator to the actual running instances. At this point that Activator no longer has any role in proceedings.

The exact timing of this update depends mostly on how much traffic has piled up and how long it takes to launch instances to serve it. Imagine that 10,000 requests arrive and the Activator then sprayed them at the first instance foolish enough to stick its head above the trenches. Instead, the Activator throttles its proxy until capacity catches up with demand. Once requests are flowing smoothly, the Autoscaler's own logic will remove the Activator from the data path.

The final component of this triad is the Queue-Proxy. This is a small proxy process that sits between your actual software and arriving traffic. Every instance of your Service will have its own Queue-Proxy, running as a sidecar. Knative does this for a few reasons. One is to provide a small buffer for requests, allowing the Activator to have a clear signal that a request has been accepted for processing (this is called "positive handoff"). Another purpose is to add tracing and metrics to requests flowing in and out of the Service.

We'll dig into those two functions—positive handoff and metrics—later in the book.

Summary

- `kn` is a CLI tool for interacting with Knative, including Serving.
- `kn service` lets you view, create, update, and configure Knative Services, including splitting traffic between Revisions.
- Knative Serving has a `controller` process, which is a collection of components called Reconcilers. Reconcilers act as feedback controllers.
- There are Reconcilers for Serving's core record types (Service, Route, Configuration, and Revision), as well as housekeeping Reconcilers.
- Knative Serving has a `webhook` process, which intercepts new and updated records you submit. It can then validate your submissions and inject additional information.
- The Knative Pod Autoscaler is a feedback control loop. It compares the ratio of traffic to instances and raises or lowers the desired number of instances that the serving `controller` controls.
- The Activator is assigned Ingress routes when no instances are available. This assignment is made by the Autoscaler.
- The Activator is responsible for poking the Autoscaler when new requests arrive, to trigger a scale-up.
- While instances are becoming available, the Activator remains on the data path as a throttling, buffering proxy for traffic.
- When the Autoscaler believes there is enough capacity to serve demand, it removes the Activator from the data path by updating Ingress routes.
- Knative Serving's Networking is highly pluggable. Core implementations are provided for two functions: Certificates and Ingress.
- Certificate controllers accept a definition of desired Certificates and must provision new certificates or map existing certificates into your software.
- Ingress controllers accept Routes and convert these into lower-level routing or traffic management configurations.
- Ingress controller implementations include Istio-Gateway, Gloo, Ambassador, and Kourier.

References

- github.com/cppforlife/knctl

3

Configurations and Revisions

This chapter covers

- Understanding the brief history of deployments up to progressive deployment
- Learning the anatomy of Configurations
- Describing the anatomy of Revisions

My focus in this chapter is provide a guided tour of Serving's dynamic duo, Configuration and Revision. This separation into two concepts isn't for the mere joy of complexity. To explain the motivation, I first give a fictionalized account of the history of software deployment, starting somewhere in the late Triassic period up until the current, slightly more advanced era of Thought Leadership.

After the history lesson, I start with Configurations. These are the main way you'll describe your software and your intentions to Knative Serving. The coverage of Configurations is necessarily brief, because Configurations mostly exist to stamp out Revisions.

My discussion of Revisions will be substantially longer, because we have much ground to cover. We'll look at containers, container images, commands and environments, volumes, consumption limits, ports and probes, concurrency, and timeouts. The style is narrative, but you can skip things you don't care about right now and refer to them later.

Before we begin, I want you to refresh a key concept, which is that Revisions are created when a Configuration is created or changed. They don't have independent existence. In the table of contents for this chapter, you might have formed the sweet illusion that you can deal entirely with Configurations in one place and entirely with Revisions in a different place. That isn't so. The knobs and dials I describe as being part of a Revision have to get there through a Configuration. While writing about Revisions with my left hand, my right hand is running commands that refer to Configurations. If you get lost or confused, reorient yourself to this landmark: Revisions are created only when Configurations are changed or created.

3.1 Those who cannot remember the past are condemned to redeploy it

Maybe you remember what you had for breakfast. Maybe you don't. But if by lunch your stomach feels ill and you go to a doctor, she'll want to know what you ate. Telling her "well, I'm not hungry now, so I guess I ate *something*, but I'm not sure what it was" is unlikely to spark much diagnostic insight. (You'll still receive a bill.)

Left to its own devices, Kubernetes will make the world appear changeless and timeless, a permanent "steady state". Whenever the desired and actual worlds become misaligned, it takes action to reconcile the difference. Afterwards, it doesn't care that the disturbance ever occurred. It doesn't remember. The ripple of the disturbance has faded, leaving the placid pond of production[16].

When an outside observer wishes to reconstruct history, they may be out of luck.

Like the doctor, we may, for various reasons, wish to know what led us to the current situation, whether for diagnosis or treatment. When you operate without history, you're pretty much swinging across the YOLO ravine on a frayed rope. Eventually it will snap, sending your hurtling down into the metaphor-crocodiles at the bottom (I've trained them to mock you).

This has led to a small cottage industry of mechanisms for capturing history from, or injecting history into, Kubernetes. For example:

- Various fields, annotations, and metadata, such as `kubernetes.io/change-cause that` provide limited historical or causal information directly on a particular Kubernetes record.
- The inbuilt `Deployment` mechanism provided by vanilla Kubernetes maintains `deployment.kubernetes.io/revision` annotations on `ReplicaSets` that it controls, which provides a partial history of the Deployment.

[16] It can be argued that I'm wrong, that there are many ways to get a sense of history: logs, Kubernetes events, and so forth. But these can be ephemeral, and besides, alliteration is always alluring to awful authors.

- The Kubernetes auditing system can be configured to emit an extremely detailed log of changes, allowing later reconstruction of history as seen by the Kubernetes API server.
- Tools and practices for "GitOps", where changes to be submitted are first checked into a Git repository before being applied to a Kubernetes cluster.
- Specialized history-visualization tools such as Salesforce's Sloop.
- The teeming multitudes of observability and/or monitoring tools and services.

Broadly, these fill two related but distinct purposes:

1. **What's the history of the cluster?** How did it get to the current condition? None of the solutions above fully covers these. Either they focus on changes to the desired world (Sloop, GitOps), or they produce data that can hint at changes to the actual world (metrics and logs), or an incomplete mix of these (Kubernetes Audit). But not both.
2. **Can I go back in time, please?** The current version of the software is wrong, a previous version was not As wrong, so we need to switch back to the previous version. The need for time travel is projected across the entire hierarchy of control, from within a cluster way back into developer-land, because rolling back to a previous version can take many forms: a `git revert`, Spinnaker canary analysis, GitOps, and there are many others.

Knative Serving wants a more general version of the time-travel capability. It's not enough to select *a* version to run. Instead Knative Serving wants to run one or *multiple* versions concurrently. If I have versions 1, 2, and 3 of my software, I want to run a mix of 1, 2, and 3 (1 only, 1 and 2, 1 and 3 . . .). And I want to change the mix whenever I want.

Multiple versions? At once? Am I crazy? Perhaps, but to decide for yourself, I need to tell you a story.

3.2 The bedtime story version of the history of deployment as a concept

Another way to look at the two desires for causality is this:

1. Something has gone wrong. Why?
2. We changed something and stuff broke. Let's undo the change.

The latter is what I want to focus on, because it's an ancient problem. For as long as there have been production systems, there have been mandates that it cannot be allowed to stop running during business hours. An obvious logic then unfolded in the early years.

AXIOM 0 If it breaks, you're fired.

AXIOM 1 The system sometimes breaks when it is changed.

AXIOM 2 The changes that cause breakage were due to human error in either the change, or in how the change was applied, or how the change interacted with other changes.

THEOREM 1 Therefore: Don't change anything.

QED. High fives and slide rules for everyone!

But wait, here comes the boss's boss's boss . . .

AXIOM 3 We need to integrate with Grot-O-Matic 7.36, because our customer will deliver records in the form of blurry Klingon hieroglyphs via FTP-over-pigeon-droppings every 24 hours. Oh, and if you don't make this change, you're fired.

THEOREM 2 Ahem. Well. I guess we need to change something, or we're fired. But if it breaks, we're fired. Therefore: be careful, extremely careful, about changes, and slather everything that moves with documentation to prove that It Wasn't Me, Boss.

Fewer high fives this time.

And so, for example, many firms had Change Approval Boards and required would-be mutators to account for their sins in a uniform way. Then the Change Window would open every quarter, and if you were lucky, your change would make it through before the window slammed shut again. And god help you if your change got in . . . but you realized it was wrong.

Then later we developed tools to make this much less painful. For example, version control systems. These were *around* in various forms for ages, but when Git and GitHub came along they became standard, the way many folks work. At the same time folks began to evolve the concepts of Continuous Integration and Continuous Deployment.

Here there arose three new possibilities, the Blue/Green deployment, the Canary Deployment, and the Progressive Deployment.

3.2.1 *The blue/green deployment*

You have a version of your software already running and serving traffic. Let's call this "Blue" (see figure 3.1).

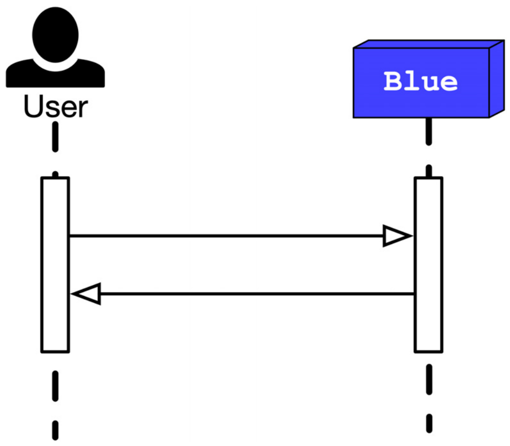


Figure 3.1 Blue.

You now wish to deploy a new version of your software, let's call it "Green".

A first approach might be to stop Blue, then deploy Green. The time between "Stop Blue" and "Start Green" is scheduled downtime. That was roughly the state of the world for Theorem 2 (see figure 3.2).

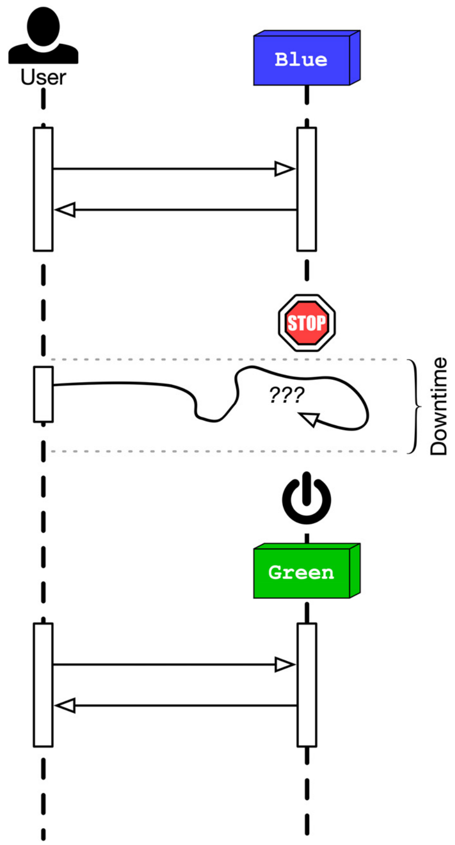


Figure 3.2 **Scheduled downtime Blues.**

Scheduled downtimes are still downtimes. It would be nice if we *didn't* have to stop Blue first. And thanks to the magic of load balancers and proxies and gateways and routers, we don't have to. What we do instead is (see figure 3.3):

1. Start Green
2. Switch traffic from Blue to Green
3. Stop Blue

From here, certain tools will tick-tock between Green and Blue as the running version. They take turns. This approach is popular because all the software needs to do is look at a name or label to see what's in production ("I see we're running Green right now") and to pick the other value ("So I will call the next version Blue during this process").

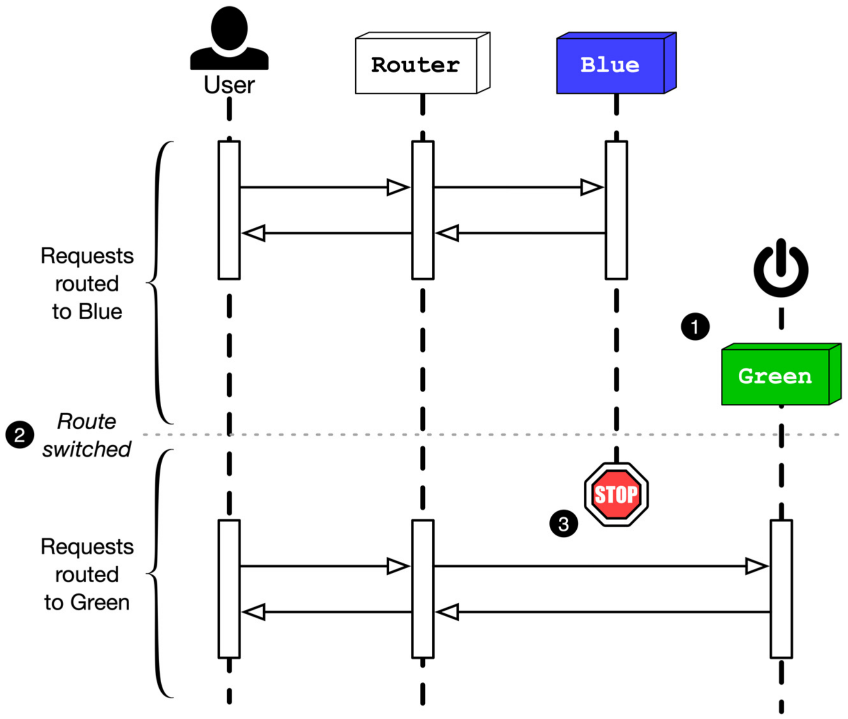


Figure 3.3 A blue/green deployment.

The system managing Blue/Green deployment won't need to maintain state about what's what.

Other systems prefer to keep the meanings stable. The running system is always Blue, the next version is always Green. That works fine so long as something will keep records.

Upgrades without taking a scheduled downtime is the basic motivation for blue/green deployments. But there are other benefits. One is that we can now ensure Green is "good" before we switch to Blue. Or alternatively, if Green is "bad", we can more easily roll the system back to Blue, because our muscle for switching traffic is well-developed. To ensure rollbacks are fast, we can keep Blue running for a little while until Green has proved itself worthy of our trust.

3.2.2 *The canary deployment*

Blue/green deployments are the minimum you should accept from any system, tool, or crazy shell script written by the longest-serving engineer, and so on, that's passed off as continuous deployment. Done properly, blue/green deployment is a safe way to deploy software.

But like many conservative, ultra-safe systems, it can be wasteful. Here's an example (assuming that my production system is always called Blue and my next version is always called Green):

During the normal production steady state, I need enough capacity to run Blue. But during the blue/green deployment, I need enough capacity for Blue *and* Green. In fact, I may need additional capacity on top of that to deal with things such as database migrations, files being downloaded to new instances of Green, additional consumption due to new Green features, and so on, as well as the overhead imposed on my control plane by the business of rolling out and cutting over to Green. And for safety, I want to keep Blue around until I'm satisfied that Green won't need to be rolled back.

We have two considerations that are at odds. The first is efficiency, the second is safety. A canary deployment helps with both of these problems.

In a canary deployment (see figure 3.4), we roll out a reduced-size sample of Green to run alongside Blue. For example, it might be that in our normal situation, we'd deploy 100 copies of our software. Instead of having 100 Blue and 100 Green during the blue/green process, we might start with 100 Blue and 1 Green. This single copy is the "canary"[17].

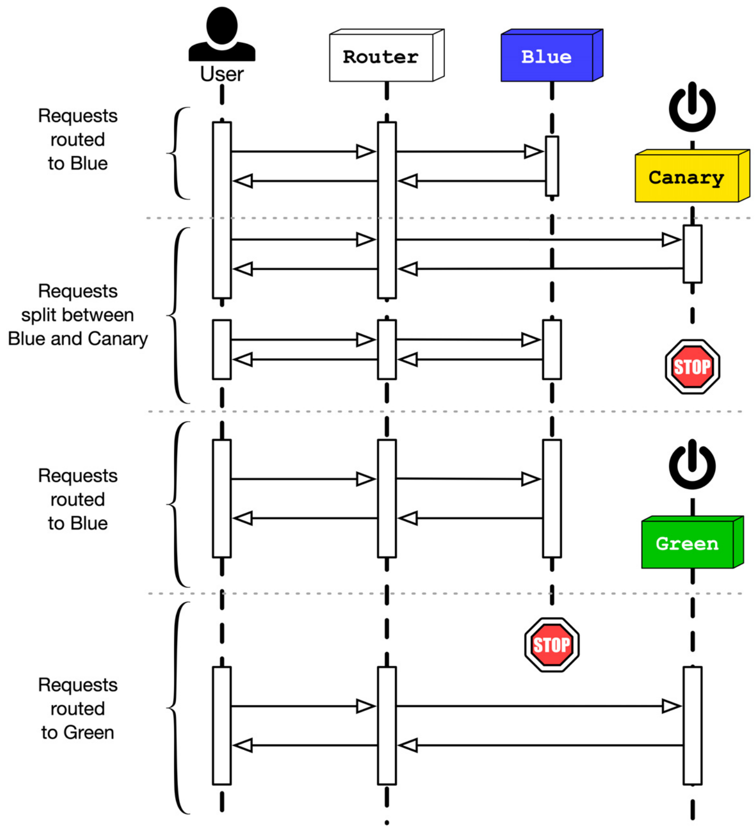


Figure 3.4 A canary deployment.

[17] "Canary" here is an analogical reference to the birds Victorian-era coal miners brought with them to deep pits and also, one supposes, to their version of tech conventions. Carbon monoxide, somewhat like vaporware announced during a keynote speech, is colorless, odorless, and lethal, but it can build slowly. The canary, being small, would die earlier than miners and so they would get early warning of the danger.

Instead of cutting *all* traffic over to Green, we'll instead send a fraction of requests to it and see what happens. Then we might raise the number of Green copies to 10. If we're satisfied with how they run, we'll then proceed to fully deploy Green. We'll then cut over and immediately remove Blue. After all, our canaries established that Green was safe, so rollback speed is a less critical consideration.

3.2.3 Progressive deployment

Mind you, what we're doing is still fairly wasteful, insofar as we reach a peak level of capacity consumption that's around 2x the steady state level. Now we arrive at the third evolution of our approach: progressive deployment.

In progressive deployment, we keep the consumption level much closer to steady state. Say we have 100 instances of Blue. We first perform a blue/green deployment of *one* instance, instead of our entire system. Afterward, we have 99 Blue and 1 Green. We run this 1 Green as a canary for a while. If we're happy, we perform another blue/green deployment, this time for 9 instances. Afterward, there are 90 Blue and 10 Green. Then, finally, we might complete the rollout of Green, retiring Blue as we go.

We have many permutations here. For example, to limit the peak surge, we might roll out 1 instance at a time, or a fixed percentage at a time, rather than perform a blue/green deployment for the entire pool. Progressive deployment is essentially the logical endpoint that arises once you can split traffic. It limits risk through canaries, it limits utilization through upgrading a fraction at a time (see figure 3.5).

3.2.4 Back to the future

What does Knative Serving do? Blue/green? Canary? Progressive?

The answer is: all of these. Sort of.

In my previous discussion, I talked about two major themes: cluster history and safe, efficient deployments. Knative Serving sets out to answer these with two core types: the `Configuration` and the `Revision`. The connection is that each Revision is a snapshot of a Configuration, and a Configuration is the template for the "most recent" Revision. An analogy often used is to git: you can think of each Revision is a particular commit. Then a Configuration is the `HEAD` of a branch of Revisions.

How does this design connect back to my discussion above? Let's review:

- **History:** Revisions represent snapshots of Configurations over time, giving a partial history of your system.
- **Time travel and deployments:** Multiple Revisions can receive traffic for a single endpoint. This allows the blue/green, canary, and progressive deployment patterns.

But there's something new here. Previously the business of deployment was a process with a binary outcome. You had version N running, something occurs, after which you run version N+1. That process might include a period of both running, but the end point was *one version*.

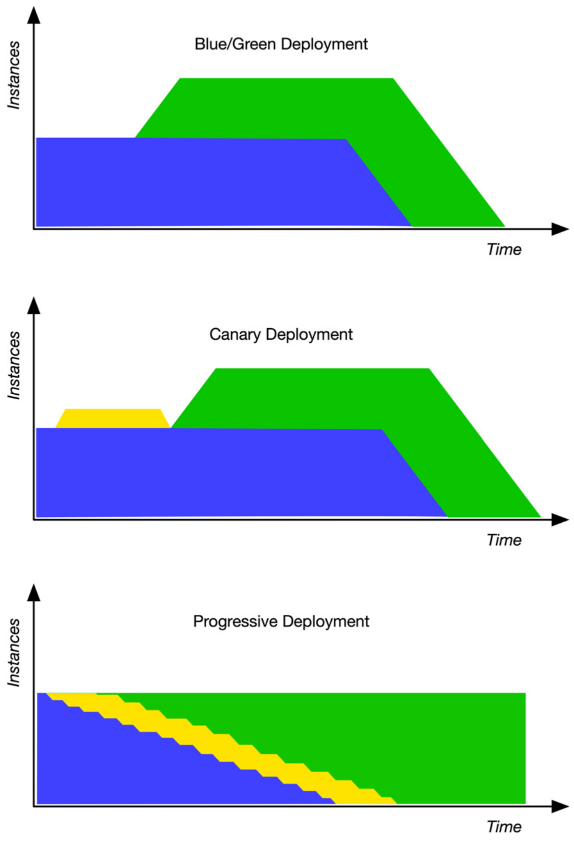


Figure 3.5 Utilization of deployments compared.

Knative Serving makes this easy, but it isn't *limited* to it. You can, if you wish, run any number of Revisions, so while the binary outcome is conventional, it isn't guaranteed. Deployment is now a fuzzy concept rather than a finite state machine.

3.3 The anatomy of Configurations

A Configuration is a definition of your software. Up to now I've avoided showing one in the flesh. I wanted to show you `kn` first and avoid being too Kubernetes-centric. But now it's time for us to accept our fates as enterprise YAML wranglers. It will be easiest for me to explain Configurations by using the YAML form.

To ease the transition, here's a `kn` command we used in chapter 2, as shown in the following listing.

Listing 3.1 The before

```
$ kn service create hello-example \
  --image gcr.io/knative-samples/helloworld-go \
  --env TARGET="First"
```

And here's an equivalent Configuration YAML file, as shown in the following listing.

Listing 3.2 The after

```
apiVersion: serving.knative.dev/v1
kind: Configuration
metadata:
  name: helloworld-example
spec:
  template:
    spec:
      containers:
      - image: gcr.io/knative-samples/helloworld-go
        env:
        - name: TARGET
          value: "First"
```

Everything from the `kn` CLI is present in the YAML version. We have a name, a container, and an environment variable. We also have a fair amount more, including a fair amount of whitespace.
This document isn't meant to be used by `kn`. Those of you who have already drunk the Kule-aid recognize it as a Kubernetes record, which would typically be submitted to Kubernetes using `kubectl apply`[18]. Consequently, it sports elements that are there to fit into Kubernetes conventions. For example, the `apiVersion` and `kind` elements are mostly there to identify the record type so that relevant controllers can be alerted to creations and updates. The `metadata` section is, under the hood, a Kubernetes type that can store many kinds of information. We provided a `name` here, but I have many more possibilities that I'm going to bring up as we go along.

Last, there's this curious little hop-skip-hop:

Listing 3.3 Yo dawg, I heard you like specs

```
spec:
  template:
    spec:
```

This isn't an accident, because there are three "things" here:

1. The outermost `spec` belongs to the `Configuration` itself. The name `spec` is another Kubernetes convention, meaning "desired world for this thing".
2. The `template` is a `RevisionTemplateSpec`, which I'll discuss in a second.
3. The innermost `spec` is a `RevisionSpec`. That is, it's the `spec` of a `Revision`.

Hopefully, this tips you off to the fact that the `template` is the "thing" that's converted into Revisions. But it goes further than that: the template is what *causes* Revisions to be created.

This is important, so I'll repeat it.

[18] Under the hood, kn is doing the same thing that kubectl is doing. It takes a YAML document and ships it off to the Kubernetes API server.

Changing the `template` *causes* Revisions to be created.

And in fact, this is true from the moment that I first submit a Configuration. I can see this using `kubectl` directly, as shown in the following listing.

Listing 3.4 Using raw kubectl

```
$ kubectl apply -f example.yaml
configuration.serving.knative.dev/helloworld-example created

$ kubectl get configurations
NAME                 LATESTCREATED              LATESTREADY                READY   REASON
helloworld-example   helloworld-example-8sw7z   helloworld-example-8sw7z   True

$ kubectl get revisions
NAME                       CONFIG NAME          K8S SERVICE NAME           GENERATION   READY   REASON
helloworld-example-8sw7z   helloworld-example   helloworld-example-8sw7z   1            True
```

I can see that by submitting the Configuration, I prompted Serving to create a Revision as well. That Revision isn't meaningfully different to one created by `kn`, as I can see in the following listing with `kn revision list`.

Listing 3.5 `kn revision list`

```
$ kn revision list
NAME                        SERVICE          GENERATION   AGE     CONDITIONS   READY   REASON
helloworld-example-8sw7z                     1            2m24s   3 OK / 4     True
```

Your eye might be drawn to `CONDITIONS` and its value, `3 OK / 4`. Despite appearances, this doesn't mean your Revision is one-quarter evil. It refers to something we've seen before: Revisions scale to zero when there's no traffic. You can see this in the following listing with `kn revision describe`.

Listing 3.6 `kn revision describe` to the rescue

```
$ kn revision describe helloworld-example-8sw7z
Name:         helloworld-example-8sw7z

# ... snipped ...

Conditions:
  OK TYPE                     AGE REASON
  ++ Ready                     2d
  ++ ContainerHealthy          2d
  ++ ResourcesAvailable        2d
   I Active                    2d NoTraffic
```

Remember that `++` means "OK". Counting the conditions from top to bottom, three of them are `++` out of four. Hence `3 OK / 4`.

As well as creating Revisions via the creation of Services or Configurations, I can also create *new* Revisions by *editing* a Configuration or Service, as shown in the following listing. In chapter 2, I used `kn service update` to amend things.

Listing 3.7 Updating using `kn`

```
$ kn service update hello-example --env TARGET=Second
```

This command amends a Service, which amends the Configuration, which causes a new Revision to pop into existence.

The equivalent would be to edit my YAML to look like the following listing.

Listing 3.8 The second YAML

```
apiVersion: serving.knative.dev/v1
kind: Configuration
metadata:
  name: helloworld-example
spec:
  template:
    spec:
      containers:
      - image: gcr.io/knative-samples/helloworld-go
        env:
        - name: TARGET
          value: "Second"
```

And then submit it with `kubectl` again in the following listing.

Listing 3.9 Amending with `kubectl apply`

```
$ kubectl apply -f example.yaml
configuration.serving.knative.dev/helloworld-example configured

$ kubectl get configurations
NAME                 LATESTCREATED              LATESTREADY                READY   REASON
helloworld-example   helloworld-example-j4gv5   helloworld-example-j4gv5   True

$ kubectl get revisions
NAME                       CONFIG NAME          K8S SERVICE NAME           GENERATION   READY   REASON
helloworld-example-8sw7z   helloworld-example   helloworld-example-8sw7z   1            True
helloworld-example-j4gv5   helloworld-example   helloworld-example-j4gv5   2            True
```

Now I can see *two* Revisions, but there's still only *one* Configuration with the name `helloworld-example`. As a helpful hint, Revisions have a generation count, which is set at its creation time. Generations are monotonic numbers. Each Revision will be a higher number than earlier Revisions, but there's no firm guarantee that all numbers will be sequential. For example, you might have deleted Revisions yourself.

3.3.1 *Configuration* `status`

Is that all that's interesting about Configurations? Not quite. We've shown the `spec` (a Desired World, in chapter 1 terms). But you also have a `status` section, which is set by the `configuration` Reconciler (an Actual World, in chapter 1 terms). I can use `kubectl` and the handy JSON utility `jq` to display my Configuration status[19], as shown in the following listing.

[19] This example of using jq to stitch up kubectl output is something of a litmus test. On one side, you have the "Unix pipes are the high watermark of software design" crowd, for whom hoarding one-liners like some kind of CLI Smaug is right and worthy. Then there are "Human-Computer interface research did not end in 1970, which is now 50 goddamn years ago" weirdoes like me, who harbor radical notions about being allowed to do damn work without having to learn Yet Another Minilanguage. This divide is a poetic illustration of why Knative is at all necessary. The developer experience for Kubernetes isn't "batteries included". It's "learn chemistry and try not to poison yourself with lead".

Listing 3.10 Looking at a status with `kubectl` and `jq`

```
$ kubectl get configuration helloworld-example -o json | jq '.status'
{
  "conditions": [
    {
      "lastTransitionTime": "2019-12-03T01:25:34Z",
      "status": "True",
      "type": "Ready"
    }
  ],
  "latestCreatedRevisionName": "helloworld-example-j4gv5",
  "latestReadyRevisionName": "helloworld-example-j4gv5",
  "observedGeneration": 2
}
```

You can see two basic sets of information. The first is `conditions`, which I talk about more later, during my discussion of Revisions. The second set of information is the trio of `latestCreatedRevisionName`, `latestReadyRevisionName`, and `observedGeneration`.

Let's start with `observedGeneration`. Earlier you saw that each Revision is given a generation number. It came from `observedGeneration`. When you apply an update to the Configuration, the `observedGeneration` gets incremented. When a new Revision is stamped out, it takes that number as its own.

`latestCreatedRevisionName` and `latestReadyRevisionName` are the same here, but they need not be. Simply creating the Revision record doesn't guarantee that actual software is up and running. These two fields make the distinction. In practice, it allows you to spot the process of a Revision being acted on by lower-level controllers.

These fields are useful for debugging. If you submit an updated Configuration but don't otherwise see expected behavior, compare them. For example, suppose I update my Configuration from `foo-1` to `foo-2`, but didn't see any change in behavior when sending HTTP requests. If I check and see that `latestCreatedRevisionName` is `foo-2` and `latestReadyRevision` is `foo-1`, then I know something is wrong with `foo-2` that merits further investigation.

3.3.2 Taking it all in with kubectl describe

The observant among you have noticed that `kn` has talked about Services, but I've talked about Configurations. This is basically because `kn` doesn't treat Configurations as a standalone concept, it instead sweeps them into Services as the unit of interaction. Given Knative's goals of simplifying and smoothing out the developer experience, that's quite reasonable.

It *does* make it a bit trickier to get a nice readout on a Configuration by itself, though. For that purpose, I need to drop down from `kn` to `kubectl`. The helpful `describe` subcommand allows me to take a closer look at the Configuration in the following listing, as Kubernetes sees it.

Listing 3.11 Inspecting a Configuration with `kubectl describe`

```
$ kubectl describe configuration helloworld-example

Name:          helloworld-example
Namespace:     default
Labels:        <none>
Annotations:   serving.knative.dev/creator: jacques@example.com   ❶
               serving.knative.dev/lastModifier: jacques@example.com
API Version:   serving.knative.dev/v1
Kind:          Configuration
Metadata:
  Creation Timestamp:  2019-12-03T01:17:28Z
  Generation:          ❷
  Resource Version:    8778016
  Self Link:
     /apis/serving.knative.dev/v1/namespaces/default/configurations/helloworl
     d-example
  UID:                 ac192f54-156a-11ea-ae60-42010a800fc4
Spec:
  Template:
    Metadata:
      Creation Timestamp:  <nil>
    Spec:      ❸
      Container Concurrency:  0
      Containers:
        Env:
          Name:   TARGET
          Value:  Second
        Image:    gcr.io/knative-samples/helloworld-go
        Name:     user-container
        Readiness Probe:
          Success Threshold:  1
          Tcp Socket:
            Port:  0
        Resources:
      Timeout Seconds:  300
Status:     ❹
  Conditions:
    Last Transition Time:        2019-12-03T01:25:34Z
    Status:                      True
    Type:                        Ready
  Latest Created Revision Name:  helloworld-example-j4gv5
  Latest Ready Revision Name:    helloworld-example-j4gv5
  Observed Generation:           2
Events:      ❺
  Type    Reason              Age   From                      Message
  ----    ------              ----  ----                      -------
  Normal  Created             14m   configuration-controller  Created
     Revision "helloworld-example-8sw7z"
  Normal  ConfigurationReady  14m   configuration-controller  Configuration
     becomes ready
  Normal  LatestReadyUpdate   14m   configuration-controller
     LatestReadyRevisionName updated to "helloworld-example-8sw7z"
  Normal  Created             6m28s  configuration-controller  Created
     Revision "helloworld-example-j4gv5"
  Normal  LatestReadyUpdate   6m24s  configuration-controller
     LatestReadyRevisionName updated to "helloworld-example-j4gv5"
```

You'll see a lot of information here, loosely approximating the shape of the underlying record. Several highlights include:

1. **`Annotations`, which are key-value metadata attached to the records. In this case you can see that Knative Serving has added annotations identifying me as the user who created and last modified the Configuration.**
2. **The `generation` is visible here under `Metadata`.**
3. **Our good friend `spec.template.spec` shows up here again.**
4. **Our other good friend `status` is also visible too.**
5. **`Events` is a log of changes that have occurred, reported to Kubernetes.**

This last one deserves a bit of commentary. The `Events` list here is a mechanism that Kubernetes provides to applications and extensions. It stores events in a nice, somewhat structured way.

THE KUBERNETES EVENTS SYSTEM

The Kubernetes Events system has two caveats you should be mindful of. One is that it's an opt-in mechanism. Software running on Kubernetes, or which extends Kubernetes, is under no obligation to emit events to Kubernetes. For certain software, this Events section is blank. Knative Serving is a good citizen in this regard and will send meaningful events for Kubernetes to record and display.

But that leads to the second problem. Even if you're dealing with well-behaved software that plays nicely with Kubernetes Events, there's no guarantee that all Events will be captured or stored or retained for long periods or protected from deletion. The client API that software calls to pass on an Event doesn't return errors, so well-behaved software may be yelling into the void. Once the event reaches the API server it's about as safe as any other Kubernetes record. It can be deleted by another controller on purpose or accidentally. And, because Events typically share resources with all other records, Kubernetes performs rolling truncation of Events. The command today that reports a bunch of Events may be ominously silent tomorrow.

The upshot is: the *presence* of an Event is meaningful, it means that the described occurrence did occur. But the *absence* of an Event should not be relied on when forming theories or diagnoses of behavior. It might be absent because the occurrence hasn't happened, or it might be absent because Kubernetes, for whatever reason, never received or stored the event, or because Kubernetes received it but has since deleted it. Absence of evidence isn't evidence of absence.

3.4 The anatomy of Revisions

I was deliberately brief in my discussion of Configurations, because their primary mission is to stamp out Revisions. Keep in mind the parent/child, template/rendered relationship between a Configuration and its Revisions. In what follows, I'll spend more time pointing out the various kinds of settings and fields that can be placed onto a Revision. But *you won't set these directly*. You'll instead apply updates to a Configuration, or to a Service, which will ultimately lead to a new Revision being stamped out.

In practical terms, this means part of the YAML you see will be from Revisions. But much of it will be from Configurations.

Those of you with a Kubernetes background will begin to ask: Why does a Revision look so much like a Kubernetes Pod?

One reason is that Knative Serving's mission is to improve the developer experience of Kubernetes. That doesn't mean that the whole of Kubernetes is exposed, and it doesn't mean the whole of Kubernetes is hidden. It's case-by-case.

When a feature is provided that's identical to the underlying system, it doesn't necessarily hurt to provide it with an identical name. For example, the `serviceAccountName` field serves the same basic purpose in Knative and Kubernetes, so why not call it the same thing?

As of this writing, Knative achieves this by internally storing part of the configuration in a Kubernetes PodSpec. But it doesn't expose the *whole* of a PodSpec, only a selected whitelist of fields. To this whitelisted set of fields, it adds two of its own, `ContainerConcurrency` and `TimeoutSeconds`, which I'll discuss in this chapter.

Note that I said: as of writing. Knative has only whitelisted a handful of PodSpec fields, and it only uses PodSpecs internally as an implementation convenience. But this is an implementation detail, not guaranteed to remain stable. PodSpecs include many knobs and dials that, it might be cogently argued, don't belong there for any reason other than implementation considerations. It's possible in future that Knative will expose other fields in a different way, or at a different level in its control hierarchy, or introduce new concepts altogether. It will be best to *ignore* the implementation detail.

Consider Revisions to be their own thing.

3.4.1 Revision basics

As you saw in chapter 2, `kn` gives us the basic information about a Revision. To recap, see the following listing.

Listing 3.12 What `kn` tells you

```
$ kn revision describe helloworld-example-8sw7z

Name:        helloworld-example-8sw7z
Namespace:   default
Age:         1d
Image:       gcr.io/knative-samples/helloworld-go (at 5ea96b)
Env:         TARGET=First
Service:

Conditions:
  OK TYPE                  AGE REASON
  ++ Ready                  1d
  ++ ContainerHealthy       1d
  ++ ResourcesAvailable     1d
   I Active                 1d NoTraffic
```

The key items here are the Name and the Namespace.

By default, the name is automatically generated when the Revision is created. It doesn't need to be. I can use kn to create a revision with a name of my own choosing, as shown in the following listing.

Listing 3.13 A Revision by any other name would smell as sweet

```
$ kn service update hello-example --revision-name this-is-a-name
# ... updates

$ kn revision list
NAME                           SERVICE         GENERATION   AGE    CONDITIONS   READY   REASON
hello-example-this-is-a-name   hello-example   6            10s    4 OK / 4     True
hello-example-jnspq-7          hello-example   5            24h    3 OK / 4     True
```

The Service name here has been baked into the Revision name by Knative Serving as an anti-collision measure.

Of course, I could achieve the same in YAML. First, I need to edit my Configuration YAML, as shown in the following listing.

Listing 3.14 Naming the next Revision in the Configuration YAML

```
apiVersion: serving.knative.dev/v1
kind: Configuration
metadata:
  name: helloworld-example
spec:
  template:
    metadata:
      name: this-too-is-a-name    <- You can see that I've added the name in a new metadata section.
    spec:
      containers:
      - image: gcr.io/knative-samples/helloworld-go
        env:
        - name: TARGET
          value: "It has a name!"
```

This section can also accept any other standard Kubernetes metadata. What does that include? Quite a lot, including metadata added automatically by Knative. To see more, we need to peek with our kubectl + jq waltz again. Starting with the metadata shown in the following listing.

Listing 3.15 Revision

```
$ kubectl get revision helloworld-example-8sw7z -o json | jq '.metadata'

{
  "annotations": {
    "serving.knative.dev/creator": "jacques@example.com"          (1)
  },
  "creationTimestamp": "2019-12-03T01:17:28Z",
  "generateName": "helloworld-example-",
  "generation": 1,
  "labels": {        2
    "serving.knative.dev/configuration": "helloworld-example",
    "serving.knative.dev/configurationGeneration": "1",
    "serving.knative.dev/service": ""
  },
  "name": "helloworld-example-8sw7z",  3
  "namespace": "default",
  "ownerReferences": [
    {
      "apiVersion": "serving.knative.dev/v1",
      "blockOwnerDeletion": true,
      "controller": true,
      "kind": "Configuration",
      "name": "helloworld-example",
      "uid": "ac192f54-156a-11ea-ae60-42010a800fc4"
    }
  ],
  "resourceVersion": "8776259",
  "selfLink":
     "/apis/serving.knative.dev/v1/namespaces/default/revisions/helloworld-
     example-8sw7z",
  "uid": "ac1a8358-156a-11ea-ae60-42010a800fc4"
}
```

Some highlights:

1. **`Annotations`, which together with ...**
2. **... `labels` captures a fair amount of useful information, which I'll outline soon.**
3. **The `name` and `namespace`, which essentially tell you what your Revision is called, and where it lives in Kubernetes-land. These are the same values that `kn` shows as `Name` and `Namespace`.**

What about `ownerReferences`? It's interesting at one level, but it's strictly speaking an implementation detail. Try not to fixate on it too closely. The same information is more easily found in the `annotations` and `labels`, as shown in table 3.1.

Table 3.1 Important Labels and Annotations on Revisions

Name	Type	Description
`serving.knative.dev/configuration`	label	Which Configuration is responsible for this Revision?
`serving.knative.dev/configurationGeneration`	label	When the Revision was created, what was the current `generation` number of the Configuration?
`serving.knative.dev/route`	label	The name of the Route that currently sends traffic to this Revision. If this value is unset, no traffic is being sent.
`serving.knative.dev/service`	label	The name of the Service which, through a Configuration, is responsible for this Revision. When this is blank, it means that there's no Service above the Configuration.

Table 3.1 Important Labels and Annotations on Revisions *(continued)*

Name	Type	Description
`serving.knative.dev/creator`	annotation	The username who is responsible for the Revision being created. `kn` and `kubectl` both submit this information as part of their requests to the Kubernetes API server. Typically, it will be an email address.
`serving.knative.dev/lastPinned`	annotation	This is used for garbage collection.
`client.knative.dev/user-image`	annotation	This is the value of the `--image` parameter used with `kn service`.

The names of labels and annotations follow a pattern: `<subject area>.knative.dev/<subject>`. This allows each of the subprojects to namespace their own annotations without trampling each other.

3.4.2 Container basics

Much of the "meat" of what you'll provide to a Revision lives in the `containers` section. This name is confusing: Knative Serving only allows a single container to be defined on a Revision (hence on a Configuration or on a Service). And the confusion continues, because `containers` is an array, so you must use YAML syntax for an array, as shown in the following listing.

Listing 3.16 The array of one

```
apiVersion: service.knative.dev/v1
kind: Revision
# ...
spec:
  containers:
  - name: first-and-only-container
    image: example.com/first-and-only-container-image
```

Suppose I now add a second container by updating a Configuration in the following listing.

Listing 3.17 Too many containers

```
apiVersion: service.knative.dev/v1
kind: Configuration
# ...
spec:
  template:
    spec:
      containers:
      - name: first-and-only-container
        image: example.com/first-and-only-container-image
      - name: sike-there-are-two-now
        image: example.com/second-image
```

This won't be acceptable to Knative in the following listing.

Listing 3.18 Computer says nein

```
$ kubectl apply -f example.yaml

Error from server (InternalError): error when applying patch:

# ... lots of angry-looking JSON here

for: "example.yaml": Internal error occurred: admission webhook
     "webhook.serving.knative.dev" denied the request: mutation failed:
     expected exactly one, got both: spec.template.spec.containers
```

The key here was "expected exactly one, got both", which refers to my attempt to double-dip.

In my example, I gave the container a `name`. Technically this isn't necessary. But names are a good idea, even simple ones. Many monitoring and debugging tools now slurp data out of the Kubernetes API; in future, others may be more Knative-centric or Knative-aware. Either way, giving the container a name makes it easier to understand, identify, and correlate with other systems.

3.4.3 Container images

The container image[20] is the software that will ultimately be run on something, somewhere.

> **NOTE** Container images—originally called Docker images, now better referred to as OCI images—are primarily about the shipment of bits that will wind up looking like a disk to your software. But they can also carry a bunch of additional settings and instructions: environment variables, startup commands, user names, and so on. Container images are interpreted and executed by container runtimes. Docker is the best-known, including its offspring `runc` and `containerd`. But there are now others: CRI-O, gVisor, Kata, Firecracker, and Project Pacific are independent implementations that can create identical runtime behavior, often with other desirable features.

In specifying a container, you must provide an `image` value. This will be a reference that allows a container runtime to fetch the image from an image registry. I showed this earlier in my examples of Configurations and Revisions.

There are however two other relevant keys to know about: `imagePullPolicy` and `imagePullSecrets`. Both of these are intended for consumption by a container runtime.

The `imagePullPolicy` setting is an instruction about "when" to pull an image to a Kubernetes node. It is one of those annoying details which surfaces at the wrong level

[20] It irks me that the motivational analogy for Docker containers was shipping containers but that "container" refers to the *running process* rather than the *bag of bits*. Instead we say, "container *image*". Reflecting the analogy back, we'd be calling container ships "containers" and calling containers "container rectangular prisms".

of abstraction, but which is nevertheless important to know about. You can give it three values: `Always`, `Never`, and `IfNotPresent`.

The `Always` policy ignores any local caching that the container runtime has and forces a re-pull anytime the Revision is launched on a Kubernetes Node. The `Never` policy prevents any attempt to pull and relies on that relevant image being pre-populated into a local cache. `IfNotPresent` says "use a cached copy if you have one, otherwise pull it".

As a rule, you don't need to set this. If you do set it for Configurations, `IfNotPresent` is a safe and efficient choice. If you're setting it for a raw Kubernetes record such as a PodSpec, then you're in a world of hurt. I'll return to this topic when we reach "From conception to production".

The `imagePullSecrets` setting is another Kubernetism poking up through the dirt. You might be used to slinging `docker pull` commands about willy-nilly to get public images. This is fine and well, but not all container images are public. And further, not all container registries are prepared to talk to unidentified strangers. A kind of authentication is required.

Kubernetes has a `Secret` record type which, among other things, can be used for image registry credentials. Like all Kubernetes records, a Secret must have a `name` that can be used to identify it and refer to it. It's this `name` to which the `imagePullSecrets` will refer.

Suppose I have an image that lives in a private repo at `registry.example.com`. I might have put credentials for `registry.example.com` into a Secret called `registry-credentials-for-example-dot-com`. Then this happens, as shown in the following listing.

Listing 3.19 Can you keep a secret?

```
apiVersion: service.knative.dev/v1
kind: Configuration
# ...
spec:
  template:
    spec:
# ...
      imagePullSecrets:
      - name: registry-credentials-for-example-dot-com
```

When it pulls the container image from `registry.example.com`, the container runtime will use the credentials provided by the Secret.

As with the `containers` section, `imagePullSecrets` is an array. In a raw Kubernetes PodSpec, this makes sense, since it allows multiple containers to be defined. Because each container can potentially come from a different registry, it's necessary to allow multiple sets of credentials.

For Knative Serving it makes less sense, because you only make a single container definition. Unlike `containers`, Knative won't impose a maximum of one entry, you may have as many as you wish. When the container image is pulled, the relevant credentials will be used.

IMAGE NAMING

Image names are, incidentally, also a mess. At the time of me writing this chapter, all of these are legal image names:

- `ubuntu`
- `ubuntu:latest`
- `ubuntu:bionic`
- `library/ubuntu`
- `docker.io/library/ubuntu`
- `docker.io/library/ubuntu:latest`
- `docker.io/library/ubuntu@bcf9d02754f659706860d04fd261207db010db96e782e2eb5d5bbd7168388b89`

More to the point, they're identical. They all refer to the same thing, because if you don't specify `docker.io`, it's assumed on your behalf.

"Very convenient", you might be thinking. Well, maybe. Suppose I instead ask for `example.com/ubuntu/1804@bcf9d02754f659706860d04fd261207db010db96e782e2eb5d5bbd7168388b89`. And further suppose that it is bit-for-bit identical with the others. Is this the same `image`? The answer is: no. Not from the point of view of the way images are named and addressed.

"But that makes sense", you say. "They're different URLs, so they should be treated as different". But what happens when you want to pull from a private repository behind your firewall? Suddenly, everything is hard, because you (1) can't reach `docker.io`, and (2) can't simply rename the images, because that would make them "different".

The problem is that no distinction is made between identity and location. Knative cannot fully resolve this mess.

Knative Serving's `webhook` component will "resolve" the container image name you give it into a full name with a digest. For example, if you told Knative that your container was `ubuntu`, it would dial out to Docker Hub to work out the full path including the digest, for example, `docker.io/library/ubuntu@bcf9d02754f659706860d04fd261207db010db96e782e2eb5d5bbd7168388b89`.

This resolution happens right before the Revision gets created, because the `webhook` component gets to act on an incoming Configuration or Service records before the rest of Serving sees them.

You can see the resolved digest in two different ways. Let's first look at it in the following listing with `kubectl` and `jq`.

Listing 3.20 What `kubectl` sees

```
$ kubectl get revision helloworld-example-8sw7z -o json | jq '.status.imageDigest'

"gcr.io/knative-samples/helloworld-
        go@sha256:5ea96ba4b872685ff4ddb5cd8d1a97ec18c18fae79ee8df0d29f446c5efe5f50"
```

The `gcr.io/knative-samples/helloworld-go` I recognise from before. The rest of it, the `@sha256:`… stuff, is what Knative resolved and recorded. It's guidance to the container runtime that it should ask for an *exact* version of the container image identified by `gcr.io/knative-samples/helloworld-go`. The `sha256:` bit tells it to verify the exact identity by using the SHA-256 hashing algorithm. If the registry doesn't have an entry that hashes to that digest value, there will be a 404 error.

`kn` does something slightly different.

Listing 3.21 Seeing the digest with `kn`

```
$ kn revision describe helloworld-example-69cbl

Name:        helloworld-example-69cbl
Namespace:   default
Age:         4h
Image:       gcr.io/knative-samples/helloworld-go (at 5ea96b)
# ... other stuff I'm ignoring right now
```

You can see `(at 5ea96b)` on the `Image` field. It's the first 6 hexadecimal digits of the full SHA-256 digest value.

What about collisions? Is it possible that two images will have the same 6 first hex digits? At one level: yes, absolutely. In terms of uniquely identifying a given image from the universe of all images, 6 hex digits isn't enough, because it can express "only" millions of permutations, instead of quintillions for the full digest. But you're not comparing the universe of *all* images, only the universe of images of that base URL. The odds of collisionmostly become noise unless you're doing something Very Interesting (please email me to tell me what it is). Six digits is easier to compare with the Mark I eyeball and doesn't cause terminal wrapping. You accept it for Git, after all.

As it happens, resolving images to the full URL with a digest is one of the best and smartest things Knative does on your behalf. I'll return to this at length in "From conception to production".

3.4.4 The command

Up to now, I've thrown container images at Knative and magic has happened: they get converted into running containers with little fuss. But that hasn't wholly been due to Knative's efforts. Let me demonstrate with `kn` action, as shown in the following listing.

Listing 3.22 The Knative doesn't know where to begin

```
$ kn service update hello-example --image ubuntu

Updating Service 'hello-example' in namespace 'default':

RevisionFailed: Revision "hello-example-flkrv-9" failed with message:
     Container failed with: .
```

Not the most helpful message. Let's look more closely at the Revision in the following listing.

Listing 3.23 What's going down?

```
$ kn revision describe hello-example-flkrv-9

Name:         hello-example-flkrv-9
Namespace:    default
Age:          2m
Image:        ubuntu (pinned to 134c7f)
Env:          TARGET=Second
Service:      hello-example

Conditions:
  OK TYPE                     AGE REASON
  !! Ready                    20s ExitCode0
  !! ContainerHealthy         20s ExitCode0
  ?? ResourcesAvailable        2m Deploying
   I Active                   11s TimedOut
```

This is slightly more helpful. I can at least see that `Ready` and `ContainerHealthy` are `!!`—that is, bad.

`!! Ready` means that the container won't come up because Kubernetes doesn't know how to run it. Or, rather, Kubernetes can't guess at what it is I want to run. Here I used `ubuntu`, which out of the box has hundreds of executables. Which one did I want to bring to life? It has no idea.

Meanwhile, `ResourcesUnavailable` is `??` (unknown) because, if the container can't come up, it doesn't hit resource limits.

What *actually* happened, though? There are two parts to the answer.

1. The container image I nominated doesn't have a defined `ENTRYPOINT`[21]. When the container runtime picks it up, it can't find out what command to run by inspecting the container image itself.
2. I didn't set a `command` field on my Configuration either. If I had, it would have been passed into the container runtime as a parameter.

Because it's set by someone closer to production, a `command` setting will override an `ENTRYPOINT`. Hence you get this basic set of combinations:

- `ENTRYPOINT` with `command` ⇒ `command`
- `ENTRYPOINT` without `command` ⇒ `ENTRYPOINT`
- `command` without `ENTRYPOINT` ⇒ `command`
- Neither `command` nor `ENTRYPOINT` ⇒ It goes kerflooie.

Your first instinct might be to try for a sneaky `bash -c echo Hello, World!` here, as a cost-cutting measure. It was certainly my first thought. But it won't do what you want either. Knative will observe that the process exited, which violates its expectations.

[21] In addition to ENTRYPOINT, images can also have a CMD. They sorta kinda do the same thing and for the purposes of our discussion it won't matter, so I'm going to keep pretending that only ENTRYPOINT is relevant.

Most of the time you shouldn't use command; you should rely on the ENTRYPOINT set by the container image you nominate. This is for a number of reasons. The most important is that it's easier. Whomever builds the container image, whether that's you or someone else, probably intends for it to be used as-is. Especially if it's going to be used by Knative.

If you *do* use command, there's one more thing to know about: args. As you might imagine, this is an array of arguments that will be passed to whatever command you defined.

But to reiterate, you probably shouldn't be using command. This is baked into kn, which doesn't expose a way to set one. Later in the book I'll be talking about how to best build images for Knative, meaning that you can safely forget command and args for good.

3.4.5 *The environment, directly*

You've already seen the easy way to add or change environment variables: use kn with --env. I used it in chapter 2 to advance the "hello, world" state-of-the-art. Many systems use, or at least support, setting environment variables as a configuration mechanism. Often this is an alternative to command line arguments or configuration files.

Whether adding new variables or updating existing variables (and thereby creating a new Revision), I use --env, as shown in the following listing.

Listing 3.24 Adding another environment variable

```
$ kn service update hello-example --env AGAINPLS="OK"

# ... Output from updating the service ...

$ kn revision describe hello-example-gddlw-4
Name:        hello-example-gddlw-4
Namespace:   default
Age:         16s
Image:       gcr.io/knative-samples/helloworld-go (pinned to 5ea96b)
Env:         AGAINPLS=OK, TARGET=Second
Service:     hello-example

# ... Conditions ...
```

If this seems too easy, you can use YAML again, by setting an env. As the name suggests, env sets environment variables. In Configuration YAML, the env parameter is set on the lonely occupant of containers. See the following listing.

Listing 3.25 More YAML again

```
apiVersion: service.knative.dev/v1
kind: Configuration
# ...
spec:
  template:
    spec:
```

```
    containers:
    - name: first-and-only-container
      image: example.com/first-and-only-container-image
      env:
      - name: NAME_OF_VARIABLE
        value: value_of_variable
      - name: NAME_OF_ANOTHER_VARIABLE
        value: yes, this is valuable too.
```

As you can see, you may set as many `name` and `value` pairs as you wish, the `env` section is an array. It's not required to use SHOUTY_SNAKE_CASE for `name`, but it's idiomatic.

One thing worth noting is that this is more verbose. In `kn` I could provide `--env KEY=VALUE` and get that set for me. With the YAML approach I need to explicitly identify the key and the value.

Remember that Knative Serving will spit out a new Revision every time you touch the `template`. That includes environment variables, which may be used to change system behaviour. Knative's dogma is that a Revision should be a faithful snapshot. If configuration can be changed out-of-band, then it will not be possible to later know how a system was configured at a particular time. Yes, it's the problem of history that I spent so much time talking about earlier in this chapter.

Knative's approach isn't without drawbacks. Firstly, updating configuration now costs you a redeployment. If your software starts fast, that might well be fine. If, for whatever reason, your software takes a long time to deploy or become ready, then tweaking configuration values may become prohibitively expensive. A school of thought (championed by Netflix, among others) is that configuration ought to be distributed independently from the code that obeys it, meaning that the deployment of configuration changes is decoupled from the deployment of software. This enables configuration changes to be made much more quickly.

On the downside, history is now sprinkled into different places again, meaning that reconstruction is back to correlation of independent timelines. If you've built powerful automation and consistent tooling, this is less of a problem, but that "if" can be a mighty big "if". Knative's decision emphasizes simplicity and safety by pushing all changes through the same mechanism.

Apart from environment variables that you set yourself, Knative Serving will inject four additional variables. These are:

- `PORT` is the HTTP port your process should listen on. You can configure this value with the `ports` setting (I'll get to it before long). If you don't, Knative will typically pick one for you. Now, it might be something predictable like `8080`, but that is*n't* guaranteed. For your own sanity, only listen on the port you find in `PORT`.
- `K_REVISION` is the name of the Revision. This can be useful for logging, metrics, and other observability tasks. Also fun for party tricks.
- `K_CONFIGURATION` is the name of the Configuration from which the Revision was created.
- `K_SERVICE` is the name of the Service owning the Configuration. If you're creating the Configuration directly, there will be no Service. In that case, the `K_SERVICE` environment variable will be unset.

3.4.6 The environment, indirectly

When I said environment variables get snapshotted, I wasn't telling the whole story (in the degenerate argot of today's youth, this is called "lying"). It's true that directly setting variables with `name` and `value` under `env` will be snapshotted into a Revision. Once this snapshot is taken, the value is frozen for all time, or until the next cosmic whoopsie in your cluster, whichever comes first (never bet against heat death).

But there are two alternative ways of injecting environment variables: `--env-from` / `envFrom` and `valueFrom`. What they have in common is that you don't provide the values of variables directly, and `envFrom` goes further and even does away with providing a name. In both cases, the values come from either a `ConfigMap` or a `Secret`.

Which means, to start with, you need ConfigMaps and Secrets from which to draw values. These are Kubernetes records and `kn` doesn't support them directly. To begin, in the following listings I need to create several and ship them off with `kubectl`.

Listing 3.26 Listing 3.26 The `ConfigMap` and `Secret`

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-configmap
data:
  foo: "bar"
---
apiVersion: v1
kind: Secret
metadata:
  name: example-secret
type: Opaque
data:
  password: <...redacted but it's definitely certainly not 'password123'...>
```

Listing 3.27 Applying the YAMLs

```
$ kubectl apply -f example-configmap.yaml example-secret.yaml
configmap/example-configmap created
secret/example-secret created
```

The first and easiest way to use these is with `--env-from`. This essentially says, "I want you to look up this record, then create variables from what you find under `data`". In the previous examples, the ConfigMap has `foo: bar` and the Secret has `password: <redacted>`. When I do this (see the following listing):

Listing 3.28 Listing 3.28 Setting variables with `kn` and `--env-from`

```
$ kn service update hello-example \
    --env-from config-map:example-configmap \
    --env-from secret:example-secret

# ... Output from update ...
```

Then inside the container, there will be two additional environment variables, `foo=bar` and `password=<redacted>`.

There's an annoyance here, which is that it's difficult to tell whether you've done this correctly. `kn` is whisper silent, as shown in the following listing, on environment variables that are injected using `--env-from`.

Listing 3.29 Too quiet for my liking

```
$ kn revision describe hello-example-gkfmx-7
Name:         hello-example-gkfmx-7
Namespace:    default
Age:          12s
Image:        gcr.io/knative-samples/helloworld-go (pinned to 5ea96b)
Env:          AGAINPLS=OK, TARGET=Second
Service:      hello-example
```

I can use `kubectl describe` for a more verbose look at the same information.

Listing 3.30 Did it work or not?

```
kubectl describe pod/hello-example-gkfmx-7-deployment-6cb9fbbd58-8mm7b
Name:           hello-example-gkfmx-7-deployment-6cb9fbbd58-8mm7b
Namespace:      default
# ... snip

Containers:
  user-container:

# ... snip

    Environment Variables from:     ❶
      example-configmap  ConfigMap  Optional: false
      example-secret     Secret     Optional: false
    Environment:           ❷
      AGAINPLS:          OK
      TARGET:            First
      PORT:              8080
      K_REVISION:        hello-example-gkfmx-7
      K_CONFIGURATION:   hello-example
      K_SERVICE:         hello-example

# ... snip
```

Note that there is distinction being drawn:

❶ **"Environment Variables from" validates that the ConfigMap and the Secret are being used.**
❷ **"Environment" shows me stuff that was injected explicitly.**

If you are so inclined, it's possible to do this in YAML with `envFrom`, as shown in the following listing.

Listing 3.31 Using `envFrom` to stamp out environment variables

```
apiVersion: serving.knative.dev/v1
kind: Configuration
metadata:
  name: values-from-example
spec:
  template:
    spec:
      containers:
      - image: example.com/an/image
        envFrom:
        - configMapRef:
            name: example-configmap
        - secretRef:
            name: example-secret
```

This mechanism is relatively convenient, because it will take everything in the ConfigMaps and Secrets you provide and stamp out environment variables. If you have software that expects a bunch of variables set, then it's easier to do it through ConfigMaps than to laboriously concatenate all of the settings into a very long `kn` command.

But it's not always convenient. Sometimes you have a big bag of values in a ConfigMap or Secret that were not originally intended to be environment variables. In this situation, you want to be able to pick and choose which values will be imported from the available selection.

This brings us to the mysterious contender, `valueFrom`. It looks much like `envFrom`, but with several subtle and important differences. For one thing, it's not exposed through `kn`, so it's all YAML from here. It also has a slightly different structure, because of the need to be able to select specific values. The selections are achieved by `configMapKeyRef` and `secretKeyRef`. Unfortunately they are a little bit on the chatty side, as shown in the following listings.

Listing 3.32 Using `valueFrom` to pull in values

```
apiVersion: serving.knative.dev/v1
kind: Configuration
metadata:
  name: values-from-example
spec:
  template:
    spec:
      containers:
      - image: example.com/an/image
        env:
        - name: FIRST_VARIABLE
          valueFrom:
            configMapKeyRef:
              name: example-configmap
              key: firstvalue
        - name: PASSWORD
          valueFrom:
            secretKeyRef:
              name: example-secret
              key: password
```

Listing 3.33 Applying the YAMLs

```
$ kubectl apply -f example.yaml
configuration.serving.knative.dev/values-from-example created
```

How can I see if this worked? As with `--env-from`, I can't see it through `kn`. Or at least, not directly (see the following listing).

Listing 3.34 `kn` does not show the resolved value

```
$ kn revision describe values-from-example-626da
Name:        values-from-example-626da
Namespace:   default
Age:         48m
Image:       example.com/an/image (at 1a2bc3)
Env:         FIRST_VARIABLE=[ref]
             PASSWORD=[ref]
Service:

Conditions:
  OK TYPE                     AGE REASON
  ++ Ready                    48m
  ++ ContainerHealthy         48m
  ++ ResourcesAvailable       48m
   I Active                   38m NoTraffic
```

You can see that there are references, butt not what they resolve to. This is one advantage of the `env.valueFrom` approach over the `envFrom` approach. In this case, `kn` will at least point to the fact that a variable exists.

Now to the lying bit: I didn't lie. What I said was completely accurate: A Revision is a snapshot of a Configuration. What's snapshotted isn't the *value* of an environment variable, but rather the exact configuration, which might so happen to include the value of environment variables. When I use `valueFrom`, I'm snapshotting the *reference* to a variable, not the value that could have been found on the other side of the reference at the moment of snapshotting.

This opens the door back to the independent updating of configuration without the updating of Configuration. That is, if it makes sense, you can change environment variables by modifying the `ConfigMap` or `Secret` that a Revision's `valueFrom` points to.

There's a caveat, which is that the change won't be effective until the Revision is relaunched. These references are resolved to actual values at container creation time. They aren't updated dynamically. If you update the ConfigMap or Secret that you referred to, that update won't be reflected in the running Revision.

To pick up the change, the Revision must be scaled to zero and then relaunched. This isn't in your direct control and so, in practice, you shouldn't rely on this mechanism for fast configuration changes. In particular, you shouldn't rely on it to rotate credentials quickly. Your choices are:

1. To edit the Configuration to force the creation of a new Revision that takes over from the previous Configuration, accepting the cost thereof.
2. Use an alternative mechanism for configuration key/values, such as Netflix Eureka, that's more proactive in managing TTLs or pushing new values to consumers, and to use out-of-band secrets management systems such as Vault or CredHub.

Which should you choose? That's partly a matter of taste. My advice is that you should prefer to edit the Configuration whenever possible for data you'd put into a ConfigMap. For a Secret, you should strongly consider using a credential manager, because keeping secret material in environment variables leads to a fascinating kind of security hell.

If you absolutely must have a secret or sensitive material in your environment, then for pity's sake use a Secret and `envFrom` *and* also do a new Revision whenever you rotate it. Yes, I know it's a schlep. But you want to make key material as inconvenient to reach as possible.

3.4.7 Configuration via files

Passing configuration via the command line is easy: use `args`. Via the environment is also easy: use `env` or `envFrom`. But these options have two problems.

Firstly, certain software requires parameter files, or you might prefer parameter files over other possibilities. For these cases, the command line and environment variables won't do.

Second, command lines and environment variables aren't a safe place for secrets to hang out. Too many tools and systems have a way of laying eyes on a command line or an environment variable. Exfiltration opportunities abound. Can you SSH into the running container? Run `ps` on the container or on the Node underlying it? Do you have monitoring system agents that extract environment variables? Which can be configured to do so? Are you checking secrets into git? The list goes on and on (and can be sung as a hymn to the tune of "We're So Boned, Time To Update My LinkedIn Profile").

One way out of this is to take your Secrets and ConfigMaps and to expose them as files in a filesystem. This first of all enables grumpy old software the luxury of not changing. Second, it behaves like a filesystem, adding another permissions hoop attackers will need to hop through. Finally, they get mounted as `tmpfs` volumes. They never touch a disk and become inaccessible once the container goes away.

Let's start with the `kn`-centric view of things by mounting a secret into our container, as shown in the following listing.

Listing 3.35 Volumes of secrets

```
$ kn service update hello-example --mount /sikkrits=secret:example-secret

Updating Service 'hello-example' in namespace 'default':
# ...
```

The key here is the `--mount` parameter, which maps from `example-secret` into `/sikkrits`. The `secret:` prefix tells `kn` what kind of record it will be asking Knative to map; the alternative option is `configmap:` for ConfigMaps.

Now we try to see in the following listing what we've done using `kn revision describe`.

Listing 3.36 The secret Secret

```
$ kn revision describe hello-example-yffhm-12

Name:        hello-example-yffhm-12
Namespace:   default
Age:         3m
Image:       gcr.io/knative-samples/helloworld-go (pinned to 5ea96b)
Env:         TARGET=Second
Service:     hello-example
# ... Conditions table
```

It gives no sign of the secret being mounted. This is true also for ConfigMaps as well. If I want to see what happened, I need to pop the bonnet ("crack the hood" for my American friends) and cop a squizz ("take a look") at the raw YAML with `kubectl` and `jq`, as shown in the following listing.

Listing 3.37 Volumes and Mounts

```
$ kubectl get -o yaml revision hello-example-yffhm-12

apiVersion: serving.knative.dev/v1
kind: Revision
metadata:
# ... lots of YAML
spec:
  containers:
    name: example-container
    # ... more YAML
    volumeMounts:
    - mountPath: /sikkrits
      name: exsec-9034cf59
      readOnly: true

  volumes:
  - name: sikkrits-9034cf59
    secret:
      secretName: example-secret
# ... still more YAML
```

You'll see here that the configuration is in two places: `volumeMounts`, under the lonely member of `containers`, and `volumes`, which hangs directly off `spec`. These different levels reflect YAML's meaningful whitespace. They also reflect another Kubernetism bubbling up into Knative. I'll take a second to explain.

Raw Kubernetes allows more than one container in a PodSpec. Containers might wish to share one or more filesystems. There needs to be (1) a way to list all the volumes that might exist and (2) a way to decide which containers can see which volumes. Dumping everything into a single pile might be convenient at first, but down the road it leads to bugs and security hassles.

Knative only allows the single container, so the positioning of `volumes` is in a sense inconsequential. It's another minor rule you need to obey if and when you're reading or writing the YAML.

Incidentally, the business of volumes shows up in `kn` in a confusing way. As well as `--mount`, you'll find there's a `--volume` option as well. The help text for both is nearly identical. Which should you use? You should stick to `--mount`. It does more or less what one might expect in terms of creating a directory, putting ConfigMaps and Secrets onto a volume and then mounting it for you.

3.4.8 Probes

Broadly speaking, software is dead or alive. When it's alive, it's ready or it's not ready. This is, at least, one of the ways Kubernetes, and therefore Knative, sees your software: as having the properties of *liveness* and *readiness*. In raw Kubernetes, you're given the ability to set `livenessProbes` and `readinessProbes` on your containers. Knative exposes this functionality, but with caveats.

First: what are probes? A probe is a simple mechanism that Kubernetes can use to determine the liveness or readiness of the software. Typical liveness probes include stuff such as, "is it listening on port 3030?" or "if I run this shell command inside the container, does it exit with code 0?". Typical readiness probes are mostly centered on making HTTP requests to known endpoints and expecting to get a `200 OK` response.

Superficially these might look the same. For example, both liveness and readiness probes might be checking for an HTTP response or sniffing a TCP port. But they're distinct. Software that's otherwise alive might not be ready for traffic. For example, during a long startup, the software is *alive* but it's not *ready*. This leads to different treatment for each kind of probe. When liveness checks fail, Kubernetes will eventually kill the container and relaunch it someplace else. When readiness checks fail, Kubernetes will prevent network traffic from reaching the container.

What does it look like? See the following listing.

Listing 3.38 Knative Probes

```
apiVersion: service.knative.dev/v1
kind: Configuration
# ...
spec:
  template:
    spec:
      containers:
      - name: first-and-only-container
        image: example.com/first-and-only-container-image
        livenessProbe:
          httpGet:
            path: /deadoralive
        readinessProbe:
          tcpSocket:
```

The first thing to note is that you can pick between `httpGet` and `tcpSocket` for your probes. The key fields for these two types are:

Type	Field	Description	Required
`httpGet and tcpSocket`	`host`	A hostname or IP address.	No
`httpGet`	`path`	An HTTP path.	Yes
`httpGet`	`scheme`	One of "http" or "https", defaults to "http"".	No
`httpGet`	`httpHeaders`	If you really need these, see the Kubernetes docs.	No

You can also set configurations that can be applied to either of the probe types. For example, you can make a `livenessProbe` wait for five seconds by using `initialDelaySeconds: 5`. Or you can require three successful probings in a row with `successThreshold: 3`.

If you came from Kubernetes, these features of probes are familiar to you. You also may be wondering: what happened to `port`? The answer is that Knative takes control of this value to satisfy its "Runtime Contract". It modifies any probes so that their `port` value is the same as the `port` value of the container itself, which will be the same as the `PORT` environment variable that's injected.

A slight quirk of this behavior is that `tcpSocket:` can hang out by itself without needing anything underneath it. I think that looks a little weird, but it's allowed in this case.

If you don't provide one or both probes, Knative Serving will create `tcpSocket` probes with `initialDelaySeconds` set to zero. By setting to zero, Knative is telling Kubernetes to immediately begin checking for liveness and readiness, in order to minimize the time it takes for an instance to begin serving traffic.

If I may be frank: probes aren't likely to be the most pressing thing to think about. Unless you have a proved need to adjust the defaults, you might as well save yourself the YAML. `kn` sees things this way, as it doesn't provide a means for setting or updating probes.

3.4.9 Setting consumption limits

Knative lets you set minimum and maximum levels for CPU share and bytes of RAM. This is another case of directly exposing the underlying Kubernetes feature, which is called `resources`.

In practice, you're mostly going to find yourself using this to set minimum levels, which is known to Kubernetes as `requests`.

You can use `kn` to adjust these, as shown in the following listing.

Listing 3.39 Requesting CPU and RAM

```
$ kn service update hello-example \
  --requests-cpu 500m \
  --requests-memory 256Mi
```

The `500m` format refers to "milliCPUs", or thousandths of a CPU. In this case, it's for 500 milliCPUs, which is half a CPU. However, what "half a CPU" means depends on where you're running Knative, you'll need to consult your vendor or provider documentation.

The memory format for `256Mi` is referring to mibibytes (not megabytes), which is the value we'd typically think of as 256 megabytes (not mibibytes). It confuses me too, but *mostly* you can substitute `Mi` for `MB` in your head and get it right. The same goes for `Gi` / `GB` and (lucky you!) `Ti` / `TB` as well.

The upper ceiling are known as `limits` and follow the same format, as shown in the following listing.

Listing 3.40 Limiting CPU and RAM

```
$ kn service update hello-example \
  --limits-cpu 800m \
  --limits-memory 512Mi
```

And, of course, there's a YAML equivalent too, as shown in the following listing.

Listing 3.41 Requesting and limiting in YAML

```
apiVersion: service.knative.dev/v1
kind: Configuration
# ...
spec:
  template:
    spec:
      containers:
      - name: first-and-only-container
        image: example.com/first-and-only-container-image
        resources:
          requests:
            cpu: 500m
            memory: 256Mi
          limits:
            cpu: 800m
            memory: 512Mi
```

You're less likely to make use of limits, because most of the time you want "burstable" behavior. That means that the container process is guaranteed to get its `requests` allocations and will "burst" to consume any spare capacity for either that the operating system is willing to allocate it. This is a useful property for helping containers to launch as quickly as possible, because it's typical for launching processes to be doing a whole bunch of preliminary bookkeeping and preparation that isn't yet about directly serving traffic.

And now for a rant

Serverless isn't, I wish it was, episode eleventy jillion.

What happens if you don't set `limit` and `range`? Nothing special, really. Left to its own devices, Kubernetes will place a completely undefined workload any old where and then leave it to fend for itself against other workloads landing on the same machine. Unless you set `limit` and `range` records, Knative will accept whatever Kubernetes dishes up as default values. Default values are configurable by the platform engineers who set up and operate the Kubernetes cluster, using `LimitRange` records. On GKE, for example, this is configured so that `requests.cpu` is `100m`—setting a floor CPU allocation of 10%.

I would prefer not to go down a rabbit hole here, because I happen to know it's a deep and elaborate rabbit hole. Somewhat-efficient packing of workloads is, after all, part of the superhero origin story for Kubernetes, so it should come as no surprise that there are many knobs and levers to be twisted or pulled by relevant persons. But this necessity has led to a complicated set of rules and ideas which no developer should be required to care about. As an exercise, look up documentation on Quality of Service levels, then try to reassure yourself that you can reliably predict what Kubernetes will do in times of trial and tribulation.

To be sure, Autoscalers solve part of this problem, but, as I'll repetitively repeat in the upcoming chapter on autoscaling, Autoscalers aren't magical. And neither is the Kubernetes scheduler. Both must work in a world where raw compute resources such as CPU and RAM aren't completely fungible. There are boundaries to what can be done, set by the capacity of the nodes that Kubernetes is managing. A container has to sit *somewhere* and its activities consume *something*. The mechanisms of `request` and `limit` are there so that you can provide hints to the Kubernetes scheduler about what that will look like. The reality that there are discrete machines leaks up through the nice abstraction of Revisions.

It turns out that Kubernetes, the closed-loop-feedback champion, has a giant gaping open loop at its heart: container placement. Once placed, the container is placed for good. The Kubernetes scheduler doesn't perform rebalancing of workloads. Rebalancing that does occur is as a side-effect of other causes, such as container crashes or autoscaling.

Is there any hope for the future? Maybe. One line of attack is VM-based runtimes such as Firecracker or Spherelets. Because these are virtual machines, they can be more easily and robustly relocated between physical nodes without appearing to be restarted, meaning that transparent rebalancing can occur without needing to modify the Kubernetes scheduler. Another more science-fiction-y line of attack will be to unbundle the resources offered by compute nodes and have directly network-connected chunks of RAM, CPUs, and so on.

3.4.10 Container concurrency

Speaking in broad terms, the purpose of the Autoscaler is to ensure that you have "enough" instances of a Revision running to serve demand. One meaning of "enough" is to ask "how many requests are being handled concurrently per instance?"

Which is where `containerConcurrency` comes in. It's your way of telling Knative how many concurrent requests your code can handle. If you set it to 1, then the Autoscaler will try to have approximately one copy serving each request. If you set it to 10, it will wait until there are 10 concurrent requests in flight before spinning up the next instance of a Revision. That is at least *approximately* what happens, because the Autoscaler has a fair few knobs and dials that affect what it does, not to mention a moderate amount of internal subtlety that I'll need to explain carefully.

You can set a concurrency *limit* with `kn`, as shown in the following listing.

Listing 3.42 Listing 3.42 Using `kn` to set container concurrency limits

```
$ kn service update hello-example --concurrency-limit 1

# ... Output from the update ...

$ kn revision describe hello-example-pyhcm-6

Name:          hello-example-pyhcm-6
Namespace:     default
Age:           2m
Image:         gcr.io/knative-samples/helloworld-go (pinned to 5ea96b)
Env:           AGAINPLS=OK, TARGET=Second
Concurrency:
  Limit:       1
Service:       hello-example
```

Note the new section under `Concurrency`, which in turn gives a `Limit`. The concurrency limit is a hard threshold for scaling. If average concurrent requests rise above this number, the Autoscaler will create more instances. There is also another setting, `--concurrency-target`. This works differently: instead of setting a maximum level of concurrency, it sets a desired level of concurrency. Right now, you can use `--concurrency-limit` and Knative will set `--concurrency-target` to the same level. In the chapter on Autoscaling, I'll break this down further.

Naturally, you can set this value in the YAML too, as shown in the following listing.

Listing 3.43 But is it YAML scale?

```
apiVersion: service.knative.dev/v1
kind: Configuration
# ...
spec:
  template:
    spec:
      containerConcurrency: 1
```

The YAML here does the same as `--concurrency-limit`, setting an upper maximum on concurrent requests being served per instance. There isn't an equivalent in the YAML for `--concurrency-target`.

If you don't use `--concurrency-limit` or set `containerConcurrency` in YAML, the value will default to 0. In turn sets up a whole bunch of other default settings that I'll ignore for now. What should you set it to? That's up to your judgment. Leaving it unset, that is, leaving it at 0, is basically OK. Autoscaling up from zero instances and back down to zero instances will occur.

If you have a closer insight into what concurrency makes sense for your software, you should take advantage of it. For example, you might have a system which is strongly thread-bound, so that a pool of four threads can handle four requests simultaneously. In this case it probably makes sense to set the value to four, or perhaps five to account for other kinds of buffering.

But remember, because we find it easier to build complex systems than to build simple ones, performance tuning will always be a mostly empirical affair. You need to apply load and observe performance, then adjust your settings.

NOTE Wouldn't this be easier if it were expressed as Requests Per Second (RPS) instead of concurrent requests? Yes, it would, and the Autoscaler can be configured to use RPS targets instead. In the Autoscaler chapter I'll explain how to do that. But here's a teaser: concurrent requests and RPS are actually closely related anyhow. If you have one, you can typically derive the other. I'll explain why when we get to autoscaling.

3.4.11 Timeout seconds

Knative Serving is based on a synchronous request-reply model, and so as a matter of necessity, it needs timeouts. The `timeoutSeconds` setting lets you define how long Knative Serving will wait until your software begins to respond to a request.

The default value is generous: five minutes. More specifically, 300. Note that this is not a duration value, it's an integer value. You don't say "300s" or "5m". You say "300".

On the upside, the default value is pretty much guaranteed to avoid flakiness due to slow responses. On the downside, if you have a bug that causes stalled responses, you're going to see the Autoscaler busily stamping out copies as unattended requests pile up.

This setting isn't directly surfaced through `kn`, and instead has to be set using `kubectl apply`. Out of the box, you can set values up to 600 — ie, ten minutes. If you attempt to set a higher value, Knative will complain. Suppose that I want the visually-distinct `9999` as my value. First, I'd tinker with the Configuration record in the following listing.

Listing 3.44 Putting up big numbers

```
apiVersion: service.knative.dev/v1
kind: Configuration
# ...
spec:
  template:
    spec:
      timeoutSeconds: 9999
```

Then, when I use kubectl to apply the change, the computer will say no, as shown in the following listing.

Listing 3.45 Nein!

```
$ kubectl -f example.yaml

error: configurations.serving.knative.dev "hello-example" could not be
     patched: Internal error occurred: admission webhook
     "webhook.serving.knative.dev" denied the request: validation failed: Saw
     the following changes without a name change (-old +new):
     spec.template.metadata.name
*{*v1.RevisionTemplateSpec}.Spec.TimeoutSeconds:
        -: "300"
        +: "9999"

expected 0 <= 9999 <= 600: spec.template.spec.timeoutSeconds
```

This error message is relatively helpful, in that it identifies what the offending change was (300 to 9999), what the inoffensive expectations were (between 0 and 600), and which component took offense (the webhook).

The downside to this sanity check is that maybe you have a good reason for letting something run for more than 5 or 10 minutes. Batch or batch-like scenarios, in particular, typically want to run for as long as it takes.

Knative Serving's timeout limit can be raised by tinkering with the installation configuration. But it's unlikely that you, as a developer, will have the authority to set such values, because they can impact everything that runs on Knative. In these cases, you will need to undertake the mature engineering step of engaging in a Jell-O™ fight over whether the value should be raised (or, come to that, lowered).

Summary

- Deployment processes have improved over the years from scheduled downtimes, to blue/green deployment, to canary deployment, finally to progressive deployment.
- Blue/green deployment works by launching the next version of software (blue) alongside the existing version (green), then switching over traffic when the new version is ready.
- Canary deployment works by first rolling out one or a few copies of the next version of the software and seeing if they're stable. If they are, further deployment, usually blue/green, occurs.
- Progressive deployment combines elements of both blue/green and canary deployments, focusing on progressively moving traffic from existing software to new software.
- Knative Serving supports all these patterns of deployment.
- In addition, Serving is able to run multiple versions of software at the same time. This is made possible using Revisions.

- Configurations are a definition of the software you want to run on Knative Serving.
- Revisions are created when Configurations are created or changed.
- Specifically, changes to the `spec.template.spec` settings in a Configuration will trigger the creation of new Revisions.
- Configuration `status` provides information about what Revision is currently running.
- Revisions have a `container`, which must include an `image`. You should also provide a `name` for debuggability.
- You can set `imagePullSecrets` if you're using private image repositories.
- You can set `imagePullPolicy`, but probably won't need to.
- Knative will try to run the image you give it, first by looking for an `ENTRYPOINT` on the image itself, then by looking for a `command` on the Revision. If these are both missing, the Revision won't work.
- While you can configure `command` and `args`, you probably shouldn't. Instead, build and use images that have `ENTRYPOINT`s.
- You can set environment variables directly using the `env` setting.
- You can also set environment variables indirectly, using `envFrom`. These values can be pulled from Kubernetes ConfigMaps and Secrets.
- Variables you set using `env` are snapshotted by the Revision. Variables you set via `envFrom` are not, meaning that they might change between Revision launches.
- You can mount configuration files easily with `kn`. Less easily by using `kubectl` with `volumeMounts` and `volumes`.
- You can define liveness and readiness probes for your software.
- If you don't define probes, Knative will assume that it can probe for an HTTP server at a known port. If doesn't get a `200 OK`, Knative assumes something is broken.
- You can set upper and lower bounds on CPU and RAM allocations for your Revision instances. You can use `kn` or `kubectl` with `requests` and `limits`.
- You can tell the Autoscaler how many requests your software can handle simultaneously by setting the container concurrency.
- If you don't set container concurrency, Knative will set reasonable defaults for its Autoscaler behavior.
- You can tell Knative how long it should wait for your software to respond to requests. The default is five minutes, you can set values up to 10 minutes.

References

- kubernetes.io/docs/tasks/debug-application-cluster/audit/
- github.com/salesforce/sloop
- Richardson, Alexis. "What is GitOps, Really?". www.weave.works/blog/what-is-gitops-really

- Anderson, Evan and Gerdesmeir, Dan. *Knative Serving API Specification*, "Container". Version 1.0.1. github.com/knative/docs/blob/master/docs/serving/spec/knative-api-specification-1.0.md#container
- kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/
- The Knative Authors. *Knative Runtime Contract*, "Meta Requests". kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/
- Shan et al., "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation". *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*. www.usenix.org/system/files/osdi18-shan.pdf
- github.com/rakyll/hey
- Accessed Wed 22 Jan 2020.

index

manning