**Chapter 8. Accessing apps: Services, routing, and service discovery**

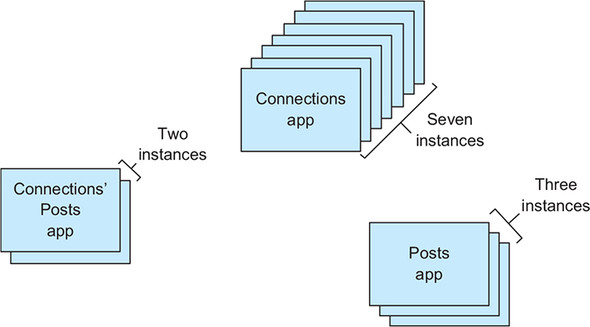
*This chapter covers*

* Single services representing multiple app instances
* Server-side load balancing
* Client-side load balancing
* Dynamic routing to service instances
* Service discovery

I’ve already been going on a bit about apps being deployed as multiple instances but needing to behave as a single logical entity. You learned that you must keep your apps stateless so that one request to an app isn’t dependent on previous requests having hit the same instance. You saw how configurations need to be carefully managed across all instances so as to ensure the same outcome regardless of which instance ends up serving a particular request, even during application lifecycle events. At this point, I want to formalize this *single logical entity*, and in doing so you’ll be able to get rid of the brittle connectivity between the three microservices in our sample application.

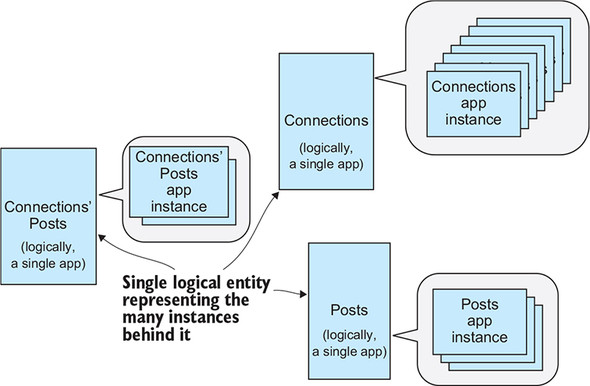
Bear with me a moment while I start with first principles. They’re important. You know your apps will be deployed as multiple instances, thereby affording you an effective way to scale deployments to meet demand, and to create a more resilient system. [Figure 8.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig01) shows multiple instances for each of the three apps that make up our running example.

**Figure 8.1. In cloud-native software, apps are deployed as multiple instances. For the software to function predictably, you want each set of app instances to operate as a single logical entity.**



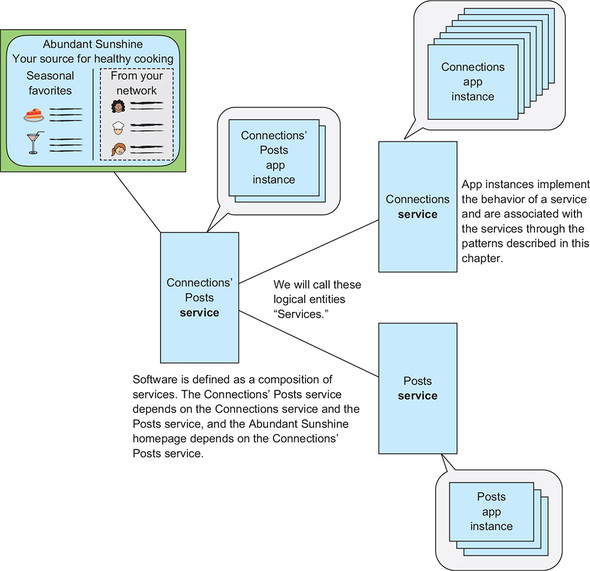
In previous chapters, you’ve been connecting these apps to each other without looking at the details of how it’s done. Turning to that now, for each set of instances, let’s introduce a box that represents the *single logical app* abstraction. As a part of that, as shown in [figure 8.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig02), I’ve more precisely labeled each piece; I’ve labeled the logical entities with the app name and more accurately labeled the instances as, well, instances.

**Figure 8.2. Each set of app instances is represented by a logical entity that defines the behavior of the app. That behavior is expected from all instances.**



You can then push the implementation details of each app into the background, as I’ve done in [figure 8.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig03) (just for a moment—don’t worry, we’ll come back to the details shortly). Another thing I’ve done in this diagram is label the logical entities as “Services.” I’ve already used this term in a different context, but the way I use it here is totally consistent with the way I’ve used it before. Previously, I referred to a “service” as a component that was used by your app code (a database or a message bus, for example). But recall that apps are most often software components used by other apps; thus they are, in fact, services.

**Figure 8.3. Your software is defined as a composition of services. Each service is implemented as a set of service instances.**



If you now focus on the (logical apps)/services, you can define your software as the set of connected services. As you know, the Connections’ Posts service depends on both the Connections service and the Posts service, as shown in [figure 8.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig03). The client of the Connections’ Posts service is, in this deployment, the Abundant Sunshine web page.

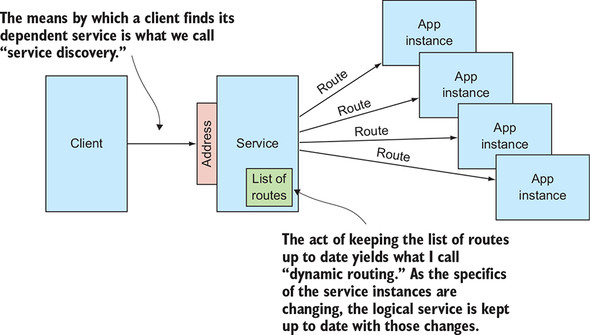
This chapter is fundamentally about these services, and two aspects of them in particular. First, it’s about how these services are tied to the app instances they represent (routing) and second, how the services are found and addressed by their clients (service discovery). Routing and service discovery may be implemented in numerous ways, and you, the software architect/developer, must understand these so that you can come up with the best design for your software.

At the moment, this probably all feels a little abstract, so I’ll start the chapter with familiar, concrete examples. Then I’ll drill into the topic of routing. For cloud-native software, this must be *dynamic* routing, the means by which incoming requests will reach the ever-changing set of app instances captured in the service abstraction. I’ll cover both traditional load balancing and client-side load balancing. In the former, client calls pass through a centralized load balancer that routes requests to instances, and in the latter, load balancing is embedded within the client. I’ll then turn to how a client of a service will find and address it. This will naturally lead us into a conversation on name servers and DNS. And then you’ll finally do it—fix the brittle service configurations in our running example.

**8.1. The service abstraction**

It’s easy to talk about services at a high level. I’ve been doing so for more than half of this book, but I’ve been a bit vague and want to fix that now. I want to start with a simple mental model, shown in [figure 8.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig04).

**Figure 8.4. The single logical app represented by a set of app instances is a *service*. The protocol that allows a client to find and access its dependent services is *service discovery*. The means for distributing incoming requests across the set of app instances is *dynamic routing*.**



There you can see four *app instances* and the service that represents the single logical app that’s implemented by those instances. On the left side of the service is a client that’s accessing the service via the service address. The means by which the client finds that service is called *service discovery*. When a request comes into the service, it’s routed to one of the instances. As you know, the service instances are going to be changing all the time—sometimes there’ll be two, sometimes ten. Their IP addresses are also changing. In order for an instance to accurately represent the implementation at any given moment, the service must be kept up to date with the list of instances it will route to. This is what I refer to as *dynamic routing*. I refer to this diagram repeatedly throughout the chapter, so you might want to bookmark this page.

|  |
| --- |
|  |

**Note**

Some design decisions around service addressing, routing, and service discovery will be made at the time of software deployment, and others will be made at software development time.

|  |
| --- |
|  |

Like virtually everything we’ve talked about so far, the patterns for handling cloud-native services are implemented both in the software you’ll be building and in the platforms your software will run on. Ultimately, your job as a software architect or developer is to ensure that your implementation enables different deployment options, or, if you build a pattern into the implementation itself, you need to clearly understand the implications of those choices. This chapter is designed to help you understand the techniques and trade-offs so you can make the right choices around service handling.

To start on the journey of understanding these design choices, let’s look at a couple of examples.

**8.1.1. Service example: Googling**

Let’s begin with something you’re completely familiar with, though you probably haven’t thought of it in the context of services and services discovery. Let’s look at what happens when you Google something. When you type [www.google.com](http://www.google.com/) into a browser, the service client in this case, you’re addressing the Google search service by name. The request for the Google homepage, however, is sent to a specific IP address, and the means of translating the name into that address is via Domain Name System (DNS) resolution.

The details of the DNS that powers the internet are complex, and the implementation is a highly distributed, hierarchical system with rules for how data is propagated through it. For simplicity, let’s use ping to obtain an IP address for the name [www.google.com](http://www.google.com/):

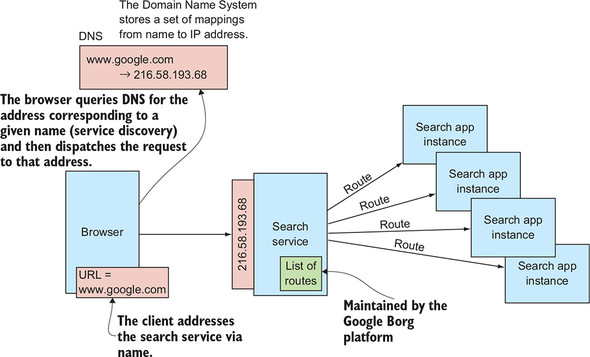
$ ping www.google.com

PING www.google.com (216.58.193.68): 56 data bytes

64 bytes from 216.58.193.68: icmp\_seq=0 ttl=53 time=19.189 ms

In reality, this name can resolve to many different IP addresses, but we need only one to make the point here. [Figure 8.5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig05), which is a more detailed version of [figure 8.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig04), now shows this search service IP address. When the client refers to the service via name, the service discovery process consults DNS for the name https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/arrow.jpg IP address mapping, and the service is then addressed via this IP address.

**Figure 8.5. The Google search service is addressed via the name**[**www.google.com**](http://www.google.com/)**, and that name is mapped to a concrete IP address via DNS. The Google platform keeps the list of routes to search service instances up to date, and load-balances requests among them.**



On the right-hand side of the service abstraction are the routes used to direct traffic to the instances that implement the service. Although I’ve never been part of the Google Site Reliability Engineering (SRE) team, it’s a reasonable assumption that at this IP address sits a load balancer that distributes incoming traffic over a set of instances of the Google homepage app. The instances of this app are constantly changing, so the list of routes that the service contains must be kept up to date, and this is done by the Google platform itself.**[**[**1**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fn1)**]** [Figure 8.5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig05) makes note of the role that Borg plays in the process of dynamic routing.

***1***

*Google has written about this platform in the 2015 paper “Large-Scale Cluster Management at Google with Borg,” which is available at*[*http://mng.bz/pgv5*](http://mng.bz/pgv5)*.*

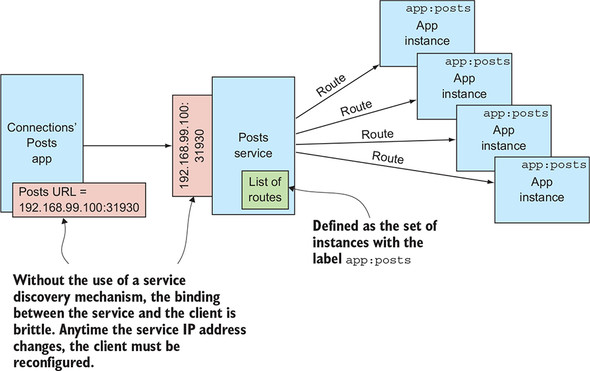
The basic services pattern comes into play for a simple operation that you likely do many times a day. You use a name, [www.google.com](http://www.google.com/), to address a service. DNS is used as a part of the service discovery process, mapping this name to an IP address. A load balancer, whose list of routes is kept up to date by the Google platform itself, routes traffic to app instances that implement the service.

Let’s look at a second example that illustrates two things. First, it shows what happens when no service discovery process is being used, and second, it offers more insight into the task of maintaining dynamic routing tables.

**8.1.2. Service example: Our blog aggregator**

Let’s look at our running example of the blog aggregator. In particular, you’ll look at the Posts service because you’re running multiple instances of the app and therefore need dynamic routing. You can also look at a client of the service, specifically the Connections’ Posts app. Just as with the Google example, you’ll look at both the right-hand side (dynamic routing) and the left (service addressing and service discovery) of the service abstraction; [figure 8.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig06) again provides a more detailed version of [figure 8.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig04).

**Figure 8.6. Until now, the configuration of your blog aggregation software lacked the use of any type of service discovery protocol and bound the Connections’ Posts service to the Posts service via IP address. This results in a brittle deployment.**



The Connections’ Posts service, which is the Posts service client, is configured with an IP address to the Posts service; looking at the env variables defined in the cookbook-deployment-connectionposts.yaml file, you can see that the Posts URL is set to something like http://192.168.99.100:31930/posts?userIds=. You might recall that anytime you re-created the Posts *service*, you were forced to reconfigure the Connection’ Posts app, providing a new IP address/port combination in this URL. You were forced to do this because you weren’t using service discovery. At the end of this chapter, you’ll fix this. In [figure 8.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig06), you can see that the service is assigned this IP address, and that value is also hardcoded into the service client.

On the right-hand side of the service are the routes. Kubernetes has a simple and elegant way of designating which app instances a service fronts: tags and selectors. Each instance of the Posts service is tagged with the key/value pair app:posts. The service is defined via a selector, designating that this service represents the list of app instances with the app:posts tag. This is depicted on the right-hand side of [figure 8.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig06). Kubernetes itself implements processes that continually keep the list of routes up to date so that anytime the service is accessed, the request will be routed to one of the current instances.

Although I hope that these two examples have helped explain the key parts of your mental model for services, I also hope that you’re left looking for more. I haven’t yet covered any concrete implementations, nor have I said much about the trade-offs I’ve alluded to. Let’s dig deeper now, starting with the right-hand side of your service abstraction.

**8.2. Dynamic routing**

We need to talk about two elements on the right-hand side of the service abstraction: the means by which the list of app instances is kept up to date, and the routing of traffic to those service instances. I refer to the latter as *load balancing*, and I want to cover two approaches: server-side load balancing and client-side load balancing.

**8.2.1. Server-side load balancing**

Surely because it’s the most commonly implemented, server-side load balancing is likely familiar to you. In a deployment that implements this pattern, there’s a component that performs the operation of accepting incoming requests and sends those requests on to one of the corresponding instances. In my customer base, I typically see both hardware-based and software-based load balancers (including F5, nginx, and offerings from entrenched networking companies like Cisco and Citrix) as well as load-balancing services from all of the major cloud providers (such as Google, Amazon, and Microsoft).

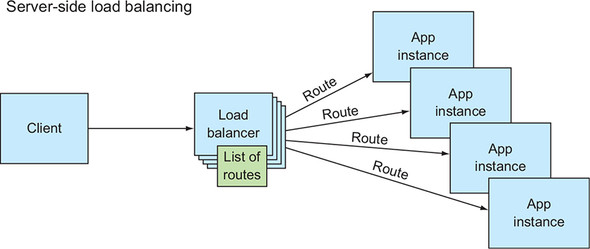
Load balancing is generally done at the TCP/UDP protocol level, or at the HTTP level. The way the load balancer chooses an instance to route to will vary; for example, round-robin or random. The details of these selection algorithms aren’t something you should concern yourself with. I will intentionally omit the details here because for your cloud-native software, you absolutely shouldn’t depend on any of these specifics. Just as your cloud-native apps shouldn’t be built so as to have the processing for one request depend on a previous request having reached exactly the same instance before, so too should you not depend on the instances being cycled through in any particular order. And because load balancers often allow you to turn on session affinity, otherwise known as *sticky sessions*, I must reiterate that you shouldn’t do so. Apps that depend on sticky sessions aren’t cloud-native.

Centralized load balancing has several advantages:

* The technology is mature. The load balancer implementations I’ve mentioned here have been evolving for several decades and as a result are robust.
* A centralized implementation is often easier to reason about than a highly distributed one.
* Configuration of a single, centralized entity is often easier than a highly distributed one.

On the other hand, a single entity can represent a single point of failure in a system, but in actuality, server-side load balancers are almost always deployed as a cluster, both for scale and resilience. [Figure 8.7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig07) shows a client request passing through a server-side load balancer (depicted as a cluster) that distributes the load across all of the service instances.

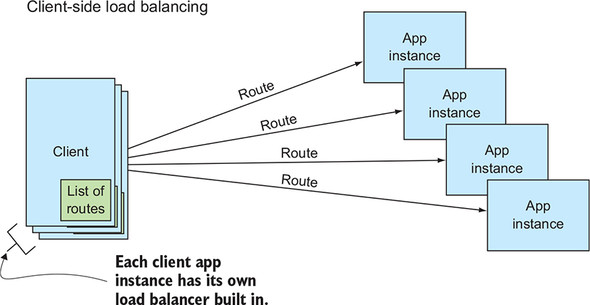
**Figure 8.7. With centralized, or server-side, load balancing, a client request is handled by a cluster of load balancers that has a list of routes to service instances. The load balancer distributes the client requests across different app instances.**



**8.2.2. Client-side load balancing**

If you take the idea of a cluster of load-balancers to an extreme, you can distribute the load-balancing components so widely that they’re included in the clients themselves, as shown in [figure 8.8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig08). Note that each instance of the client has its own load-balancing capability built in.

**Figure 8.8. With client-side load balancing, the client sends requests directly to the service instances and performs the task of distributing requests across all instances. The list of routes to service instances is maintained within the client itself.**



Client-side load balancing has gained in popularity because the number of microservices making up our software implementations has increased dramatically, and commensurate with that, so has the number of network requests flowing through the system. Pulling the load balancer into the client effectively eliminates one hop across the network, and at scale, this can make a noticeable difference in performance. Compare [figure 8.7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig07) to [figure 8.8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig08), and you can see that the client accesses service instances directly in the latter.

For client-side load balancing, you almost always use a framework that will either pull libraries into your application binary (as with Netflix Ribbon, <https://github.com/Netflix/ribbon>), or will employ other techniques such as sidecars (as with Istio, <https://istio.io/>). Either way, before you jump to the conclusion that you always want to optimize for performance and therefore will always use client-side load balancing, consider the following ramifications:

* If you’re bundling libraries into your code, updates to the client-side load-balancing framework will require that you rebuild your applications.
* Configuration of the load-balancing functionality may be more difficult.
* You’ll need to learn the details of how to use the particular client-side load-balancing capability that you choose. You’re likely already well versed in making TCP or HTTP requests from your client by using libraries that are well tested and ubiquitous. You’re now learning a new protocol.
* Most important, you’ll be limiting the deployment options for your apps. By choosing to use Ribbon, for example, you make it far more difficult to use a server-side load balancer in which corporate policies may be enforced.

Whether your software will use client-side or server-side load balancing is likely a decision that’s influenced by corporate standards, and will certainly be heavily influenced by the architectural principles agreed upon within your development teams. Regardless of whether you use client-side load balancing, or server-side, you need to understand one more pattern, even if your platform implements it for you: how the list of service instances is kept up to date for the routing function.

**8.2.3. Route freshness**

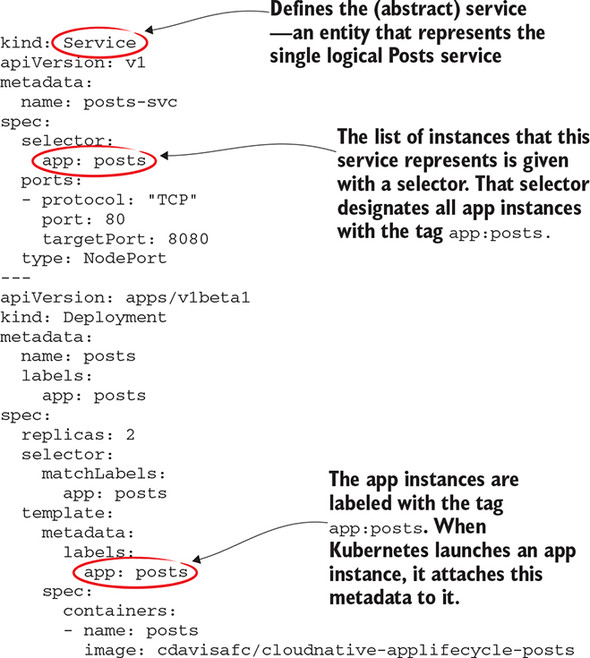
On the surface, keeping this list of routes up to date seems simple: when a new instance is created, you need to add its address to the list, and when an instance is disposed of, you need to remove it from the list. But something so simple in concept gets complex in a highly distributed, constantly changing environment. Looking back at the previous chapter’s discussion around application lifecycle, recall that you already considered some edge cases—for example, one in which an application is denied the opportunity to announce its impending departure before a sudden crash. If accuracy of the routing table depended on such edge cases never happening, your system wouldn’t work well.

At the core of a properly functioning system is a control loop (yes, another control loop!) whose job it is to constantly assess the actual state of the deployment and ensure that this reality is reflected in the routing tables. Your cloud-native platform will provide the core capabilities on which this depends, and your job is to ensure that your app presents the information needed for the platform to properly do its job. For route freshness, this means two things: (1) providing information so that the platform can build an accurate model of the actual state of the system, and (2) providing a means by which the instances implementing a service are identified.

The first portion of this has already been covered in [chapter 7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07). You’re responsible for implementing endpoints that the platform can use to assess the health of the app. Recall that you configured Kubernetes, your cloud-native platform, to implement probes of these health endpoints, and these are used to build that model of the state of the system.

The second part to keeping the routes fresh is a means by which the set of app instances that should be on the list can be identified. Again, the way this is accomplished depends on the platform, and in Kubernetes this is done with tags and selectors. Although I haven’t previously presented the details, this has been included in the deployment of our software all along. [Figure 8.9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig09) shows a portion of the deployment manifest for Posts, cookbook-deployment-posts.yaml.

**Figure 8.9. The manifest for the Posts (abstract) service and the service instances that implement it. The list of service instance routes is kept up to date via a control loop that uses the service selector to find all app instances that meet certain criteria (in this case, the label app: posts).**

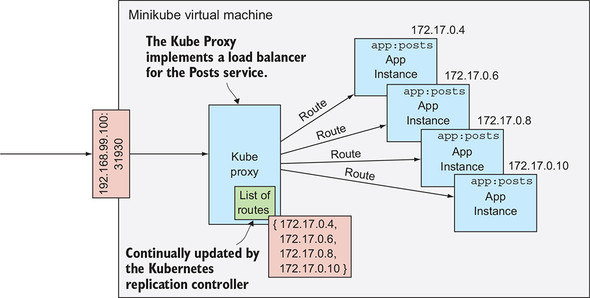


Here you can see that Kubernetes appropriately calls the service abstraction a Service, and part of the definition is a selector with the tag app:posts. Further down, in the definition of the app instance, you see that instances will be tagged with metadata, including the app:posts tag. The control loop that keeps the list of routes up to date will issue the appropriate query against the model of the actual state of the system and will update the routing tables accordingly.

Okay, you now have enough information about dynamic routing and load balancing that I can present to you our concrete implementation of the right-hand side of the Posts service. What I present here is specific to the Minikube-based deployment that we’ve been using throughout the text.

If you go all the way back to [figure 8.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig06), it’s tempting to think of the service depicted there as the load balancer, but it’s just an abstraction. In our Minikube-based deployment, the load balancer is implemented with a single instance of a Kubernetes component called the Kube Proxy (Minikube is a nonproduction Kubernetes deployment, so single points of failure are acceptable). As the name indicates, Kube Proxy is just that: a proxy that takes incoming requests and routes them to the appropriate backends. In Kubernetes, each app instance is assigned its own IP address and, as you saw just a moment ago, a control loop is continuously querying the set of instances for those with the app:posts tag. The resultant system is shown in [figure 8.10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig10). To be clear, this is an implementation of server-side load balancing.

**Figure 8.10. Concrete implementation of the Posts service running in Minikube. The Kube Proxy is a load-balancer implementation and includes the list of IP addresses to all instances of the Posts service.**

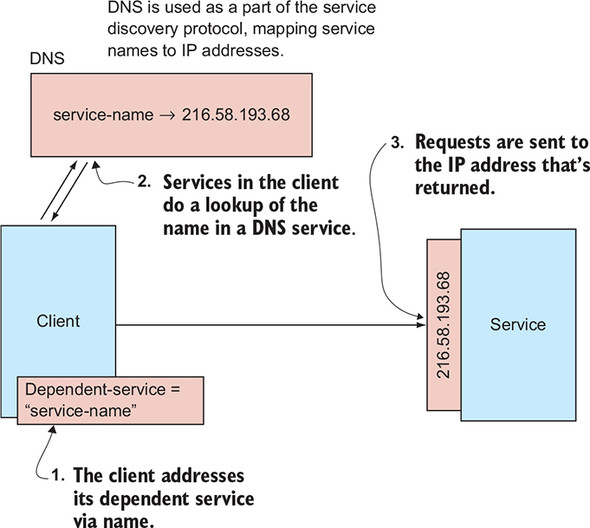


Having covered the right-hand side of the service abstraction depicted in [figure 8.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig04) (and many of the figures following), let’s now turn to the left side, which shows how the service is accessed. When you created the Posts service, Minikube dynamically assigned a port on which the Kube proxy is listening—the IP address of the Minikube virtual machine, along with this port, is what you’ve been using to access the service. But this is brittle. When the service address changes, the client must be reconfigured; this is because you haven’t been using any service discovery process in your sample. A client can refer to a service in a better way through that service discovery process we’ll dig into now.

**8.3. Service discovery**

At the core, what you need is a simple abstraction that loosely couples a client from the (changing) address of the service it depends on. It’s not complex. You just need a naming service. [Figure 8.11](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig11) depicts a simple protocol that allows the client to refer to a service by name; an address lookup on that name is performed, and connectivity is established.

**Figure 8.11. At the time of service access, the service discovery protocol allows a client to refer to a service by name, resulting in a more resilient binding between them.**



There are two parts to making this pattern work. First, there needs to be a way for entries to be placed into the name service. Second, there must be a way for an address to be fetched, given a name. Once again this sounds simple; it sounds like a simple map. But as soon as you do this in the context of a distributed system, it gets trickier. The good news is that you don’t have to solve this yourself. Naming systems abound, and your job is simply to use them effectively.

But before moving on to how you’ll use them, let me talk about the characteristics of a naming system in the context of your cloud-native apps. You already have a pretty solid appreciation for the fact that in a highly distributed software topology, having replicated pieces that operate independently is crucial. Your name service is itself a replicated, distributed system. Without getting into the details of the CAP Theorem,**[**[**2**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fn2)**]** naming services are generally configured to favor availability over consistency, a choice that suits this particular use case. To see why, let’s consider what happens when a client accesses a service.

***2***

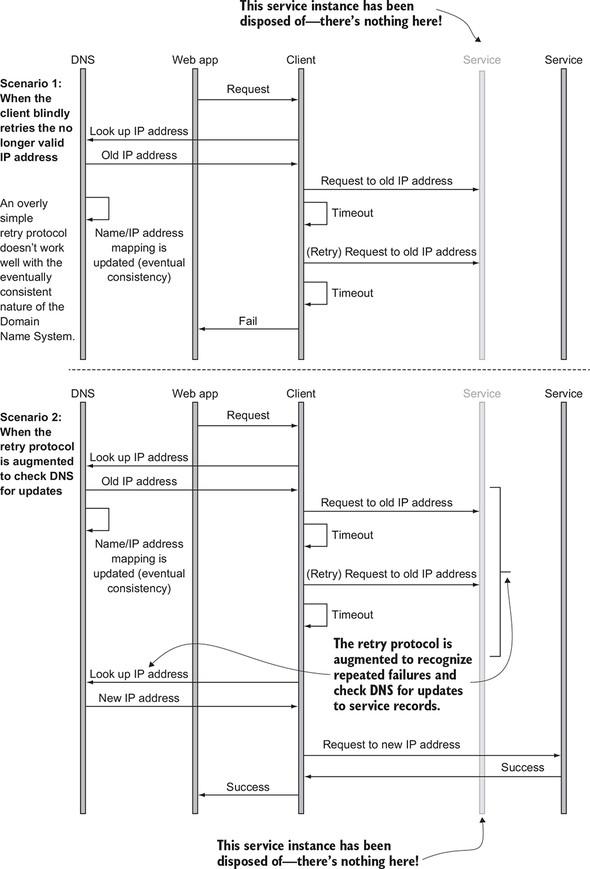
*The CAP theorem, proven by computer scientist Eric Brewer, states that of the three—Consistency, Availability, and Partition-tolerance (CAP)—only two can be realized in a distributed data store. See*[*http://mng.bz/DVlV*](http://mng.bz/DVlV)*.*

Favoring availability over consistency means that when the client asks the naming service for an address, it will always get an answer, but that answer may be stale. Incorrect answers will only be given when a service is available at a new address or is no longer available at an old address, but that latest information hasn’t yet propagated throughout the entire system; the naming-service node that’s answering the question for the client is a bit out of date. But a well-constructed naming system will minimize the windows in which this inconsistency may occur. Because a client can do nothing to reach a service without name resolution, and because most of the time the answer given by the naming service will be accurate, favoring availability over consistency is good. But, because inconsistencies may happen, albeit rarely, the client of the name resolution system must account for this possibility.

As the developer of the client app, you’re responsible for implementing necessary compensating behaviors. This isn’t the only case where you have to create an implementation that adapts to certain inconsistencies, but the good news is that some basic patterns, such as retries, which you’ll be studying in detail in the next chapter, serve to help out in many cases. So let’s add some basic retries to your service discovery protocol.

Suppose you have a service that recently experienced some lifecycle events that have disposed of an instance and have created a new one. When a client goes to access that service, it consults DNS to obtain the IP address where it can be reached, but, because it favors availability over consistency, the DNS responds with the IP address of the now disposed-of service instance. The client attempts to access that service and, of course, receives no response. It may retry the request another time or two (and I cover retries in far more detail in the next chapter), but eventually it will fail. This behavior is shown in the upper part of [figure 8.12](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig12).

**Figure 8.12. The service discovery protocol must account for the eventually consistent nature of the Domain Name System.**



Knowing that DNS is eventually consistent, if you adjust this behavior just a bit, you get far better results. After failing a couple of retries, you can ask DNS for an IP address again, and because in the meantime DNS was updated and is now consistent, you get a new IP address. Your attempt to access the service at the new address is successful, and your client can now complete its job. The lower part of [figure 8.12](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig12) shows this protocol. Depending on the framework that you’re using within your client code, you may or may not be explicitly responsible for this implementation; the framework may transparently implement this protocol for you. It’s, of course, important that you clearly understand whether you’re personally responsible for the implementation.

But what happens if there is, in fact, a service on the old IP address, but it’s a different service? This is a far more dangerous proposition. The short answer to this conundrum is that naming services should never, ever be used as a security implementation. The service implementation and/or deployment must implement access control mechanisms so that unauthorized access isn’t permitted. With such an implementation in place, client access to a stale IP address will be met with an error message indicating that access was denied, and the client can respond appropriately. Knowing that the access control issue may be a result of a stale IP address means that, as a client developer, you could check back with the naming service to see if an updated IP address is available, and if so, perform a retry.

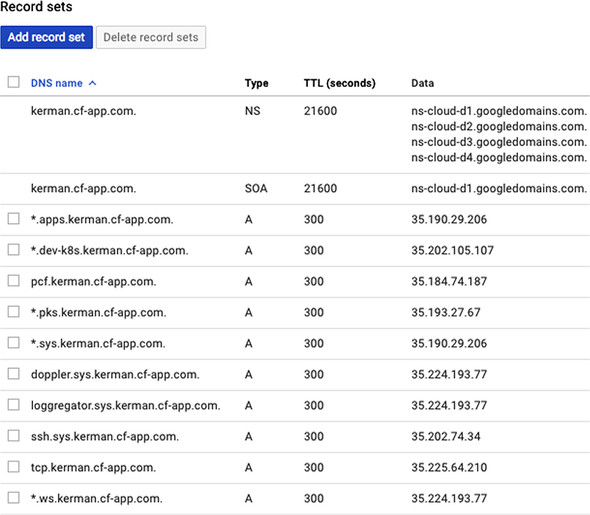
This discussion into the edge cases surrounding service discovery foreshadows a deeper discussion on compensating mechanisms that’s coming in [chapters 9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09) and [10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10). Let’s leave this discussion for now and look at concrete implementations of the core service discovery pattern.

**8.3.1. Service discovery on the web**

You already looked at this scenario earlier in the chapter: what happens when you access [www.google.com](http://www.google.com/) by using your web browser? The browser implements the client-side protocol of accessing the naming service (in this case, DNS) and then dispatches the request to that address. But how did the appropriate entries make it into DNS in the first place? You know that this is done by explicitly putting entries into the registry.

[Figure 8.13](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig13) shows the console for Cloud DNS, a DNS interface that Google provides within its Google Cloud Platform (GCP). There you can see records mapping names to IP addresses. In this case, these were created as part of an installation of Cloud Foundry onto GCP.

**Figure 8.13. DNS entries that map domain names to IP addresses**

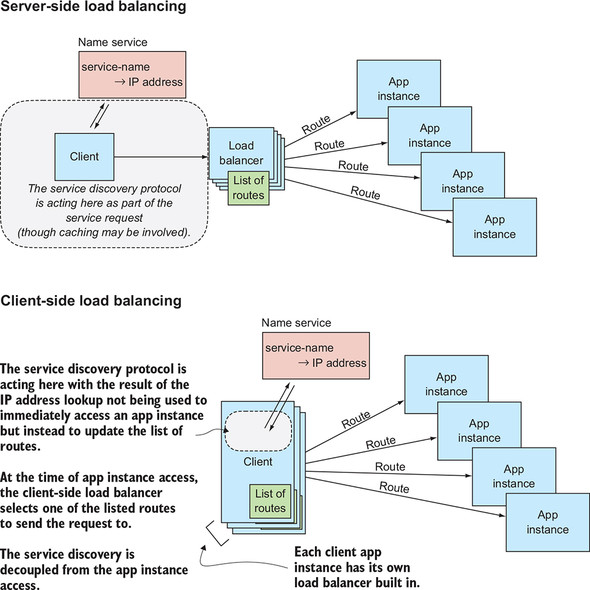


In this scenario, the DNS service of the web is providing the implementation of the naming service, entries were placed into DNS via a software deployment process, and when you access the URL pcf.kerman.cf-app.com, your web browser interrogates DNS to obtain the IP address 35.184.74.187 and fetch your Cloud Foundry Operations Manager application.

**8.3.2. Service discovery with client-side load balancing**

As you’ve seen, service discovery is all about allowing a service to be found at a particular address without tightly coupling that address into the client implementation. The protocol is used both when load balancing is implemented server-side and when it’s implemented client-side, but some differences exist. [Figure 8.14](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig14) makes this clear. The difference is, if you will, temporal.

**Figure 8.14. Service discovery differs depending on whether client-side load balancing or server-side load balancing is used.**



With server-side load balancing, the name resolution is usually done as part of the invocation of the service—after consulting DNS, a request is dispatched to the service. On the other hand, with client-side load balancing, the service discovery protocol is used to update the list of routes that are part of the client-side load balancer. To some extent, you’ve blended load balancing and service discovery, so it bears us looking at the details for a moment.

I already mentioned Netflix Ribbon and implementation of a client-side load balancer. This framework supports a programming model in which the client code can refer to its dependent service via name. Using the Spring Framework, this may be done with a class annotation such as the following:

@RibbonClient(name = "posts-service")

Later in the code, you can then use that name, via restTemplate, for example, to contact the service:

String posts

= restTemplate.getForObject("http://posts-service/posts", String.class);

The lookup of the address and request dispatch is implemented with a combination of the Spring Framework and the Ribbon client. But service discovery also depends on the registration of name/address pairs into the name service. Here, where you’re using client-side load balancing, you need a special service to facilitate this. Netflix Ribbon is almost always used in combination with another service, Netflix Eureka (<https://github.com/Netflix/eureka>), the service discovery service.

In this case, a Eureka service must be running; it’s the naming service that resolves IP addresses from a given name. The simplest way of registering a service instance with Eureka is to once again use the Spring Framework. Any application that includes the Spring Boot starter for Eureka in the class path and has the coordinates of the Eureka service configured in will automatically be registered with Eureka by the Spring Framework. It’s part of the application lifecycle that the Spring Framework is managing for you.

**8.3.3. Service discovery in Kubernetes**

The last example that I want to cover sets the stage for adding service discovery into our running sample implementation, thereby eliminating the brittle configuration that has plagued us through the early chapters of the book. The pattern, of course, is the same as for the last two examples: there’s some type of a name service, a process for placing entries into that name service, and another protocol for obtaining IP addresses from a name.

Kubernetes provides an implementation of a DNS service called—wait for it—*CoreDNS*. (Okay, saying it that way was a bit more fun when the included DNS was called *Kube-DNS*. In late 2018, Kube-DNS was effectively replaced with the next-generation CoreDNS.) Although it’s an optional component, I have yet to find a Kubernetes deployment that fails to install it by default. It’s deployed as an app (in a pod) into your running Kubernetes cluster. Other portions of the Kubernetes platform, as well as elements within your app code, will interface with it to perform the registration and lookup actions that make up the service discovery protocol. You can see that CoreDNS is running by executing the following command:

$ kubectl get pods --namespace=kube-system

NAME READY STATUS RESTARTS AGE

coredns-86c58d9df4-8mfq8 1/1 Running 0 6d19h

coredns-86c58d9df4-sfqjm 1/1 Running 0 6d19h

etcd-minikube 1/1 Running 0 6d19h

kube-addon-manager-minikube 1/1 Running 0 6d19h

kube-apiserver-minikube 1/1 Running 0 6d19h

kube-controller-manager-minikube 1/1 Running 0 6d19h

kube-proxy-jwcmg 1/1 Running 0 16h

kube-scheduler-minikube 1/1 Running 0 6d19h

storage-provisioner 1/1 Running 0 6d19h

In this output, you might notice that CoreDNS is running as a cluster of two pods. Naming services are a critical component in the system that makes up your software and must therefore be deployed in a highly resilient manner. Multiple instances contribute to that resilience.

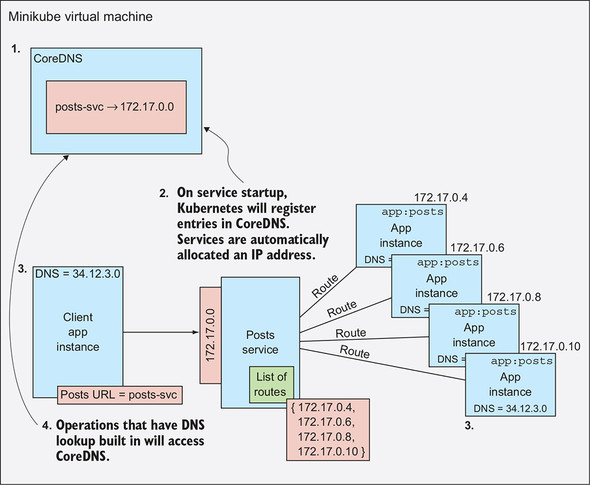
Starting with one part of the service discovery protocol, DNS registration is done automatically by Kubernetes when services are created. The names that Kubernetes registers for a specific service may be set explicitly in the deployment manifest, or defaults may be derived from standard service fields such as the service name.

For the other part of the protocol, the lookup, the approach is simple. CoreDNS acts just like any other DNS. Any processing that would normally interface with a DNS service—using restTemplate to make an HTTP request, for example—will do those lookups against CoreDNS. Kubernetes ensures that the address for CoreDNS is configured into the running pods.

[Figure 8.15](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08fig15) puts it all together:

1. The Kubernetes cluster houses a DNS service called CoreDNS.
2. On startup, a service’s name and address are added to the CoreDNS service.
3. All pods (apps) running in the Kubernetes environment have the address of the CoreDNS service configured in.
4. Any DNS-accessing operations, such as making an HTTP request to a URL that contains a name, access the CoreDNS service to resolve the address.

**Figure 8.15. Kubernetes provides an implementation of the service discovery protocol with the inclusion of a domain name service and processes that automatically create and access entries in that registry.**



You’re now ready to apply all this newfound knowledge to our blog aggregation example.

**8.3.4. Let’s see this in action: Using service discovery**

The time has come! You will now get rid of the brittle configuration between the various services that make up our sample application. When you’re done here, no longer will one service address another via IP address, and no longer will that previously brittle configuration need updating when the IP address for a service changes. Instead, you will use a DNS service to implement a service discovery protocol. Or better put, the platform you deploy the apps to, in this case, Kubernetes, will implement that protocol for you.

**Setting up**

At this point, I refer you to the setup instructions for running the samples in earlier chapters in this text; there are no new requirements for running the sample in this chapter.

You’ll be accessing files in the cloudnative-servicediscovery directory, so in your terminal window change into that directory.

**Running the apps**

Because I’ve made changes to the deployment manifests, and because there are no longer any brittle steps in configuring components of the deployment, I suggest that you delete the entire deployment of your sample app, including the databases and config server components, as well as all of the Kubernetes services. This will allow you to see clearly how much simpler your deployment is with the addition of automated service discovery (instead of you having implemented the service discovery protocol by hand). You may do this by running the script I’ve provided as follows:

$ ./ deleteDeploymentComplete.sh all

Looking at what I’ve provided here, I’ll first point out that this directory is sparse. It contains only a couple of utility scripts and the deployment manifests for our sample application. There’s no source code whatsoever, and this is telling. Remember the point that I made early on in the chapter? I said design decisions are just as likely to be a deployment time concern as a development one. Because the code that made calls to dependent services, from the Connections’ Posts app to the Posts service, for example, was already using techniques that used DNS services, replacing the brittle IP addresses with names in the app deployment manifests required no code changes. The deployment manifests point to the Docker images I created for [chapter 7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07).

Let’s start by deploying the two database services and the Spring Cloud Configuration Server; this is done with the following three commands:

kubectl apply -f mysql-deployment.yaml

kubectl apply -f redis-deployment.yaml

kubectl apply -f spring-cloud-config-server-deployment-kubernetes.yaml

Don’t forget to re-create the cookbook database:

$ mysql -h $(minikube service mysql-svc --format "{{.IP}}") \

-P $(minikube service mysql-svc --format "{{.Port}}") -u root -p

mysql> create database cookbook;

By issuing the well-worn command, kubectl get all, you can see the deployment, service, and pods that are created as a result.

Now let’s look at changes I’ve made in the deployment manifests for the Connections service:

* I’ve updated the URI to the MySQL service to reference it by name; the definition of the relevant environment variable now reads as follows:
* - name: SPRING\_APPLICATION\_JSON
* value: '{"spring":{"datasource":{"url":

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg "jdbc:mysql://mysql-svc/cookbook"}}}'

* You can also see that referring to the SCCS is also by name:
* - name: SPRING\_CLOUD\_CONFIG\_URI

value: "http://sccs-svc:8888"

You’re now ready to launch the Connections service with the following command:

kubectl apply -f cookbook-deployment-connections.yaml

You can see the same configurations for the Posts service, which you can now launch with the following command:

kubectl apply -f cookbook-deployment-posts.yaml

Finally, in the following listing, you can see that in the Connections’ Posts deployment manifest, you now refer to Redis, SCCS, and each of the Posts and Connections services by name.

**Listing 8.1. Excerpt from cookbook-deployment-connectionsposts.yaml**

- name: CONNECTIONPOSTSCONTROLLER\_POSTSURL

value: "http://posts-svc/posts?userIds="

- name: CONNECTIONPOSTSCONTROLLER\_CONNECTIONSURL

value: "http://connections-svc/connections/"

- name: CONNECTIONPOSTSCONTROLLER\_USERSURL

value: "http://connections-svc/users/"

- name: REDIS\_HOSTNAME

value: "redis-svc"

- name: REDIS\_PORT

value: "6379"

- name: SPRING\_APPLICATION\_NAME

value: "mycookbook"

- name: SPRING\_CLOUD\_CONFIG\_URI

value: "http://sccs-svc:8888"

You can launch this service with the following command:

kubectl apply -f cookbook-deployment-connectionsposts.yaml

Did you notice that you didn’t have to edit a single one of the deployment manifests? Ah, the beauty of loose coupling through service discovery.

I want to draw your attention to two additional things in the configuration of the Connections’ Posts service.

First, the names used to refer to the Posts and Connections services aren’t followed by any port number. You might have noticed that the former configurations showed that both services were listening at the same IP address (the address of your Minikube virtual machine) but on different ports. When you replaced the URI (Universal Resource Identifier) with the service name, you not only eliminated a brittle binding to an IP address, but also changed something about the way in which traffic was routed. When the IP address was used, you routed to the Posts or Connections services through a north/south avenue. Your request traveled outside the Kubernetes environment and reentered it via the IP address of your Minikube VM. In replacing the IP address and port with the service name, the routing from Connections’ Posts to Posts stayed within the Kubernetes environment, using an east/west avenue. Also, when Kubernetes creates a service object, it assigns an internal IP address to that object, and that’s the IP address associated with the name in CoreDNS.

The second thing I want to draw your attention to is related to this first point. Notice that the port number for the Redis service is now set to 6379. In the prior configurations, you accessed the Redis service through north/south avenues, just as you did for the Posts and Connections services. But in changing the Redis hostname to the DNS-registered redis-svc, east/west routing is used, and traffic will be sent directly to redis-svc. Looking at the definition of the Redis service, you can see that it’s configured to listen on port 6379, and requests coming into that port will be passed on to the targetPort on which the pod that’s running the actual Redis service is listening.

**Listing 8.2. Excerpt from redis-deployment.yam**

kind: Service

apiVersion: v1

metadata:

name: redis-svc

spec:

selector:

app: redis

ports:

- protocol: "TCP"

port: 6379

targetPort: 6379

type: NodePort

You now have the sample application fully functional, and with one important difference from before: you can delete the Posts or Connections services and re-create them, and the client of those services, the Connections’ Posts service, needn’t be reconfigured or even redeployed. Because access to the dependent services is facilitated with the service discovery protocol, your cloud-native software deployment is tolerant of such changes.

Adapting to the constant changes in cloud-native software deployments would be intractable without the help of a platform that provides things like health checks and route freshness. Service discovery is an equally essential protocol to use. Your job is to build your app code and deployments in a manner that allows these platforms to provide such services to your software.

**Summary**

* A simple abstraction can be used to more loosely couple clients from dependent services.
* Two main load-balancing approaches are available—centralized (or server-side) and client-side. Each has advantages and disadvantages.
* Configuration of load balancers must be dynamic and highly automated because in a cloud-native setting, the instances to which traffic is routed are changing far more frequently than they have in the past.
* Naming services such as DNS are central to the service discovery protocol that allows clients to find dependent services even in a topology that’s constantly changing.
* When using a domain name service, as a developer you must account for the fact that the name-to-IP-address tables are eventually consistent. You must account for entries potentially being out of date.
* Using a service discovery protocol yields far more resilient software deployments.