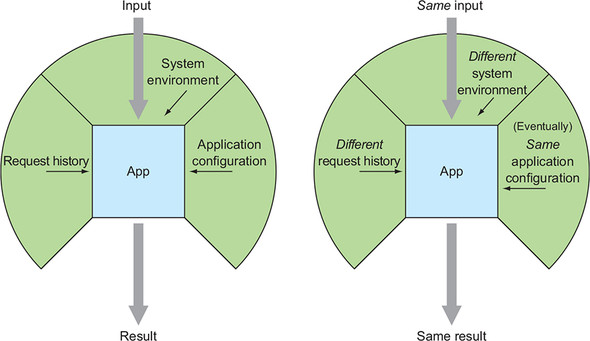
**Chapter 6. Application configuration: Not just environment variables**

*This chapter covers*

* Application configuration requirements
* Difference between system and app configuration values
* Proper use of property files
* Proper use of environment variables
* Configuration servers

At the start of the preceding chapter, I presented the illustration repeated here in [figure 6.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06fig01). This diagram shows that although it seems simple that the same inputs to any of a number of application instances would yield the same results, other factors are also influencing those results—namely, the request history, the system environment, and any application configuration. In [chapter 5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05), you studied techniques for eliminating the impact of the first of these by ensuring that any state resulting from a sequence of requests would be stored in a shared backing service, and this allowed the app instances to be stateless.

**Figure 6.1. The cloud-native app must ensure consistent outcomes despite differences in contextual influencers such as request history (different app instances have processed a different set of requests) and system environment values (such as IP addresses). Application configuration should be the same across instances but may differ at times.**



This chapter addresses the remaining two influencers: the system environment and app configuration. Neither of these is entirely new in cloud-native software; app functionality has always been influenced by the context it’s running in and the configuration applied. But the new architectures I’m talking about in this book bring new challenges. I start the chapter by presenting some of those. Next, I talk about what I call the app’s *config layer*—the mechanism for having both the system environment and application configuration make their way into the app. Then I drill into the specifics of bringing in system environment values; you’ve probably heard the phrase “store the config in env variables,” and I explain that. And finally, I focus on app configuration with an eye on the antisnowflaking that I talked about in earlier chapters. I also explain the parenthetical “eventually” you see in [figure 6.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06fig01).

**6.1. Why are we even talking about config?**

Why am I even talking about application configuration? Developers know better than to hardcode config into their software (right?). Property files exist in virtually every programming framework, and we’ve had best practices for ages. Application configuration is nothing new.

But the cloud-native context *is* new—sufficiently new that even seasoned developers will need to evolve the patterns and practices they employ to properly handle app config. Cloud-native applications are inherently more distributed than they were before. We adapt to increasing workloads by launching additional instances of an app instead of allocating more resources to a single one, for example. The cloud infrastructure itself is also constantly changing, far more than the infrastructures on which we deployed apps in the last several decades. These core differences in the platform bring new ways in which the contextual influencers present themselves to the app and therefore require new ways for apps to handle them. Let’s take a quick look at some of those differences.

**6.1.1. Dynamic scaling—increasing and decreasing the number of app instances**

The preceding chapter introduced the concept of multiple app instances and should already have given you a good sense of how this impacts your designs. Here I want to draw your attention to two nuances that affect application configuration.

First, although in the past you might have had multiple instances of an app deployed, it was likely a relatively small number. When configuration needed to be applied to those app instances, doing so through “hand” delivery of configuration was tractable (even if less than ideal). Sure, you might have used scripts and other tooling to apply config changes, but you could get away with not using industrialized automation. Now, when apps are scaled to hundreds or thousands of instances, and those instances are constantly being moved around, a semiautomated approach will no longer work. Your software designs, and the operational practices around them, must ensure that *all* app instances are running with the same configuration values, and you must be able to update those values with zero app downtime.

The second factor is even more interesting. Until now, I’ve addressed app configuration from the viewpoint of the results generated by the app; I’ve focused on the need for any of the instances of an app to produce the same output, given the same input. But the configuration of an app can also markedly impact how consumers of that app find and connect with it. To be direct, if the IP address and/or port of an app instance changes, and by some measure we’d consider this (system) configuration data, does the app bear any responsibility for making sure those changes are known to all possible consumers of the app? The short answer is yes, and I will cover this, though not fully until [chapters 8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08) and [9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09) (this is the essence of service discovery). For the time being, please simply appreciate that app configurations have a network effect.

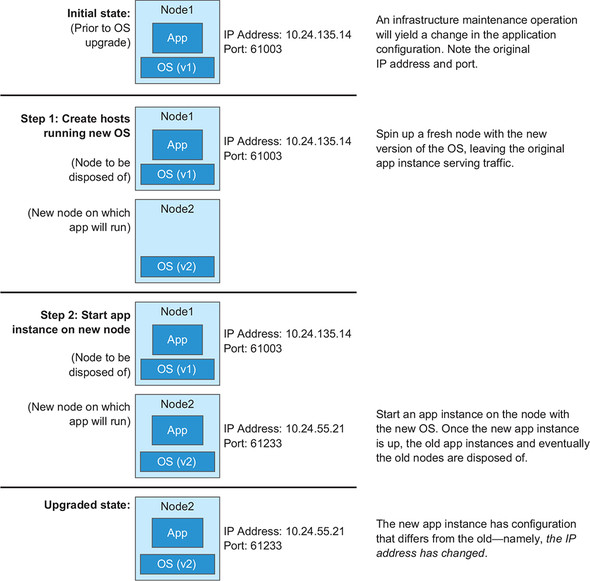
**6.1.2. Infrastructure changes causing configuration changes**

We’ve all heard this narrative: the cloud brought with it the use of lower-end, commodity servers, and because of their internal architectures and the robustness (or lack thereof) of some embedded components, a higher percentage of your infrastructure may fail at any given time. All this is true, but hardware failure is still only one cause of infrastructure change, and likely a small part of it.

A much more frequent and necessary infrastructure change comes from upgrades. For example, applications are increasingly being run on platforms that provide a set of services over and above raw compute, storage, and networking. No longer must application teams (dev and ops) supply their own operating system, for example. Instead, they can simply send their code to a platform that will establish the runtime environment, and then deploy and run the app. If the platform, which from an app perspective is part of the infrastructure, needs an upgrade to the version of the operating system (because of an OS vulnerability, for example), then that represents a change in the infrastructure.

Let’s stay with this example of upgrading the OS, which is illustrative of many types of infrastructure changes in that it requires an app to be stopped and restarted. That’s one of the advantages of cloud-native, apps: you have multiple instances that keep the system from incurring downtime as a whole. Before taking down an instance still running on an old OS, a new instance of the app will first be started on a node that already has the new version of the OS. That new instance will be running on a different node and clearly in a different context from the old, and your app and the software as a whole will need to adapt. [Figure 6.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06fig02) depicts the various stages in this process; note that the IP address and port differ for the application before and after the upgrade.

**Figure 6.2. Application configuration changes are often brought about from changes in the infrastructure, which are either anticipated (as depicted here, a rolling upgrade) or unexpected.**



An upgrade isn’t the only deliberate cause of infrastructure changes. An interesting security technique gaining widespread acceptance calls for application instances to be frequently redeployed because a constantly changing attack surface is harder to penetrate than a long-lived one.**[**[**1**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06fn1)**]**

***1***

*See “The Three Rs of Enterprise Security: Rotate, Repave, and Repair,” by Justin Smith at*[*http://mng.bz/gNKe*](http://mng.bz/gNKe)*for more information.*

**6.1.3. Updating application configuration with zero downtime**

So far, I’ve given examples of changes that are, if you will, inflicted on the application from an external source. Scaling the number of instances of an app isn’t directly applying a change to that instance. Instead, the existence of multiple instances imposes contextual variability across the set.

But sometimes an app running in production simply needs to have new configuration values applied. For example, a web application may display a copyright at the bottom of each and every page, and when the calendar turns from December to January, you want to update the date without redeploying the entire app.

Credential rotation, whereby the passwords used by one system component to gain access to another are periodically updated, serves as another example, one that’s commonly required by the security practices of an organization. This should be as simple as having the team that’s operating the application in production (which hopefully is the same team that built it!) provide new secrets, while the system as a whole continues to operate normally.

These types of contextual changes represent changes in the application configuration data and, in contrast to things like infrastructure changes, are generally under the control of the app team itself. This distinction may tempt you to handle this type of variability in a manner that might be a bit more “manual.” But as you’ll see shortly, from the app perspective, handling intentional changes and imposed changes by using similar approaches isn’t only possible, but highly desirable.

That’s the trick in all these scenarios: to create the proper abstractions, parameterizing the app deployment so that the elements that will vary across different contexts can be injected into the app in a sensible way and at the right time. Just as with any pattern, your aim is to have a tried, tested, and repeatable way to design for these requirements.

The place to start with this repeatable pattern is at the application itself—creating a technique to clearly define the precise configuration data for an application that will allow for values to be inserted as needed.

**6.2. The app’s configuration layer**

If you’re reading a book about cloud-native software, you’ve almost assuredly heard of the Twelve-Factor App ([https://12factor.net](https://12factor.net/)), a set of patterns and practices that are recommended for microservices-based applications. One of the most frequently cited factors is #3: “Store config in the environment.” Reading the admittedly brief description at <https://12factor.net/config>, you can see that it advises that configuration data for an app be stored in environment variables.

Part of the valid argument for this approach is that virtually all operating systems support the concept of environment variables, and all programming languages offer a way to access them. This not only lends to application portability but also can form the basis of consistent operating practices, regardless of the types of systems your app is running on. Following this guidance in Java, for example, you could have code such as the following to access and use the configuration data stored in those environmental variables:

public Iterable<Post> getPostsByUserId(

@RequestParam(value="userIds", required=false) String userIds,

HttpServletResponse response) {

String ip;

ip = System.getenv("INSTANCE\_IP");

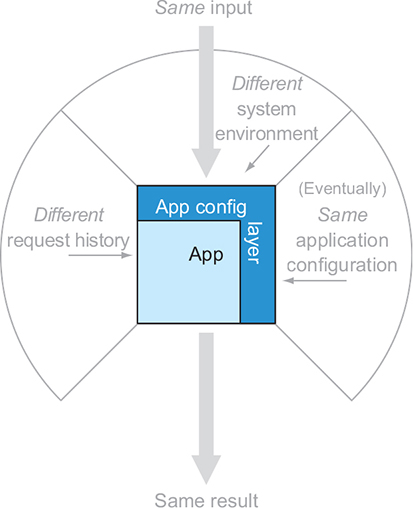
...

}

Although this approach will certainly allow your code to be used in different environments, a couple of flaws arise from this simple advice, or at least the preceding implementation of it. First, environment variables aren’t the best approach for all types of config data. You’ll see shortly that they work well for system config, but less so for application config. And second, having System.getenv calls (or similar in other languages) spread throughout your codebase makes keeping track of your app configuration difficult.

A better approach is to have a specific configuration layer in your application—one place that you can go to see the configuration options for an app. As you progress through this chapter, you’ll see that there are differences in the way system environment configuration and app configuration are handled, but this app config layer is common to both ([figure 6.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06fig03)). In Java, this comes through the use of property files, and most languages provide a similar construct. Although you’re almost certainly familiar with the use of property files, I have a way of looking at them that I want to share—one that will set you up well when we get into the differences between system and app configuration later in the chapter.

**Figure 6.3. Apps have a specific configuration layer that supports both system environment and application configuration. This layer allows the implementation to use values regardless of how their values are provided.**



The biggest advantage of the approach I describe here may be that property files are a single logical place for all configuration parameters to be defined. (You may have several property files, but they’re generally all placed in the same location in a project structure.) This allows a developer or application operator to easily review and understand the configuration parameters of an application. Remember my earlier remark about the System.getenv calls being scattered throughout a body of code? Imagine that you’re a developer taking over an existing codebase and you have to “spelunk” through dozens of source code files to even see what data is acting as input to the app. Ugh. Property files are good.

The biggest disadvantage with the way property files are used today is that they’re usually bundled into the deployable artifact (with Java, into the JAR file), and the property files often carry actual configuration values. Recall from [chapter 2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02) that one of the keys to optimizing an application’s development-to-operations lifecycle is that you have a single deployable artifact that’s used throughout the entire SDLC. Because the context will be different in the various development, test, and production environments, you might be tempted to have different property files for each, but then you’d have different builds and different deployable artifacts. Do this, and you’re back to providing ample opportunity for the proverbial “it works on my machine” to creep back in.

|  |
| --- |
|  |

**Tip**

The good news is that you have an alternative to having different property files for different deployments.

|  |
| --- |
|  |

And this is where my tweak comes in. I’ve come to think of the property file first as a specification of the configuration data for the application, and second as a gateway to the application context. The property file defines variables that can be used throughout the code, and the values are bound to those variables from the most appropriate sources (system env or app config) and at the right times. All languages provide a means for accessing the variables defined in these property files throughout the code. I’ve already been using this technique in the code samples.

Let’s look at the application.properties file for the Posts service.

**Listing 6.1. application.properties**

management.security.enabled=false

spring.jpa.hibernate.ddl-auto=update

spring.datasource.username=root

spring.datasource.password=password

ipaddress=127.0.0.1

To trace the thread from property files to code, let’s have a closer look at the ipaddress property. I haven’t drawn your attention to it yet, but I’ve been printing the IP address that an app instance is serving traffic on within the log output. When you run this software locally, the value of 127.0.0.1 will be printed. But you might have noticed that when you deployed the services to Kubernetes, the log files were reporting that same IP, incorrectly. This is because I’ve been doing the very thing I just said isn’t so good: binding values to those variables directly in the property file. I’ll start fixing that shortly; I’ll talk about how our properties get their values in the next two sections. Right now, I want to focus on the property file as an abstraction for the app implementation. In the PostsController.java file, you find the following code.

**Listing 6.2. PostsController.java**

public class PostsController {

private static final Logger logger

= LoggerFactory.getLogger(PostsController.class);

private PostRepository postRepository;

**@Value("${ipaddress}")**

private String ip;

@Autowired

public PostsController(PostRepository postRepository) {

this.postRepository = postRepository;

}

...

}

The local variable, ip, is drawing its value from the ipaddress environment variable. Spring provides the @Value annotation to make this simple. Putting the pieces together: the application source defines data members that may have their values injected, and it draws those values from the defined properties. The property file lists all of the configuration parameters, not only to facilitate the entry of those values into the application, but also to provide the developer or operator a specification of the configuration data for the app.

But, again, it’s not at all good that you have the 127.0.0.1 value hardcoded in the property file. Some languages, like Java, provide an answer to this by allowing you to override property values when you launch an app. For example, you could start your Posts service with the following command, providing a new value for ipaddress:

java -Dipaddress=192.168.3.42 \

-jar cloudnative-posts/target/cloudnative-posts-0.0.1-SNAPSHOT.jar

But I want to draw you back to factor #3, “Store config in the environment.” This advice points to something important. It’s true that moving value bindings out of property files and into the command line eliminates the need for different builds for different environments, but the different start commands now offer a new way for config errors to creep into your operational practices. If, instead, you store the IP address in an env variable, then you can use the following command to launch the app in any of the environments. The app will simply absorb the context that it’s running in:

java -jar cloudnative-posts/target/cloudnative-posts-0.0.1-SNAPSHOT.jar

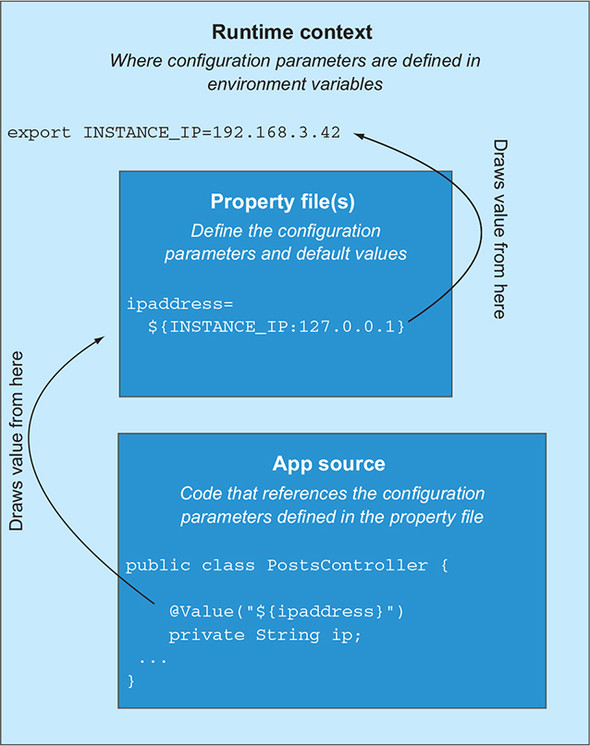
Some language frameworks support mapping env variables to application properties. For example, with the Spring Framework, setting the env variable IPADDRESS will cause that value to be injected into the ipaddress property. We’re getting there, but I’m going to add one more abstraction to give you even more flexibility and code clarity. I want to update the ipaddress line in the property file to this:

ipaddress=${INSTANCE\_IP:127.0.0.1}

This line now states that the value for ipaddress will be specifically drawn from the env variable INSTANCE\_IP, and if that env variable isn’t defined, ipaddress will be set to the *default* value of 127.0.0.1. You see, it’s okay to have values in the property file as long as they represent sensible defaults, and you’re intentional about how the values will be overridden when the default isn’t correct.

Let’s put all of this together in a diagram—[figure 6.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06fig04). The application source references properties that are defined in the property file. The property file acts as a specification of the configuration parameters for the app and will clearly indicate which values may come from env variables.

**Figure 6.4. The application source references properties that are defined in the property file. The property file acts as a specification of the configuration parameters for the app, and can indicate that values should come from env variables (INSTANCE\_IP).**



Property files written in this way are compiled into a single deployable artifact that can now be instantiated into any environment; the artifact is expressly designed to absorb the context of that environment. This is goodness—and a key pattern for properly configuring your applications in a cloud context!

But I haven’t given you the whole story. Everything I’ve said is 100% true, but through omission, I’ve implied that the property files always source values from environment variables (though I did hint that this might not always be the case). That is but one place from which configuration data can come; there are alternatives. And the differences are generally drawn along the lines of whether you have system config or application config data. Let’s look at each of those now.

**6.3. Injecting system/environment values**

What I mean by *system values* are those that the application developer or operator aren’t in direct control of. Whoa—what? In the world I’ve spent most of my career in, this is an absolutely crazy concept. Computers and computer programs are deterministic, and if you supply all the inputs in the same way, you can totally control the output. A suggestion to cede some of that control would make many software professionals uncomfortable. But moving to the cloud necessitates exactly that. It takes us all the way back to the concept I talked about in [chapter 2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02): *change is the rule, not the exception*. Giving up some control also allows systems to operate more independently, ultimately making the delivery of software more agile and productive.

System variables reflect the part of the application context that’s generally supplied by the infrastructure. I suggest that it represents the state of the infrastructure. As we’ve already discussed, our job as developers is to ensure that app outcomes are consistent, despite running in a context that’s not known a priori and is constantly changing.

To explore this a bit further, let’s look at a concrete example: including the IP address in logging output. In the past, you might not have thought of the IP address as something that was constantly changing, but in the cloud it is. App instances are constantly being created and each time will receive a new IP address. Including the IP address in log output is particularly interesting when running in a cloud setting because it allows you to track which specific instance of an app served a particular request.

**6.3.1. Let’s see this in action: Using ENV variables for configuration**

To get started, I invite you back to the cloudnative-abundantsunshine repository, specifically to the cloudnative-appconfig directory and module. Looking at the implementation for the Connections’ Posts service, you can see that the property file already reflects the ipaddress definition shown in the previous section. It reads as follows:

ipaddress=${INSTANCE\_IP:127.0.0.1}

The app needs the ipaddress value, and the infrastructure has such a value. How, then, do you connect the two? This is where factor #3 nailed it: environment variables are the constant in virtually all environments; the infrastructure and platforms know how to supply them, and application frameworks know how to consume them. Using this ubiquity is important. It allows you to establish best practices, regardless of whether your app is running on Linux (any of the flavors!), macOS, or Windows.

To see all of this in action, I want to deploy the latest version of the app into Kubernetes.

**Setting up**

Just as with the examples of the previous chapters, in order to run the samples, you must have these standard tools installed:

* Maven
* Git
* Java 1.8
* Docker
* Some type of a MySQL client, such as the mysql CLI
* Some type of a Redis client, such as redis-cli
* Minikube

**Building the microservices (optional)**

I will have you deploy the apps into Kubernetes. To do so, Docker images are required, so I’ve prebuilt those images and made them available in Docker Hub. Therefore, building the microservices from source isn’t necessary. That said, studying the code is illustrative, so I invite you to follow some of these steps, even if you don’t build the code yourself.

From the cloudnative-abundantsunshine directory, check out the following tag and then change into the cloudnative-appconfig directory:

git checkout appconfig/0.0.1

cd cloudnative-appconfig

Then, to build the code (optional), type the following command:

mvn clean install

Running this command builds each of the three apps, producing a JAR file in the target directory of each module. If you want to deploy these JAR files into Kubernetes, you must also run the docker build and docker push commands as described in the “[Using Kubernetes requires you to build Docker images](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05sb02)” sidebar in [chapter 5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05). If you do this, you must also update the Kubernetes deployment YAML files to point to your images instead of mine. I won’t repeat those steps here. Instead, the deployment manifests I provide point to images stored in my Docker Hub repository.

**Running the apps**

If you don’t already have it running, start Minikube as described in [section 5.2.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05lev2sec2) of [chapter 5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05). To start with a clean slate, delete any deployments and services that might be left over from your previous work. I’ve provided you a script to do that: deleteDeploymentComplete.sh. This simple bash script allows you to keep the MySQL and Redis services running. Calling the script with no options deletes only the three microservice deployments; calling the script with all as an argument deletes MySQL and Redis as well. Verify that your environment is clean with the following command:

$kubectl get all

NAME READY STATUS RESTARTS AGE

pod/mysql-75d7b44cd6-jzgsk 1/1 Completed 0 2d3h

pod/redis-6bb75866cd-tzfms 1/1 Completed 0 2d3h

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 2d5h

service/mysql-svc NodePort 10.107.78.72 <none> 3306:30917/TCP 2d3h

service/redis-svc NodePort 10.108.83.115 <none> 6379:31537/TCP 2d3h

NAME READY UP-TO-DATE AVAILABLE AGE

deployment.apps/mysql 1/1 1 1 2d3h

deployment.apps/redis 1/1 1 1 2d3h

NAME DESIRED CURRENT READY AGE

replicaset.apps/mysql-75d7b44cd6 1 1 1 2d3h

replicaset.apps/redis-6bb75866cd 1 1 1 2d3hNAME

Note that you’ve left MySQL and Redis running.

If you’ve cleared out Redis and MySQL, deploy each with the following commands:

kubectl create -f mysql-deployment.yaml

kubectl create -f redis-deployment.yaml

Once completed, the deployment will be as depicted in [figure 6.5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06fig05). You’ll have one each of the Connections and Posts services, and two instances of the Connections’ Posts service. To achieve this topology, for now, you may still have to edit deployment manifests. These steps, summarized next, are detailed in [chapter 5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05):

1. Configure the Connections service to point to the MySQL database. Look up the URL with this command and insert into the appropriate position in the deployment manifest:
2. minikube service mysql-svc \

--format "jdbc:mysql://{{.IP}}:{{.Port}}/cookbook"

1. Deploy the Connections service with the following:

kubectl create -f cookbook-deployment-connections.yaml

1. Configure the Posts service to point to the MySQL database. Use the same URL that you obtained with the command in step 1 and insert it into the appropriate position in the deployment manifest.
2. Deploy the Posts service:

kubectl create -f cookbook-deployment-posts.yaml

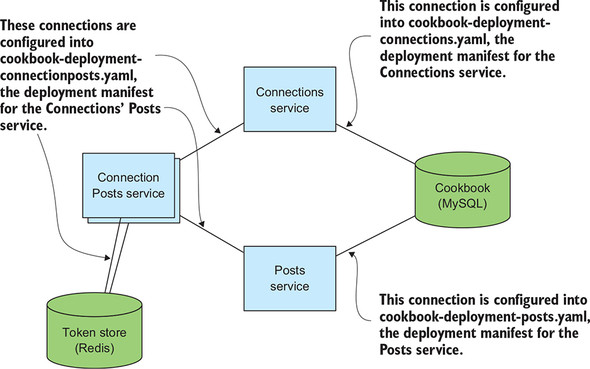
1. Configure the Connections’ Posts service to point to the Posts, Connections, and Users services, as well as the Redis service. These values can be found with the following commands, respectively:

|  |  |
| --- | --- |
| Posts URL | minikube service posts-svc --format "http://{{.IP}}:{{.Port}}/posts?userIds=" --url |
| Connections URL | minikube service connections-svc --format "http://{{.IP}}:{{.Port}}/connections/" --url |
| Users URL | minikube service connections-svc --format "http://{{.IP}}:{{.Port}}/users/" --url |
| Redis IP | minikube service redis-svc --format "{{.IP}}" |
| Redis port | minikube service redis-svc --format "{{.Port}}" |

1. Deploy the Connections’ Posts service:

kubectl create -f cookbook-deployment-connectionsposts.yaml

**Figure 6.5. This software deployment topology currently requires a great deal of hand edits of connections between the services. These manual configurations will be progressively eliminated as you proceed through more cloud-native patterns.**



Your deployment is now complete, but I’d like to draw your attention to the lines in the deployment manifests that get to the topic at hand: configuration of system values. The following shows a portion of the deployment manifest for the Connections’ Posts service:

**Listing 6.3. cookbook-deployment-connectionsposts.yaml**

apiVersion: apps/v1

kind: Deployment

metadata:

name: connectionsposts

labels:

app: connectionsposts

spec:

replicas: 2

selector:

matchLabels:

app: connectionsposts

template:

metadata:

labels:

app: connectionsposts

spec:

containers:

- name: connectionsposts

image: cdavisafc/cloudnative-appconfig-connectionposts:0.0.1

**env:**

**- name: INSTANCE\_IP**

**valueFrom:**

**fieldRef:**

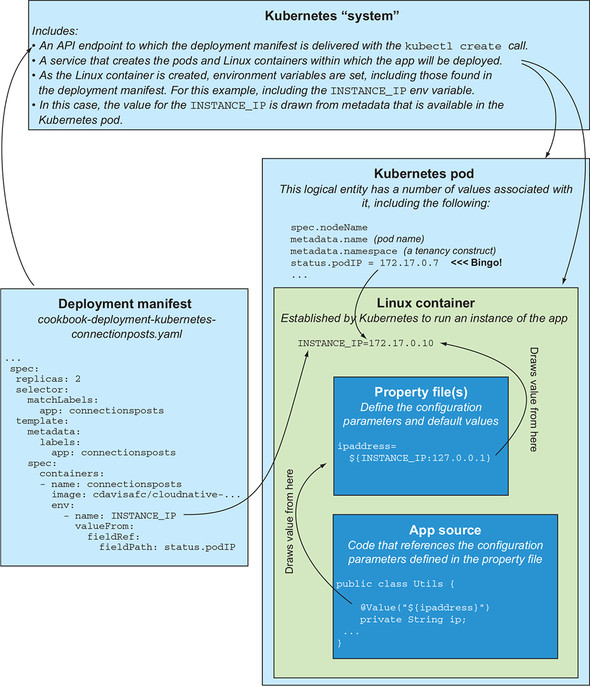
**fieldPath: status.podIP**

As part of the specification for the service, you see a section labeled env. Yep, that’s exactly where you define environment variables for the context in which your app will run. Kubernetes supports several ways of supplying a value. In the case of INSTANCE\_IP, it draws the value from attributes supplied by the Kubernetes platform itself. Only Kubernetes knows the IP address of the pod (the entity in which the app will run), and that value can be accessed in the deployment manifest via the attribute status.podIP. When Kubernetes establishes the runtime context, it seeds it with the INSTANCE\_IP value that, in turn, is drawn into the application through the property file.

[Figure 6.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06fig06) summarizes all of this. Notice that the box labeled “Linux Container” is exactly that of [figure 6.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06fig04). What you’re seeing here is the app config layer running in a Kubernetes context. [Figure 6.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06fig06) shows how that context interfaces with the app config layer. The diagram shows a lot of sophistication:

* Kubernetes has an API that allows for a deployment manifest to be supplied.
* That deployment manifest allows for env variables to be defined.
* When the deployment is created, Kubernetes creates a pod and containers within the pod, and seeds each of those with a host of values.

**Figure 6.6. Responsible for the deployment and management of the app instances, Kubernetes establishes the environment variables defined in the deployment manifest, drawing values from the infrastructure entities it has established for the app.**



But despite this relative complexity, what’s in the Linux container remains simple; the app will draw values from env variables. The app is shielded from all of the Kubernetes complexity by using env variables as the abstraction. This is why factor #3 of the Twelve-Factor App is so spot on; it drives simplicity and elegance.

If you look at the code in [listing 6.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06ex03), you’ll see that a Utils Java class is used to generate a tag that concatenates the IP address and port that the app is running on. This tag is then included in the log output. When an instance of this class is created, the Linux container has already been initialized, including having the INSTANCE\_IP environment variable set. This has the result of initializing the ipaddress property that’s then drawn into the Utils class with the @Value annotation. Although it’s not related to the topic of environment variables, for completeness I’ll also point out that I’ve made the class ApplicationContextAware and have implemented a listener that waits for the embedded servlet container to be initialized. At that time, the port that the app is running on has been set and can be looked up through EmbeddedServletContainer.

**Listing 6.4. Utils.java**

public class Utils implements ApplicationContextAware,

ApplicationListener<ServletWebServerInitializedEvent> {

private ApplicationContext applicationContext;

private int port;

@Value("${ipaddress}")

private String ip;

public String ipTag() {

return "[" + ip + ":" + port +"] ";

}

@Override

public void setApplicationContext(

ApplicationContext applicationContext)

throws BeansException {

this.applicationContext = applicationContext;

}

@Override

public void onApplicationEvent(ServletWebServerInitializedEvent

embeddedServletContainerInitializedEvent) {

this.port = embeddedServletContainerInitializedEvent

.getApplicationContext().getWebServer().getPort();

}

}

Okay, time to see all of this in action.

If you’ve re-created your MySQL service, be sure to create the cookbook database by connecting to the server with a MySQL client and issuing the create database command. For example:

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

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

mysql> create database cookbook;

Query OK, 1 row affected (0.00 sec)

In addition to what I detail here, you’re welcome to stream the logs from both the Connections and Posts services, but what I really want to home in on is the log output for the Connections’ Posts service. Let’s invoke this service a few times. Recall that the first step is to authenticate, and then you can access the posts for your connections with a simple curl command:

# authenticate

curl -X POST -i -c cookie \

$(minikube service --url connectionsposts-svc)/login?username=cdavisafc

# get the posts – repeat this command 4 or 5 times

curl -i -b cookie \

$(minikube service --url connectionsposts-svc)/connectionsposts

Kubernetes doesn’t support aggregated log streaming, which is why I’ve had you invoke the service several times before looking at the logs. You can now, however, look at the logs from both instances with a single command:

$ kubectl logs -lapp=connectionsposts

...

... : Tomcat started on port(s): 8080 (http) with context path ''

... : Started CloudnativeApplication in 16.502 seconds

... : Initializing Spring FrameworkServlet 'dispatcherServlet'

... : FrameworkServlet 'dispatcherServlet': initialization started

... : FrameworkServlet 'dispatcherServlet': initialization completed

... : Starting without optional epoll library

... : Starting without optional kqueue library

... : [172.17.0.7:8080] getting posts for user network cdavisafc

... : [172.17.0.7:8080] connections = 2,3

... : [172.17.0.7:8080] getting posts for user network cdavisafc

... : [172.17.0.7:8080] connections = 2,3

...

... : Started CloudnativeApplication in 15.501 seconds

... : Initializing Spring FrameworkServlet 'dispatcherServlet'

... : FrameworkServlet 'dispatcherServlet': initialization started

... : FrameworkServlet 'dispatcherServlet': initialization completed

... : Starting without optional epoll library

... : Starting without optional kqueue library

... : [172.17.0.4:8080] getting posts for user network cdavisafc

... : [172.17.0.4:8080] connections = 2,3

... : [172.17.0.4:8080] getting posts for user network cdavisafc

... : [172.17.0.4:8080] connections = 2,3

Looking through this example, you can see that you have output from both instances of the Connections’ Posts service. The logs aren’t interleaved. This command simply accesses the logs from one instance and dumps them out, and then does the same for the next instance. You can, however, see where the output has come from two different instances, because the IP addresses for each have been reported; one instance has IP address 172.17.0.7, and the other has 172.17.0.4. Here you can see that two requests went to the instance serving traffic on 172.17.0.4, and two requests went to the instance at 172.17.0.7. Kubernetes has instantiated values into the environment variables present in the context of each instance, and the app has drawn the value in through the property files that were crafted to access environmental variables. It’s a good design.

Let’s take a look at the environment variables in the running containers. You may do this by executing the following command, replacing the pod name with your own:

$ kubectl exec connectionsposts-6c69d66bb6-f9bjn -- env

PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin...

CONNECTIONPOSTSCONTROLLER\_POSTSURL=http://192.168.99.100:32119/posts?userIds=

CONNECTIONPOSTSCONTROLLER\_CONNECTIONSURL=http://192.168.99.100:30955/

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg connections/

CONNECTIONPOSTSCONTROLLER\_USERSURL=http://192.168.99.100:30955/users/

REDIS\_HOSTNAME=192.168.99.100

REDIS\_PORT=31537

INSTANCE\_IP=172.17.0.7

KUBERNETES\_PORT\_443\_TCP\_PROTO=tcp

...

Among the long list of values printed out, you’ll see INSTANCE\_IP. At your direction (recall the lines in the deployment YAML for the Connections’ Posts service), Kubernetes set that value to the IP address of the pod. This is an application configuration value that’s set by the system in which your app is running.

Hopefully, this exercise has helped make things clear. But even so, I want to offer one other insightful tool. I haven’t told you yet that I have something running as a part of each of the services. Through the magic of Spring Boot, the application automatically implements an endpoint where you can view the environment in which your apps are running. Run the following command to see the output:

curl $(minikube service --url connectionsposts-svc)/actuator/env

The JSON output is lengthy, but you’ll see that it includes some of the following:

...

"systemEnvironment": {

"PATH": "/usr/local/sbin:/usr/local/bin:...",

"INSTANCE\_IP": "172.17.0.7",

"PWD": "/",

"JAVA\_HOME": "/usr/lib/jvm/java-1.8-openjdk",

...

},

"applicationConfig: [classpath:/application.properties]": {

...

"ipaddress": "172.17.0.7"

},

...

Among the available data is the IP address you’ve been manipulating. Under the systemEnvironment key, you see a map that includes the INSTANCE\_IP key with the value 172.17.0.7. You can also see under the applicationConfiguration key a map that includes the same address associated with the key ipaddress. The connection was established just as you intended.

Looking through other values in this output, there appear to be many environment variables, and indeed there are. But you can see that many other contextual values are also reported. You see, for example, the process ID (PID), the operating system version (os.version), and many other values that aren’t stored in env variables. This drives home the point that environment variables aren’t the only contextual values for your app. The /actuator/env endpoint reports on the broader superset. I’d now like to move on to another part of that application context and a different way of bringing in values.

**6.4. Injecting application configuration**

For the type of configuration data that you just looked at, with values that are part of the runtime environment and managed by the runtime platform, using environment variables is natural and effective. But when I first started working with cloud-native systems, I struggled with rationalizing factor #3 (store config in the environment) with some of the other things we need for managing application configuration. Ultimately, the answer is that there are better ways to manage application configuration data. That’s what I want to take you through now.

When it comes to getting an application running in production, I’d argue that the configuration data is just as important as the implementation itself, for without the proper configuration, the software just won’t work. This calls for applying the same level of rigor to managing app config data as you apply to managing code. In particular:

* The data must be persisted and access controlled. This is so similar to the way you handle source code that one of the most common tools used for this is a source code-control (SCC) system, like Git.
* You’ll never set config data by hand. If you need to change a configuration, you’ll make a change to the source code-controlled (Git) repository and invoke some action to apply that config to the running system.
* Configurations must be versioned so that you can consistently re-create deployments based solely on a specific version of the app, tied together with a specific version of the configuration. It’s also essential to know what properties were being used at what times so that operational behaviors (good and bad) can be correlated to the configuration that was applied.
* Some configuration data is sensitive, such as credentials used for intercomponent communication in a distributed system. This brings added requirements that are addressed by special-purpose configuration repositories such as Vault from HashiCorp.

The first part of the answer for managing application configuration data is that it’s managed in what I refer to as a *configuration data store.* (I’m avoiding the use of *configuration management database* because that term comes with a bit of baggage. It implies particular patterns that aren’t the ones that apply in the cloud-native world.) The configuration store will simply house key/value pairs, maintain a version history, and have various access control mechanisms applied.

And the second part of the answer for managing application configuration is that a service facilitates the delivery of this versioned, access-controlled data to the application. This service is provided by a config server. Let’s begin adding this to our running example.

**6.4.1. Introducing the configuration server**

At the moment, our implementation offers some level of control at the Connections’ Posts service. A user must authenticate before the service will deliver results. But the two services that ultimately provide the data, the Connections and Posts services, remain wide open. Let’s secure these services with secrets. We’ll use secrets instead of user authentication and authorization because these services won’t be called by a specific logged-in user, but rather will be called by another software module (in our case, the Connections’ Posts service). For example, you use the Posts service here to get the posts for the set of users being followed by the logged-in user, but in another setting, you might use the same service to get the posts for any bloggers who are currently trending.

I’ve implemented secrets in the example by configuring a secret into both service being secured (the Connections and the Posts services), and by configuring the same secret into the client (the Connections’ Posts service). Before looking at the implementation in detail, let’s first look at how to manage these values.

First, you want to create a source-code repository to hold the secrets. You can create a repo from scratch, or to make things easier, you can fork a super-simple repo that I have at <https://github.com/cdavisafc/cloud-native-config.git>. You’ll need to fork it so you can commit changes as you go through the exercise. In there, you’ll see something that looks suspiciously close to a properties file: mycookbook.properties. This file contains two values—the secret that will protect the Posts service and another that will protect the Connections service:

com.corneliadavis.cloudnative.posts.secret=123456

com.corneliadavis.cloudnative.connections.secret=456789

Now you’ll establish the service that will manage access to these configuration values, and for that you’ll use Spring Cloud Configuration (<https://github.com/spring-cloud/spring-cloud-config>). The Spring Cloud Configuration Server (SCCS) is an open source implementation that’s well suited to managing data for distributed systems (cloud-native software). It runs as an HTTP-based web service and provides support for organizing data around your complete software delivery lifecycle. I refer you to the README in the repository for further details, but demonstrate a few key capabilities here.

Let’s start putting the pieces together. First, check out the following repository tag:

git checkout appconfig/0.0.2

Next, let’s get SCCS up and running. Fortunately, there’s already a Docker image for the server, and I’ve provided you a Kubernetes deployment manifest. Before creating the pod with the usual command, fork the <https://github.com/cdavisafc/cloud-native-config.git> repository, and then replace the URL in the following snippet of the deployment manifest with the URL to your repository:

env:

- name: SPRING\_CLOUD\_CONFIG\_SERVER\_GIT\_URI

value: "https://github.com/cdavisafc/cloud-native-config.git"

Then create the service with the following command:

kubectl create -f spring-cloud-config-server-deployment.yaml

After the server is up and running, you can access the configurations with the following command:

$ curl $(minikube service --url sccs-svc)/mycookbook/dev | jq

{

"name": "mycookbook",

"profiles": [

"dev"

],

"label": null,

"version": "67d9531747e46b679cc580406e3b48b3f7024fc8",

"state": null,

"propertySources": [

{

"name": "https://github.com/cdavisafc/cloud-native-

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg config.git/mycookbook.properties",

"source": {

"com.corneliadavis.cloudnative.connections.secret": "456789",

"com.corneliadavis.cloudnative.posts.secret": "123456"

}

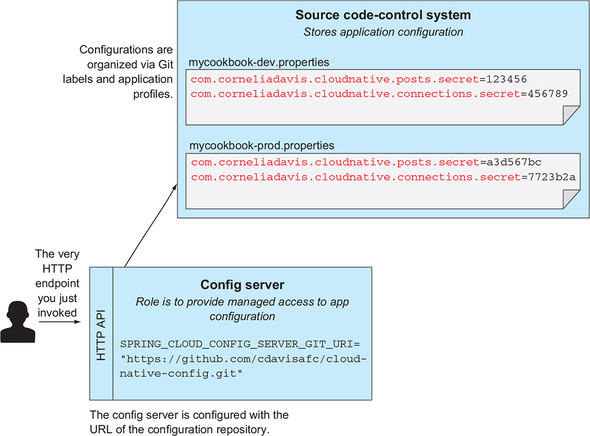
}

]

}

SCCS supports tagging configurations with both Git labels and application profiles. My sample config repository includes two configuration files for the mycookbook application—one for dev and one for prod. Executing the preceding curl command and replacing /dev with /prod will show the values for the production profile. What you’ve now established is shown in [figure 6.7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06fig07): a GitHub repository that stores configs, and a configuration service that manages access.

**Figure 6.7. Application configuration is facilitated through the use of a source code-control system that will persist configuration values, and a configuration service that provides managed access to that data.**



Let’s look at both ends of the relationship that you’re securing with your secrets. In the following listing, the Posts (and in the same way, the Connections) service will now check that the passed secret matches what has been configured in, and the Connections’ Posts service will pass the secret that has been configured into it.

**Listing 6.5. PostsController.java**

public class PostsController {

...

**@Value("${com.corneliadavis.cloudnative.posts.secret}")**

**private String configuredSecret;**

...

@RequestMapping(method = RequestMethod.GET, value="/posts")

public Iterable<Post> getPostsByUserId(

@RequestParam(value="userIds", required=false) String userIds,

@RequestParam(value="secret", required=true) String secret,

HttpServletResponse response) {

Iterable<Post> posts;

**if (secret.equals(configuredSecret)) {**

logger.info(utils.ipTag() +

"Accessing posts using secret " + secret);

// look up the posts in the db and return

...

} else {

logger.info(utils.ipTag() +

"Attempt to access Post service with secret " + secret

+ " (expecting " + password + ")");

response.setStatus(401);

return null;

}

}

...

}

In the Connections’ Posts service, the secret that’s configured in will be passed in the request to the Connections or Posts services, as shown in the following listing.

**Listing 6.6. ConnectionsPostsController.java**

public class ConnectionsPostsController {

...

@Value("${connectionpostscontroller.connectionsUrl}")

private String connectionsUrl;

@Value("${connectionpostscontroller.postsUrl}")

private String postsUrl;

@Value("${connectionpostscontroller.usersUrl}")

private String usersUrl;

**@Value("${com.corneliadavis.cloudnative.posts.secret}")**

**private String postsSecret;**

**@Value("${com.corneliadavis.cloudnative.connections.secret}")**

**private String connectionsSecret;**

@RequestMapping(method = RequestMethod.GET, value="/connectionsposts")

public Iterable<PostSummary> getByUsername(

@CookieValue(value = "userToken", required=false) String token,

HttpServletResponse response) {

if (token == null) {

logger.info(utils.ipTag() + ...);

response.setStatus(401);

} else {

ValueOperations<String, String> ops =

this.template.opsForValue();

String username = ops.get(token);

if (username == null) {

logger.info(utils.ipTag() + ...);

response.setStatus(401);

} else {

ArrayList<PostSummary> postSummaries

= new ArrayList<PostSummary>();

logger.info(utils.ipTag() + ...);

String ids = "";

RestTemplate restTemplate = new RestTemplate();

// get connections

**String secretQueryParam = "?secret=" + connectionsSecret;**

ResponseEntity<ConnectionResult[]> respConns

= restTemplate.getForEntity(

connectionsUrl + username + secretQueryParam,

ConnectionResult[].class);

ConnectionResult[] connections = respConns.getBody();

for (int i = 0; i < connections.length; i++) {

if (i > 0) ids += ",";

ids += connections[i].getFollowed().toString();

}

logger.info(utils.ipTag() + ...);

**secretQueryParam = "&secret=" + postsSecret;**

// get posts for those connections

ResponseEntity<PostResult[]> respPosts

= restTemplate.getForEntity(

postsUrl + ids + secretQueryParam,

PostResult[].class);

PostResult[] posts = respPosts.getBody();

for (int i = 0; i < posts.length; i++)

postSummaries.add(

new PostSummary(

getUsersname(posts[i].getUserId()),

posts[i].getTitle(),

posts[i].getDate()));

return postSummaries;

}

}

return null;

}

...

}

Aside from certain things you’d never do in a real implementation, and I’ll come back to those in a moment, none of this is causing you any surprise. But look at the way that the configuration value is brought into the app. The property file for the Connections’ Posts service is as follows.

**Listing 6.7. Connections’ Posts application.properties**

management.endpoints.web.exposure.include=\*

connectionpostscontroller.connectionsUrl=http://localhost:8082/connections/

connectionpostscontroller.postsUrl=http://localhost:8081/posts?userIds=

connectionpostscontroller.usersUrl=http://localhost:8082/users/

ipaddress=${INSTANCE\_IP:127.0.0.1}

redis.hostname=localhost

redis.port=6379

com.corneliadavis.cloudnative.posts.secret=drawFromConfigServer

com.corneliadavis.cloudnative.connections.secret=drawFromConfigServer

As I’ve already talked about, the properties defined here may simply be acting as placeholders. Both the secrets have values that read drawFromConfigServer. (This isn’t an instruction, but rather is arbitrary. It could equally have been set to foobar.) And then the Connections’ Posts controller has lines that read like this:

@Value("${com.corneliadavis.cloudnative.posts.secret}")

private String postsSecret;

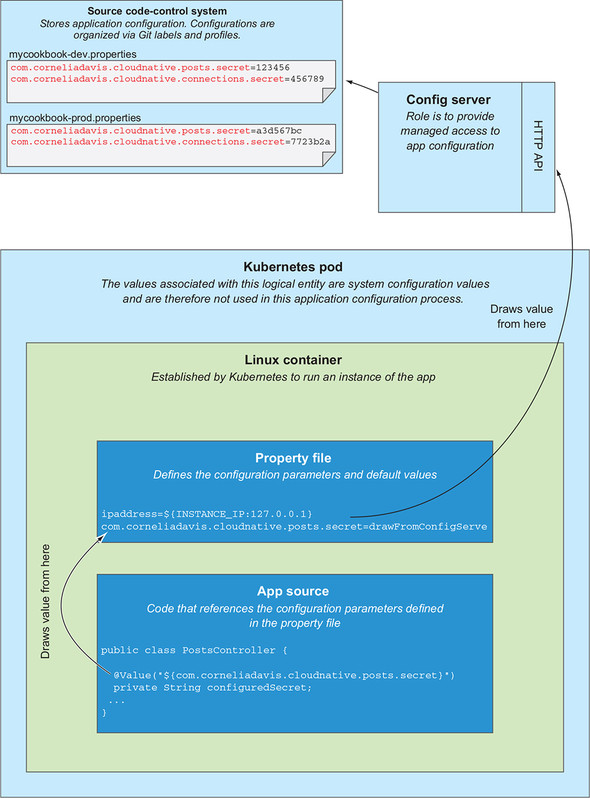
@Value("${com.corneliadavis.cloudnative.connections.secret}")

private String connectionsSecret;

This looks familiar, because it’s exactly the same technique that was used to draw in the INSTANCE\_IP system config value. And that’s exactly the point. The application config layer takes exactly the same form, whether the values are system/environment values or application configuration values being injected.

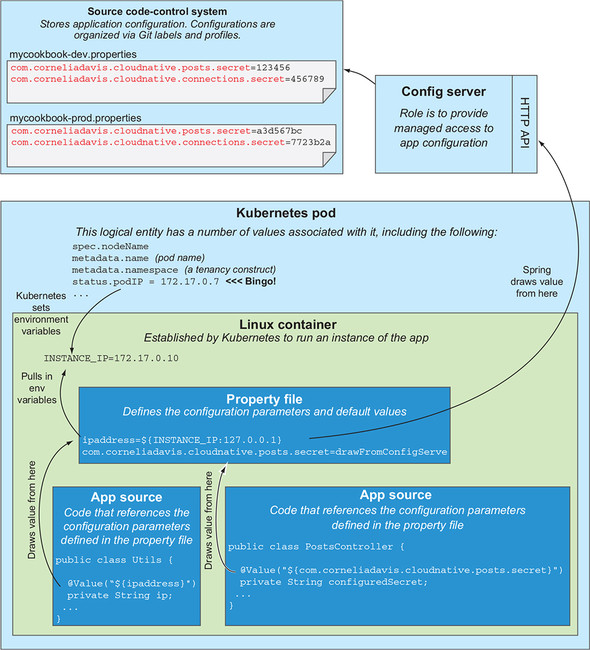
[Figure 6.8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06fig08) shows how application config data makes its way into the running application. Notice that the application configuration layer, with the property file at the center, remains as simple as shown in [figure 6.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06fig04). The only thing that has changed is that the configuration server is supplying the bindings to the variables defined in the property file.

**Figure 6.8. The application configuration layer depends on the property file as the means for injecting values; those values come in through the use of a config server.**



Now let’s put both the application and system configuration together in one diagram; see [figure 6.9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06fig09). This is exactly what’s implemented in our sample application. Again, note that the patterns used in the application configuration layer are the same for both types of config data. What differs is the way the values are injected into that application configuration layer. For system data, it’s handled by the platform (in this case, Kubernetes) and is well served through the use of env variables. For application configuration data, you’re using Spring (there are other options—look for future blog posts from me on the subject), which makes calls to the HTTP interface for the Spring Cloud Configuration Server; this approach allows you to version the app config data.

**Figure 6.9. The property file acts as the common configuration layer for both system config and application config. System config data is injected via env variables, and app config data is injected via a configuration service.**



**6.4.2. Security adds more requirements**

I’ve created the implementation this way to make it easy to reason about the main design patterns for configuring cloud-native applications. But there are several things that you wouldn’t do as I have here:

* You would never pass secrets on the query string; they’d instead be passed in an HTTP header or in the body.
* You definitely wouldn’t print the values of secrets in log files.
* In the configuration store, you’d at the very least encrypt any sensitive values. SCCS does support encryption, and technologies such as HashiCorp’s Vault provide additional services for credential management.
* You’ll notice that every method in the Posts and Connections controllers now has effectively the same code wrapping the method functionality. This boilerplate distracts from the main functionality of the method and is repeated too frequently. Most modern programming frameworks provide security-related abstractions that allow for this functionality to be configured more elegantly.

**6.4.3. Let’s see this in action: Application configuration using a config server**

Okay, so your implementation will work beautifully, provided the secrets configured into all of the apps match. That goes directly to one of the key concerns with configuring cloud-native apps: they’re highly distributed! Notice that the mycookbook properties aren’t defined for a single app; the same configuration is used across different microservices. I have a single place where I configure the secret, and my operational practices will draw them together in the right way.

With all of that laid out, let’s verify that what we’ve designed works as advertised.

**Setting up**

If you’ve already followed the setup instructions from the example in [section 6.3.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06lev2sec4), you needn’t do more here. As always, you’re welcome to build the executables from source, build the Docker images, and push them to your Docker Hub repository. But I’ve already done that and made them available to you in Docker Hub. All of the configuration files point to the appropriate Docker images.

**Running the apps**

First, clean up the set of microservices you’ve deployed to Kubernetes; recall that I provided a script that allows you to do this in one shot by typing this command:

./deleteDeploymentComplete.sh

Before you redeploy the services, you want to connect the deployment process to the configuration server. You’ve already deployed the config server (if you haven’t yet done this as described previously, please do so now). Now you must inject the coordinates of that config server into the implementation so that the Spring Framework can use that connection to find and inject the configuration values. In the Kubernetes deployment manifests for each of the services, you’ll find the definition of, what else, an environment variable that has the URL for SCCS. You need to provide your specific URL in all three places—the Posts, Connections, and Connections’ Posts deployment manifests. You can obtain the correct value with the following command:

minikube service --url sccs-svc

Assuming you’ve left the Redis and MySQL services running, those URLs don’t need to be updated. You can deploy the Posts and Connections services with the following two commands:

kubectl create -f cookbook-deployment-connections.yaml

kubectl create -f cookbook-deployment-posts.yaml

Again, you must now update the deployment manifest for the Connections’ Posts service to point to the property URLs for the Posts and Connections services. Recall that you can obtain these values as follows:

|  |  |
| --- | --- |
| Posts URL | minikube service posts-svc --format "http://{{.IP}}:{{.Port}}/posts?userIds=" --url |
| Connections URL | minikube service connections-svc --format "http://{{.IP}}:{{.Port}}/connections/" --url |
| Users URL | minikube service connections-svc --format "http://{{.IP}}:{{.Port}}/users/" --url |

Now deploy the Connections’ Posts service with the following:

kubectl create -f cookbook-deployment-connectionsposts.yaml

Invoke the Connections’ Posts service just as you previously have, by first authenticating and then fetching the posts:

# authenticate

curl -X POST -i -c cookie \

$(minikube service --url connectionsposts-svc)/login?username=cdavisafc

# get the posts

curl -i -b cookie \

$(minikube service --url connectionsposts-svc)/connectionsposts

Nothing has changed, huh? Just as you should expect, but let’s take a quick peek under the covers by looking into the log files for the Posts service:

... : [172.17.0.4:8080] Accessing posts using secret 123456

... : [172.17.0.4:8080] getting posts for userId 2

... : [172.17.0.4:8080] getting posts for userId 3

You can see that the Posts service has been accessed using the secret 123456. Obviously, the secrets were properly configured into both the caller (Connections’ Posts) and the callee (Posts) services.

Now, what happens when you need to update application configuration? You want new values injected equally in all application instances for a single service, and of course, when values are to be inserted into different services, that must be coordinated as well. Let’s try this out. The first thing I’ll ask you to do is update the secret values for the dev profile of mycookbook; you can change the values to anything you like. You must then commit those changes to your repo and push to GitHub. From the cloud-native-config directory:

git add .

git commit -m "Update dev secrets."

git push

If you now issue the final curl that accesses the Connections’ Posts data, it all works as expected. But if you look at the Posts log file again, you’ll see that the Posts access still uses, and successfully, the secret 123456. And here we come to the topic of the next chapter—application lifecycle. You must be deliberate about when configuration changes are applied. From the example so far, you can see that the new credentials haven’t yet been applied.

But they will be on startup, so I’ll have you restart the Posts service by deleting the pod. Because you’ve instantiated a Kubernetes deployment that specifies that one instance of the Posts service should always be running, Kubernetes will immediately create a new pod for a new instance of the Posts service:

kubectl delete pod/posts-66bcfcbbf7-jvcqb

Now curl the Connections’ Posts service again. You’ll see two things:

* First, the service invocation will fail.
* And second, looking at the log file for the Posts service shows you why:

... : [172.17.0.7:8080] Attempt to access Post service with secret 123456 (expecting abcdef)

When you deleted and re-created the Posts service, it picked up the new configuration values. However, the Connections’ Posts service still has the old ones configured in. The old are sent; the new are expected. Clearly, you need to coordinate the update of configuration across instances and sometimes across services, but before you can go there, you must study application lifecycle concerns and patterns. That is the topic of the next chapter.

**Summary**

* Cloud-native software architecture requires a reevaluation of the techniques you use for app configuration. Some existing practices remain, and some new approaches are useful.
* Cloud-native application configuration is not as simple as just storing config in environment variables.
* Property files remain an important part of proper handling of software configuration.
* Using environment variables for configuration is ideally suited to system config data.
* You can use cloud-native platforms such as Kubernetes to deliver environment values into your apps.
* App config should be treated just as source code is: managed in a source code repository, versioned, and access controlled.
* Configuration servers, such as Spring Cloud Configuration Server, are used to deliver configuration values into your apps.
* You now have to think about when config is applied, which is inherently related to the cloud-native application lifecycle.