**Chapter 7. The application lifecycle: Accounting for constant change**

*This chapter covers*

* Zero-downtime upgrades: blue/green and rolling
* Canaries
* Credential rotation patterns
* App lifecycle and troubleshooting
* Application health checks

The application lifecycle seems pretty basic: an app gets deployed, is started, is running for a bit, and is eventually shut down. Aside from the chaos that ensues when that “shutting down” part happens unexpectedly, this lifecycle is generally boring (or so we hope). Why, then, have an entire chapter dedicated to the topic?

Before I answer that question, let me first be clear on my definition of *application lifecycle*. The application lifecycle I cover here is distinctly different from the software development lifecycle (SDLC) that I’ve already talked about a great deal. The latter is about the phases your software goes through in the *development* and *delivery* of software—from design, to being under development, to first passing unit tests and then integration tests, to delivering it to production.

The application lifecycle, on the other hand, is all about the stages an application goes through after it’s ready for production deployment. The central concern isn’t the work going into the development or management of the software, but the state of the application itself. Is it deployed? Running? Stopped (either via crash or intentional stop)? Although it’s natural to consider an app being deployed as a part of the SDLC, the focus here is on the running state of an application.

To help you understand the content of this chapter as well as provide a basis for later explanations, [figure 7.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig01) depicts a fairly vanilla sequence of stages an app goes through in its lifetime. You’ll notice that I’ve excluded the creation of the deployable artifact from the app lifecycle; that’s part of the SDLC. On the other hand, you might find it curious that I’ve included having a provisioned environment and, later, a disposed-of environment, within the app lifecycle. I’ll ask for your patience; I promise this will become clear soon.

**Figure 7.1. A simple depiction of the stages in an app’s lifecycle**



So apps are started and stopped. What makes this interesting in our cloud-native setting? As it happens, both of the factors that characterize cloud-native apps: that they’re highly distributed and constantly changing.

Let’s start with the former. You’ve already learned that even when multiple instances of an app are deployed, collectively they need to behave as a single logical entity. How, then, do you properly handle something like applying new configuration to an app, or deploying a new version? Do you do this for all the apps in lockstep, or is there another approach? (Look back at [figure 6.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06fig01), which indicates that “eventually” all instances must have the same configuration; this was a foreshadowing of material I cover in this chapter.)

As for constant change, recall that apps will regularly be moved around, because of either a failure or a management event such as addressing a vulnerability in the operating system. Aside from this leading to many starts and stops, there are cascading effects we need to account for. For example, when another microservice depends on an app that has just been started, information about the newly running app might need to be made available to those dependent apps.

Application lifecycles for the cloud-native app are different from those of apps running in decades past, and this places new requirements on app design. This is why I’ve focused this entire chapter on the app lifecycle.

After briefly reviewing some operational concerns that intersect with the app lifecycle, I will talk about the application lifecycle for multiple instances of an app (which is what you’ll have pretty much always). As is already abundantly clear, your software consists of many apps working together, so I’ll talk about the need to be aware of the ways that application lifecycle events for one app affect another. Then I’ll cover the ephemeral context in which our apps now live and what that means for your app designs. In an environment that’s constantly changing, it becomes paramount that app health can be accurately assessed, and if necessary, acted upon, so you’ll see a section on that. And finally, I’ll talk briefly about the serverless, or rather, function-as-a-service programming paradigm from the lens of the app lifecycle.

**7.1. Having empathy for operations**

The app lifecycle is more about ops than development, but as a developer, you need to deliver software that can be effectively managed in production. Having empathy for the application operator is a theme that runs through this text. Heck, these days you’re likely to be doing ops as well as dev, so attention placed here is self-serving. Let’s have a brief look at some of the main operational concerns, with an eye on the impact of the cloud-native lifecycle:

* ***Manageability—***One of the first concerns for operations is the sustainability of managing application deployments. Wherever possible, management functions should be automatable, and when any management tasks are necessary, they should be done efficiently and reliably. The way you design your software can have a marked impact. For example, as you’ll see, a change in app configuration almost always necessitates a restart of the app, so you should carefully decide whether to make something configuration or input data.
* ***Resilience—***By the end of this book, you’ll have a solid appreciation for platforms that keep apps running on your behalf, even while a constant stream of changes are happening around them. I sometimes like to think of these platforms as robots—robots that are handling a whole host of operational tasks that humans used to do. But here’s the thing: robots don’t read release notes. I make this statement with a wink, and definitely metaphorically, but the intent holds as true for the app lifecycle as it does for many other aspects of your cloud-native software. For example, if a system like Kubernetes will stand up a new instance of an app when one has failed, it must have a nonsubjective way of detecting that an app has failed or is failing. It’s up to you, the application developer, to ensure that the platform has a fail-safe way of detecting when an app has failed.
* ***Responsiveness—***Users of your software must receive outputs in a timely manner, and what is considered timely depends on the use case. For example, if a user is uploading a PowerPoint deck to SlideShare, it’s okay if that deck isn’t available to others for several minutes, allowing time for any format conversions to take place. On the other hand, if a user is trying out a news aggregation website for the first time and sees that pages are taking tens of seconds to render, it will likely be their last visit. Many factors will affect the responsiveness of your app, and the way the app lifecycle relates to user actions is one. For example, if an app is started up only after a user request has been made, the user will feel the cost of that startup. In the former example, they probably won’t notice. In the latter, that cost could mean the difference between a retained or lost customer.
* ***Cost management—***One of the great promises of the cloud is that of cost efficiency. Instead of provisioning, configuring, and managing infrastructure for peak loads, or, inevitably, an overestimation of peak loads, you’re able to use only the resources you need at the moment. Being able to scale application capacity out or in depending on the traffic load, even being able to optimize away any idle computing time, is a powerful lever in IT cost management. These scaling operations mean new apps are started and old ones are stopped. Handling these app lifecycle events gracefully is essential.

In the remainder of this chapter, I’ll cover many of the patterns that well serve the preceding list of concerns.

**7.2. Single-app lifecycle, multiple-instance lifecycles**

Let’s start with a concrete example: our Posts service. In [chapter 6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06), you added some configuration to this service, a secret used for authorization. Let’s assume you have more than one instance of the app, each of which is running with the same configuration (the same secret). Everything is humming along just fine, the app is functioning exactly as you want, and then, oops, you inadvertently check your mycookbook.properties file that contains the secret into a public GitHub repository.

Although you quickly realize your mistake and pull the file, your secret may have been compromised, so you need to rotate this credential. To do so, you’re going to change your configuration source file and, via the config server, deliver that configuration to all of the running apps. And here’s where you need to think about the application lifecycle.

|  |
| --- |
|  |

**Warning**

Yes, I realize that in the running code sample, I’ve had you do exactly the thing that I’ve now characterized as a problem: storing credentials in a public GitHub repository. I’ll direct your attention to the comments in [chapter 6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06), which I’m repeating here: secrets should be stored in a repository expressly designed for handling sensitive information—something like HashiCorp’s Vault or Pivotal’s CredHub. I’m using GitHub in our samples only to keep things simple.

|  |
| --- |
|  |

I say, “deliver that configuration to the running apps,” but that statement is a bit disingenuous. What I mean is that I want you to roll the application—to restart it with the new configuration applied. That’s exactly what I had you do at the end of [chapter 6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06); I had you delete the Posts pod, causing Kubernetes to create a new one. This is lesson number one when it comes to application configuration and the app lifecycle.

|  |
| --- |
|  |

**Note**

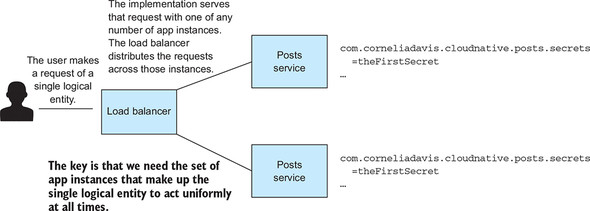
When new configuration is being applied to your running app, the app should be re-created, and hence restarted, with the new configuration applied.

|  |
| --- |
|  |

Some of you might feel uncomfortable with this guidance. It might seem wasteful to restart the app when a cost is associated with that; it takes time to start an app. And re-creating the runtime environment goes even further and costs even more. And if you know Spring well, you might already be thinking of the easily enabled /refresh endpoint that, when invoked, will refresh the application context without completely restarting the app. I suggest that use of /refresh is a hack that can easily lead to unmanageable application deployment.

To explain why, let’s consider the following scenario. Say you have two instances of the Posts service deployed. As I’ve already covered, one of the tricks with cloud-native application architectures when you’re running more than one instance of an app is to have those multiple instances serving results as a single logical application. When a request is made to the Posts service, it will result in the same outcome regardless of which instance is reached. [Figure 7.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig02) shows two instances of the Posts service fronted by a load balancer.

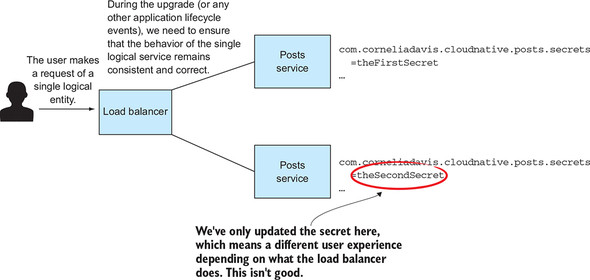
**Figure 7.2. Both instances of the Posts service are running with the same configuration and act as a single logical application.**



Now let’s consider what would happen if you curled the /refresh endpoint. That curl command would reach only one instance, effectively updating the secret only for that instance. Now, as seen in [figure 7.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig03), one instance is running with theFirstSecret, and the other is running with theSecondSecret configured in. It’s now easy to see that when a request such as the following is issued, if the load balancer directs the request to the first instance, the request will succeed, but if it reaches the second, it will fail. Our two instances are definitely not acting as a single logical app:

$ curl http://myapp.example.com /posts?secret=theFirstSecret

**Figure 7.3. Because the two instances of the Posts service have different configurations applied, the outcome of issuing a request will differ greatly depending on whether a request is routed to the first instance of the app or the second. This must be avoided, even during a zero-downtime upgrade.**



Okay, so you might think the answer is simple: just roll through all the instances and update the configuration. Ultimately, you’d be right, but rolling through the instances in such a way that things stay running throughout is a bit tricky. You must consider factors including the way all app instances are cycled through and the state of the system as a whole during the upgrade process.

To hit all the instances, you might be tempted to just issue another /refresh curl command, but there’s no guarantee that this command will reach the other app instance. This is why using the /refresh URL is a hack. You’re trying to use a user’s interface to perform a management function. Upgrading the configuration of all app instances is absolutely a management function, and you need to use a tool that will operate on each and every instance, and you need to be in control of this. You can’t be at the mercy of a load balancer. The management function of upgrading the configuration of all the instances must sit behind the load balancer, not in front of it.

I will show you one of those control tools (spoiler alert: it’s in Kubernetes) when you look at a concrete example in [section 7.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07lev1sec3), but for now let’s continue the discussion assuming such a control plane function exists. Although the mechanism of using the /refresh URL isn’t right, the intent is to refresh the application configuration, and you want to apply this new configuration all with zero downtime. Let’s consider three options:

* Change the configuration while the app is running.
* Stand up a second set of instances, all of which will have the new configuration applied, and then switch all traffic from the first set to the second in one fell swoop. This is a *blue/green deployment*.
* Roll through the application instances, replacing a subset of them with the new, and then moving on to the next subset. This is a *rolling upgrade*.

I’d like to immediately rule out the first option for a couple of reasons. First, many applications and application frameworks apply configuration changes only at app startup time. For example, with the way that we’ve used .property files in our samples, a good practice for sure, config changes won’t be applied until the application context is refreshed, and this is pretty close to an app restart (and exactly what the /refresh endpoint does).

In addition, applying configuration changes without restarting may bring your application into a state that can’t be reproduced. Let’s say, for example, an app loads reference data on startup, and the location from which that data is loaded is supplied in a configuration parameter. If you change that configuration parameter and trigger a load of data from the new location, the reference data loaded in memory would be a combination of the first load and the second, and the functionality of the app will reflect that state. Now suppose the app crashes and you want to troubleshoot. There’s no way for you to obtain an instance that has the same state that the crashed app did. As you’ve already seen repeatedly, reproducible app deployments are absolutely essential in a cloud context in which app instances are frequently being created.

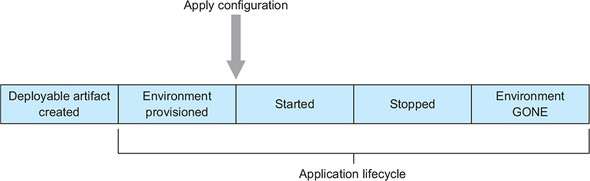
|  |
| --- |
|  |

**Note**

Applying application configuration at startup time ([figure 7.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig04)) greatly simplifies operational practices and is therefore strongly recommended.

|  |
| --- |
|  |

**Figure 7.4. Application configuration is best applied when the application is started. Most application frameworks naturally function this way.**



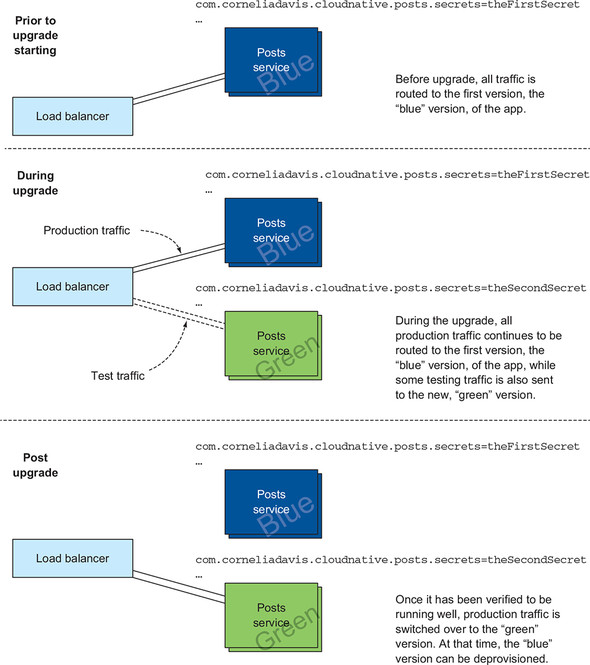
The second and third options are both valuable in the context of cloud-native apps. Let’s look more deeply at each of them.

**7.2.1. Blue/green upgrades**

Used to update the configuration or version of your running app, the blue/green deployment is, particularly from a developer perspective, the simplest approach. It starts when you have one version of the app running, the “blue” version, and wish to deploy a new version, the “green” one.

[Figure 7.5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig05), which assumes that multiple instances of your cloud-native app are deployed, depicts the process. To start, you have a load balancer serving traffic to all blue instances of the app. In the next step, you deploy a complete set of instances of the new, green version but leave all production traffic routed to the blue. Now you can check to see that the green instances are operating correctly by sending traffic to them, and after that’s verified, you cut all traffic over from the blue version to the green.

**Figure 7.5. The blue/green deployment is used when an app can’t tolerate having multiple versions running side by side (when the multiple versions can’t be made to act as a single logical instance).**



|  |
| --- |
|  |

**Note**

What makes blue/green deployments simpler than rolling upgrades from the viewpoint of your application design is that with the former, you have only one version of the app running at a time.

|  |
| --- |
|  |

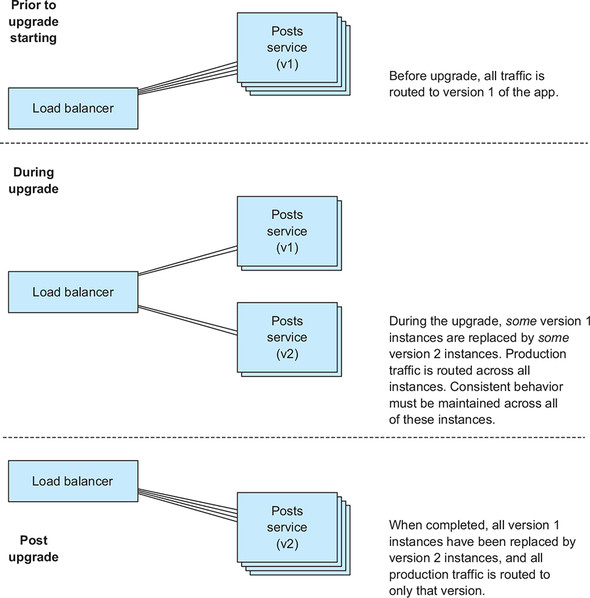
This preceding NOTE is interesting indeed. The idea that you have in production only a single version of an app at a time is what you’re used to, but when you eliminate this assumption, it affords you a great deal of power. The rolling upgrade is one of those things.

**7.2.2. Rolling upgrades**

A *rolling upgrade* is also used to perform a zero-downtime update to a running application, and just as with blue/green deployments, when completed, all traffic to the app will be routed to the new version of an app. During the upgrade process, however, things are markedly different.

[Figure 7.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig06) depicts the process. At the onset, all traffic is being load-balanced across the multiple instances of the current version of the app. During the upgrade, you’ll incrementally take a subset of the current version instances offline and bring a number of instances of the new version of the app online to replace them. As you do so, traffic is load-balanced across instances of the original version of the app *and* instances of the new version of the app. Traffic is being served to more than one version of the app! After the first batch is running well, you upgrade the next batch, and so on. The rolling upgrade is complete when all old versions of the app have been replaced by new.

**Figure 7.6. During rolling upgrades, production traffic is routed to more than one version of the app. The app must be able to function as a single logical whole, even as requests are being distributed over different versions.**



I once again draw your attention back to [figure 6.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06fig01), which indicates that eventually all instances of an app will be running with the same configuration. Now you can see concretely what I mean.

|  |
| --- |
|  |

**Note**

During a rolling upgrade, application traffic will be handled by different versions of the same app.

|  |
| --- |
|  |

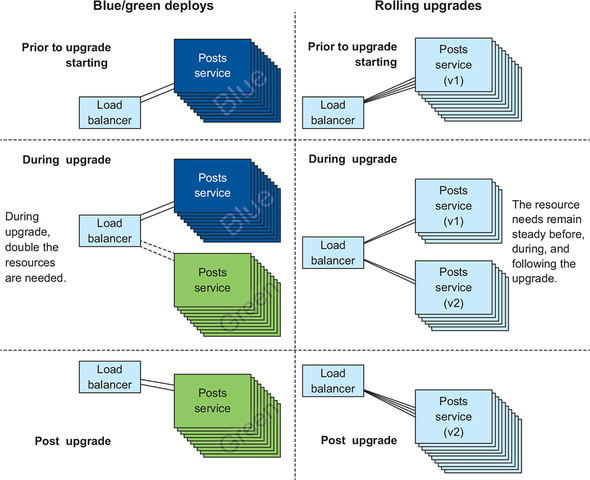
The point in the above NOTE is interesting indeed. Recall one of our fundamental premises: the set of independent application instances should all operate collectively, as a whole. The outcome of invoking an app should be the same regardless of which app instance responds to the invocation request. Think about this for a moment. This means that you, the application architect/developer, are responsible for making sure that your app design supports this deployment pattern. Now, let me be clear here: this may not always be possible, but if you can realize this characteristic in your design, then and only then will a rolling upgrade deployment pattern be available.

**7.2.3. Parallel deployments**

No question, it takes more care to create software that allows for rolling upgrades (and shortly I’ll take you through a code-based, concrete example of this). Therefore, you might wonder what you get from this added effort, if in the end a blue/green and a rolling upgrade yield the same result—all instances have been switched to the new version. The short answer is, “a lot,” but let me draw your attention to two points in particular.

First, a blue/green deployment requires more resources than the rolling. During the upgrade, in order to keep the application capacity stable, you’ll need as many instances of the green as you do of the blue. Only after the switch is flipped and all the traffic is routed to the new version may the resources being used by the old be freed. With the rolling upgrade, you choose the batch size—the number of instances that will be replaced per cycle—and with that, you’re able to control the resources required of the upgrade process. You can see this contrast in [figure 7.7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig07).

**Figure 7.7. One consideration when deciding between blue/green and rolling upgrades is the resource requirement. Note that during a blue/green deployment, double the resources are needed for the Posts API, whereas the rolling upgrade has only a slight increase in the resource requirements.**



Second, an app design that allows for multiple versions to be running at once offers more benefits than the rolling upgrade. One of those is agility.

Your cloud-native software is made up of many apps, with multiple instances of each app running at any given time, and a major goal in this modern era is the frequent evolution of that software. You know from our prior conversations that requiring all of the different microservices that make up the software to upgrade in lockstep effectively thwarts that speed. Approaches such as these yielded the nightmarish Gantt charts of the past that tracked dozens or hundreds of dependencies across various components and teams, and forced alignment before production changes could be made.**[**[**1**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fn1)**]**

***1***

*Wikipedia’s entry on Gantt charts (*[*http://mng.bz/O2za*](http://mng.bz/O2za)*) provides more information.*

Now consider that your app may be used from many other components of the software. When you want to upgrade your app, do you want all of your consumers to have to adjust to the new at the same time? Of course, the answer is no, and if you’ve built your app in such a way that multiple versions can be running side by side, you can have some consumers using the old, while other consumers are using the new. We call this *parallel deploys*.

Another use case for parallel deploys is to support experimentation. The example I always use to explain this concept is that of an e-commerce recommendation engine. Although I don’t have direct knowledge of this, I’d be willing to bet that Amazon has multiple versions of its recommendation engine running at once. The algorithms that suggest related items to a shopper are complex, almost certainly driven by models produced through machine learning. Slight tweaks to an algorithm, or even only different configurations to a model, can produce different click-through rates and purchase volumes. To optimize results, a retailer can have these multiple versions running side by side for a time, analyze the results, and then keep the better-performing version.

Parallel deployments are powerful, but they do place an increased importance on the versioning of your software. When traffic is being routed to more than one version of an app, the version must be identifiable, and the version of running software is ultimately made up of the version of the deployable artifact and the version of the configuration applied to the running instances.

|  |
| --- |
|  |

**Note**

Running App Version = Version of Deployable Artifact + Version of the App Config

|  |
| --- |
|  |

Controlling the version of your deployable artifact should be handled by your build pipelines. Application configuration versioning is handled through source code control. If you’re using Git, for example, the app operator can use the commit SHA (Secure Hash Algorithm) as the version.

To summarize the main points of this section:

* You need to have multiple app instances act collectively even during an upgrade application lifecycle event.
* Building apps in a way that allows multiple versions to run side by side allows for rolling upgrades to be used (and provides other benefits as well).
* Rolling upgrades have benefits over blue/green deploys.

Finally, it’s worth pointing out that any app that allows for rolling upgrades can also be deployed in a blue/green fashion. However, the opposite isn’t true. To use a rolling upgrade, your app must be designed to allow different instance versions and configurations to be running at the same time; with a blue/green deploy, only one version/configuration is serving traffic at once, so this special handling isn’t required.

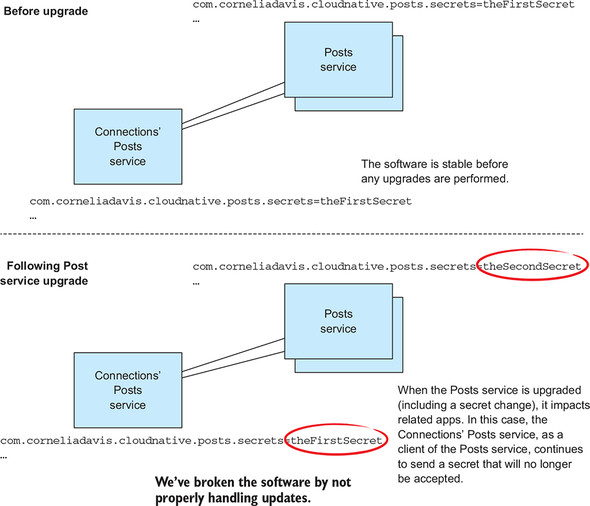
Proper behavior of an app across multiple instances during an app lifecycle event such as an upgrade is only half of the picture. It’s also important to consider how an app upgrade affects the clients of that app. How does the application lifecycle of one app affect those of another? Let’s cover that now.

**7.3. Coordinating across different app lifecycles**

Coming back to our Posts service, in [figure 7.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig03) it was clear that the two app instances wouldn’t provide consistent behavior, an obvious problem. And now that you’ve just digested the content of the previous section, you can see that this app can’t be upgraded in a rolling fashion. Let’s say that you’ve updated the app by using a blue/green deployment. The Posts service now has a new secret configured in.

This brings us to another problem: application lifecycle events affect not only the app to which they’re being applied, but also affect other, dependent apps. For example, because the Connections’ Post service is a client of the Posts service, you see how application lifecycle events of one app affect a dependent app. In this specific scenario, when you rotate the credential in the Posts service, you also need to rotate it in the Connections’ Posts service. This dependency is depicted in [figure 7.8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig08).

**Figure 7.8. Application lifecycle events must be coordinated across different components of your software.**



The question, then, is how to coordinate these updates. Clearly, it’s not in any type of a “transaction.”**[**[**2**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fn2)**]** Your software is a distributed system, and as has been made abundantly clear, maintaining autonomy is an important characteristic for the apps that make up your cloud-native software. If you can update clients and services only in lockstep, you’ve lost a great deal of that autonomy.

***2***

*Incurring downtime for your software while you update all of the dependent pieces is one type of transaction, one that clearly isn’t zero downtime.*

Instead, you’ll design your apps so that application lifecycle events across different apps can proceed independently, all while keeping the software fully functional with zero downtime. No single pattern provides the solution for all cases. As the application architect/developer, it’s your job to design the right algorithm.

|  |
| --- |
|  |

**Note**

You must design your apps and document the API so that any lifecycle events that have an impact on dependent services are eliminated, are minimized, or can be adapted to by those clients.

|  |
| --- |
|  |

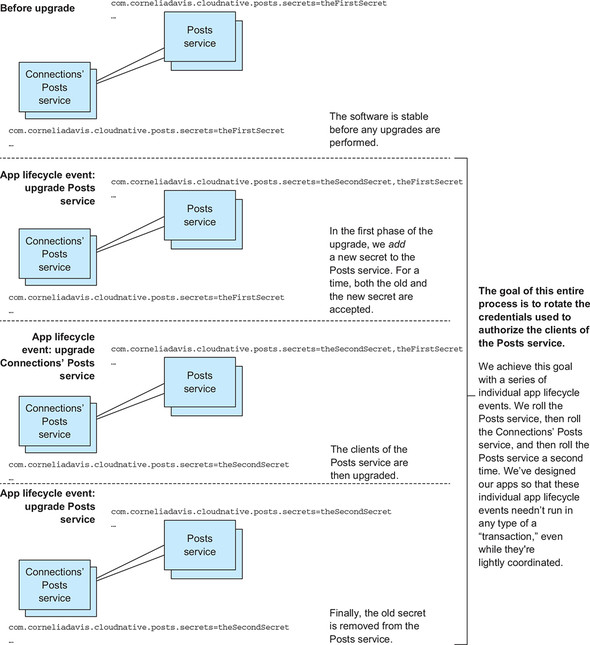
Okay, I concede, this is all a bit abstract. Let’s turn back to our sample app to make this more understandable. Recall that earlier you had compromised the secret for your Posts service and as a result needed to update the configuration of your running app. You do so by restarting all app instances with the new configuration applied. For the moment, don’t concern yourself with whether you do so using a blue/green deployment or a rolling update. The first, naïve approach poses the following challenge:

* If you first update the Posts service, requests coming from the Connections’ Post service, which are sending over the old secret, will fail.
* If you first update the Connections’ Posts service, it will start sending the new credential, and requests to the Posts service, which still has the old, will fail.
* We’ve already established that you can’t update them at the same time without incurring downtime.

So you have to be cleverer with your application design. For credential rotation, there’s a commonly used pattern. The key to this technique is that you’ll implement a phased approach to updating the secret; in one stage, the client service accepts more than one secret. [Figure 7.9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig09) depicts the following flow:

* Before you start the update, you have the same secret configured into both the client and the server.
* You update the Posts service app, *adding* a new credential to a *list* of accepted credentials. The authorization semantics are that any of the credentials in the list will allow access (in most cases, the list is limited to two credentials). Notice that the client is still using the old, but because that secret is still in the service list, the requests will succeed.
* You then update the Connections’ Posts app, replacing the old credential with the new. Because the new secret has already been configured into the Posts app, which now supports a list of credentials, requests from the new client instances will succeed.
* Finally, after the upgrade of the client is complete, the Posts service app can be updated to remove the old secret.

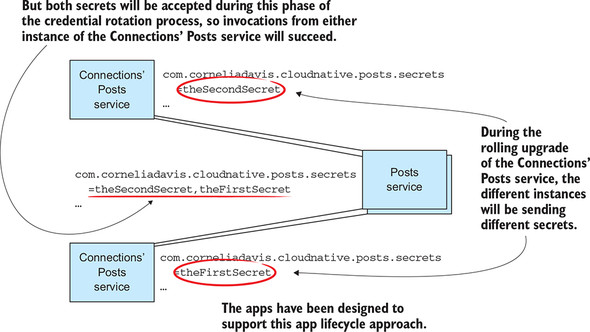
**Figure 7.9. This credential rotation pattern is an exemplar of the consideration that the software architect/developer must pay mind to. The goal is to ensure that the software has zero downtime during application lifecycle events that affect numerous apps.**



If you, like I, have been in the software business for some time, you might initially have a negative reaction to this operational flow. It involves a whole lot of redeploys, each of which comes through a disposal and creation of new app instances. But those old instincts need to be unlearned. Cloud-native apps are designed for this kind of ephemerality—remember, change is the rule, not the exception—and allow us to use this to make our software more robust and manageable.

Finally, I want to point out that this design also allows you to perform upgrades of each of the apps in a rolling style. In the first phase of your upgrade, the Connections’ Posts service is using the old secret, and both the old and new instances of the Posts service accept that credential even while the new instances already have added the updated secret. In the next phase, some instances of the Connections’ Posts service will be sending over the old secret while others will be sending the new; again, the Posts service accepts both. After the Connections’ Posts service is fully updated, it will be sending only the new credential, so during the second update of the Posts service, both old and new versions will succeed. This is depicted in [figure 7.10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig10).

**Figure 7.10. During the rolling upgrade of the Connections’ Posts service, the different instances will be sending different secrets. The apps have been designed to support this app lifecycle approach.**



Having had the architectural discussions, let’s now get some practical experience with rolling upgrades as well as the coordination of lifecycle events across apps. We’ll do so with proper credential rotation in our sample application—fixing the gotcha from the end of [chapter 6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06).

**7.4. Let’s see this in action: Credential rotation and app lifecycle**

To follow along with the code (I’ll have you run things in a moment), make sure you have the master branch checked out of our sample repository and change into the application lifecycle directory:

git checkout master

cd cloudnative-applifecycle

In this example, you’ll implement the credential rotation pattern depicted in [figure 7.9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig09). You’ll update the Posts service in a rolling fashion, then the Connections’ Posts service in a rolling fashion, and then finally the Posts service, again in a rolling fashion. Notice that each app has two instances deployed; in the starting state, the credentials configured in match across the instances of an app, and they’re coordinated across the Connections’ Posts service and the Connections and Posts services. I’ll remind you that each of the three apps is drawing its configuration, via the config server, from the single mycookbook.properties file that’s checked into GitHub. Note that this is a configuration file for the composition of the three apps that collectively form the My Cookbook software.

To follow the credential rotation pattern described in [section 7.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07lev1sec3), you need to update the Posts and the Connections implementations so that they can hold a list of valid secrets, any of which may be used to invoke the service. The storage of the credentials occurs within the singleton instance of the Utils class, the key parts of which are shown in the following code.

**Listing 7.1. Utils.java**

public class Utils implements

ApplicationContextAware, ApplicationListener<ApplicationEvent> {

// <lines omitted for brevity>

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

private String configuredSecretsIn;

private Set<String> configSecrets;

// <lines omitted for brevity>

@Override

public void onApplicationEvent(ApplicationEvent applicationEvent) {

if (applicationEvent instanceof ServletWebServerInitializedEvent) {

ServletWebServerInitializedEvent

servletWebServerInitializedEvent

= (ServletWebServerInitializedEvent) applicationEvent;

this.port = servletWebServerInitializedEvent...

} else **if (applicationEvent instanceof ApplicationPreparedEvent) {**

**configSecrets = new HashSet<>();**

**String secrets[] = configuredSecretsIn.split(",");**

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

**configSecrets.add(secrets[i].trim());**

**logger.info(ipTag()**

**+ "Posts Service initialized with secret(s): "**

**+ configuredSecretsIn);**

}

}

public String ipTag() { return "[" + ip + ":" + port +"] "; }

**public boolean isValidSecret(String secret) {**

**return configSecrets.contains(secret);**

}

// The following method is included only to facilitate some

// logging that wouldn't exist in production.

public String validSecrets() {

String result = "";

for (String s : configSecrets)

result += s + ",";

return result;

}

}

First, I draw your attention to the onApplicationEvent method, specifically the case where you handle the ApplicationPreparedEvent. Without going into the details of the rich set of application lifecycle events implemented through the Spring Framework, know that the ApplicationPreparedEvent is triggered when the application has been fully initialized. The configuredSecretsIn string has been initialized, via the config server, from the com.corneliadavis.cloudnative.posts.secrets property. What you’re doing here is parsing it and loading the values in a Set so that the test for validity is trivial, as seen in the definition of the isValidSecret method.

Now, looking at the implementation of the Posts controller in the following listing, you can see that you simply need to check whether the secret passed in is valid before proceeding with processing. When the secret is invalid, you print out both the secret that was passed in as well as the valid one or ones that are configured into the app. In a real application, you wouldn’t print these values in the log, but doing so here is helpful for experimenting with these concepts.

**Listing 7.2. Method from PostsController.java**

@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 (utils.isValidSecret(secret)) {**

logger.info(utils.ipTag()

+ "Accessing posts using secret " + secret);

if (userIds == null) {

logger.info(utils.ipTag() + "getting all posts");

posts = postRepository.findAll();

return posts;

} else {

ArrayList<Post> postsForUsers = new ArrayList<Post>();

String userId[] = userIds.split(",");

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

logger.info(utils.ipTag()

+ "getting posts for userId " + userId[i]);

posts = postRepository.findByUserId(

Long.parseLong(userId[i]));

posts.forEach(post -> postsForUsers.add(post));

}

return postsForUsers;

}

} else {

logger.info(utils.ipTag()

+ "Attempt to access Post service with secret " + secret

+ " (expecting one of " + utils.validSecrets() + ")");

response.setStatus(401);

return null;

}

}

This covers the two key parts of the service side of the pattern: (1) secrets are configured on application startup, and (2) to support the zero-downtime credential rotation pattern, the service allows more than one valid secret at a time. I’ve presented only the code from the Posts service here, but the structure is identical in the Connections service.

Let’s now look at the client side, in the implementation of the Connections’ Posts app in [listing 7.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07ex03). The basic structure is similar to that of the Posts and Connections apps. You use the Utils class to process the secrets configuration. Then in the app controller, in which the calls are made to the Posts and Connections services, you access the values through the singleton utils object. Looking first at the controller code, you can see that it’s straightforward. You access the posts or connections secret that’s configured into the app by asking for the value from the utils object, and send it on the query string.

**Listing 7.3. Method from ConnectionsPostsController.java**

@RequestMapping(method = RequestMethod.GET, value="/Connections' Posts")

public Iterable<PostSummary> getByUsername(

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

HttpServletResponse response) {

// <lines omitted for brevity>

// get connections

**String secretQueryParam** ***1***

= "?secret=" + utils.getConnectionsSecret();

ResponseEntity<ConnectionResult[]> respConns

= restTemplate.getForEntity(

connectionsUrl + username + secretQueryParam,

ConnectionResult[].class);

// <lines omitted for brevity>

**secretQueryParam = "&secret=" + utils.getPostsSecret();** ***2***

// get posts for those connections

ResponseEntity<PostResult[]> respPosts

= restTemplate.getForEntity(

postsUrl + ids + secretQueryParam,

PostResult[].class);

// <lines omitted for brevity>

}

* ***1* Accesses the connections secret configured into the app**
* ***2* Accesses the posts secret configured into the app**

Although most of the Utils.class here is similar to the same class in the Posts app, it has one nuance. Notice that although the com.corneliadavis.cloudnative.connections.secrets and com.corneliadavis.cloudnative.posts.secrets properties are drawn from the mycookbook.properties file and may each contain a list of secrets, in the Connections’ Posts service, you need only the most recent one. You’ll establish an operational practice, and this must be covered in the documentation for the Connections’ Posts service, where the newest secret is always in the first position in the list. As you can see, you store in the state of the singleton utils object only one secret for each of the Posts and Connections services. To make that clear, the property that’s configured into the app is the single secret, even though the property file is carrying more than one. Again, the following code includes logging output that isn’t suitable for a production system but is a valuable teaching tool.

**Listing 7.4. Method from the Utils class of the ConnectionsPosts service**

@Override

public void onApplicationEvent(ApplicationEvent applicationEvent) {

if (applicationEvent instanceof ServletWebServerInitializedEvent) {

ServletWebServerInitializedEvent

servletWebServerInitializedEvent

= (ServletWebServerInitializedEvent) applicationEvent;

this.port = servletWebServerInitializedEvent...;

} else if (applicationEvent instanceof ApplicationPreparedEvent) {

connectionsSecret = connectionsSecretsIn.split(",")[0];

postsSecret = postsSecretsIn.split(",")[0];

logger.info(ipTag()

+ "Connection Posts Service initialized with Post secret: "

+ postsSecret + " and Connections secret: "

+ connectionsSecret);

}

}

Okay, so let’s put this to work.

**Setting up**

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

* Maven
* Git
* Java 1.8 (optional—needed only if you plan to build the container images yourself)
* Docker (optional—needed only if you plan to build the container images yourself)
* 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)**

Because I will have you deploy the apps into Kubernetes, and in order to do so Docker images are required, I’ve prebuilt those images and made them available in Docker Hub. Therefore, building the microservices from source isn’t necessary.

If you haven’t already done so, check out the master branch; and from the cloudnative-abundantsunshine directory, change into the cloudnative-applifecycle directory:

git checkout master

cd cloudnative-applifecycle

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 don’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 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. No Kubernetes services are deleted—something that will likely save you the step of configuring URLs into each of the app deployment manifests.

Verify that your environment is clean with the following command:

$ kubectl get all

NAME READY STATUS RESTARTS AGE

pod/mysql-75d7b44cd6-s8zcr 1/1 Running 0 70m

pod/redis-6bb75866cd-kf99k 1/1 Running 0 72m

pod/sccs-787888bfc-x9p2m 1/1 Running 0 73m

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

service/connections-svc NodePort 10.103.148.230 <none> 80:30955/TCP

service/connectionsposts-svc NodePort 10.104.253.33 <none> 80:31742/TCP

service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP

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

service/posts-svc NodePort 10.110.192.11 <none> 80:32119/TCP

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

service/sccs-svc NodePort 10.107.16.107 <none> 8888:30455/TCP

NAME READY UP-TO-DATE AVAILABLE AGE

deployment.apps/mysql 1/1 1 1 70m

deployment.apps/redis 1/1 1 1 72m

deployment.apps/sccs 1/1 1 1 73m

NAME DESIRED CURRENT READY AGE

replicaset.apps/mysql-75d7b44cd6 1 1 1 70m

replicaset.apps/redis-6bb75866cd 1 1 1 72m

replicaset.apps/sccs-787888bfc 1 1 1 73m

Note that this leaves MySQL and Redis running. If you’ve cleared out Redis and MySQL, deploy each of these with the following commands, and create the cookbook database with the next two commands:

kubectl create -f mysql-deployment.yaml

kubectl create -f redis-deployment.yaml

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

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

mysql> create database cookbook;

After the following steps are completed, the deployment will be as depicted in the first stage of [figure 7.9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig09). You’ll have two each of the Connections and Posts services, and two instances of the Connections’ Posts service. To achieve this topology, for now, you still have to edit deployment manifests. These steps, summarized here, are detailed in [chapter 5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05):

1. Configure the Connections and Posts services with the following values:

|  |  |
| --- | --- |
| MySQL URL | minikube service mysql-svc --format "jdbc:mysql://{{.IP}}:{{.Port}}/cookbook" |
| SCCS URL | Minikube service sccs-svc --format "http://{{.IP}}:{{.Port}}" |

1. Deploy the Connections service:

kubectl apply -f cookbook-deployment-connections.yaml

1. Deploy the Posts service:

kubectl apply -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}}" |
| SCCS URL | Minikube service sccs-svc --format "http://{{.IP}}:{{.Port}}" |

1. Deploy the Connections’ Posts service:

kubectl apply -f cookbook-deployment-connectionsposts.yaml

You can test out your deployment by issuing curl commands against any of the microservices. The Posts and Connections services require that the secret be passed in the query string, and the Connections’ Posts service requires login before content will be served. The commands are as follows:

curl -i $(minikube service --url connections-svc)/connections?secret=anyval

curl -i $(minikube service --url connections-svc)/users?secret=anyvalue

curl -i $(minikube service --url posts-svc)/posts?secret=foobar

curl -X POST -i -c cookie \

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

curl -b cookie \

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

|  |
| --- |
|  |

**Tip**

A simple way to look up the secrets configured into the software is to curl the Posts and Connections services with ?secret=anyvalue, and look at the logs; recall that you’re printing the accepted value to the log.

|  |
| --- |
|  |

Let’s now perform the first part of the upgrade process. You’ll do this by updating the config file for the software deployment and then rolling the Posts service. Recall that you’re storing the config in GitHub and are using SCCS (Spring Cloud Config Server) to deliver that configuration to the application instances. Edit the mycookbook.properties file in the cloud-native-config repo by updating the following line:

com.corneliadavis.cloudnative.posts.secrets=originalSecret

Add a new secret to the front of the list:

com.corneliadavis.cloudnative.posts.secrets=newSecret,originalSecret

You must commit and push these changes to your GitHub repository. You now want to perform a rolling upgrade on the Posts service, and you’ll use the kubectl apply command to do so. Because this configuration change is done via SCCS, and that property change is therefore not seen by Kubernetes, you must do something to cause Kubernetes to cycle the application instances. The trick I’m using is having an env variable in the deployment YAML that you can bump when you want Kubernetes to roll through your app instances. You simply must change the value of VERSIONING\_TRIGGER to something new. This is done in the cookbook-deployment-posts.yaml file:

- name: VERSIONING\_TRIGGER

value: "1" ***1***

* ***1* Updates the value to something new—I typically just increment the number**

Before running the command to initiate the rolling upgrade, in a terminal window, set a watch on the pods currently running in your environment:

watch kubectl get pods

Now start the rolling upgrade with the following command:

kubectl apply -f cookbook-deployment-posts.yaml

In the watch window, you’ll see new instances of the Posts service being created and old ones being terminated and then eventually discarded. This is performing the upgrade process that’s depicted in [figure 7.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig06). (I don’t know about you, but watching this type of application lifecycle automation the first time was pretty darn cool for me!) Now let’s curl the Posts service, experimenting with various secrets, starting with a bad secret:

curl -i $(minikube service --url posts-svc)/posts?secret=aBadSecret

Have a look into the logs for the two pod instances. In one, you’ll find a message that reads something like the following:

Attempt to access Post service with secret aBadSecret (expecting one of newSecret,oldSecret,)

I’ll once again remind you that you should never send secrets out to log files, but you’re doing so here because it’s helpful in exercising our examples. This message shows you that the new configuration has been applied to the Posts service. You can now invoke the Posts service with either of the secrets and receive a response.

The Connections’ Posts service is still using the oldSecret, so let’s update it now.

You’ve already updated the configuration in GitHub, so you must have Kubernetes do only a rolling upgrade. You’ll do so by editing the cookbook-deployment-connectionsposts.yaml file, bumping the VERSIONING\_TRIGGER just as you’d done with the Posts service:

- name: VERSIONING\_TRIGGER

value: "1" ***1***

* ***1* Updates the value to something new—I typically just increment the number.**

Now issue the following command to initiate the rolling upgrade:

kubectl apply -f cookbook-deployment-connectionsposts.yaml

Both after and throughout the upgrade, you can invoke the Connections’ Posts service. Depending on which instance is routed to, either the old or the new secret will be sent to the Posts service. Again, because of the pattern you’ve implemented, everything will function as expected.

Finally, after the Connections’ Posts service has been fully upgraded, you can remove the old secret from the configuration (don’t forget to commit and push to GitHub), update the deployment YAML to bump the VERSIONING\_TRIGGER, and execute the following command:

kubectl apply -f cookbook-deployment-posts.yaml

After Kubernetes has completed the rolling upgrade for the Posts service, the credential rotation will be complete. You just upgraded your software with *absolutely no downtime* and were able to use a rolling upgrade (application lifecycle automation built into Kubernetes) because your software was specifically designed to support it.

**7.5. Dealing with ephemeral runtime environments**

It should be clear by now that one of the main characteristics of cloud-native apps is that they’re constantly disposed of and created anew. You just experienced that in the credential rotation example, in which you rolled through all instances of both the Posts and the Connections apps twice, first to add a new secret and then later to remove the old. Again, I understand your possible aversion to this practice; it goes against the deeply held belief that stable is good and change is bad, but in the cloud, change is inevitable and can even bring a new type of stability.

Like many other cloud-native patterns and practices, however, embracing this new paradigm brings cascading effects—new concerns for you, the application developer/architect. The main one that I want to talk about right now is the impact that the ephemerality of these runtime contexts has on the manageability of our apps. Under that heading, I want to talk about two topics: (1) troubleshooting and (2) repeatability.

Let me talk about the latter first, only briefly because this is a point I’ve made several times already. The main point is that an application deployment (the specific bits that are deployed, the way that it’s running, and the context it’s running in) should be 100% reproducible. Cloud-native application platforms provide many features that support this goal, so learning the best practices for using such platforms is important. But, because I’m about to say a bit more about troubleshooting, let me point out that the best of these platforms, when properly configured, will not allow you to ssh into the runtime environment. Why? Simply put, to make sure that you can’t create a running instance of your app that’s not reproducible.

Let’s suppose you were allowed to ssh into the container in which your app was running. While there, you do a few things that you think are only for troubleshooting, things that aren’t part of the production configuration of the app; for example, you might open ports to allow monitoring tools in, or you might install additional packages. You fix the problem, and because you’re a responsible engineer, you go back into the source or configuration and reflect the changes there. You leave that container running—why not; it’s working fine. But the container is now a snowflake; although you think you’ve reflected all of the “fixes” back in the app and config source, it’s possible that only this unique container has just the right config to make it work. Later, when the instance is replaced (for any number of reasons), the new app instance may (or may not) bring back old problems. Restricting operations that allow for snowflakes to be created greatly enhances manageability of software deployments.

Which brings me to my second point. If you can’t ssh into an instance, how will you figure out what’s going on when your apps aren’t working as expected? Short answer: logging and metrics. As a software developer, you play a crucial role in ensuring that the logs and metrics feeds contain information sufficient to diagnose problems. And here comes another twist in this plot, one that’s specifically related to application lifecycle: it’s possible that by the time you’re troubleshooting, the app instance with which the trouble arose is no longer in existence. If the app crashed, the container is likely to have been discarded, a new instance taking its place.

The practice of disposing of problematic app instances is so common in cloud-native platforms that the Twelve-Factor App guidance includes factor #11, “Treat logs as event streams.” The point of this particular bit of advice is to establish a contract, or API, for the “publication” of log data so that platforms running the cloud-native apps can take care of making that data available, even while *it* maintains control of the application lifecycle. In a story that rings familiar, because you’ve seen similar arguments when I described using env variables for configuration data, stdout and stderr streams exist in pretty much every runtime environment, and every programming language/framework supports writing to those streams. And it’s a simple standard; there’s no need for locating log files stored in bespoke directories or even doing log file rotations. All of the log data is streamed out and can then be handled by the platform itself.

|  |
| --- |
|  |

**Tip**

So this brings us to the punch line: write your logs to stdout and stderr.

|  |
| --- |
|  |

This doesn’t mean that you should simply have System.out calls throughout your code. There’s a reason that logging frameworks such as Apache Log4j or Logback were invented. But these may be configured, and often by default are, to direct any logging output to stdout and stderr.

The metrics story isn’t as straightforward, mainly because metrics data is inherently more structured than log data and varies widely across different applications and platforms. There isn’t one set of metrics data that platforms and tooling can be built around to provide all of the needed insight. That said, the topic is important, and some de facto standard practices and tools have begun to emerge; [chapter 11](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11) covers some of these.

Hopefully, by now it’s clear that although I have a whole chapter on the topic, I briefly include troubleshooting here because of the impact that the application lifecycle has on it. Just as it is with core application logic, how you handle observability data must also account for the inherently more dynamic nature of the application lifecycle. I emphasized a word a few paragraphs ago (“it”) that stresses the point: in the cloud-native world, humans don’t control the application lifecycle; systems do (and in the best cases, intelligent systems such as Kubernetes or Cloud Foundry). Systems demand much stronger contracts or APIs, and you’re responsible for ensuring that your applications meet those contracts.

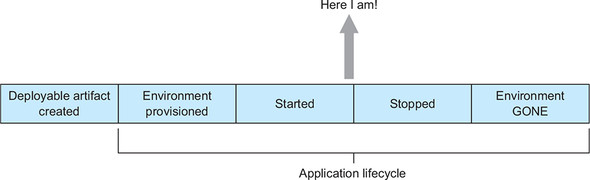
**7.6. Visibility of app lifecycle state**

In [section 7.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07lev1sec3), I talked about application lifecycle concerns across different but related apps. Specifically, you looked at how an operational need is addressed through a series of lifecycle events. There’s another element to that cross-app relationship that I want to talk about now, and it arises when one app needs to know about app lifecycle events of another.

For example, as you well know (and by this time are completely irritated by), when the Posts service is re-created for any reason (and I do mean the Kubernetes *service*, not the pods), you need to redeploy the Connections’ Posts service with a configuration that includes the new URL to that service. If instead you could have the latter app automatically updated upon such a change in the former, your experience would be far better. (Bear with me just a bit longer; you’re only a few pages away from fixing this!)

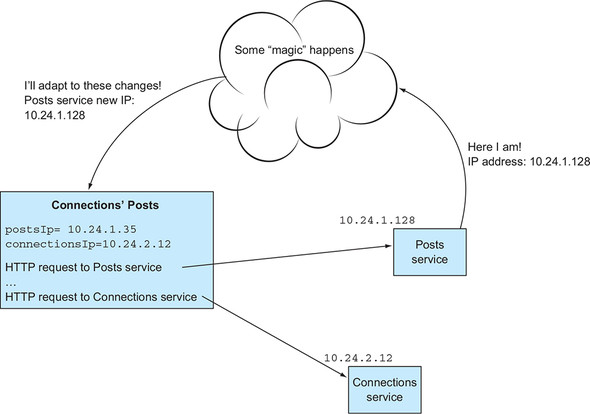
From an application lifecycle perspective, this means that the Connections’ Posts service depends on knowing when lifecycle events happen to the Posts service. In the simplest sense, the Posts service is responsible for making its lifecycle state available. For example, [figure 7.11](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig11) shows an application lifecycle event being broadcast when an app is started.

**Figure 7.11. Application startup is an important event that many other components in the cloud-native software will be interested in.**



Taking this one step further, [figure 7.12](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig12) shows a dependency on this lifecycle event. In this diagram, initially the Connections’ Posts service was reaching the Posts service at IP address 10.24.1.35, but when a new Posts service is started at IP address 10.24.1.128, it’s responsible for broadcasting that information, and the Connections’ Posts service needs to be updated with that IP address.

**Figure 7.12. Apps have the responsibility for broadcasting lifecycle events because other components will be affected. Those other components likewise have the responsibility for adapting to those changes.**



The exact mechanics of how this broadcast from Posts is handled, where it’s published, and how interested parties can pick up the pertinent information isn’t germane at this point in our discussion. That’s represented in [figure 7.12](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig12) with a nebulous “magic” blob, and we’ll come back to that later. Until now, you’ve been implementing that entire protocol—looking up the new IP and port with the minikube service list command and editing YAML files. But this needs to be automated, and the main point should be clear: app lifecycle events on Posts affect other parts of your software, and you must specifically account for this.

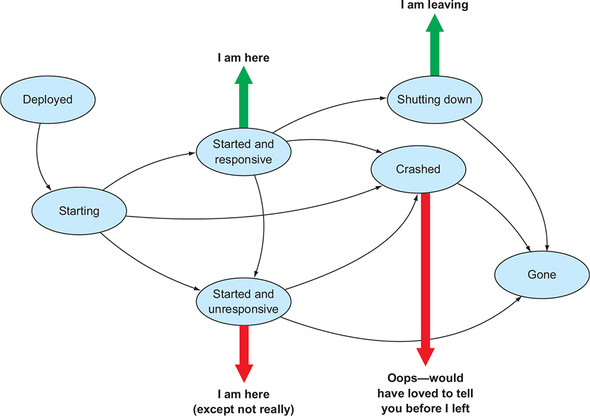
When talking about the responsibilities on either side of the relationship, I’ve been a bit vague. I admit that I’ve implied that you, the developer, are responsible for broadcasting or consuming the events. Although the full answer is a bit complicated, and you’ll study this in more detail as you progress through the text, the short one is this: if you’re using a cloud-native application platform, it will generally take care of these concerns for you. For the moment, I just want you to have an appreciation for this dependency.

Although you may understand the problem, I hope that you’re reading what I’ve written thus far with a healthy dose of skepticism. In [figure 7.12](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig12), you’re looking at a bunch of coordination across many components that are highly distributed over an unreliable network, yet I’ve suggested that proper operations are dependent on broadcasted lifecycle events reaching all interested parties.

Here’s the truth: the broadcasted event as I’ve described it here should be thought of as an optimization. The reason I started with an optimization is that it illustrates the challenge beautifully. You have a whole bunch of pieces that need to act in concert to make modern software work. In an abstract sense, it’s simple: an app lifecycle event needs to be received by those that will be affected by the state change. But the reality is more complicated: all this needs to happen in the face of a multitude of failure scenarios. App lifecycle events won’t always be generated; when they are, they’ll sometimes be lost; and even when not lost, they sometimes won’t be recognized or operated on by the components that should do so.

To understand this, let me present in [figure 7.13](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig13) a far less trivialized application lifecycle than the one shown in [figures 7.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig01) and [7.11](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig11). In addition to modeling possible failure scenarios, [figure 7.13](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig13) also shows a richer set of possible transitions between the application lifecycle states. After an app is deployed, the startup may succeed (Started and Responsive), or it may fail—the app might be running but not running well (Started and Unresponsive), or it may outright crash. An app that started successfully and was operable for some time may also crash or, worse, may stay running but be unresponsive. Whether gracefully shut down or not so gracefully, at some point the app and its environment will be gone.

**Figure 7.13. Application lifecycle states and transitions between them. In the happy path, when apps are started and stopped, there’s an opportunity to broadcast app lifecycle change events. But when something goes awry, state-change broadcasts will be incorrect or may never even be generated.**

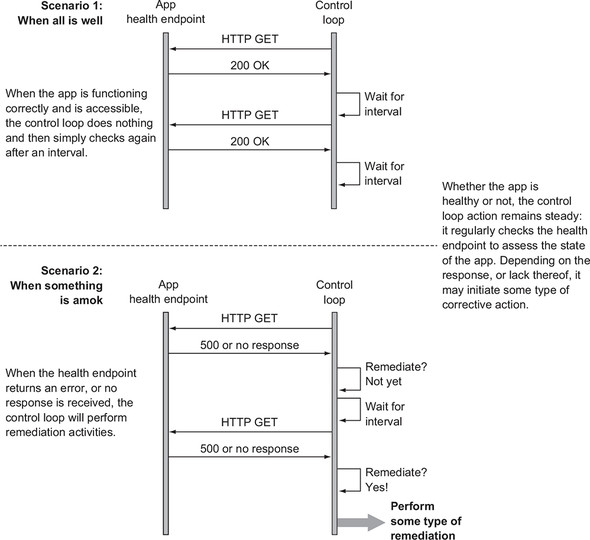


Expanding our scenario a bit, realize that the Connections’ Posts service needs to know not only when new Posts services are started, but also when they go away. This diagram allows you to study both cases. Note that I’ve added a few thick arrows with annotations. These are equivalent to the single arrow from [figure 7.11](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig11) but updated to the more complex lifecycle model. The annotations at the top show the broadcast of “happy path” events: when the app successfully starts, it announces its existence, and when it’s being gracefully shut down, it can first let the world know it’s going away. But what happens when, as shown in the downward-facing arrows, the startup isn’t so great, or when the app crashes without warning, robbing it of the opportunity to let the world know of its impending doom?

Without getting overly dramatic, this is part of the magic of cloud-native. Apps are built not only to work in the happy path, but also to continue working, or self-heal when things go awry. To handle this particular circumstance, the answer is health endpoints combined with health checks/responders, and you, as the developer, play a critical role in the coordination among these things. The concept is simple: a health endpoint presents data representing the state of an app, and health checks/responders implement some type of control loop that interrogates, and acts upon that status. This addresses the aforementioned deficiencies of lost lifecycle events with redundancy. The interrogating control loop does so continuously (say, every 10 seconds), so if one or even a couple of communication attempts are unsuccessful because of momentary lapses, the next will succeed and the system continues functioning. This is another example of what I’ve already referred to as *eventual consistency*.

[Figure 7.14](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig14) presents two sequence diagrams depicting this basic pattern. Whenever asked, the app responds with its current lifecycle state, and the control loop regularly asks and responds appropriately. The first sequence diagram shows what happens when the system is functioning properly: the control loop checks the app health endpoint, and upon getting a response indicating that everything is fine, simply waits for the next interval and asks again. The second sequence diagram shows what happens when the app is running but unresponsive, or the health endpoint returns an error. One failure won’t necessarily trigger remediation actions, but repeated ones will (you’ll see this in action in a moment).

**Figure 7.14. Control loops play an important role in cloud-native systems, providing redundancy that compensates for “glitches” in the system.**



I’ve made the point that as the developer, you play a critical role in making this whole thing work, but I want to be clear here that I don’t consider you responsible for both sides of that protocol. The control loop will come from the cloud-native platform—Kubernetes, Cloud Foundry, or similar.**[**[**3**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fn3)**]** You’re responsible for the app health endpoints, and for building apps that can be fixed—by creating new instances—when they’re experiencing problems. Right now, we’ll focus on the endpoint. The remediation stuff—well, that’s what this book is all about. What you’re responsible for is implementing a health endpoint that accurately reflects the state of the app. You may want to ensure that any connectivity to persistent services such as databases are functioning, for example.

***3***

*If your application platform doesn’t have these types of control loops, it isn’t a cloud-native platform.*

The system as a whole is built to be resilient to glitches—network outages, app failures, and more—and the control loop provides needed redundancy. And now it’s clear why the earlier broadcast-based design should be thought of as an optimization. Rather than waiting for the next control loop cycle to broadcast the status, the lifecycle state-change event immediately initiates the broadcast. If, for any reason, that broadcast event is lost, the next time the control loop fires, there will be another.

**7.6.1. Let’s see this in action: Health endpoints and probes**

Okay, enough abstract. Let’s make this real with an example. If you ran the samples from earlier in the chapter, you’re already set up. The code you’ve been running already includes what I want to demonstrate here.

Implementing the polling approach I described previously, you’ll add a /healthz endpoint into each of the services. The code, slightly contrived, simply checks a Boolean class member. When it’s set to true (the default), it returns a success status code, and when it’s set to false, the application sleeps for a long time, effectively rendering it unresponsive. You have an app that’s in the Started and Unresponsive state, as follows.

**Listing 7.5. Method from Posts\_Controller.java**

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

public void healthCheck(HttpServletResponse response)

throws InterruptedException {

if (this.isHealthy) response.setStatus(200);

else Thread.sleep(400000);

}

The other half of the demonstration, the control loop that continuously polls the health endpoint retrieving the data and acting upon it as appropriate, comes from Kubernetes. There are some new lines in the Kubernetes deployment manifest, as follows.

**Listing 7.6. Excerpt from cookbook-deployment-posts.yaml**

livenessProbe:

httpGet:

path: /healthz

port: 8080

initialDelaySeconds: 60

periodSeconds: 5

These lines configure that control loop, the liveness probe, so that every 5 seconds, Kubernetes will send an HTTP GET request to the /healthz endpoint of this pod (Kubernetes also supports TCP liveness probes). When a new pod is started, Kubernetes will wait 60 seconds before the control loop is initiated, offering time for the service to initialize before it expects the health endpoint to accurately reflect the app status. If at any time Kubernetes receives an error status code, or no response, it will restart the container. Let’s see this in action.

**Seeing this in action**

If you don’t have the software running, follow the steps in the “Running the apps” portion of [section 7.5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07lev1sec5). Please stream the logs from each of your two Posts service pods in two side-by-side terminal windows. With my two current pods, I’d use the following commands in those two separate windows:

$ kubectl logs -f posts-439493379-0w7hx

$ kubectl logs -f posts-439493379-hfzt1

To verify that all is well, you can curl any of the Posts endpoints, including the /healthz endpoint. You will, of course, see activity in the logs:

$ curl $(minikube service --url posts-svc)/posts?secret=newSecret

[

{

"id": 7,

"date": "2019-02-17T05:42:51.000+0000",

"userId": 2,

"title": "Chicken Pho",

"body": "This is my attempt to re-create what I ate in Vietnam..."

},

{

"id": 9,

"date": "2019-02-17T05:42:51.000+0000",

"userId": 1,

"title": "Whole Orange Cake",

"body": "That's right, you blend up whole oranges, rind and all..."

},

{

"id": 10,

"date": "2019-02-17T05:42:51.000+0000",

"userId": 1,

"title": "German Dumplings (Kloesse)",

"body": "Russet potatoes, flour (gluten free!) and more..."

},

{

"id": 11,

"date": "2019-02-17T05:42:51.000+0000",

"userId": 3,

"title": "French Press Lattes",

"body": "We've figured out how to make these dairy free, but just as

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

}

]

$ curl -i $(minikube service --url posts-svc)/healthz

HTTP/1.1 200

X-Application-Context: mycookbook

Content-Length: 0

Date: S un, 17 Feb 2019 06:13:34 GMT

Now, to place one of your Posts service instances into the Started and Unresponsive state, issue the following curl command:

$ curl -i -X POST $(minikube service --url posts-svc)/infect

Keep an eye on your log streams. Within 5 to 10 seconds, you’ll see lines such as the following emitted to one of your two log streams, and the log-streaming session will terminate; the log streaming terminates when the container to which it is connected goes away. That infect endpoint simply flips the isHealthy Boolean in the Posts service:

... ConfigServletWebServerApplicationContext : Closing

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg org.springframework.boot.web.servlet.context.AnnotationConfigServletWeb

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg ServerApplicationContext@27c20538: startup date [Sun Feb 17 06:03:15 GMT

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg 2019]; parent: org.springframework.context.annotation.AnnotationConfig

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg ApplicationContext@2fc14f68

... o.s.j.e.a.AnnotationMBeanExporter : Unregistering JMX-exposed beans on

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

... o.s.j.e.a.AnnotationMBeanExporter : Unregistering JMX-exposed beans

... j.LocalContainerEntityManagerFactoryBean : Closing JPA

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg EntityManagerFactory for persistence unit 'default'

But discarding the old container isn’t all that the control loop in Kubernetes did. The loop also started a new instance of the app in a new container. If you start streaming the logs from that new container, which you can do by issuing the kubectl logs command again (note that because Kubernetes restarts only the container and not the pod, the pod name will be unchanged), you’ll see that the app once again is up and running:

$ kubectl logs -f posts-5876ffd568-gr5bf

... s.c.a.AnnotationConfigApplicationContext : Refreshing org.

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg springframework.context.annotation.AnnotationConfigApplicationContext

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg @2fc14f68: startup date [Sun Feb 17 06:15:30 GMT 2019]; root of context

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

... trationDelegate$BeanPostProcessorChecker : Bean 'configuration

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg PropertiesRebinderAutoConfiguration' of type [org.springframework

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

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg $$EnhancerBySpringCGLIB$$3cd10333] is not eligible for getting

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg processed by all BeanPostProcessors (for example: not eligible for

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg auto-proxying)

. \_\_\_\_ \_ \_\_ \_ \_

/\\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \ \ \ \

( ( )\\_\_\_ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )

' |\_\_\_\_| .\_\_|\_| |\_|\_| |\_\\_\_, | / / / /

=========|\_|==============|\_\_\_/=/\_/\_/\_/

:: Spring Boot :: (v2.0.6.RELEASE)

...

... o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s):

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg 8080 (http) with context path ''

... c.c.c.config.CloudnativeApplication : Started

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg CloudnativeApplication in 15.74 seconds (JVM running for 16.605)

This example demonstrates that making the state of an app available allows a system such as Kubernetes to appropriately respond when the inevitable changes happen to an app. As a developer of cloud-native applications, it’s your responsibility to produce the appropriate implementation for the model being used in your cloud-native application runtime.

What I’ve described in this section is a polling approach, but I’d like to point out that control loops that broadcast heartbeats are another way of implementing this pattern. Using this technique, components are continually broadcasting their lifecycle state with one control loop, and entities that have an interest in this app’s status will be listening for these events and responding appropriately. I’ll remind you of the discussion of [chapter 4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04): what I’m describing here is an event-driven pattern. As the architect/developer, you must understand the architectural patterns of the software as a whole and design/implement appropriately. The key with either approach is that control loops provide the redundancy that compensates for the uncertainty inherent in distributed systems.

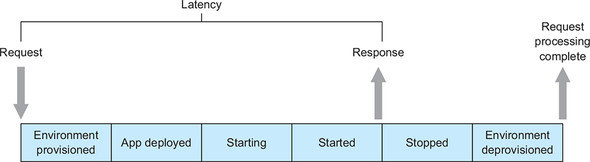
**7.7. Serverless**

This isn’t a book, or even a chapter, on serverless computing. But especially here in this chapter on the application lifecycle, having a brief look allows you to understand certain elements of cloud-native software more deeply. As is usually pointed out within the first few moments of a discussion on serverless, the name is a bit of a misnomer because the functions being executed in this style are absolutely running on servers. It’s just that the developer is completely unconcerned with these details. Instead, the serverless system takes care of it all.

Developer productivity is certainly one of the goals of this computing style, but so is operational efficiency and economics, because most serverless systems charge for use only for the time that a function is executing. Regardless of the specific goals, the thing that I want to focus on is the serverless platform; that platform is very much a cloud-native platform. Let’s start by looking at the model in the most basic way.

At the most basic level, serverless computing has an application running only when it’s actively processing to produce a response to an event. If you view that from an application lifecycle perspective, it’s only when a request comes in that a runtime environment is provisioned, the app is deployed and started, and the request processing is done. And after it has run, the runtime environment is disposed of; see [figure 7.15](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig15).

**Figure 7.15. All lifecycle stages are passed through for a single-function invocation.**

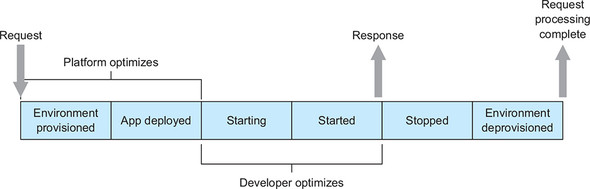


This application lifecycle isn’t unfamiliar. What is unconventional is that all the stages, from provisioning through disposal, happen with each invocation. What I like best about serverless computing is that this extreme serves to amplify the patterns for cloud-native software. For example, if the runtime environment is entirely re-created for each invocation, the apps can never, ever depend on internal state from previous invocations. Buh-bye, sticky sessions, for example.

But there’s more that I want to draw your attention to that’s specific to cloud-native apps running in a serverless setting. This has to do with efficiency and latency. Looking at [figure 7.15](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig15), it’s obvious that a lot needs to happen between a request for processing and the completion thereof. How can all of this happen while meeting requirements for responsiveness? The short of it is through optimizations, some of which you’re responsible for.

[Figure 7.16](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07fig16) again shows our serverless lifecycle stages, but this time with a few notations overlaid. The early stages of the app lifecycle are entirely handled by the system, and indeed serverless platforms specifically focus on, among other things, making environment provisioning and app deployment fast. Most are built on containers and use formats for the deployable artifacts that allow for rapid deployment. As a developer, you’re responsible for, and have control over, only the application startup and its actual execution. You must focus on making that happen as quickly as is necessary.

**Figure 7.16. Serverless computing requires a platform that optimizes the early stages, and the developer is responsible for optimizing the starting and execution of the app.**



Let me be blunt: not all of your processing is best accomplished in a serverless setting. If you have a workload that takes little time to run and is constantly being requested, particularly if the startup “cost” is greater than that of executing the function, you’re likely better off starting up one or more instances of the app and having requests served from those already running instances. If, on the other hand, you have processing that’s less frequently run and takes far more time to run than it takes to provision, deploy the app, and start it up, then the serverless paradigm might be ideal.

If you’re building an app that will run in a serverless context, you must pay special attention to the startup costs and ensure that they don’t dwarf those of the function execution. You have several levers to control that. First, the programming language you use has a direct impact. Starting a JVM can take tens of seconds, which would be awful if the code execution time was in the milliseconds. And second, even after you choose a language, be sure to minimize what you need to load into the runtime to support your app functionality. For example, don’t include dependencies in your code that are never used; you’ll pay the price of slow startup for no reason.

Now, I do want to point out that most of the serverless platforms in the market today implement optimizations to lessen the impact of the app lifecycle in its purest form. For example, app environments are often preserved and reused for requests that come in relatively close in time to one another, but it’s clearer here than in other platforms that no app should use such features. Again, that’s one of the things I like most about serverless: that it makes clear the need for cloud-native patterns.

**Summary**

* In a cloud-native setting, you must think about an app’s lifecycle and treat it as a single logical entity even while each app instance has its own independent lifecycle.
* You must also pay careful attention to how app lifecycle events affect other apps that form the broader piece of software.
* Only if multiple instances of an app can tolerate different configurations running at the same time can a rolling upgrade be used. Otherwise, a blue/green deployment must be used. Both can be done with zero downtime.
* A carefully constructed credential rotation pattern can be done with rolling upgrades.
* Intentionally replacing app instances can serve such patterns, so do that. Let go of the biases of the past.
* Application logs should be sent to stdout and stderr, where most cloud-native platforms will process them.
* Application state must be made available so that system health can be maintained and dependent apps can appropriately adapt to changes.
* Serverless is an extreme form of cloud-native processing that uses most of the patterns covered in this text.