**Chapter 5. App redundancy: Scale-out and statelessness**

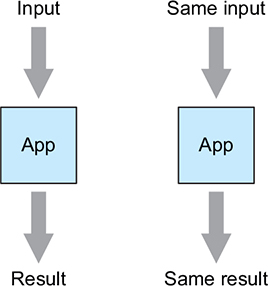
*This chapter covers*

* Scale-out as a central tenet of cloud-native apps
* Pitfalls of stateful apps in cloud-native software
* What it means for an app to be stateless
* Stateful services and how they’re used by stateless apps
* Why sticky sessions shouldn’t be used

The title says “Scale-out,” but really, it’s not just a matter of scaling. There are many reasons for what is probably the core tenet of cloud-native software: redundancy. Whether your application components are micro or macro, whether they can be configured via environment variables or have config baked into property files, whether they fully implement fallback behaviors or not, a key to change tolerance is that there is no single point of failure. Apps always have multiple instances deployed.

But then because any one of those multiple instances of the app can satisfy a request (and you’ll see shortly why *that* is essential), you need those multiple instances to behave as a single logical entity. [Figure 5.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05fig01) depicts this clearly.

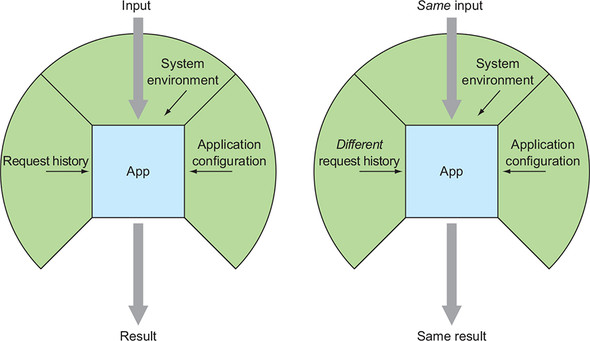
**Figure 5.1. Given the same input, the result produced by an app instance must be the same, regardless of whether there are one, two, or one hundred instances of the app.**



Although this seems simple enough, it can be a bit tricky, because the context in which each app instance is running will vary. The input coming in with an invocation of the app (I’m not implying any particular invocation pattern here—could be request/response or event-driven) isn’t the only thing that affects the result. Each app instance will be running in its own container—which could be a JVM, a host (virtual or physical), or a Docker (or similar) container—and environment values in that context can influence the execution of the app. Application configuration values will be supplied to each app instance as well. And, of primary interest to us in this chapter, the user’s history of interaction with the app also has a marked effect.

[Figure 5.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05fig02) wraps context around the app instances and shows the challenges to achieving parity across the external influences on the app.

**Figure 5.2. The cloud-native app must ensure consistent results in spite of possible differences in other contextual influencers.**



[Chapter 6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06) addresses the system environment and app configuration influencers. In this chapter, I cover the request history.

I begin by covering the benefits of deploying multiple instances of an app, and you’ll immediately see what happens when this intersects with stateful apps. I do this in the context of the cookbook example we started with in the previous chapter, breaking that monolith into individual microservices that are then independently deployed and managed. I introduce local state into one of those microservices, specifically storing authentication tokens. Yes, sticky sessions are a common pattern for addressing this type of session state, but it’s a bad idea in the cloud, and I’ll cover why. I also introduce the notion of a *stateful service*, a special type of service that’s carefully designed to handle the complexities of state. And finally, I’ll show you how to keep separate the parts of your software that are stateful from the parts that are stateless.

**5.1. Cloud-native apps have many instances deployed**

In the cloud, the prevailing model for increasing or decreasing application capacity to handle changing request volumes is to scale horizontally. Rather than adding or reducing capacity to a single application instance (vertical scaling), increased or decreased request volumes are handled by adding or removing app instances.

That isn’t to say that app instances can’t be allocated with substantial compute resources; Google Cloud Platform (GCP) is now offering a machine type with 1.5 TB of memory, and AWS is offering one with nearly 2 TB. But changing the specs of the machine that an app is running on is a substantial event. Say, for example, you’ve estimated that 16 GB of memory is ample for your application, and for some time things are working fine. But then your request volume picks up, and you’d now like to add to that capacity. There’s no way in cloud environments such as AWS, Azure, or GCP to change the machine type for a running host. Instead, you’d have to create a new machine with 32 GB of memory (even if you need only around 20 GB, because there’s no machine type with 20 GB of memory), deploy your app there, and then figure out how to move your users over to the new instances with as little disruption as possible.

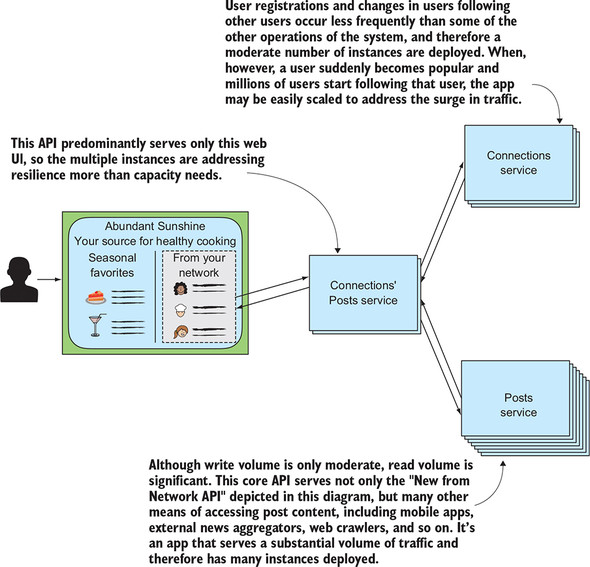
Contrast that with the scale-out model. Instead of provisioning your app with 16 GB of memory, you provision four instances with 4 GB each. When you need more capacity, you simply request a fifth instance of the app, make that instance available by, for example, registering it with a dynamic router, and now you’re running with a collective 20 GB of memory. Not only are you given finer-grained control over the consumption of resources, but the mechanics of achieving the greater scale are far easier.

But flexible scaling isn’t the only motivation for having multiple instances of an app. So too are high availability, reliability, and operational efficiency. Going all the way back to the first example in this book, in the hypothetical scenario depicted in [figures 1.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fig02) and [1.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fig03), it was the multiple instances of an application that allowed Netflix to remain operable in the face of an AWS infrastructure outage. Clearly, if you deploy your app as a singleton, it’s a single point of failure.

Multiple instances also bring benefits when it comes to operating software in production. 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 the platform, and it will establish the runtime environment and deploy the app. If the platform (which from an app perspective is part of the infrastructure) needs to upgrade the OS, ideally the app should remain running throughout that upgrade. While a host is having its OS upgraded, the workloads running thereon must be stopped, but if you have other app instances running on other hosts (recall the discussion in [chapter 3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03) around app instance distribution), you can roll through your hosts one at a time. While one app instance is taken offline, other app instances are still serving traffic.

Finally, when you bring together app scale-out and a decomposed, microservices-based software architecture, you gain a great deal of flexibility in the overall resource consumption of the system. As demonstrated by the cooking community software that I introduced in the previous chapter, having separate app components allows you to scale the Posts API, which is being invoked by many more clients than those depicted in our scenario to date, to handle large volumes of traffic, while other apps such as the Connections API have fewer instances handling much smaller volumes. See [figure 5.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05fig03).

**Figure 5.3. Architecting apps for multi-instance deployments affords significant gains in the efficiency of resource use as well as resilience and other operational benefits.**



**5.2. Stateful apps in the cloud**

As you can see, the need for things such as flexible scaling, resilience, and operational efficiencies leads to having multiple instances of an app running as a part of your software solutions. These same goals, as well as the multi-instance architecture I’ve just described, all have a strong relationship to the statefulness or statelessness of your apps. But rather than discuss these elements only in the abstract, let’s turn to a concrete example to start.

I’m going to begin with the application from the previous chapter, in particular, the one that was written in the request/response style. This implementation is heavily stateful in that it even stores all of the app data, users, connections, and posts in an in-memory database. It’s also a monolithic application. The Connections’ Posts, Connections, and Posts services are all part of the same project and ultimately compiled into the same JAR file (Java archive). To start, you’ll fix both of these things.

You’ll find the source code for this example in the cloud-native repository, in particular in the cloudnative-statelessness directory/module. You’ll begin with an implementation that first demonstrates the downsides of carrying state in your apps and you’ll later implement a solution. After cloning the repository, please check out a specific tag and move into the cloudnative-statelessness directory:

git clone https://github.com/cdavisafc/cloudnative-abundantsunshine.git

git checkout statelessness/0.0.1

cd cloudnative-statelessness

**5.2.1. Decomposing the monolith and binding to the database**

Let’s first look at how I’ve broken the previously monolithic application into three separate services. The cloudnative-statelessness directory now holds only a pom.xml file and a subdirectory/submodule for each of the three microservices. Two of the microservices, cloudnative-posts and cloudnative-connections, are completely standalone. Each has no dependency on any of the other microservices.

The third microservice, the cloudnative-connectionsposts app, is also mostly disconnected from the other two, with the only real indication of any dependency seen in the application.properties file:

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

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

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

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

INSTANCE\_IP=127.0.0.1

INSTANCE\_PORT=8080

Recall that this app makes a request to the *Connections* service to obtain the list of individuals a particular user follows, and then makes a request to the *Posts* service to obtain the posts from any of those individuals. URLs are configured into this app to facilitate reaching those services over HTTP. (Note that configuring these URLs into the application.properties file is a cloud antipattern and will be corrected in the next chapter.)

Turning now to the storage of the user, connections, and posts data, whether you’re running this app in the cloud or not, it almost certainly requires that the data for the app isn’t just held in memory but is also stored on persistent disk somewhere. That I previously stored it only in an in-memory H2 database was purely out of convenience; the persistent storage of the data wasn’t germane to our conversation in the previous chapter. Ultimately, we also have to concern ourselves with the resilience of this persisted data, and I’ll soon talk about this in more depth. I’ve added a dependence on a MySQL database in the pom.xml files of the Connections and Posts apps. The POM file now includes both of the following dependencies:

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

The H2 dependency was previously present; the MySQL dependency is new. I’ve kept the H2 dependency primarily to allow it to be used in testing. If a MySQL URL is provided on startup, Spring Boot JPA will instantiate and configure the MySQL client. Otherwise, it will use H2.

Let’s get this code up and running.

**Setting up**

Just as with the examples of the previous chapter, in order to run the samples, you must have standard tools installed; the last two on this list are new:

* Maven
* Git
* Java 1.8
* Docker
* Some type of a MySQL client, such as the mysql command-line interface (CLI)
* Some type of a Redis client, such as redis-cli

**Building the microservices**

From the cloudnative-statelessness directory, 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.

**Running the apps**

Before running any of the microservices, you need to start a MySQL service and create the cookbook database. To start the MySQL service, you’ll use Docker. Assuming you have Docker installed, you can do so with the following command:

docker run --name mysql -p 3306:3306 -e MYSQL\_ROOT\_PASSWORD=password \

-d mysql:5.7.22

To create the database, you can connect to your MySQL server via a client tool of your choice. Using the mysql CLI, you can type the following:

mysql -h 127.0.0.1 -P 3306 -u root –p

Then type in the password, password. At the MySQL command prompt, you can execute the following command:

mysql> create database cookbook;

Now you’re ready to run the apps—plural. That, right there, is the first point: because you’ve separated out the software into three independent microservices, you need to run three different JAR files. You’ll be running each app locally, so each Spring Boot app server (which is Tomcat by default) must start on a different port. You’ll supply this, and for the Posts and Connections services, the URL to the MySQL service on the command line. So in three terminal windows, execute the following three commands:

java -Dserver.port=8081 \

-Dspring.datasource.url=jdbc:mysql://localhost:3306/cookbook \

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

java -Dserver.port=8082 \

-Dspring.datasource.url=jdbc:mysql://localhost:3306/cookbook \

-jar cloudnative-connections/target/

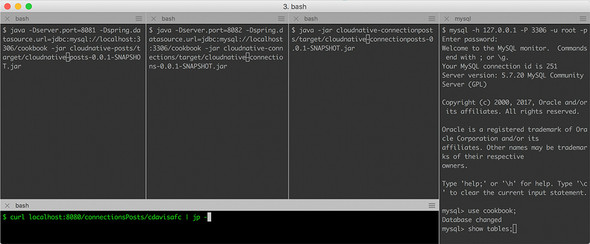
https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg cloudnative-connections-0.0.1-SNAPSHOT.jar

java -jar cloudnative-connectionposts/target/

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg cloudnative-connectionposts-0.0.1-SNAPSHOT.jar

I like to set up my terminal as shown in [figure 5.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05fig04): I’m in the cloudnative-statelessness directory in all windows. This arrangement allows me to look in on the database on the far right, watch the log output for each of the three microservices across the top (I run the Java commands I’ve just given you in each of these three windows), and execute curl commands to test my services in the larger window in the lower left.

**Figure 5.4. My terminal configuration allows me to send requests to the microservices while watching the results in the other windows.**



Next, you can execute the following curl commands to exercise each microservice:

curl localhost:8081/posts

curl localhost:8082/users

curl localhost:8082/connections

curl localhost:8080/connectionsposts/cdavisafc

In particular, watch all three of the upper windows when you execute the final curl command. You’ll see how that single request ultimately touches all three microservices.

With this version of the software, you’re now able to realize a deployment topology such as that depicted in [figure 5.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05fig03); the different apps are independently scaled. Your current deployment is, however, rather noncloud: everything is running locally, you have only single instances of each app, and your configuration is embedded within the JAR files with the inclusion of values in the application.properties files. The apps, however, are stateless. You could stop and start any of the app instances without data loss.

Without explicitly focusing on it, I’ve just snuck in an implementation of the pattern that allows you to make your apps stateless; I’ve moved the state into an external store. The reason I haven’t focused on that so far, however, is that connecting to an external database is something that’s so familiar to most of you that the points I want to make around statelessness might be lost. So to study statelessness, I want to reintroduce state into one of the microservices—a type of state that can easily creep into your designs if you don’t focus on keeping it out.

|  |
| --- |
|  |

**Note**

A common way for state to creep in is through session state.

|  |
| --- |
|  |

**5.2.2. Poorly handling session state**

Up until now, a client of the Connections’ Posts service could simply provide a username on the query string and retrieve the posts written by people followed by that individual. You could request that set of posts for any user in the system, unencumbered. What you want to do, however, is allow Max to retrieve only the posts of people he follows, and Glen to retrieve only the posts of the people he follows. To facilitate this, you’ll ask a client of the Connections’ Posts service to authenticate before serving any content.

To follow along, please check out the following tag for the same repository you’ve been working with:

Git checkout statelessness/0.0.2

I’ve added a login controller to the implementation of the Connections’ Posts service. Invoking the login functionality, which for simplicity takes only a username as input, will generate a token that’s then passed to subsequent invocations of the Connections’ Posts service. If the passed-in token is found to be valid, the set of posts is returned; if the token is invalid, an HTTP 1.1/401 Unauthorized response is returned.

The login controller is part of the Connections’ Posts service and is found in the LoginController.java file. As you can see in the following code, after the login token is created, it’s stored in an in-memory hash map that associates a token to a username.

**Listing 5.1. LoginController.java**

package com.corneliadavis.cloudnative.connectionsposts;

import ...

@RestController

public class LoginController {

@RequestMapping(value="/login", method = RequestMethod.POST)

public void whoareyou(

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

HttpServletResponse response) {

if (username == null)

response.setStatus(400);

else {

UUID uuid = UUID.randomUUID();

String userToken = uuid.toString();

CloudnativeApplication.validTokens.put(userToken, username);

response.addCookie(new Cookie("userToken", userToken));

}

}

}

That in-memory hash map is declared in the CloudnativeApplication.java file, as shown next.

**Listing 5.2. CloudnativeApplication.java**

public class CloudnativeApplication {

public static Map<String, String> validTokens

= new HashMap<String, String>();

public static void main(String[] args) {

SpringApplication.run(CloudnativeApplication.class, args);

}

}

I’ve made one other relevant change to the method that serves posts from a user’s connections, as shown in the following code, an excerpt of code found in the ConnectionsPosts.java file. Rather than providing a username as a part of the service URL, the service takes no usernames, but instead looks for a token in the cookie that is passed to the service.

**Listing 5.3. ConnectionsPostsController.java**

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

public Iterable<PostSummary> getByUsername(

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

HttpServletResponse response) {

if (token == null)

response.setStatus(401);

else {

String username =

CloudnativeApplication.validTokens.get(token);

if (username == null)

response.setStatus(401);

else {

// code to obtain connections and relevant posts

return postSummaries;

}

}

return null;

}

You can rebuild the application and redeploy the Connections’ Posts service to test this functionality. From the cloudnative-statelessness directory, run the following command to build the project:

mvn clean install

Now, just as you’ve done previously, run the microservices in three terminal windows with the following commands:

java -Dserver.port=8081 \

-Dspring.datasource.url=jdbc:mysql://localhost:3306/cookbook \

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

java -Dserver.port=8082 \

-Dspring.datasource.url=jdbc:mysql://localhost:3306/cookbook \

-jar cloudnative-connections/target/

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg cloudnative-connections-0.0.1-SNAPSHOT.jar

java -jar cloudnative-connectionposts/target/

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg cloudnative-connectionposts-0.0.1-SNAPSHOT.jar

The Connections’ Posts service is no longer invoked with a username as a part of the URL. Calls to the new endpoint prior to performing any login will result in an HTTP error. To see this, include the –i switch in your curl command as follows:

$ curl -i localhost:8080/connectionsposts

HTTP/1.1 401

X-Application-Context: application

Content-Length: 0

Date: Mon, 27 Nov 2018 03:42:07 GMT

You’ll log in with the following command; use one of the usernames preloaded in the sample data:

$ curl -X POST -i -c cookie localhost:8080/login?username=cdavisafc

HTTP/1.1 200

X-Application-Context: application

Set-Cookie: userToken=f8dfd8e2-9e8b-4a77-98e9-49aaed30c218

Content-Length: 0

Date: Mon, 27 Nov 2018 03:44:42 GMT

And now when you invoke the Connections’ Posts service, passing the cookie with the -b command line switch, you’ll receive this response:

$ curl -b cookie localhost:8080/connectionsposts | jp -

[

{

"date": "2019-02-01T19:09:41.000+0000",

"usersname": "Max",

"title": "Chicken Pho"

},

{

"date": "2019-02-01T19:09:41.000+0000",

"usersname": "Glen",

"title": "French Press Lattes"

}

]

I haven’t called it out explicitly yet, but you might have noticed that our implementation is no longer stateless. The valid tokens are stored in memory. Lest you think this example is contrived—a hash map in my main Spring Boot app—I assure you it’s quite representative of a pattern commonly found in apps today. As part of my work at Pivotal, I recently brought a new caching product to market (Pivotal Cloud Cache) and had many conversations with developers and architects who were looking for effective ways of handling the HTTP session state that many of their apps were using. Although I’m not expressly using any of the HTTP interfaces here, to keep the code as simple as possible, the basic structure is the same.

Despite this supposed limitation in the implementation, if you rerun that last curl repeatedly, you’ll consistently receive the expected response. If it’s all working fine, then what’s the problem? The issue is that at this point, you’re still running the app in a noncloud and non-cloud-native setting. You have only a single instance of each app, and provided they remain functional and the Connections’ Posts service remains running, your software functions as expected.

But you know that things are always changing in a cloud setting, and from the previous section, you also know that apps almost always have multiple instances deployed, so let’s run a few experiments.

First, let’s simulate the cycling of the Connections’ Posts service. In a real setting, this could happen if the app itself crashed, but even more likely it happens because a new version of the app is being deployed or the infrastructure is going through a change that causes the app to be re-created in a refreshed infrastructure context. To simulate this, stop the app by pressing Ctrl-C in the right window and rerunning the java command:

java -jar cloudnative-connectionposts/target/

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg cloudnative-connectionposts-0.0.1-SNAPSHOT.jar

When you now attempt the curl command and pass in the valid authentication token, you’ll receive the HTTP 1.1/401 Unauthorized response:

$ curl –i -b cookie localhost:8080/connectionsposts

HTTP/1.1 401

X-Application-Context: application

Content-Length: 0

Date: Mon, 27 Nov 2018 04:12:07 GMT

I’m certain this doesn’t surprise you. You’re well aware that the tokens are stored in memory, and when you stopped and restarted the application, you lost everything that was in memory. The main realization that I hope you have, however, is that you no longer can count on this type of application cycling not happening. Change is the rule, not the exception.

Let’s now look at a second scenario: deploying multiple instances of the app. As soon as you have those multiple instances, you’ll need a load balancer to route traffic among them. You could establish this yourself, running something like nginx and configuring all of your instances into it, but this is exactly what cloud platforms do for you, so I will take this as an opportunity to introduce you to one. As it happens, Kubernetes has an easy-to-use, locally deployable version that makes it a perfect option for what I want to demonstrate. It has a vibrant community around it that can offer support when needed, and I have to tell you, Kubernetes is an awesome piece of technology that I love working with. If you have familiarity with and access to another cloud platform such as Cloud Foundry, Heroku, Docker, OpenShift, or others, by all means, run these experiments there.

**Introducing a cloud-native platform**

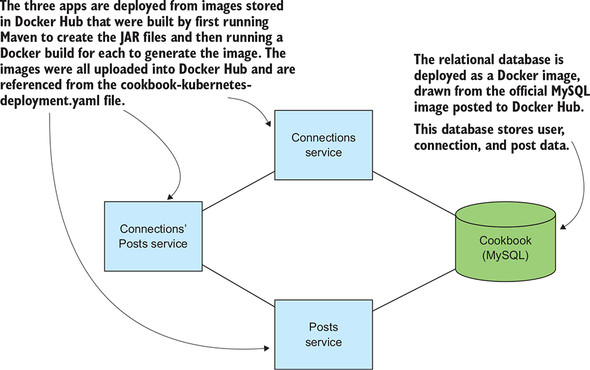
Kubernetes is a platform for running applications. It includes capabilities that allow you to deploy, monitor, and scale your apps. It brings the type of health monitoring and automatic remediation that I’ve talked about in earlier chapters of this book and will allow you to test various examples and patterns in a compelling way. For example, when app instances are lost for any reason, Kubernetes will launch new instances to replace them. As I’ve already mentioned several times (and will continue to do so), a cloud-native platform will provide support for, and even implementations of, many of the patterns I cover in this book.

To run an application in Kubernetes, it must be containerized. You must have a Docker image (or similar) that contains your app. I won’t cover containerization in detail, but will give you the steps to perform so that your app can be bundled into a Docker image and made available to Kubernetes.

The distribution of the open source Kubernetes project that you’ll be using here is called Minikube (<https://github.com/kubernetes/minikube>). Whereas in production Kubernetes would always be deployed as a multinode distributed system (probably across availability zones, as we’ve talked about in preceding chapters), Minikube gives you a single-node deployment that allows you to get up and running quickly on your own workstation. The installation instructions for Minikube are included in the README file of the GitHub repository and provide steps for running on Linux, Windows, and macOS. Prior to installing Minikube, you should also install the Kubernetes CLI, kubectl (<https://kubernetes.io/docs/tasks/tools/install-kubectl/>). After you’ve addressed the prerequisites (for example, having VirtualBox installed on your machine) and installed Minikube, you’ll be ready to deploy our sample app on Kubernetes.

I’ve provided you deployment manifests for all the components that make up our software example. At this stage, you have the four components you previously ran locally: the MySQL database and each of the three microservices (Connections, Posts, and Connections’ Posts). To deploy the software in the topology shown in [figure 5.5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05fig05), proceed as follows.

**Figure 5.5. The deployment topology of the cookbook software, refactored into separate components and deployed into a cloud setting. At the moment, each app has only a single instance deployed.**



**Deploying and configuring the database**

You’ll run the exact same Docker image in Kubernetes as you did when running our example locally. The deployment manifest you’ll use is mysql-deployment.yaml. You instruct Kubernetes to launch and manage this component with the following command:

kubectl create -f mysql-deployment.yaml

You can watch the status of this deployment as follows:

$ kubectl get all

NAME READY STATUS RESTARTS AGE

pod/mysql-75d7b44cd6-dbnvp 1/1 Running 0 30s

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

service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 14d

service/mysql-svc NodePort 10.97.144.19 <none> 3306:32591/TCP 6h14m

NAME READY UP-TO-DATE AVAILABLE AGE

deployment.apps/mysql 1/1 1 1 30s

NAME DESIRED CURRENT READY AGE

replicaset.apps/mysql-75d7b44cd6 1 1 1 30s

You’ll notice that the output displays numerous entities that have a MySQL association. Here’s a brief overview: you have a *deployment* of MySQL that’s running the app in a *pod* (Docker images run in pods) and can be accessed via the MySQL *service*. *Replica sets* indicate the number of copies of a particular workload that are to be running.

To create the cookbook database, you’ll use the same mechanism you did when running locally. You need only know the connection string to pass to your MySQL client. This is something that you can obtain from Minikube with the following command:

minikube service mysql –url

If you’re using the mysql CLI for access, you can execute the following command to access the database and then the next command to create the database:

$ 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)

Your database server is now running, and the database your app will use is created.

**Configuring and deploying the Connections and Posts services**

The ways in which the Connections and Posts services are configured and deployed are virtually identical. Each must know the connection string and credentials for the MySQL database you’ve just deployed, and each will be run in its own container (and Kubernetes pod). There’s a deployment manifest for each service, and you must edit each to insert the MySQL connection string. To obtain the URL that will be inserted into each file, execute the following command:

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

The response when you run this command is as follows:

jdbc:mysql://192.168.99.100:32713/cookbook

Now edit the cookbook-deployment-connections.yaml and cookbook-deployment-posts.yaml files, replacing the string <insert jdbc url here> with the jdbc URL returned from the preceding minikube command. For example, the final lines of cookbook-deployment-kubernetes-connections.yaml will look something like this:

- name: SPRING\_APPLICATION\_JSON

value: '{"spring":{"datasource":{"url":

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg "jdbc:mysql://192.168.99.100:32713/cookbook"}}}'

You can then deploy both services by executing the following two commands:

kubectl create -f cookbook-deployment-connections.yaml

kubectl create -f cookbook-deployment-posts.yaml

Running kubectl get all again will show that you now have two microservices running in addition to the MySQL database:

$ kubectl get all

NAME READY STATUS RESTARTS AGE

pod/connections-7dffdc87c4-p8fc8 1/1 Running 0 12s

pod/mysql-75d7b44cd6-dbnvp 1/1 Running 0 13m

pod/posts-6b7486dc6d-wmvmv 1/1 Running 0 12s

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

service/connections-svc NodePort 10.106.214.25 <none> 80:30967/TCP

service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP

service/mysql-svc NodePort 10.97.144.19 <none> 3306:32591/TCP

service/posts-svc NodePort 10.99.106.23 <none> 80:32145/TCP

NAME READY UP-TO-DATE AVAILABLE AGE

deployment.apps/connections 1/1 1 1 12s

deployment.apps/mysql 1/1 1 1 13m

deployment.apps/posts 1/1 1 1 12s

NAME DESIRED CURRENT READY AGE

replicaset.apps/connections-7dffdc87c4 1 1 1 12s

replicaset.apps/mysql-75d7b44cd6 1 1 1 13m

replicaset.apps/posts-6b7486dc6d 1 1 1 12s

To test that each service is running correctly, use the following two commands to retrieve the sample connections and posts that have been loaded into your database:

$ curl $(minikube service --url connections-svc)/connections

[

{

"id": 4,

"follower": 2,

"followed": 1

},

{

"id": 5,

"follower": 1,

"followed": 2

},

{

"id": 6,

"follower": 1,

"followed": 3

}

]

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

[

{

"id": 7,

"date": "2019-02-03T04:36:28.000+0000",

"userId": 2,

"title": "Chicken Pho",

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

},

{

"id": 8,

"date": "2019-02-03T04:36:28.000+0000",

"userId": 1,

"title": "Whole Orange Cake",

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

},

{

"id": 9,

"date": "2019-02-03T04:36:28.000+0000",

"userId": 1,

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

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

},

{

"id": 10,

"date": "2019-02-03T04:36:28.000+0000",

"userId": 3,

"title": "French Press Lattes",

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

}

]

**Configuring and deploying the Connections’ Posts service**

Finally, let’s deploy the service that collects and returns the set of posts made by the folks that a particular individual follows. This service doesn’t access the database directly; instead, it makes service calls to both the Connections and Posts services. With the deployments you’ve just executed in the immediately preceding section, these services are now running at the URLs you’ve just tested, and you need only configure those URLs into the Connections’ Posts service. You’ll do this by editing the deployment manifest, cookbook-deployment-connectionsposts-stateful.yaml. There are placeholders for three URLs, and they should be filled in with values obtained by issuing the following commands:

|  |  |
| --- | --- |
| 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 |

The final lines of your deployment manifest will look something like the following:

- name: CONNECTIONPOSTSCONTROLLER\_POSTSURL

value: "http://192.168.99.100:31040/posts?userIds="

- name: CONNECTIONPOSTSCONTROLLER\_CONNECTIONSURL

value: "http://192.168.99.100:30494/connections/"

- name: CONNECTIONPOSTSCONTROLLER\_USERSURL

value: "http://192.168.99.100:30494/users/"

Finally, you’ll deploy the service with the following command:

kubectl create \

-f cookbook-deployment-connectionsposts-stateful.yaml

You can now test this service just as you have before, by executing the following commands:

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

curl -X POST -i -c cookie \

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

curl -i -b cookie \

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

|  |
| --- |
|  |

**I promise this will get better soon.**

I know that all of this manual configuration is frustrating you, but fear not. The next chapter covers application configuration, and with the use of proper practices, some of this tedium will be eliminated.

|  |
| --- |
|  |

To set up for the next demonstration, I’m going to have you stream the logs for this Connections’ Posts service. In a new terminal window, execute the following command, providing the name of the connectionsposts pod (you can see this from running the kubectl get pods command):

kubectl logs –f pod/<name of your connectionsposts pod>

You can now repeat the final curl command shown previously and see the resultant activity in those connectionsposts logs. You’re up and running. And provided you don’t cycle the Connections’ Posts service, this deployment will continue serving your users quite well. But what happens when you scale the service to multiple instances? To do so, execute the following command:

kubectl scale --replicas=2 deploy/connectionsposts

Executing the kubectl get all command again shows several interesting things:

NAME READY STATUS RESTARTS AGE

pod/connections-7dffdc87c4-cp7z7 1/1 Running 0 10m

pod/connectionsposts-5dc77f8bf9-8kgld 1/1 Running 0 5m4s

pod/connectionsposts-5dc77f8bf9-mvt89 1/1 Running 0 81s

pod/mysql-75d7b44cd6-dbnvp 1/1 Running 0 36m

pod/posts-6b7486dc6d-kg8cp 1/1 Running 0 10m

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

service/connections-svc NodePort 10.106.214.25 <none> 80:30967/TCP

service/connectionsposts-svc NodePort 10.100.25.18 <none> 80:32237/TCP

service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP

service/mysql-svc NodePort 10.97.144.19 <none> 3306:32591/TCP

service/posts-svc NodePort 10.99.106.23 <none> 80:32145/TCP

NAME READY UP-TO-DATE AVAILABLE AGE

deployment.apps/connections 1/1 1 1 10m

deployment.apps/connectionsposts 2/2 2 2 5m4s

deployment.apps/mysql 1/1 1 1 36m

deployment.apps/posts 1/1 1 1 10m

NAME DESIRED CURRENT READY AGE

replicaset.apps/connections-7dffdc87c4 1 1 1 10m

replicaset.apps/connectionsposts-5dc77f8bf9 2 2 2 5m4s

replicaset.apps/mysql-75d7b44cd6 1 1 1 36m

replicaset.apps/posts-6b7486dc6d 1 1 1 10m

You can see that a second pod running an instance of the Connections’ Posts service has been created. But there remains only a single connectionsposts-svc service. This single service will now load-balance requests across both instances of the app. You can also see that both the deployment and the replication controllers are showing a desire to have two instances of the app running at any point.

|  |
| --- |
|  |

**Don’t do this in production!**

I can’t help but point out that issuing a command to provision a second instance of the Connections’ Posts app *at the command line* should *never be done in production*. As soon as you’ve done so, you’ve created a snowflake, a software topology that isn’t recorded anywhere.

The deployment manifest that you used to initially deploy the software captures the initial topology, and likely what you expect is running in production, but reality has diverged from that which is recorded. When individuals manually apply changes to a running system, that system is no longer reproducible from the artifacts that are recorded and controlled as a part of your operational practices.

The right practice for achieving this scale-out is to modify the deployment manifest, check it into a control system, and apply it to the cloud environment. Kubernetes supports this, updating a running system rather than producing a new one with a command such as kubectl apply. I take the shortcut here only for simplicity.

|  |
| --- |
|  |

Before testing your application functionality in the new deployment topology, please stream the logs for the new app instance. In another terminal window, execute the kubectl logs –f command with the name of the new pod, as you can see in the kubectl get all output:

kubectl logs -f po/<name of your new pod>

Now, let’s test the application functionality again by issuing your final curl command a few more times:

curl -i -b cookie \

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

Keep an eye on the two windows in which you’re streaming the application logs. If the load balancer routes traffic to the first instance, you’ll see activity in that log stream and results will be returned as expected. If, however, traffic is routed to the second instance, that log will show there was an attempt to access the app with an invalid token, and a 401 will be returned. But of course, the token *is* valid; the trouble is that the second instance doesn’t know about that valid token.

I’m sure we’d all agree this is a terrible user experience. Sometimes the curl command returns the requested information, and sometimes it reports that the user is unauthenticated. When the user then scratches his head and thinks, “Aren’t I already logged in?” and refreshes the page, he might get a valid response, but then the next request again reports unauthenticated.

|  |
| --- |
|  |

**Using Kubernetes requires you to build Docker images**

You might notice that I haven’t asked you to build the application as with previous examples. I’ve skipped the steps here because in order to deploy apps to Kubernetes, they must be containerized, and the build process is somewhat involved. You must create JAR files, run a docker build command to create the Docker images, upload those images into an image repository, update the deployment manifest to point to those images, and then do the deployment.

I’ve included the Dockerfile that I used to create each of the container images. But for brevity, the deployment manifests I provide point to the Docker images I’ve already produced and uploaded into my own Docker Hub repository. The commands I’ve executed to do all of this are as follows:

Build source:

mvn clean install

Create Docker images:

docker build --build-arg \

jar\_file=cloudnative-connectionposts/target\

/cloudnative-connectionsposts-0.0.1-SNAPSHOT.jar \

-t cdavisafc/cloudnative-statelessness-connectionsposts-stateful .

docker build --build-arg \

jar\_file=cloudnative-connections/target\

/cloudnative-connections-0.0.1-SNAPSHOT.jar \

-t cdavisafc/cloudnative-statelessness-connections .

docker build --build-arg \

jar\_file=cloudnative-posts/target/cloudnative-posts-0.0.1-SNAPSHOT.jar \

-t cdavisafc/cloudnative-statelessness-posts .

Upload to Docker hub:

docker push cdavisafc/cloudnative-statelessness-connectionposts-stateful

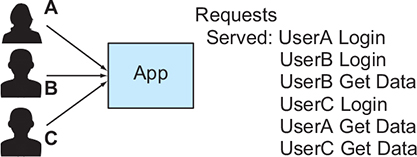
docker push cdavisafc/cloudnative-statelessness-connections

docker push cdavisafc/cloudnative-statelessness-posts

|  |
| --- |
|  |

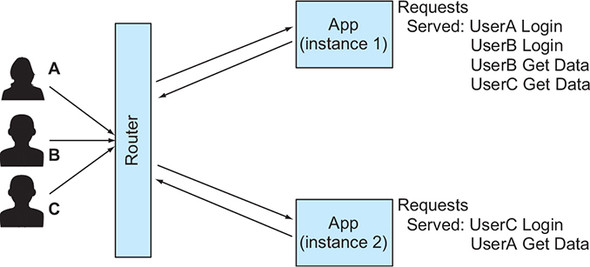
Although in this simple example it’s easy to see why this occurs, let’s look at what’s going on from an architectural perspective. [Figure 5.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05fig06) depicts the desired behavior of our app. I’m showing a single entity for the Connections’ Posts app to designate that *logically* you have only a single app. As I pointed out at the beginning of the chapter, the same input should yield the same result, regardless of the number of instances of the app and which app a particular request is routed to.

**Figure 5.6. Logically, you have a single app that serves a series of requests. Because each user logs in before trying any Get Data calls, you expect each of those will succeed.**



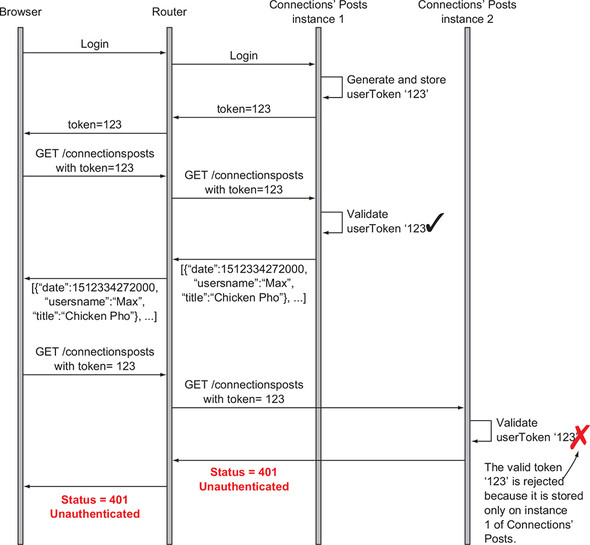
But you haven’t achieved this in this current implementation because your app isn’t stateless. As a simple way to visualize what is happening, [figure 5.7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05fig07) separates your single logical app into two instances and distributes the sequence of requests across those instances. In this diagram, I haven’t changed the sequencing of the requests but only distributed them across different application instances. It’s now easy to see that if your app considers only the local requests, in many cases your application functionality will be compromised.

**Figure 5.7. When a single logical entity is deployed as multiple instances, care must be taken to ensure that the distributed set of events continues to be treated as a whole.**



Turning back to our cookbook example, [figure 5.8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05fig08) summarizes the behavior that your current implementation presents.

**Figure 5.8. When you add a second instance of the Connections’ Posts app, its local list of valid tokens differs from that of the first. This is exactly the problem with stateful apps in cloud-native software.**



How do you then solve this problem? Let’s consider one “solution” that’s in relatively widespread use today, but isn’t appropriate in the cloud setting—sticky sessions. (I’ll then present the cloud-native solution.)

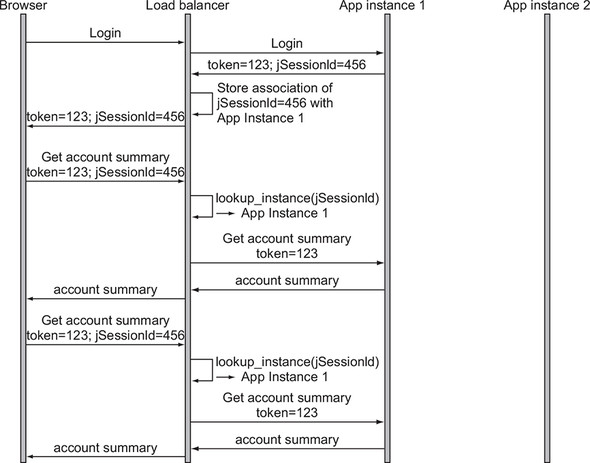
**5.3. HTTP sessions and sticky sessions**

What about sticky sessions? For almost as long as we’ve been using load balancers to distribute requests across multiple instances of applications—and this is far longer than those apps have been designed in a cloud-native manner—we’ve been using sticky sessions. Why can’t we continue to use those to deal with stateful services?

First, let me briefly explain the technique. *Sticky sessions* are an implementation pattern in which an app includes a session ID in the response to a first request from a user—a fingerprint for that user, if you will. That session ID is then included, usually via a cookie, in all subsequent requests. This effectively allows the load balancer to keep track of individual users who are interacting with an app. That load balancer, which is responsible for deciding where to send a specific request, will remember which instance was first accessed and will then do its best to route all requests carrying that session ID to the same app instance. If that app instance has a local state, requests consistently routed to that instance will have that local state available.

[Figure 5.9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05fig09) shows that sequence: when a session ID is present in the request, the router looks up the instance to which that ID corresponds and sends the request to that app instance.

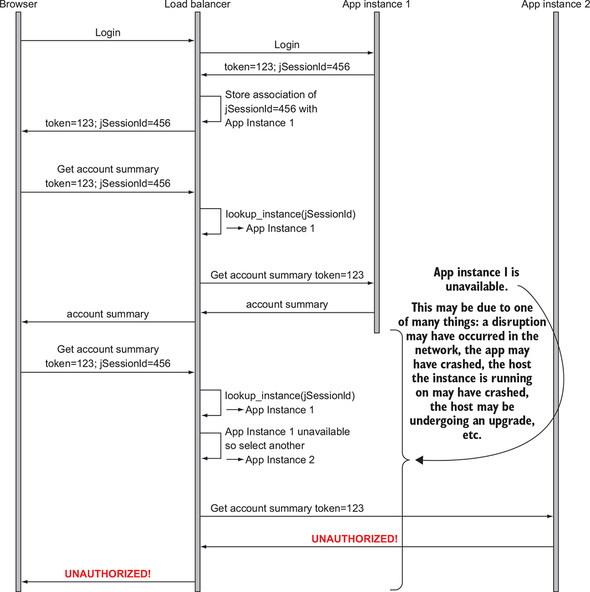
**Figure 5.9. Sticky sessions are implemented by the load balancer and are used to tie a specific user to a specific instance of an app.**



Isn’t that an easier solution than ensuring that each and every one of your apps is completely stateless?

Did you catch the part about “do its best”? Despite its best efforts, the router may not be able to send the request to the “right” instance. That instance may have vanished or may be unreachable because of network anomalies. In [figure 5.10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05fig10), for example, app instance 1 is unavailable, so the router will send the user request to another app instance. Because that instance doesn’t have the local state that app instance 1 had, the user will once again experience the negative behavior that I demonstrated previously.

**Figure 5.10. A load balancer supporting sticky sessions will attempt to stick specific users to specific app instances, but may be forced to send a request to a different instance. In cloud-native software, you must assume that instances come and go with some regularity. If you don’t, the user experience suffers.**



Developers have, for some time, justified the use of sticky sessions, arguing that anomalies such as disappearing instances or network outages are rare, and that when they do occur, subpar user experiences, while undesirable, are acceptable. This is a poor argument for two reasons. First, the recycling of app instances is increasingly common because of either unanticipated or deliberate changes to the infrastructure. Second, it isn’t difficult to implement something better—namely, persisting session state in a connected backing-store, and this is an approach that brings with it many other advantages. Let’s have a look at that now.

**5.4. Stateful services and stateless apps**

The right approach to solving this problem is suggested in the title to this chapter: make apps stateless. To first demonstrate the antipattern—stateful services—and now show the good pattern, I’ve chosen the specific example of user authentication because it’s an area where I’ve so often seen subpar solutions implemented. To excuse the use of sticky sessions, people often make the argument that we can’t have users providing their credentials with every request. Although that argument is sound, it doesn’t mean the app instances need to hold that state.

**5.4.1. Stateful services are special services**

Of course, there has to be state somewhere. As a whole, an application must carry state somewhere in order to be of any use. Your banking website wouldn’t provide much value if you couldn’t see your account balances, for example. So when I suggest that apps be stateless, what I’m really saying is that we needn’t have state everywhere in our architecture.

|  |
| --- |
|  |

**Note**

Cloud-native applications have places where state lives, and just as important, places where it doesn’t.

|  |
| --- |
|  |

The app is stateless. State lives in data services. This is the narrative that we hear a lot these days. I confess that I’m not the type of person to follow any guidance unless I understand why the advice is sound, so let’s do a proof by briefly studying what we’d have to do to keep our software running well if we carried a lot of state in our apps. Remember, one of the primary things we have to anticipate is constant change.

In this case, anytime that the internal state of an app changes, and that state is either in memory or on local disk, that state needs to be preserved, just in case the app instance is lost. Several approaches for preserving state exist, but all involve replication. One option, which is completely independent of any knowledge of the application logic, is to take snapshots; simply make copies of memory and disk at a certain interval.

But as soon as we start talking about snapshots, we have a host of decisions to make around how to produce and manage those snapshots. How often do you do them? How do you ensure that the snapshots are consistent (that is, capture a state that isn’t inadvertently transitory because changes occurred in the middle of taking a snapshot)? How long will it take to recover them? Trade-offs of recovery time objectives (RTOs) against recovery point objectives (RPOs) are inherently complex. Even if you aren’t familiar with the details of snapshotting, RTO, and RPO (I spent more than a decade working for a major storage system vendor https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/smile.jpg), this description alone has probably been sufficient to raise your stress level a bit.

Another approach to making copies of data for resilience is to make the replication step a part of the data storage. Here the application events trigger both the storage of a primary copy of data as well as one or more replicas. To ensure that replicas are available when the primary vanishes for any reason, the copies are distributed across failure boundaries: copies are stored on different hosts, in different availability zones, or on different storage devices. But then, as soon as these processes cross such boundaries, you have a distributed system dealing with data, and frankly, these are hard problems to solve.

You’ve surely heard of the CAP theorem, which states that only two of the three attributes of *consistency*, *availability*, and *partition* tolerance can be maintained in any system. Because a distributed system will always suffer from occasional network partitions, distributed data systems can only be *either* consistent *or* available. Detailed study of the CAP theorem and the challenges of distributed stateful services is beyond the scope of the discussion at hand, but the following note makes my point.

|  |
| --- |
|  |

**Note**

Handling state in cloud-based systems, which by definition are highly distributed, requires special care and complex algorithms. Rather than solving those problems in each and every app that makes up our software, we’ll concentrate the solutions in only specific parts of our cloud-native architecture. Those parts are the *stateful services*.

|  |
| --- |
|  |

So, you put the state in specially designed stateful services and remove the state from your apps. In a moment, you’ll do that in the context of our cookbook application. But first I want to point out a few other advantages of stateless apps, those beyond avoiding the complexity of distributed data resilience. When your apps are stateless, a cloud-native application platform can easily create new instances of an app when older instances are lost. It need only start a new instance from the same base state that it started the original, and it’s good to go. The routing tier can evenly distribute the load across multiple instances, the number of which can be adjusted based on the volume of requests that need to be handled; no operations beyond registering the coordinates of the new instances are necessary. Instances can be moved around with ease. Need to upgrade a host that your instances are running on? No problem, just start a new instance and route traffic.

You can reasonably manage multiple *versions* of an app all deployed and running side by side (recall the importance of parallel deploys in our earlier conversations on continuous delivery). You can have some of your traffic routing to the newest version and other traffic reaching prior versions, and despite the variability in the app components, state continues to be handled consistently in the stateful services.

Now, to be clear, there’s absolutely nothing wrong with having an app store data in memory and even on local disk. But that data can be counted on to be there only for the duration of the single app invocation that generated it in the first place. A concrete example comes with an app that will load an image, process it in some way, and return a new rendering thereof. The processing could be multistep and store interim results on disk, but that local storage is ephemeral, existing only until the final image has been generated and is returned to the caller. At the risk of beating a dead horse, you can’t count on any of that data being available when the next request comes into that same app instance.

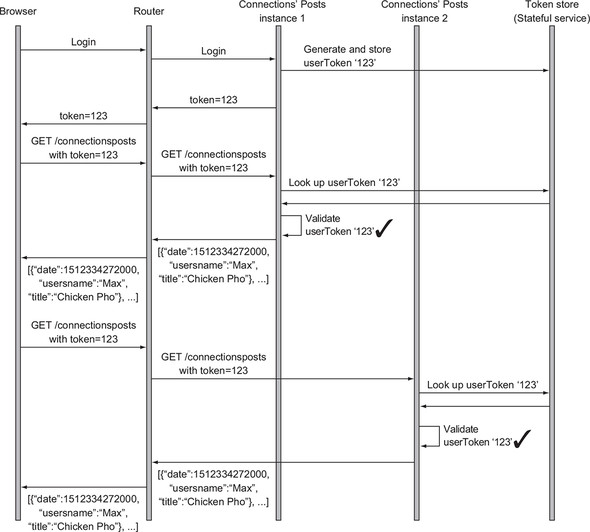
The job of developers, then, is to be explicit about which state is important to preserve and which is not, and to design their apps so that any data that’s needed across invocations is placed in the stateful services. With careful consideration of this design element, you have the best of both worlds. A portion of your overall implementation can be managed to handle the scale and fluidity of the system it’s running on (the stateless app), and a portion can be designed to handle the trickier task of managing data (state).

**5.4.2. Making apps stateless**

Let’s now turn back to our cookbook example. When you left off, you had implemented some simple user authentication. But because you stored valid tokens in memory, if a request was routed to an instance that wasn’t storing a user’s token, they’d be asked to log in, even if they already had.

The solution is simple: you’ll introduce a key/value store that will be used to persist valid login tokens, and you’ll bind your app to that stateful service. Every instance of the app will include that binding (how the binding is recorded foreshadows the next chapter on app configuration), so any request can be routed to any app instance, and the valid tokens will be accessible. [Figure 5.11](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05fig11) reflects this topology, updating [figure 5.8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05fig08), which contained the prior flow.

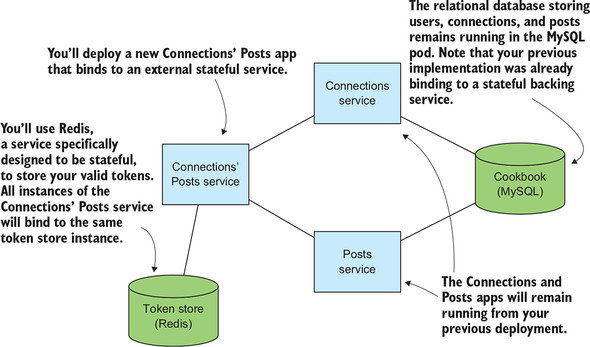
**Figure 5.11. Storing valid tokens in a stateful service that’s bound to all instances of the Connections’ Posts service allows apps to be stateless but our overall software solution to be stateful.**



I’ve implemented this solution within the cloudnative-statelessness directory/module of the cloud-native repository. The deployment topology that you’ll achieve with the following instructions is shown in [figure 5.12](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05fig12). The solution is in the master branch of your repository, so if you had previously cloned and checked out an earlier tag, you can switch to the master branch with the following:

git checkout master

**Figure 5.12. Your new deployment replaces the Connections’ Posts service with a stateless version and adds a Redis server to store authentication tokens.**



The implementation uses Redis as the stateful service. You’ll run that service via another deployment to your Minikube environment. Execute the following command to start the Redis server:

kubectl create -f redis-deployment.yaml

After the container is running, you can connect to it by using the Redis CLI (or another client if you prefer) with the following command, and you can view the keys stored in Redis as well:

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

-p $(minikube service redis-svc --format "{{.Port}}")

> keys \*

You’ll now replace the prior Connections’ Posts service with a new one. The code changes to the app are minimal.

First, in the CloudnativeApplication Spring Boot configuration and main class, delete the local storage for tokens and add configuration for the Redis client that will be used to connect to your token store.

**Listing 5.4. CloudnativeApplication.java**

public class CloudnativeApplication {

@Value("${redis.hostname}")

private String redisHostName;

@Value("${redis.port}")

private int redisPort;

@Bean

public RedisConnectionFactory redisConnectionFactory() {

return new LettuceConnectionFactory

(new RedisStandaloneConfiguration(redisHostName, redisPort));

}

public static void main(String[] args) {

SpringApplication.run(CloudnativeApplication.class, args);

}

}

Next, instead of storing tokens in the now deleted local storage, in the LoginController.java code you use the Redis client to store the token to the username pair in your external stateful store.

**Listing 5.5. LoginController.java**

...

*CloudnativeApplication.validTokens.put(userToken, username*); ***1***

ValueOperations<String, String> ops = **this.template**.opsForValue();

ops.set(userToken, username);

* ***1* We’ve removed this**

Then, instead of retrieving tokens from the now deleted local storage, in the ConnectionsPostsController you use the Redis client to retrieve a username via the token that was supplied via the cookie.

**Listing 5.6. ConnectionsPostsController.java**

...

*String username*

*= CloudnativeApplication.validTokens.get(token);* ***1***

* ***1* We’ve removed this**

ValueOperations<String, String> ops = this.template.opsForValue();

String username = ops.get(token);

To make clear what you’re doing here, I will have you delete the old version of the Connections’ Posts app with the following command:

kubectl delete deploy connectionsposts

Before you deploy the new version of the Connections’ Posts app, you must configure the Redis connection information into the deployment manifest. Edit the cookbook-deployment-connectionsposts-stateless.yaml file, inserting the Redis hostname and port into the appropriate locations. You can obtain the hostname and port values with the following two commands:

minikube service redis --format "{{.IP}}"

minikube service redis --format "{{.Port}}"

When completed, this YAML file will appear similar to this:

- name: CONNECTIONPOSTSCONTROLLER\_POSTSURL

value: "http://192.168.99.100:31040/posts?userIds="

- name: CONNECTIONPOSTSCONTROLLER\_CONNECTIONSURL

value: "http://192.168.99.100:30494/connections/"

- name: CONNECTIONPOSTSCONTROLLER\_USERSURL

value: "http://192.168.99.100:30494/users/"

- name: REDIS\_HOSTNAME

value: "192.168.99.100"

- name: REDIS\_PORT

value: "32410"

You can now deploy the new app with the following command:

kubectl create \

-f cookbook-deployment-connectionsposts-stateless.yaml

And test your software with the usual series of commands:

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

curl -X POST -i -c cookie \

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

curl -i -b cookie \

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

The POST curl command will have created a new key in the Redis store, something you can see by executing the keys \* command, run via the Redis CLI. And now let’s get the topology back to what you had when you saw the problems with the stateful app. Scale your Connections’ Posts app to two instances with the following:

kubectl scale --replicas=2 deploy/connectionsposts

You can now stream the logs for both instances (using the kubectl logs –f <podname> command as you’ve done previously) and repeat the final curl command to see that both instances are now able to see all valid tokens and correctly serve responses:

curl -i -b cookie \

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

Yes, it really is that simple. Sure, you’ll have to be deliberate about making your apps stateless, perhaps breaking old habits, but the pattern is straightforward, and the resultant benefits immense. A limited few of you will be working on the more difficult problems of managing distributed, stateful services, but the majority of us (myself included) can easily make our apps stateless and simply take advantage of the hard work and innovation happening in the realm of cloud-native stateful services.

But our application topology is now more complex, and with the greater distribution comes several challenges. What happens when you need to move your stateful service to new coordinates? Say it gets a new URL, or you need to update the credentials you’re using to connect to it (something we haven’t even touched upon yet). What happens when access to the stateful service is disrupted, even for a moment? We’ll address these challenges and more as we move further through the material in the book. Next up, you’ll learn how to handle application configuration in such a way that you can easily adapt to the changing nature of your cloud and application requirements.

**Summary**

* Stateful applications don’t work well in a cloud-native context.
* A sequence of interactions with a user, thought of logically as captured in session state, is a common way for state to creep into your applications.
* Stateful services are a special type of service that must address the significant challenges of data resilience in a distributed, cloud-based setting.
* Most apps should be stateless and should offload the handling of state to these stateful services.
* Making apps stateless is simple and, when done, realizes significant advantages in a cloud setting.