18

Microservices with Docker and Kubernetes

This chapter covers

- Running Camel on Docker
- Getting started with Kubernetes
- Running and debugging Camel on Kubernetes
- Understanding essential Kubernetes concepts
- Building resilient microservices on Kubernetes
- Testing Camel microservices on Kubernetes
- Introducing fabric8, Kubernetes Helm, and OpenShift

If you've been developing Java applications for many years, the transition to running them on the cloud is a big leap. You not only need to learn and master new concepts and primitives from the container-based world, you also make an architectural change from a monolithic to a microservice style.

As a developer, this can be a daunting mountain to climb, so in this chapter we'll help you focus on one thing at a time and climb that mountain step by step. We'll be in the developer role and show you how to develop, build, and run Camel microservices on Docker and Kubernetes all running locally on your computer.

You aren't required to use any on-premises cluster infrastructure or sign up with an online cloud provider. As developers, we feel comfortable if we have control and can do it all from our own computers. You don't have to worry about whether your computer is powerful enough, because what we'll do can run on any reasonable computer. At the time of this writing, the author of this chapter is using a four-year-old MacBook Air equipped with only 8 GB of memory and a mediocre CPU.

Docker came out a number of years ago as an elegant solution for packaging and running applications. Section 18.1 starts with Docker; we'll show you how to take any Java project and build it as Docker images and run it using Docker containers.

Docker is so popular that you may have heard that it solves all your problems. Although Docker provides a great portable container format for shipping applications between environments and allows you to run these applications anywhere, it's just one part of a bigger puzzle. When you move beyond the single-container phase and run your applications in a cluster of nodes, you need something more. You need powerful clustering facilities such as service discovery, load balancing, fault-tolerance, container scheduling, rolling deployment to minimize downtime, mounting persistent volumes, and configuration management. That's where Kubernetes enters the stage. It has all of that, and it's the third-generation container platform from Google, which brings tons of wisdom about running containers reliably at scale.

When getting started with microservices built and deployed in Docker and managed by Kubernetes, it helps to be running a local environment that's used for development purposes. In section 18.2, you'll learn how to set up a local Kubernetes cluster. In section 18.3, you'll get down to action and run Camel microservices in the local Kubernetes cluster.

Section 18.4 covers essential Kubernetes concepts that you should learn in order to be familiar with container-based platforms. These concepts are universal, so if you find yourself using an alternative cloud or container platform, you can apply the information from this section to your situation as well.

A distributed system such as a microservices architecture will affect developers who now must even more rigorously build their applications with fault-tolerance in mind. Section 18.5 shows what can go wrong when you don't do so. We'll play dirty and cause chaos in the local cluster by killing containers so you can see what happens. With this insight and practical experience, you'll learn how to develop your Camel applications to be resilient and fault-tolerant. You'll also learn to use what Kubernetes brings to the table in this matter via support for self-healing. Section 18.6 covers the basics of writing and running unit tests with Kubernetes.

We end this chapter by stepping beyond Kubernetes to briefly cover three projects: fabric8, Helm, and OpenShift, which all bring different value to Kubernetes users. Let's start with the little blue whale known as Docker.

18.1 Getting started with Camel on Docker

Docker, Docker, Docker. Unless you've been living under a rock for the last couple of years, you've likely heard of Docker. You may even have become tired of hearing that it's the best thing since sliced bread and that it'll change everything.

We've certainly had our dose of Docker and therefore won't spend too much time introducing you to Docker and explaining its installation and use. Docker is a key component in Kubernetes and therefore also in the way we build, package, and deploy our applications in clustered platforms—maybe already today and certainly in the near future. With this in mind, we think it's important to show you how to build and run your Java applications (and Camel) with Docker and Kubernetes.

We'll do this by developing two small microservices that use Camel and run on two popular microservices frameworks: Spring Boot and WildFly Swarm. The microservices we'll build, deploy, and run are illustrated in figure 18.1.

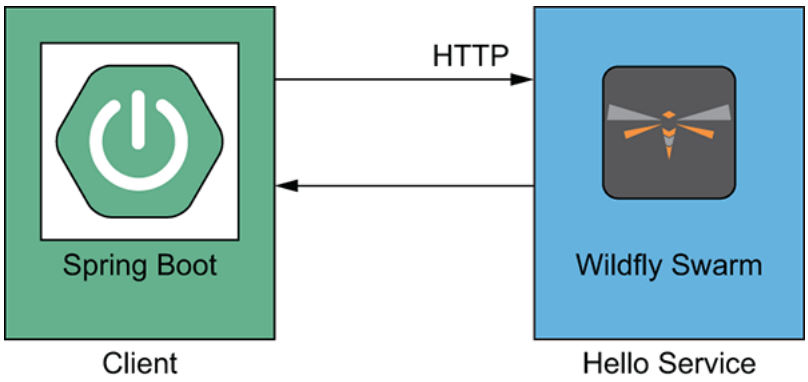

Figure 18.1 The microservice on the left runs on Spring Boot and is a client application that periodically calls the Hello service and logs its response. The microservice on the right runs using WildFly Swarm and hosts a Hello service returning a canned response.

This section and the following two sections will take you on a journey from developing the microservices and running them standalone, to using Docker and finally using Kubernetes. Metaphorically, we're climbing up a container mountain. The steps to the top of the Kubernetes mountain are as follows:

1. Develop the two microservices
2. Run the microservices standalone
3. Build the microservices as Docker images
4. Run the microservices using Docker

In the following sections (18.2 and 18.3) you'll continue using Kubernetes:

1. Install and run Kubernetes
2. Run the microservices using Kubernetes

Put on your climbing gear, because here goes.

18.1.1 Building and running Camel microservices locally

Chapter 7 covered building Camel microservices with WildFly Swarm and Spring Boot. To quickly build the microservices for this chapter, you'll use the same approach; you'll visit the websites http://start.spring.io and http://wildfly-swarm.io/generator.

In the interest of not repeating ourselves, the book's source code contains the ready-built microservices in the chapter18/standalone directory. The highlights are as follows.

WildFly Swarm includes a Camel route that can be written as one line:

```
from("undertow:http://localhost:8080/").bean(hello);
```

The called bean then returns a canned response that includes the hostname:

```
public String sayHello() throws Exception {
  return "Swarm says hello from " + InetAddressUtil.getLocalHostName();
}
```

The Spring Boot client is simple as well, as it's just a Camel route:

```
from("timer:foo?period=2000")
  .to("http4://localhost:8080")
  .log("${body}");
```

Running the example locally

You can now run this example locally by first starting the WildFly Swarm application:

```
cd hello-swarm
mvn wildfly-swarm:run
```

Then from another shell, start the client:

```
cd client-spring
mvn spring-boot:run
```

The client will then call the Hello service every other second and log the response:

```
[0 - timer://foo] route1 : Swarm says hello from davsclaus.air
[0 - timer://foo] route1 : Swarm says hello from davsclaus.air
```

When you run multiple Java JVMs on your computer, you may have had the problem that one JVM can't start because it wants to use a TCP port that's already taken by another running JVM. We'd have this problem in this example if we didn't avoid this deliberately.

Running Spring Boot without an embedded HTTP server

Spring Boot makes it easy to embed an HTTP server using the spring-boot-starter-web dependency. But for simple standalone clients, this may not be needed, so you can use the spring-boot-starter dependency instead. Ironically, doing so makes it a bit harder to keep Spring Boot running. But Camel makes this easy by turning on the run controller in the application.properties file:

```
camel.springboot.main-run-controller=true
```

Spring Boot will now keep running until the JVM is terminated.

SPRING BOOT WITH EMBEDDED HTTP SERVER AND HEALTH CHECKS

Running Spring Boot without an embedded HTTP server is recommended in only some situations, such as for development purposes or for running as tiny a JVM as possible without having to include an HTTP server. On the other hand, when running Spring Boot in a cloud platform such as Kubernetes, it's recommended to include an embedded HTTP server with Spring Boot to more easily facilitate health checks, which help to achieve robust and highly available applications.

You're now ready to climb the next step up the Kubernetes mountain.

18.1.2 Building and running Camel microservices using Docker

Docker and Kubernetes run applications as containers that are loaded from Docker images. How do you build a Docker image? That's a good question; building a Docker image is easy and hard at the same time.

It's easy because the Docker images are defined using a Dockerfile, which you can easily write on your own. A Dockerfile is just a text file with that very same name. For example, the Dockerfile you'll use to run the Spring Boot client microservice contains just three lines of text:

```
FROM openjdk:latest
COPY maven /maven/
CMD java -jar maven/spring-docker-2.0.0.jar
```

A Docker image is a compressed TAR file that includes the Dockerfile in the root alongside other files you want to include in the image. The Spring Boot Docker image consists of only two files:

```
maven/spring-docker-2.0.0.jar
Dockerfile
```

This sounds easy, so why is building a Docker image also hard? Well, if the IT industry were simple, you wouldn't need to buy and read a 900-page book to know everything about Apache Camel. And with Docker im-

ages, things get more complicated as you need to set up user permissions, set environment variables, mount volumes, map network ports, run commands, and much more.

Luckily, some great tooling can assist you, especially for Maven-based Java projects. To build a Docker image, you should use the Docker Maven plugin from fabric8.

Fabric8 Docker Maven plugin

The fabric8 project has great tooling for Java developers to work with Docker and Kubernetes. To build Docker images, you'll use docker-maven-plugin by adding the plugin to the two microservices. The Spring Boot client is the easiest, as all you have to do is add the code in the following listing.

Listing 18.1 Building a Docker image for the Spring Boot microservice

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.23.0</version>
  <configuration>
    <images>
      <image>
    <name>camelinaction/spring-docker:latest</name>    (1)
```

(1) Name of the Docker image

```
        <build>
    <from>openjdk:latest</from>    (2)
```

(2) Base Docker image

```
          <assembly>
      <descriptorRef>artifact</descriptorRef>    (3)
```

❸

What to include in the Docker image

```
          </assembly>
<cmd>
 java -jar maven/${project.artifactId}-${project.version}.jar    ❹
```

❹

Command to start the application

```
</cmd>
        </build>
      </image>
    </images>
  </configuration>
</plugin>
```

A Docker image must have a name in this syntax:

```
organization/name:version
```

In this example, we use `camelinaction` as the organization and `spring-docker` as the name, and `latest` as the version ❶.

USING PLACEHOLDERS IN A DOCKER IMAGE NAME

The Docker Maven plugin supports defining the Docker image name with placeholders, as shown here:

```
<name>%g/%a:%l</name>
```

This uses the Maven group ID as the organization, Maven artifact ID as the name, and the Maven version as the version (unless the version is SNAPSHOT, and then `latest` is used). You can find more details in the docker-maven-plugin documentation: https://dmp.fabric8.io/#image-configuration.

Docker images are layered, and you need to specify the parent image, which in our example must include Java, so you choose `latest` with `openjdk` ❷. The assembly configures which files to include in the Docker image. Because you're using Spring Boot as a fat JAR, you need to include only one file, which is the artifact built ❸. And finally, you specify how to run the application in the run configuration ❹. You can now build the Docker image by running the following:

```
cd chapter18/docker/client-spring
mvn install docker:build
```

This builds the Docker image and pushes it to the local Docker image repository. You should be able to see the built image by running the `docker images` command:

```
$ docker images
REPOSITORY                    TAG     IMAGE ID      CREATED         SIZE
camelinaction/spring-docker   latest  a0ba61b85c6f  8 minutes ago   657 M
```

WHY ARE DOCKER IMAGES SO LARGE?

If you look at the output from the `docker images` command in the preceding example, it shows that the camelinaction/spring-docker image is 657 MB. Docker images are layered, and an image includes all the parent layers and their dependencies. In this case, the base image is openjdk, which includes a Linux distribution and Java that end up taking up the vast majority of the size.

There's no point in running the Spring Boot client yet, because you need to build and run the WildFly Swarm microservice first.

18.1.3 Building a Docker image using the Docker Maven plugin

You'll also use docker-maven-plugin to build the WildFly Swarm microservice. This requires a bit more work than with Spring Boot, as shown in the following listing.

Listing 18.2 Building a Docker image for the WildFly Swarm microservice

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.23.0</version>
  <configuration>
    <images>
      <image>
    <name>camelinaction/helloswarm-docker</name>    1
```

1

Name of Docker image

```
        <build>
    <from>openjdk:latest</from>    2
```

2

Base Docker image

```
          <assembly>
      <inline>    3
```

3

What to include in the Docker image

```
        <fileSets>    3
         <fileSet>    3
          <directory>target</directory>    3
          <outputDirectory>.</outputDirectory>    3
          <includes>    3
           <include>*-swarm.jar</include>    3
          </includes>    3
         </fileSet>    3
        </fileSets>    3
      </inline>    3
          </assembly>
```

```
    <cmd>
      java -jar maven/${project.artifactId}-${project.version}-swarm.jar    ❹
```

❹

Command to start the application

```
    </cmd>
          </build>
          <run>
            <ports>
        <port>8080:8080</port>    ❺
```

❺

Network port mappings

```
            </ports>
          </run>
        </image>
      </images>
    </configuration>
  </plugin>
```

As with Spring Boot, you define the name of your Docker image ❶. Because you're using Java, you start off with the openjdk image ❷. When using WildFly Swarm with Docker, you want to run it as a fat JAR (or fat WAR). Because the wildfly-swarm Maven plugin generates the fat JAR with `-swarm` as its suffix, you need extra configuration to work around that. What you can do is inline the assembly and use file sets to include the correct fat JAR ❸. The fat JAR can then easily be run from Java using `java -jar` as specified in the `run` command ❹. Since the WildFly Swarm microservice is hosting an HTTP service, you need to expose this in the Docker port mappings ❺. The syntax is `external-port:internal-port`. People usually map port numbers using the same internal and external port numbers, so you typically see `2181:2181, 8080:8080`, and so forth. With this port mapping, you can let the Spring Boot microservice call the Hello service within the WildFly Swarm application on port 8080.

The Docker image can be built by running the following:

```
cd chapter18/docker/hello-swarm
mvn install docker:build
```

When the image is built, you can see it from the local Docker image repository:

```
$ docker images
REPOSITORY                        TAG     IMAGE ID      CREATED         SIZ
camelinaction/helloswarm-docker   latest  a0d46b208134  7 minutes ago   755
```

Okay, you should now be ready to run this example using Docker.

18.1.4 Running Java microservices on Docker

You first need to run the WildFly Swarm microservice that hosts the HTTP service the client will be calling. You can run this from a shell:

```
docker run -it -p 8080:8080 camelinaction/helloswarm-docker:latest
```

When running Docker applications, you can run them in the background or foreground. Well, that's not entirely true. You can run the application in interactive mode using `-it`, which means the shell will tail the logs and keep the container running until the shell is terminated with Ctrl-C. This makes it nice and easy for developers to run something and quickly stop, and then do code changes, rebuild, and run again.

Notice that you also specify a port mapping with `-p 8080:8080`. You might think you must do this because the Spring Boot client has to access the HTTP service on this port. But that's not exactly why. Using `-p 8080:8080` creates a port forwarding between the host operating system and Docker. This allows you to call the HTTP service from a web browser on `localhost:8080` as if the application were running natively on your host. Open a web browser and enter the following URL:

```
http://localhost:8080
```

You should see a response from WildFly Swarm such as this:

```
Swarm says hello from f1c5a96e250c
```

Now that WildFly Swarm is up and running, you're ready to start the Spring Boot client. You can do this by running the following Docker command:

```
docker run -it camelinaction/spring-docker:latest
```

The application starts up and calls the HTTP service, and all is good. Hey, wait, what's happening? The application fails, and a stacktrace is logged; why is that?

```
Message History
---------------------------------------------------------------------
RouteId   ProcessorId  Processor                    Elapsed (ms)
[route1]  [route1]     [timer://foo?period=2000    ] [        69]
[route1]  [to1    ]    [http4://localhost:8080     ] [        63]

Stacktrace
---------------------------------------------------------------------
org.apache.http.conn.HttpHostConnectException: Connect to localhost:8080
[localhost/127.0.0.1, localhost/0:0:0:0:0:0:0:1] failed: Connection refu
(Connection refused)
```

The client can't connect to localhost:8080, but you just tried that from a web browser. You called http://localhost:8080 and got back a response from WildFly Swarm. Why doesn't it work for the Spring Boot application?

The reason is related to the way Docker runs applications in isolated containers when each container is assigned its own IP address, as pictured in figure 18.2.

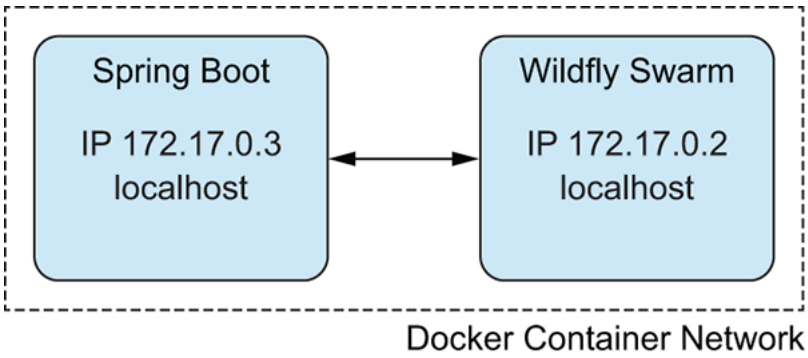

Figure 18.2 Each Docker container is isolated and has its own unique assigned IP address. The applications running inside the containers can't communicate with each other using localhost, but must use the IP address instead.

Communicating between Docker containers

Each Docker container that runs on the same Docker host runs in the same isolated Docker network. Each container gets a unique IP address assigned, and therefore the applications can connect over the network using this IP number. In figure 18.2, you can see that the Spring Boot container was assigned 172.17.0.3, and WildFly Swarm was assigned 172.17.0.2.

To fix the problem with running the Spring Boot client, you need to change the Camel route to use 172.17.0.2 instead of localhost:

```
.to("http4://172.17.0.2:8080")
```

We don't like hardcoding an IP address, so let's use a property placeholder:

```
.to("http4://{{swarm.ip}}:8080")
```

You can then specify the address in the Spring Boot application.properties file:

```
swarm.ip=172.17.0.2
```

That's a better practice.

Okay, but how do you know the IP address that Docker assigned? Frankly, you don't know. That has a lot to do with the way you run Docker and the way you may have set up Docker Container Networking. But by default, Docker assigns IP addresses from 172.17.0.2 upward. When you started

WildFly Swarm first, it was assigned the first free number, which is 172.17.0.2. But then to be sure, you need to inspect the running container to find its IP address, which you can do as follows:

```
$ docker ps
CONTAINER ID        IMAGE
f1c5a96e250c        camelinaction/helloswarm-docker:latest
```

To find the container ID of the running container, which you can then inspect, you run this:

```
docker inspect f1c5a96e250c
```

This outputs a lot of information, but within that you can find the IP address:

```
 "IPAddress": "172.17.0.2",
```

Now that you have the correct IP address and have updated the Spring Boot source code, you can rebuild and run the application again:

```
mvn clean install docker:build
```

Then run the application using Docker:

```
docker run -it camelinaction/spring-docker:latest
```

Because you're using the correct IP address, the console logs the response from WildFly Swarm:

```
route1 : Swarm says hello from f1c5a96e250c
route1 : Swarm says hello from f1c5a96e250c
route1 : Swarm says hello from f1c5a96e250c
```

Is there something wrong with the log? Why does it log such a weird name, f1c5a96e250c? There's nothing wrong with the log. When Docker runs a container, that container is assigned a unique ID that becomes the hostname.

To stop the application, you can press Ctrl-C.

JAVA INSIDE DOCKER: WHAT YOU MUST KNOW TO NOT FAIL

Running Java inside Docker containers does bring new problems. For example, Java may see that it's running on an operating system that has eight CPU cores, but in reality the Docker container has been resource-limited to one CPU core. Rafael Benevides posted a blog with good details: https://developers.redhat.com/blog/2017/03/14/java-inside-docker/.

Looking inside a running container

When running Docker containers, you need from time to time to be able to explore inside to see what's going on. You can start a new process inside the existing running container, even a shell such as bash if bash is provided by the Docker image (typically, bash or sh is provided from a base image).

For example, to look inside the WildFly Swarm container, you need to find its container ID and then start the bash shell using `docker exec` as highlighted here:

```
$ docker ps
CONTAINER ID        IMAGE
ccb99db22935        camelinaction/spring-docker:latest
f1c5a96e250c        camelinaction/helloswarm-docker:latest
$ docker exec -it f1c5a96e250c bash
root@f1c5a96e250c:/# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0     4336   760 ?        Ss+  15:17   0:00 /bin/
 -c java -jar maven/helloswarm-docker-2.0.0-swarm.jar
root         5  0.4 20.3 3090760 416828 ?      Sl+  15:17   0:29 java
 -jar maven/helloswarm-docker-2.0.0-swarm.jar
```

To quit, you run the `exit` command.

This concludes our coverage of running Camel on Docker. We want to show you the transition from running Camel standalone, via Docker, to running Camel on a real container platform such as Kubernetes.

18.2 Getting started with Kubernetes

This section will teach you the basic climbing skills so you'll learn the basics before we let you attempt climbing the Kubernetes mountains. At first, we'll walk you through installing and running a local Kubernetes cluster. We'll show you two ways to deploy and run applications in Kubernetes. In this section, we'll use the Kubernetes command-line tooling so you're aware of its existence and can gain knowledge of how to use it. The command-line tool works for any kind of application you want to run, not only Java-based applications. In section 18.3, we'll take you back to Java and Camel and use the fabric8 Maven plugin, which makes it much easier for Java developers to build, run, and debug applications in Kubernetes.

After some practice with using Kubernetes, you'll learn in section 18.4 the theories behind the most important concepts and principles of Kubernetes. We could have done this in reverse order (theory before practice), but we want you to have fun, and we think you can best learn Kubernetes by seeing something running in action first.

Kubernetes (and Docker) are native Linux technologies, so they must run in a Linux operating system. You'll need to use something called Minikube (Minikube is a local Kubernetes cluster intended for development purposes). Minikube makes it easy to run a local single-node Kubernetes cluster on a Windows, Mac, or Linux computer.

Let's get started and install Minikube.

18.2.1 Installing Minikube

Minikube requires a virtualization driver to be preinstalled. The driver varies depending on your operating system. Mac users should install the xhyve driver. Windows users should use Hyper-V. Linux users can use KVM. You can find information and links for installing these drivers on the Minikube website: https://github.com/kubernetes/minikube. From this point forward, we assume that you've installed a virtualization driver.

To install and run Minikube, you can follow the instructions from the Minikube website. For example, to install on a Mac:

```
$ curl -Lo minikube https://storage.googleapis.com/minikube/releases/
v0.24.1/minikube-darwin-amd64 && chmod +x minikube &&
 sudo mv minikube /usr/local/bin/
```

After Minikube has been installed, you should be able to run the `version` command:

```
$ minikube version
minikube version: v0.24.1
```

Minikube is now ready to be started.

18.2.2 Starting Minikube

When starting Minikube for the first time, you can provide parameters to specify the amount of memory, CPU, and disk space to allocate. The defaults are reasonable, but for our little example you don't need as much resources, so start with the following parameters:

```
minikube start --cpus 2 --memory 2048 --disk-size 10g --vm-driver xhyve
```

The last parameter is important; it specifies which VM driver to use (see Minikube documentation for details). After the installation is complete, you can get the status of Minikube:

```
$ minikube status
minikubeVM: Running
localkube: Running
kubectl: Correctly Configured: pointing to minikube-vm at 192.168.64.2
```

This means the local Kubernetes cluster is up and running.

STARTING MINIKUBE FROM SCRATCH

The good thing about Minikube is that you shouldn't fear that it'll take over your computer. At any time, you can always delete and start a new cluster. For example, if a cluster is running, you first stop it:

```
minikube stop
```

Then you can delete the cluster:

```
minikube delete
```

And then you can run `start` again, to create a new cluster:

```
minikube start --cpus 2 --memory 2048 --disk-size 10g --vm-driver=xhyve
```

If you find your Minikube installation broken, you can always delete the ~/.minikube directory and start again, as Minikube will then re-download needed Docker images and reconfigure itself.

But how do you interact with the cluster?

Interacting with Kubernetes

Kubernetes provides a command-line tool called kubectl that you use to interact with the cluster. For example, you can deploy applications, scale deployments up or down, and a lot more.

You install kubectl by following the installation guidelines from the Kubernetes website: https://kubernetes.io/docs/tasks/tools/install-kubectl/.

After you've installed kubectl and it's working, you can run the `cluster-info` command:

```
$ kubectl cluster-info
Kubernetes master is running at https://192.168.64.2:8443
```

A-ha—there's something called a Kubernetes master. We'll get back to that in section 18.4.

Minikube also comes with a web dashboard, and you can obtain the URL as follows:

```
$ minikube dashboard --url
http://192.168.64.2:30000
```

Let's open a web browser and see this dashboard. You could copy/paste that URL, but instead you can type the command without the `--url` parameter:

```
minikube dashboard
```

The dashboard then opens in your web browser. Spend a few minutes clicking the dashboard, and you'll find out that there's not much to see. That's because no applications are running in the cluster. But something is running in the cluster: Kubernetes runs a core set of services that are part of what's called the *control pane*. Because Kubernetes runs containers, you can use the Docker CLI to see this.

Using Docker CLI with Kubernetes

A good idea when working with Kubernetes from the shell is to set up the Docker environment to point to the Kubernetes cluster. You can do this from Minikube:

```
$ minikube docker-env
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.64.2:2376"
export DOCKER_CERT_PATH="/Users/davsclaus/.minikube/certs"
export DOCKER_API_VERSION="1.23"
# Run this command to configure your shell:
# eval $(minikube docker-env)
```

This outputs the command to run:

```
eval $(minikube docker-env)
```

You can now use the Docker CLI to interact with the Kubernetes cluster. For example, to list all the running Docker containers, you type this:

```
docker ps
```

And to list which Docker images are in the Kubernetes Docker repository, you type this:

```
docker images
```

STARTING AND STOPPING MINIKUBE

At any time, you can stop Minikube as follows:

```
minikube stop
```

And then later, you can start Minikube again, and it'll resume where it left off:

```
minikube start
```

After all this, your local Kubernetes cluster is up and running and ready. Now it's time to run your first application in the cluster.

18.3 Running Camel and other applications in Kubernetes

When you run applications on Kubernetes, they run as containers loaded from Docker images. The information you learned in the previous section about running Camel on Docker is required knowledge for working with Kubernetes.

To deploy an application in Kubernetes, you need two things:

- Docker image
- Kubernetes manifest

The Kubernetes manifest is a file that holds metadata about how to set up and deploy the application. This manifest file can be in either YAML or JSON format and can be written by hand. But when applications become more complex, it's recommended to generate these manifests. You'll see

how to do that in the following section. But first let's start from nothing and try to run the WildFly Swarm example you built as a Docker image.

18.3.1 Running applications using kubectl

From a shell, you can run the WildFly Swarm application using the `run` command from kubectl. Before you can do this, you need to build the Docker image with the application, and you can do that by running the following commands from a shell (section 18.3.3 dives into more detail about building and deploying to Kubernetes using Maven):

```
eval $(minikube docker-env)
cd chapter18/kubernetes/hello-swarm-docker
mvn package fabric8:build
```

The `fabric8:build` command will build the Docker image and push that to the local Docker repository, which you can list using the Docker command line as follows:

```
$ docker images
REPOSITORY                              TAG  IMAGE ID      CREATED
camelinaction/helloswarm-kubernetes  2.0  34326d1f695e  2 minutes ago
```

Now you're ready to run the application using the kubectl tool:

```
$ kubectl run hello-world --replicas=1 --labels="foo=bar"
 --image=camelinaction/helloswarm-kubernetes:2.0 --port=8080
```

This command tells Kubernetes to run a new application with the name hello-world with one instance. The application has an associated label with its key-value as `foo=bar`. The application uses the Docker image from the camelinaction organization with the name helloswarm-kubernetes and 2.0 as the version. The last parameter is used for exposing a network connection on port 8080 to the running container.

KUBECTL WITH SHELL COMPLETION

We recommend that you install shell completion for the kubectl tool, which allows you to use tab completion. For example, to show the logs of a running pod, you can type the following from the command shell:

```
kubectl log he<TAB>
```

Instead of typing `<TAB>`, you press the Tab key. This completes the name of the pod:

```
kubectl log hello-world-1096927984-m0jpk
```

Otherwise, you'd have to type the name. You can find instructions on installing shell completion at the Kubernetes website: https://kubernetes.io/docs/tasks/tools/install-kubectl/#enabling-shell-autocompletion.

This application runs inside Kubernetes as a Docker container, in what Kubernetes calls a *pod* (pods will be covered in section 18.4). To list the running applications, you list the pods using `get pods`:

```
$ kubectl get pods
NAME                           READY     STATUS    RESTARTS   AGE
hello-world-1096927984-m0jpk   1/1       Running   0          12m
```

Here you can see that one pod is running.

18.3.2 Calling a service running inside a Kubernetes cluster

Okay, so far so good, but how do you try calling the HTTP service that the example provides? How do you, from your host operating system in a web browser, call an HTTP URL inside the Kubernetes cluster? That is an excellent question. By default, Kubernetes doesn't expose network ports outside the cluster. What you need to do is expose the network port 8080 as a service, which also can be accessed from outside the Kubernetes cluster. This can be done using the `expose` command:

```
kubectl expose deployment hello-world --type=NodePort --name=hello-servi
```

This command exposes a service on the hello-world deployment of type `NodePort`. Because you run the Kubernetes cluster using Minikube as a single-node cluster for development purposes, all you need is a way to forward network connections from the Kubernetes master to port 8080 on the running pod. Figure 18.3 illustrates the players involved as you call the service from your web browser.

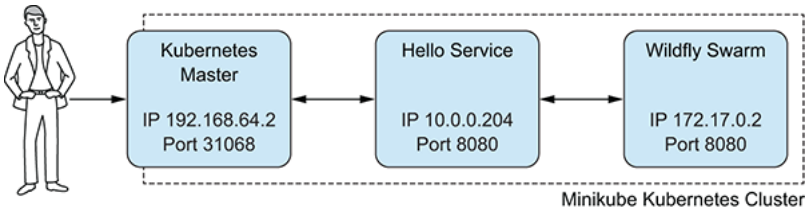

Figure 18.3 A user wants to call a service running inside the Kubernetes cluster. The service is accessible through node port 31068 on the Kubernetes master, which forwards the traffic to the Hello service running inside the cluster. The Hello service also forwards the traffic to the running pod with the targeted service being called.

Minikube is running locally on your computer as a virtual machine (VM). The VM has been assigned IP number 192.168.64.2, which is reachable from your host operating system. You can communicate with applications running inside the Minikube cluster with this IP number.

In Kubernetes, you use Kubernetes services as a level of indirection between your running pods and clients calling into these pods. (Section 18.4 goes more in depth about the Kubernetes concepts.) In figure 18.3, you can see that a Hello service sits between the Kubernetes master and the pod that runs the WildFly Swarm application. At this time, all you need to know is that by calling IP 192.168.64.2:31068, you'll end up being routed to the WildFly Swarm pod on IP 172.17.0.2:8080 inside the cluster. But wait a minute—how do you know that it's port 31068 that ends up being routed to the WildFly Swarm application? To determine this, you can use kubectl:

```
$ kubectl get service
NAME            CLUSTER-IP   EXTERNAL-IP    PORT(S)          AGE
hello-service   10.0.0.204   <nodes>        8080:31068/TCP   3h
```

You can then see the port number (highlighted here). But there's an easier way: you can ask Minikube to give you the URL to any service running in-

side the cluster. To get the URL to hello-service, you type this:

```
$ minikube service --url hello-service
http://192.168.64.2:31068
```

If you run the command without the `--url` parameter, the URL opens automatically in the browser. And you get this response:

```
Swarm says hello from hello-world-1096927984-90q0p
```

Kubernetes manifest file

The last piece we want to show you about running applications in Kubernetes is the manifest file. At the beginning of section 18.3, we mentioned that you need a Docker image and a Kubernetes manifest file to deploy an application. So far, you've deployed applications via the kubectl command, which is capable of *autogenerating* the manifest file from the information given.

At any given time, you can use the kubectl tool to output any of the resources in the cluster as a manifest file. If you run the following command, it outputs in YAML format what the Kubernetes manifest file would have to be if you were to type that by hand. The output is 58 lines, and here are the first 10 lines of the manifest file:

```
$ kubectl get deployment -o yaml hello-world
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  creationTimestamp: 2017-11-04T20:33:36Z
  generation: 1
  labels:
    foo: bar
  name: hello-world
```

If you were to write this by hand, you wouldn't need all 58 lines; some could be skipped. But you'd still end up having to write 20 or more lines,

which is tedious. Therefore, let's go back to being Java developers and use some nice tooling that can do all of this for us.

RUNNING ANY DOCKER IMAGE IN KUBERNETES

The kubectl run command can be used to run any Docker image as a container in the Kubernetes cluster. DockerHub is an online Docker image repository that hosts a lot of images. A beginner example is to run an Nginx container, which can be done as follows:

```
kubectl run my-nginx --image=nginx --port=80
```

Then you need to create a service to expose a network connection:

```
kubectl expose deployment my-nginx --type=NodePort --name=my-nginx-servi
```

This allows you to call the service from your host operating system:

```
minikube service --url my-nginx-service
http://192.168.64.2:30036
```

And you can then open a web browser and enter the URL: http://192.168.64.2:30036.

In the next section, we want to start all over again, so let's delete what we created. First, delete the deployment:

```
kubectl delete deployment hello-world
```

Then delete the service as well:

```
kubectl delete service hello-service
```

That's it. You're now ready to start again and run Java applications using Maven.

18.3.3 Running Java applications in Kubernetes using Maven tooling

Any Java project that's Maven-based can quickly be turned into being Kubernetes-ready in less time than it takes to brew a cup of coffee or tea.

You turned the Camel microservices in section 18.2 into Docker images with the help of the Docker Maven plugin. You could do the same thing now, but with the Kubernetes-ready fabric8 Maven plugin. You can quickly add this plugin to any Maven project by running the following command from the project directory:

```
cd chapter18/kubernetes/hello-swarm
mvn io.fabric8:fabric8-maven-plugin:3.5.33:setup
```

Here you run the setup goal of fabric8-maven-plugin. But pay attention to how you can refer to the plugin using full Maven coordinates. This is a Maven standard that allows you to run any Maven plugin if you specify the full Maven coordinates (`groupId:artifactId:version:goal`).

Running the setup goal will add fabric8-maven-plugin to the Maven pom.xml file, and from this point onward you can run the plugin using just this:

```
mvn fabric8:nameOfGoal
```

Now you can easily deploy this application into the running Kubernetes cluster. Before doing so, remember to set up the command shell to point to the Kubernetes master:

```
eval $(minikube docker-env)
```

And then you can deploy the application:

```
mvn fabric8:deploy
```

TIP You can use `mvn fabric8:undeploy` to uninstall a deployed application. You can also undeploy an application by deleting the deployment from the kubectl command line: `kubectl delete deployment nameOfDeployment`.

What happens now is that the plugin will scan your Maven project and then make a best effort to generate Kubernetes manifest files that fit the profile of your project. The plugin logs to the console what it has detected and what it's doing:

```
[INFO] F8: Building Docker image in Kubernetes mode
[INFO] F8: Running generator wildfly-swarm
[INFO] F8: wildfly-swarm: Using Docker image fabric8/java-jboss-openjdk8
-jdk:1.2 as base / builder
```

Here you can see it's detected that WildFly Swarm is being used, and will then adapt accordingly. You can then use the kubectl tool to see the status of the pods:

```
$ kubectl get pods
NAME                                      READY     STATUS    RESTARTS   A
helloswarm-kubernetes-2173069017-fpn7c    1/1       Running   0
```

You can see the pod is running, but you'd like to be sure and look inside the logs of the WildFly Swarm application. This can be done using kubectl:

```
kubectl logs helloswarm-kubernetes-2173069017-fpn7c
```

And you can use the `-f` option to follow the logs.

Another way to show the logs is to use fabric8-maven-plugin, as shown here:

```
mvn fabric8:log
```

This then dumps and follows the log of the running pod. To stop, press Ctrl-C.

Okay, so the WildFly Swarm application is running in the cluster. Now you want to call its service from your web browser. Yes, this is the same situation we illustrated previously in figure 18.3. What you need to do is configure the service to be a `NodePort` type.

NOTE Using the `NodePort` type works well for local development on Minikube. But in a real cluster, you should use the `LoadBalancer` type, which will let Kubernetes use any built-in load balancer from the underlying infrastructure or cloud provider.

This can be done by adding configuration to fabric8-maven-plugin in the pom.xml file. But there's an easier way. First, run this Maven goal:

```
mvn fabric8:resource
```

The plugin generates Kubernetes manifest files in the directory target/classes/META-INF/fabric8. You can then open the file kubernetes.yml and see all the glory of the Kubernetes manifest file. What you need to do is somehow merge in the details you want. The easiest way is to create a new file in your Maven project named src/main/fabric8/service.yml. The content of this file will then be merged into the service part of the Kubernetes manifest file. To enable `NodePort`, you add the following lines:

```
spec:
  type: NodePort
```

And then you can run `mvn fabric8:resource` to regenerate the Kubernetes manifest file. In target/classes/META-INF/fabric8, you can see whether your changes have been included.

Configuring NodePort in pom.xml

Instead of configuring the Kubernetes manifest in the src/main/fabric8/service.yml file, you can do most of the configuration from fabric8-maven-plugin in the pom.xml file. You may want to do this when you set only a smaller number of configurations, such as setting the

port type to be `NodePort`. But this requires a fair bit of XML, as you need to add the following lines:

```
<configuration>
  <enricher>
    <config>
      <fmp-service>
        <type>NodePort</type>
      </fmp-service>
    </config>
  </enricher>
</configuration>
```

What's best to use? Using the YAML file, you configure the Kubernetes manifest directly, whereas using pom.xml, you're using an XML configuration that resembles the Kubernetes manifest structure but isn't exactly the same. At the end of the day, it's good to familiarize yourself with the Kubernetes manifest structure, so using the YAML files is likely better.

Calling the service

To try calling the service, you need to get the service URL from Minikube:

```
$ minikube service --url helloswarm-kubernetes
http://192.168.64.2:30102
```

And then type the URL in the web browser:

```
http://192.168.64.2:30102
```

This returns a response from the running WildFly Swarm application:

```
Swarm says hello from helloswarm-kubernetes-2176083623-fshwg
```

You can also let Minikube open your web browser and call the service if you omit the `--url` parameter:

```
minikube service helloswarm-kubernetes
```

Now let's build and run the Spring Boot client that calls this service. This should be easy, right? It is—but there's always something lurking.

18.3.4 Java microservices calling each other in the cluster

What you want to do now is deploy and run the Spring Boot client microservice that should call the existing WildFly Swarm microservice. One microservice will call the other microservice, with both of them running in the Kubernetes cluster. As you may remember from section 18.1.4, when you were using Docker containers to call each other, you had to change the hostname from localhost to the IP address of the WildFly Swarm container. In Kubernetes, you use Kubernetes services when you want to connect to services—for example, other applications—over the network. Figure 18.4 illustrates this principle.

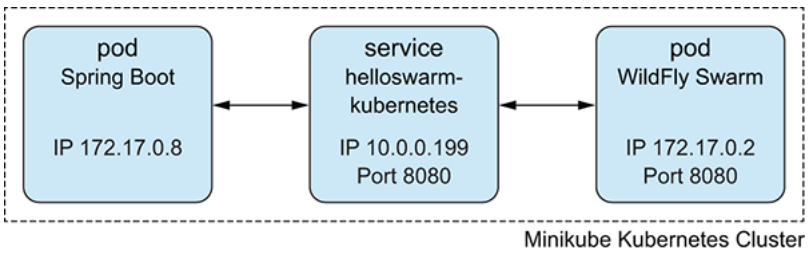

Figure 18.4 The Spring Boot microservice wants to call the Hello service from the WildFly Swarm microservice. In Kubernetes, you use services to accommodate this. The service sits between the clients and the pods hosting the service and acts as local gateway and load balancer to direct traffic to running pods that are ready to service the traffic.

What does this mean for us as Java developers? It means that every time we want to call a microservice in the cluster, we have to use a Kubernetes service. At this time, you can think of a Kubernetes service as a level of indirection that sits between you and the service. In figure 18.4, you can see the service in the middle.

Camel makes it easy to call a Kubernetes service. Here's the Camel route that would call the service:

```
from("timer:foo?period=2000")
  .to("netty4-http://localhost:8080")
  .log("${body}");
```

You need to change this to use the IP address and port of the service:

```
from("timer:foo?period=2000")
  .to("netty4-http://10.0.0.199:8080")
```

```
    .log("${body}");
```

But how do you know the IP address? As with Docker, you can use command-line tooling to retrieve such details:

```
$ kubectl describe service helloswarm-kubernetes
Name:                   helloswarm-kubernetes
Type:                   NodePort
IP:                     10.0.0.199
Port:                   http    8080/TCP
NodePort:       http    30102/TCP
```

Here you can see that the IP address is 10.0.199 and the port number is 8080. After this code change, you could build and run this application in the cluster as follows:

```
cd chapter18/kubernetes/client-spring
mvn fabric8:deploy
```

But now you hardcode an IP address and port number in the source code, and isn't this bad? Well, yes and no. The brilliance of a Kubernetes service is that it has a static IP address and port number for the entire lifecycle of the service. On the other hand, pods are temporary and can come and go, and each time the pod may have a different IP address.

But let's get rid of the IP address, as Camel makes this easy. You can refer to a Kubernetes service in Camel using the `{{service:name}}` syntax. The route can be changed to the following:

```
from("timer:foo?period=2000")
  .to("netty4-http://{{service:helloswarm-kubernetes}}")
  .log("${body}");
```

With Camel, you have another choice instead of using `{{service:name}}`: you can use the Service Call EIP and write the route as follows:

```
from("timer:foo?period=2000")
  .serviceCall("helloswarm-kubernetes")
```

```
    .log("${body}");
```

The Service Call EIP works the same as `{{service:name}}`, but the EIP makes it stand out in the route with a strong indication that it's calling a service. The book's source code contains an example of using the Service Call EIP, which you can find in the chapter18/kubernetes/client-spring-servicecall directory. The preferred Kubernetes practice is to use DNS. The preceding example can be written as follows:

```
from("timer:foo?period=2000")
  .to("netty4-http://helloswarm-kubernetes:8080")
  .log("${body}");
```

When using the DNS name, you must remember to use the port number as well. Because you're not using the default HTTP port number 80, you have to specify the port number 8080. In the earlier versions of Kubernetes, DNS wasn't supported, so your only choice was to use `{{service:name}}` or the Service Call EIP. At the time of this writing, DNS is so well supported that it's the recommended choice.

Okay, now we're happy, so let's run the client.

TIP Section 18.4 goes into more depth about Kubernetes services as well as other Kubernetes concepts.

But let's show you another feature of fabric8-maven-plugin that we like. You can use the run goal to make it appear that you're running your current Java project locally in the foreground:

```
cd chapter18/kubernetes/client-spring
mvn fabric8:run
```

What happens now is the same as fabric8:deploy, but the plugin will appear to run locally by running in the foreground and following the log. From the logs, you should see that the client is able to call the service:

```
Swarm says hello from helloswarm-kubernetes-171471280-d8r0k
Swarm says hello from helloswarm-kubernetes-171471280-d8r0k
```

```
Swarm says hello from helloswarm-kubernetes-171471280-d8r0k
```

Press Ctrl-C to stop the plugin, which undeploys the application.

Open the Kubernetes web console:

```
$ minikube dashboard
```

Then you can see your running pods and more information. Spend a couple minutes browsing the console, shown in figure 18.5.

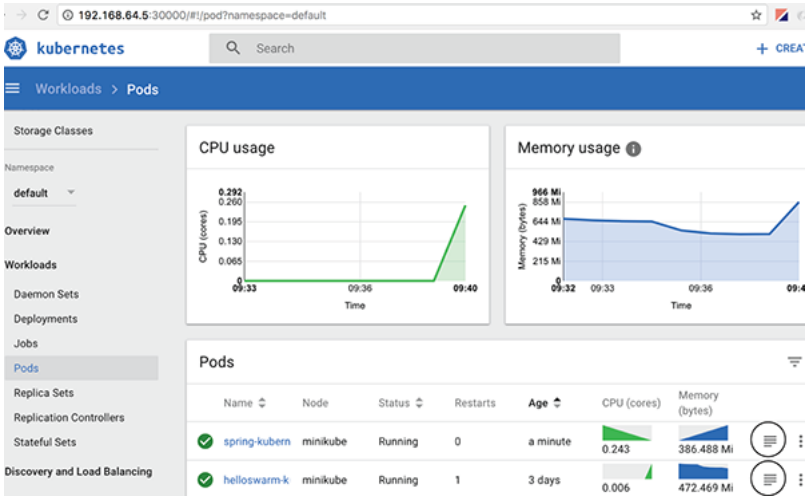

Figure 18.5 The Kubernetes web console shows the running pods. From the console, you can access the logs of the pods by clicking the second-to-last button (marked with a circle) on the right side. The two diagrams at the top show the cluster-wide CPU and memory usages. Likewise, you can see the CPU and memory usage for each pod as well in the tiny graphs shown in the table.

TIP To display CPU and memory usage graphs in the dashboard, you must first install the heapster add-on in Minikube.

As a Java developer, you might worry that having to run your Java applications in the Kubernetes environment means your life will become more problematic. For example, traditional Java applications can easily be started or debugged directly from the Java editor. Is that even possible with Kubernetes, and if so, how can you do that?

18.3.5 Debugging Java applications in Kubernetes

You can debug your Java applications running in the Kubernetes cluster, even if the cluster isn't running locally with Minikube. Imagine that you suspect that the WildFly Swarm microservice has a bug, and you want to debug the microservice while it runs in the Kubernetes cluster.

You can do this with help from the fabric8 Maven plugin, using its debug goal. But before you can use this, you must have an existing running pod.

Debugging a running pod

If you haven't deployed the WildFly Swarm application, you can do so with the deploy goal:

```
cd chapter18/kubernetes/hello-swarm
mvn fabric8:deploy
```

Now imagine that when you call the Hello service, it returns a response that we suspect is caused by a bug:

```
$ curl http://192.168.64.2:30102
Swarm says hello from helloswarm-kubernetes-3397218272-pptf4
```

You now want to debug the running pod with the WildFly Swarm microservice. This is done by running the debug goal of the fabric8 Maven plugin:

```
cd chapter18/kubernetes/hello-swarm
mvn fabric8:debug
```

What happens next is that the running pod is enabled in debug mode, which may take a little while. The plugin logs in the command shell what's happening:

```
[INFO] F8: Enabling debug on Deployment helloswarm-kubernetes
[INFO] F8: Waiting for debug pod with selector
 LabelSelector(matchExpressions=[], matchLabels={project=helloswarm
-kubernetes, provider=fabric8, version=2.0-SNAPSHOT,
 group=com.camelinaction}, additionalProperties={}) and $JAVA_ENABLE_DEB
 = true
[INFO] F8:[W] helloswarm-kubernetes-3781014250-cs2fl status: Pending
[INFO] F8:[W] helloswarm-kubernetes-3397218272-pptf4 status: Running Rea
[INFO] F8:[W] helloswarm-kubernetes-3781014250-cs2fl status: Pending
[INFO] F8:[W] helloswarm-kubernetes-3781014250-cs2fl status: Running Rea
[INFO] F8: Port forwarding to port 5005 on pod helloswarm-kubernetes
-3781014250-cs2fl using command /usr/local/bin/kubectl
[INFO] F8:
```

```
[INFO] F8: Now you can start a Remote debug execution in your IDE using
 localhost and the debug port 5005
[INFO] F8:
[INFO] F8:kubectl Forwarding from 127.0.0.1:5005 -> 5005
[INFO] F8:kubectl Forwarding from [::1]:5005 -> 5005
```

The pod has been restarted with the `JAVA_ENABLE_DEBUG` environment variable set to `true`. All the Java base images from fabric8 or JBoss support Java debugging controlled by this environment variable. They start Java with remote debugging enabled on port 5005. To make it possible to remotely debug from your host operating system, port forwarding from Kubernetes has been enabled on port 5005 as well. In other words, you can do a Java remote debug on localhost:5005, and it'll be network connected to the Kubernetes cluster and the running pod. From your Java editor such as IDEA, you can then do a remote Java debug on port 5005 and set a breakpoint in the source code, as shown in figure 18.6.

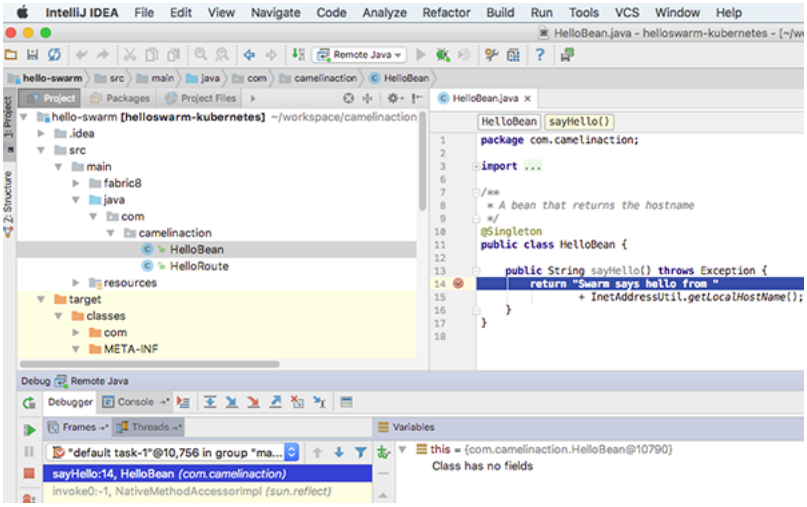

Figure 18.6 Remote debugging Java from an IDEA editor. The Java application runs in a pod in the Kubernetes cluster, which you've network connected using the fabric8-maven-plugin:debug goal to make this easy.

From the Java editor, you can then inspect what's happening, and even change the response message. For example:

```
$ curl http://192.168.64.2:30102
I changed this in the editor
```

This is awesome. You can call the service again, and the remote debugger is activated again. To stop the debugger, press Ctrl-C in the command shell where the fabric8-maven-plugin:debug is running.

TIP To speed up the debugging, you can deploy the pod in debug mode from the beginning:

```
mvn fabric8:deploy -Dfabric8.debug.enabled=true
```

Calling a service in Kubernetes keeps changing the port number

When calling a service from your host operating system into the Kubernetes cluster, you have to expose the service using the `NodePort` type. The exposed service may be re-created when the pod is redeployed or debugged. This can be annoying, because when you attempt to call the service with the old port number, it will fail:

```
$ curl http://192.168.64.2:30102
curl: (7) Failed to connect to 192.168.64.2 port 30102: Connection refus
```

Here's a trick for avoiding this. You can specify the fabric8 Maven plugin to avoid triggering a service change in Kubernetes by setting an ignore option when debugging:

```
mvn fabric8:debug -Dfabric8.deploy.ignoreServices=true
```

It also works for re-deploying as well:

```
mvn fabric8:deploy -Dfabric8.deploy.ignoreServices=true
```

You've now climbed to the top of the Kubernetes mountain, and that's enough action for today. Let's take a moment to reflect on what you've achieved.

Recap

At the beginning of this chapter, you set out on a course to build two small Java (with Camel) microservices and make them deploy in a local Kubernetes cluster, all running on your local computer. You did this in three steps. First, you built the microservices and ran them standalone. Then you moved on to add Docker into the mix; you built the microservices as Docker images, which you could run as Docker containers. Then

Kubernetes entered the stage, and you first had to install a local Kubernetes cluster called Minikube.

You learned a few tips and tricks for using the Kubernetes command-line tool (kubectl) to deploy and manage applications in the cluster. Then you returned to your Java microservices, and with the help from the fabric8-maven-plugin, you could easily build and run your microservices in the cluster. You also learned how to call from your host operating system into Kubernetes, to call services running inside the cluster. Java applications running in the cluster can also be debugged easily with the fabric8:debug goal, which you learned how to use.

Speaking of Kubernetes, let's go over the most essential concepts it brings to the table. You are now half way through this chapter, and it may be a good time for a little break.

18.4 Understanding Kubernetes

We've already had our fun with Kubernetes in the previous section, where we were running a single-node cluster using Minikube. We hope that with some practical experience, you're now better suited for what's coming in this section. First, we'll explain some of the reasons that container-managed platforms such as Kubernetes are becoming increasingly needed by businesses.

To become skilled with Kubernetes, you'll go over its essential concepts and architecture. With this information, you can then jump back into action in the following sections to scale containers and provoke failures and learn how to make Camel microservices truly cloud native.

18.4.1 Introducing Kubernetes

Today we're on the verge of changing the way we build, package, and run our applications. Traditionally, we've built monolithic applications that run on application servers, together with other applications within the same JVM (OS process). We're heading toward breaking up these big monoliths into smaller pieces known today as *microservices*. Because these microservices are decoupled from each other, we can develop, build, deploy, run, and scale them individually. From the business side of

things, we'll be able to deliver business values faster in order to keep up with today's rapidly changing business requirements.

But with a growing number of applications to be deployed, it becomes increasingly more difficult to configure, manage, and keep the infrastructure up and running. Businesses are also always on the lookout for ways to reduce costs, and one way is to better use resources in their data centers. Managing all this complexity isn't sustainable with manual procedures and labor. We need a high degree of automation and an infrastructure that's self-managed and capable of orchestrating, scheduling, configuring, handling failures in, and supervising the platform as a whole. The answer to that is Kubernetes.

Kubernetes enables developers to deploy applications themselves without requiring assistance from system administrators. The lives of system administrators are also improved as they shift from focusing on managing individual applications to instead managing Kubernetes and the rest of the infrastructure.

Kubernetes abstracts away all the hardware complexity and exposes your data centers as a unified computing resource platform. It allows you to deploy and run applications without having to worry about the hardware and servers underneath. When you deploy applications on Kubernetes, you let Kubernetes orchestrate and schedule which servers each individual component of the applications is running on. Communication between services and applications in the cluster is made easy by Kubernetes, which makes it an ideal platform for distributed applications. Kubernetes can easily scale applications up and down as needed without affecting the network communication between applications and services. For example, Kubernetes makes it possible to move workloads between nodes without affecting applications or downtime of services.

Kubernetes does all that and does it so well that it's becoming the de facto standard for running distributed applications in both the cloud and on on-premises infrastructure.

Where did Kubernetes come from?

The short answer is Google. Google has been running containers at scale in its data centers for more than 10 years. And Google has become so

good at doing this that everything in Google runs in containers. Google has built several container-management platforms over time, and Kubernetes is its last and best effort. It's based on all of Google's knowledge and experience.

In 2014, Google decided to open source its next-generation successor to Borg (its current platform), named Kubernetes. Since then, Kubernetes has become a large, open, and vibrant community with contributors from many IT vendors such as Google, Microsoft, Intel, IBM, CoreOS, Red Hat, and many more. To ensure an even playing field, Google transferred Kubernetes to the Cloud Native Computing Foundation, where the project is governed by a board comprising various stakeholders and companies. The success of Kubernetes has skyrocketed, and it's also popular in the developer community, where it's the number one project on GitHub.

Are you ready to learn about the high-level architecture of Kubernetes? Read on.

18.4.2 Kubernetes architecture

A high-level, 10,000-feet diagram of Kubernetes is sketched in figure 18.7.

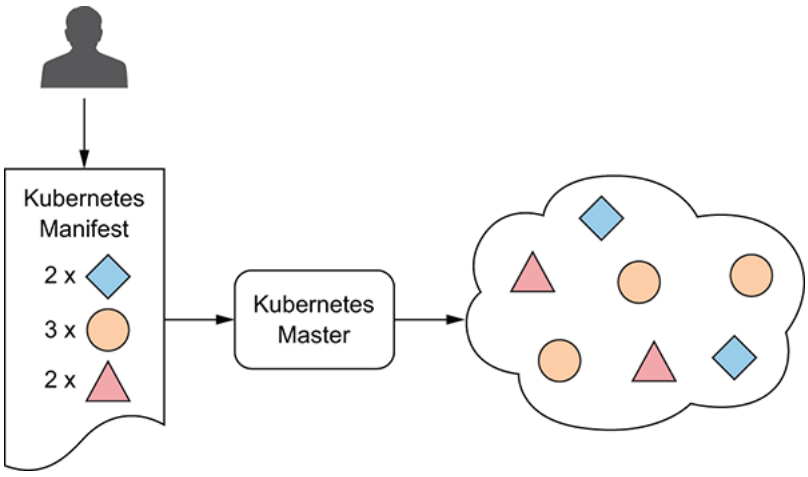

Figure 18.7 The cloud shape represents the worker nodes that Kubernetes exposes as a single container-based platform. The Kubernetes master orchestrates and supervises the platform. Users can deploy applications by applying Kubernetes manifest files to the Kubernetes master.

The Kubernetes platform includes a master node (in reality you would run multiple instances of the master node to make the cluster highly available) that orchestrates numerous work nodes (represented as the cloud). When a developer submits a list of applications (in a Kubernetes manifest file) to deploy to the master, Kubernetes then deploys these applications in the cluster of worker nodes. It doesn't matter which worker

node an application is deployed on. It's all decided by Kubernetes, which has built-in intelligence to orchestrate the platform in the best possible way.

Declarative state

After the applications are running, Kubernetes will continuously make sure that the platform runs the desired number of applications, as described in the Kubernetes manifest files. For example, in figure 18.7, three applications are set to run two, three, and two instances, respectively. If one of those instances stops for any reason (for example, the process crashes or becomes unresponsive), Kubernetes will restart it automatically. Likewise, if a worker node dies or becomes inaccessible, Kubernetes will reschedule the applications to run on the remaining working nodes.

This notion of declarative state is a key principle of Kubernetes. We as users should just specify what the desired state should be and leave it up to Kubernetes to ensure that the cluster runs accordingly.

When working with Kubernetes, you should familiarize yourself with the concepts that follow.

18.4.3 Essential Kubernetes concepts

This section covers essential Kubernetes concepts that we think are the most beneficial:

- Pods
- Labels
- Replication controllers and deployments
- Services

Pods

A *pod* is a group of one or more Docker containers (like a pod of whales). A typical pod has only one Docker container. When you need two or more containers (known as the *side-car* or *ambassador* pattern), a pod is the way to group them together.

A pod is the minimal deployment unit in Kubernetes, and pods are orchestrated, scheduled, and managed by Kubernetes. When we refer to an

application running in Kubernetes, it's running inside a pod as a Docker container. A pod is given its own IP address, and all containers within the pod share this IP as well as the same local network and filesystem. Likewise, if persistent volumes are mounted to a pod, they're also shared among the containers running in the pod.

An important detail to know about pods is that they're ephemeral (temporary). Pods can come and go in the cluster—for example, pods can be scaled down, or containers within the pod may crash. An application can run in a pod with a given IP address, and then because of a crash, the application can be started on another pod with a new IP address. This falls in line with the world of microservices and distributed systems, where things will fail, so you're strongly encouraged to write your applications with this concept in mind.

Labels

So far in this chapter, you've been running a few pods in your local Minikube cluster. When running Kubernetes for real, you'll end up with many more pods running in the cluster. You need a way of being able to organize and group these pods, and the answer to this is labels.

Labels are just key-value pairs that can be assigned to pods—for example, `tier=frontend` or `tier=backend`. A pod can also have multiple labels, such as `tier=backend,app=inventory`.

After labeling your pods, you'll be able to query the cluster using label selectors to find the pods belonging to a certain group. For example, to find all front-end pods, you can use a label selector with `tier=frontend`. You can also find all pods that aren't front end using `tier!=frontend`. As a Camel fan, you can also add a label to your pods to indicate that the pods run Camel.

You can use the `--show-labels` parameter to show all the labels for the pods:

```
$ kubectl get pods --show-labels
```

As an example, suppose you have the label `animal`; you can then list all pods running Camel using kubectl:

```
$ kubectl get pods -l animal=camel
NAME                                     READY   STATUS    RESTARTS   AGE
helloswarm-kubernetes-171471280-d8r0k    1/1     Running   1          1d
```

Kubernetes has no restrictions or standards on the key names of the labels (hence our use of `animal` in the preceding example). You can use any arbitrary name you desire. From the Kubernetes point of view, it doesn't matter whether the keys are named `tier`, `version`, `animal`, `bambi`, or `thunderstruck`.

Replication controllers and deployments

As you've learned, pods represent the deployment unit in Kubernetes. You can create, manage, and supervise your pods manually, but in real life you want your pods to stay up and run automatically without manual intervention. Instead of creating pods manually, you'd create other types of resources, such as replication controllers or deployments, to then create and supervise the pods.

A *replication controller* is a Kubernetes resource that ensures a pod is always up and running. If the pod dies or become unresponsive, the replication controller creates a new pod immediately.

The concept of replication controllers comes back to the notion of desired state in Kubernetes. The replication controllers constantly monitor the running pods and ensure that the number of pods always matches the desired state. If not enough pods are running, the controller creates new pods based on a pod template. If there are too many pods, excess pods are stopped and removed.

A replication controller has these three parts:

- *Label selector*—Groups pods controlled by the replication controller
- *Replica count*—Specifies the desired number of running instances
- *Pod template*—Describes how to create a new pod

In section 18.3, you deployed our two Camel microservices in the cluster; both had been set to a replicate count of 1 by default. You could use the replication controller to scale up the number of pods—for example, to

run two instances of the WildFly Swarm application. This can be done using the kubectl command-line tool:

```
$ kubectl scale --replicas=2 deployment/helloswarm-kubernetes
deployment "helloswarm-kubernetes" scaled
```

The list of pods now shows two pods with the WildFly Swarm application:

```
$ kubectl get pods
NAME                                    READY   STATUS    RESTARTS   AGE
helloswarm-kubernetes-171471280-d8r0k   1/1     Running   1          1d
helloswarm-kubernetes-171471280-nlpw1   1/1     Running   0          20s
```

You may have noticed that you specified deployment in the arguments to the `kubectl` command when you scaled the WildFly Swarm application up to two replicas. This is because you're using Kubernetes deployment to deploy applications, which was a concept introduced in a later release. A Kubernetes deployment is a higher-level replication controller that's capable of performing more-advanced rollout and rollback of pods. You can use the kubectl tool to list the current deployments:

```
$ kubectl get deployment
NAME                   DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
helloswarm-kubernetes   2         2         2            2           1d
spring-kubernetes       0         0         0            0           1d
```

Here you can see that we desired two pods of the WildFly Swarm application, which are also the current number of instances running. On the other hand, the Spring Boot application was scaled down to zero, which means no pods are running currently. This is a common way to stop an application—by scaling it down to zero.

TIP You can use the `describe` command from kubectl to output verbose details about any Kubernetes resource. For example: `kubectl describe deployment helloswarm-kubernetes`.

Services

Kubernetes services are brilliant. They're as simple as possible, but elegant in their solution. Why? First, let's set the scene. You've learned about pods, and replication controllers ensure that a certain number of pods are constantly running and working as intended. In a microservice or distributed architecture, your applications often need to communicate with each other, and outside the cluster as well—for example, to listen to HTTP requests coming from clients outside the cluster, to call into services hosted on these pods. These pods need a way of discovering each other so they can communicate.

You also learned that Kubernetes can scale up pods so that multiple instances run in the cluster. Therefore, when you try to communicate with pods, you shouldn't rely on their direct IP addresses because pods can come and go (they're dynamic). What you need is a level of indirection, where you can group the pods that provide the service, the way your applications communicate with them, and possibly load-balance against them.

This is exactly what a Kubernetes service is. It allows you to use a label selector to group your pods, and abstract them with a single static network IP address that you can use to discover and communicate with. A Kubernetes service is a single IP address for a group of pods that provide the same service. The IP address is static and never changes while the service exists. You previously encountered a Kubernetes service, illustrated in figure 18.3. This time, let's use the two Camel microservices with Spring Boot and WildFly Swarm as an example. In figure 18.8, we've emphasized where the Kubernetes service lives and how it sits between the Spring Boot and WildFly Swarm pods.

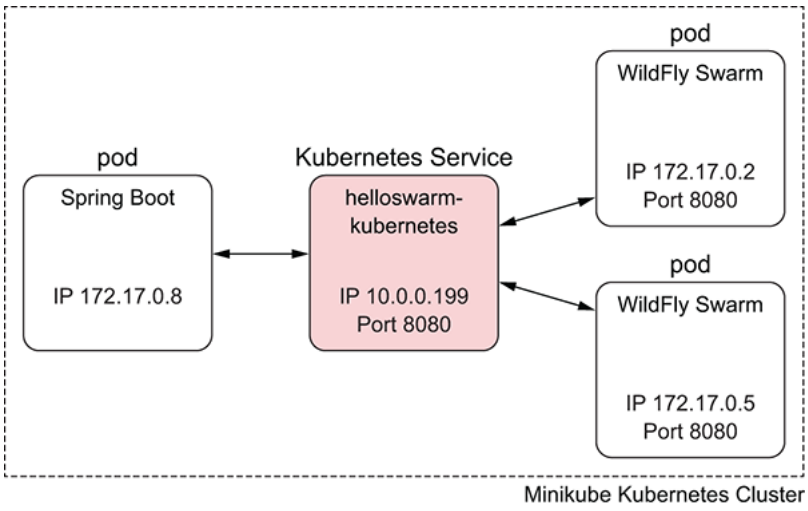

Figure 18.8 The Spring Boot pod calls the helloswarm-kuberentes service, which by itself is a static virtual network IP on 10.0.0.199 that traps the traffic and then load-balances between the running pods that provides the service.

A Kubernetes service consists of the following parts:

- Unique name
- Label selector
- Static IP address and port number

For example, you can use the kubectl tool to view details of services:

```
$ kubectl describe service helloswarm-kubernetes
Name:                   helloswarm-kubernetes
Namespace:              default
Selector:               group=com.camelinaction,
                        project=helloswarm-kubernetes,provider=fabric8
Type:                   NodePort
IP:                     10.0.0.199
Port:                   http        8080/TCP
NodePort:               http        30102/TCP
Endpoints:              172.17.0.2:8080,172.17.0.5:8080
Session Affinity:       None
```

Here you can see that the service has the static IP of 10.0.0.199 and uses port 8080. You can also see that currently the two pods that provide the service are listed with their IP addresses under `Endpoints`. The `Selector` refers the label selector that's used to gather the list of pods. You can run a `kubectl get pods` command with the selector to see the list of pods:

```
$ kubectl get pods -l group=com.camelinaction,app=helloswarm-kubernetes,
NAME                                     READY   STATUS    RESTARTS   AGE
helloswarm-kubernetes-171471280-d8r0k    1/1     Running   1          1d
helloswarm-kubernetes-171471280-nlpw1    1/1     Running   0          2h
```

As expected, the two WildFly Swarm pods provide the service.

Let's review one more time why Kubernetes services are brilliant. They're brilliant because they're an elegant abstraction. From a client point of view, a service is just a static IP address and port number that never changes. There's no need for any dynamic lookup to a service registry to obtain a list of locations where the services are currently running. It can't be simpler for clients to call a service: just a regular network call on a static IP address and port number.

And the brilliance doesn't stop there. Kubernetes also provides a service call using DNS. Instead of calling the service using its static IP and port number, you can call using its service name.

For example, you can call the WildFly Swarm service from within the Kubernetes cluster. To do this, you can start bash on the running Spring Boot pod:

```
kubectl exec -it spring-kubernetes-3845155382-ww31q bash
```

From the bash shell, you can use `curl` to call the service, at first using the direct IP addresses of the pods:

```
$ curl http://172.17.0.2:8080
Swarm says hello from helloswarm-kubernetes-171471280-d8r0k
$ curl http://172.17.0.5:8080
Swarm says hello from helloswarm-kubernetes-171471280-nlpw1
```

Then you can call the IP address of the service itself:

```
$ curl http://10.0.0.199:8080
Swarm says hello from helloswarm-kubernetes-171471280-nlpw1
```

And you can also use the DNS name to call the service. Notice how it load-balances among the two running pods, by returning a different hostname in the response:

```
$ curl http://helloswarm-kubernetes:8080
Swarm says hello from helloswarm-kubernetes-171471280-nlpw1
$ curl http://helloswarm-kubernetes:8080
Swarm says hello from helloswarm-kubernetes-171471280-d8r0k
```

Using the `kubectl exec` command is a handy trick to connect to a running pod and poke around to see what's happening from the bash shell.

Other concepts

After you have more experience using Kubernetes, you should become familiar with other concepts that are good to know. Here are a few:

- *Namespace*—A virtual cluster within the cluster. You use namespaces to separate clusters.
- *Probes*—Used for probing when pods are ready to service traffic and whether they're alive. Section 18.5 covers this topic.
- *PetSet*—Used for stateful applications.
- *Jobs*—Used for tasks that run to completion and terminate.
- *Ingress*—Used to expose services to the outside using HTTP/HTTPS routes.
- *Persistent volume claims*—Used to mount persistent volumes to pods.
- *ConfigMap*—Can be used for configuration management.
- *Secrets*—Used to store sensitive information.

TIP If you want to learn all about Kubernetes, we suggest *Kubernetes in Action* by Marko Luksa (Manning, 2017).

It's time to get back on the Camel and into action. Let's have some fun and see what happens when we release the chaos monkey and start killing pods. How resilient and well behaved do you think our two small Camel microservices are in the cluster?

18.5 Building resilient Camel microservices on Kubernetes

Chapter 7's section 7.4 covered how to design for failures in a microservice architecture. This doctrine is of the utmost importance in any distributed system. In this section, we'll play devil's advocate and cause havoc in our Kubernetes cluster so you can witness what happens when your Camel microservices are not fault-tolerant.

We'll then discuss which Kubernetes features can be used to aid developers building fault-tolerant microservices. Let's begin by setting up the scene before we start being naughty.

18.5.1 Scaling up microservices

We begin our journey with our two Camel microservices running in the Kubernetes cluster. There's a pod of each, as shown here:

```
$ kubectl get pods
NAME                                    READY   STATUS    RESTARTS   AGE
helloswarm-kubernetes-171471280-88vgw   1/1     Running   1          12h
spring-kubernetes-2151443245-27s8g      1/1     Running   0          1m
```

In case you're wondering why the WildFly Swarm pod has been restarted one time, it's because at the time it was the only pod running in the cluster, and Minikube was stopped. When Minikube is started again, it restarts the previously running pods.

In this example, the Spring Boot pod is the client that will continuously call the service running on the WildFly Swarm pod. We'll exploit this fact and see what happens with the client when the WildFly Swarm pod is being killed, scaled up and down, and what else can happen.

But from the beginning, everything is happy, and we'll follow the logs from the command shell:

```
$ kubectl logs -f spring-kubernetes-2151443245-27s8g
...
Swarm says hello from helloswarm-kubernetes-171471280-88vgw
Swarm says hello from helloswarm-kubernetes-171471280-88vgw
```

Deleting the pod

Let's first try to see what happens if we delete the WildFly Swarm pod:

```
$ kubectl delete pods helloswarm-kubernetes-171471280-88vgw
pod "helloswarm-kubernetes-171471280-88vgw" deleted
```

How does the client behave? Not too well, as you'll shortly see: the client fails and outputs exceptions in the log:

```
Caused by: io.netty.channel.ConnectTimeoutException: connection timed ou
```

But then after a little while, the client works again and logs the response from the WildFly Swarm pod:

```
Swarm says hello from helloswarm-kubernetes-171471280-b9bj5
```

Notice that the hostname in the response has changed. That's because the pod was deleted and Kubernetes created a new pod, which gets a new IP address assigned and a new hostname as well.

You can find the new IP address (the previous IP address was 172.17.0.2) by running the `describe` command on the pod:

```
$ kubectl describe pod helloswarm-kubernetes-171471280-b9bj5
IP:             172.17.0.6
```

Oh, wait a minute—you almost fooled us. You told Kubernetes to delete the pod. This isn't exactly the same as if the pod dies because the process crashes or something that isn't in the direct hands of Kubernetes. Yes, you're right in telling Kubernetes to delete a pod, as this allows Kubernetes to perform this in a more graceful manner. Let's try to be more evil and kill the pod using Docker.

Killing the pod

To use the Docker CLI, you need to prepare your command-line shell, which you do from Minikube:

```
minikube docker-env
```

Then you run the `eval` command:

```
eval $(minikube docker-env)
```

You can now use Docker to list all the running Docker containers in the Kubernetes cluster:

```
$ docker ps
CONTAINER ID    IMAGE
563728aead40    camelinaction/helloswarm-kubernetes:snapshot-...
```

You should see a big list of containers, but at the top, you may find the Docker process that runs the WildFly Swarm pod:

Now you can kill that pod using Docker and its container ID:

```
docker kill 563728aead40
```

The Spring Boot client should start failing again, and you should see errors and stacktraces in the log. But after a while, the WildFly Swarm pod comes back online, and the log outputs successful responses again:

```
Swarm says hello from helloswarm-kubernetes-171471280-b9bj5
Swarm says hello from helloswarm-kubernetes-171471280-b9bj5
```

This time, the pod was killed, and Kubernetes was able to self-heal by restarting the pod. Wait a minute, did we say *restart*? Yes. As you'll notice, the hostname in the response is the same as before you killed the pod. Also if you check the IP address of the pod, you'll see it's still the same as before. But its restart count is now 1:

```
$ kubectl get pod
NAME                                    READY   STATUS    RESTARTS   AGE
helloswarm-kubernetes-171471280-b9bj5   1/1     Running   1          24m
spring-kubernetes-2151443245-27s8g      1/1     Running   0          26m
```

Whether the pod is deleted or killed, Kubernetes is able to self-heal and either start a new pod or restart the failed Docker container in the existing pod. Either way, Kubernetes does what's necessary to keep the state of the running cluster according to the desired state. And what's the desired state? You can see this from the deployments:

```
$ kubectl get deployment
NAME                    DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
helloswarm-kubernetes   1         1         1            1           2d
spring-kubernetes       1         1         1            1           2d
```

This is what would be expected. We're not falling off our chairs yet. If you have only one instance of a pod running that provides a service, and if that pod dies, then clients attempting to call that service will fail. You can make this more resilient by scaling up the number of instances that provide the service.

Scaling up the pod

You scale up the WildFly Swarm microservice by setting the replicas on its deployment via the kubectl tool:

```
kubectl scale deployment helloswarm-kubernetes --replicas=2
```

You'd then expect the client to load-balance between the two running pods. Oh, what just happened? Did you spot a stacktrace in the client log?

```
Cannot connect to 10.0.0.199:8080
```

But among those stacktraces are some positive responses:

```
Swarm says hello from helloswarm-kubernetes-171471280-b9bj5
```

And after a little while, the client gets only successful responses:

```
Swarm says hello from helloswarm-kubernetes-171471280-pdkwv
Swarm says hello from helloswarm-kubernetes-171471280-b9bj5
Swarm says hello from helloswarm-kubernetes-171471280-b9bj5
```

What just happened? The WildFly Swarm deployment was scaled up from one to two pods, which means Kubernetes created and started a new pod. The client that continuously calls the service will now start to load-balance between both pods. That is why you can see two hostnames in the preceding logs. The load balancing in Kubernetes is by default random, so that's why you see two consecutive responses from the same pod in the last two lines.

But what about those errors and stacktraces you saw in the beginning? Those occur because when Kubernetes starts up the second pod, that pod is included in the service load balancing right away and therefore receives traffic from the calling client. But the pod isn't yet ready. It's a Java application and it takes a while to start up and initialize WildFly Swarm and Camel, and only then is it ready to receive calls.

To fix this problem, Kubernetes has a couple of health probes that should be used.

18.5.2 Using readiness and liveness probes

Kubernetes allows you to define a readiness probe on your pod's containers. The readiness probe will be called periodically to determine whether a pod is ready. Pods that are ready will be part of the active service endpoints. A readiness probe determines whether a pod can service requests from clients. The way that the readiness check is performed on the pod is specific to what's running inside the pod. Kubernetes supports three kinds of readiness or liveness probes:

- *Exec probe*—Executes a command, and the exit code determines the result
- *HTTP Get probe*—Sends an HTTP GET request, and the HTTP status code determines the result
- *TCP Socket probe*—Attempts to establish a connection via TCP to determine the result

Using HTTP checks is a common technique for checking the health of applications. Spring Boot and WildFly Swarm support this out of the box from their actuator and monitor modules.

The source code contains this example in the chapter18/kubernetes/hello-swarm2 directory. In this example, we've added the org.wildfly.swarm:monitor dependency to the Maven pom.xml file. WildFly Swarm provides a health endpoint on /health that returns an HTTP 2xx status code if WildFly is healthy.

You can use this to know when WildFly Swarm is ready. To add the readiness probe in your microservice, you can add it to the Kubernetes manifest file. This can be done by creating a deployment.yml file in the src/main/fabric8 directory with the content from the following listing.

Listing 18.3 Kubernetes manifest with readiness and liveness probes

```
spec:
  template:
    spec:
      containers:
      - env:
    livenessProbe:    (1)
```

(1) HTTP liveness probe

```
          httpGet:
            path: /health
            port: 8080
            scheme: HTTP
          initialDelaySeconds: 30
    readinessProbe:    (2)
```

(2) HTTP readiness probe

```
          httpGet:
            path: /health
            port: 8080
```

```
            scheme: HTTP
          initialDelaySeconds: 10
```

As you can see, the Kubernetes manifest file can be nested deeply, such as five levels down, before you specify the liveness ❶ and readiness ❷ probes. Notice that you specify the HTTP URL to use, which is deferred as localhost (for example, http://localhost:8080/health is the URL that Kubernetes will periodically call).

AUTOMATIC HEALTH CHECKS FOR SPRING BOOT AND WILDFLY SWARM PROJECTS

The fabric8 Maven plugin can automatically add default readiness and liveness probes for specific Maven projects such as WildFly Swarm and Spring Boot. This happens automatically when a specific dependency is declared in the pom.xml file. For WildFly Swarm, you add the `org.wildfly.swarm:monitor` dependency, and with Spring Boot it's the `org.springframework.boot:spring-boot-actuator` dependency.

That's all that's needed. The example from hello-swarm2 with the health check configuration as shown in listing 18.3 is unnecessary. You can find examples with the source code in the chapter18/hello-swarm3 and chapter18/client-spring-health directories. The fabric8-maven-plugin website provides additional documentation: https://maven.fabric8.io.

When the WildFly Swarm container is started, Kubernetes will wait 10 seconds before performing periodic readiness checks. If the container reports it's ready, it's added to the list of active endpoints on the service. But if anytime later the readiness check fails, the container is removed from the active endpoints and will no longer receive requests. If the container becomes active again, it will be re-added as an active service endpoint.

A liveness check is also periodically triggered by Kubernetes (both checks will keep running). If a liveness probe fails, Kubernetes will assume that the container is unhealthy and restart the container, or if deemed necessary re-create the pod.

This is an important difference between readiness and liveness probes. Readiness probes make sure only ready containers will receive traffic, whereas liveness probes keep the pods running by killing unhealthy containers and replacing them with healthy ones.

You've now added both readiness and liveness probes to your WildFly Swarm microservice. You then deploy this improved microservice in the Kubernetes cluster:

```
cd chapter18/kubernetes/hello-swarm2
mvn fabric8:deploy
```

After the pods have been redeployed, only one WildFly Swarm pod exists, because the Kubernetes manifest used by fabric8:deploy by default uses `replicas=1`.

Scaling up the pod with readiness and liveness probes

When you scale up the WildFly Swarm deployment, you'd expect the readiness probe to be in use and the new pod to start to receive traffic only when it's ready. The Spring Boot client shouldn't log any errors while it continuously runs and calls the service. Let's see what happens:

```
$ kubectl scale deployment helloswarm-kubernetes --replicas=2
```

While the deployment is being scaled up, you can watch the pods using the `-w` flag:

```
$ kubectl get pods -w
NAME                                   READY   STATUS    RESTARTS   AG
helloswarm-kubernetes-2700449218-5fh1w   1/1     Running   0          2h
helloswarm-kubernetes-2700449218-wh2vk   0/1     Running   0          7s
spring-kubernetes-2151443245-27s8g       1/1     Running   0          4h
NAME                                   READY   STATUS    RESTARTS   AG
helloswarm-kubernetes-2700449218-wh2vk   1/1     Running   0          9s
```

While the pod is being started, the `READY` column is listed as `0/1`, which means the pod has one container and currently none is ready. Then a little while later, the state changes and the column becomes `1/1`, which means the container is ready.

When this happens, the Spring Boot client should start to load-balance its requests between the two pods and log responses from both pods:

```
Swarm says hello from helloswarm-kubernetes-2700449218-5fh1w
Swarm says hello from helloswarm-kubernetes-2700449218-wh2vk
```

Are our microservices now resilient and fault-tolerant? Frankly, they're not. In the Spring Boot client, errors that you must deal with can still happen. You can see a problem in the client if you, for example, kill one of the running WildFly Swarm pods.

Using the Docker CLI tool, you can find the running Docker containers in the Kubernetes cluster and find the container ID for one of the running WildFly Swarm containers. You'll then kill this container and observe the Spring Boot client and see what happens:

```
docker kill 845ab817199f
```

You may have to attempt this multiple times to get to a situation where the client still fails when calling the service. We had to try three times before getting this error:

```
java.net.ConnectException: Cannot connect to 10.0.0.199:8080
```

But the error should happen only one or two times, because the Kubernetes cluster is quick to detect that the WildFly Swarm pod has died and therefore isn't ready to service traffic from the client. The subsequent logs are all successful responses from the same pod, until later when both pods are up and running and the log shows responses from both of them:

```
Swarm says hello from helloswarm-kubernetes-2700449218-5fh1w
Swarm says hello from helloswarm-kubernetes-2700449218-5fh1w
...
Swarm says hello from helloswarm-kubernetes-2700449218-wh2vk
```

Your microservices are almost resilient and fault-tolerant. Building distributed systems such as a microservice architecture is a complex task, and you should always design your applications with failure in mind.

We previously covered this topic in chapter 7, section 7.4. In that chapter, you learned how to use Camel's error handler or the Circuit Breaker EIP pattern to deal with failures when calling downstream services.

18.5.3 Dealing with failures by calling services in Kubernetes

In a distributed system, failures can happen anywhere, even when using Kubernetes. For example, the Spring Boot client can fail while calling the service from the WildFly Swarm application (which you've seen on several occasions so far in this chapter). You can reduce the chances for failure by replicating the downstream services to run on multiple pods and let Kubernetes load-balance among the active services. But even so, failures can still happen. For example, a container may just have died before Kubernetes has detected it's no longer live or ready to handle requests.

You should design for failures that can and will happen. The good news is that Camel provides a rich set of error-handling functionality. For example, you can use Camel's error handler to retry the failed service call:

```
errorHandler(defaultErrorHandler()
    .retryAttemptedLogLevel(LoggingLevel.INFO)
    .maximumRedeliveries(5)
    .redeliveryDelay(3000));

from("timer:foo?period=2000")
  .to("netty4-http://{{service:helloswarm-kubernetes}}")
  .log("${body}");
```

The error handler has been configured to retry up to 5 times and with a 3-second delay between attempts. To be able to see in the log when Camel performs a retry, you've raised the logging level to `INFO`.

As usual, we've provided an example with the source code. You can find this example in the chapter18/kubernetes/client-spring2 directory.

The updated client can be deployed to the Kubernetes cluster as follows:

```
mvn fabric8:deploy
```

With our recent improvements in the WildFly Swarm microservice using the readiness and liveness probe, it's become much tougher to provoke failures when the client calls the service. Therefore, you'll roll back the WildFly Swarm deployment to use the older version that doesn't include a readiness probe. This can be done by deploying the older version:

```
cd chapter18/kubernetes/hello-swarm
mvn fabric8:deploy
```

You should now have only one pod running the WildFly Swarm microservice:

```
$ kubectl get pods
NAME                                      READY   STATUS    RESTARTS   AG
helloswarm-kubernetes-2700449218-tmw68    1/1     Running   1          2h
spring-kubernetes-1274767787-czg6s        1/1     Running   0          2h
```

You can provoke a failure by killing the WildFly Swarm pod, which causes Kubernetes to restart the container. Because it lacks a readiness probe, the container will receive traffic it can't process, and the client will either receive a connection exception or HTTP error codes.

The client logs the following:

```
Swarm says hello from helloswarm-kubernetes-2700449218-tmw68
On delivery attempt: 0 caught: java.net.ConnectException: Cannot connect
On delivery attempt: 1 caught: java.net.ConnectException: Cannot connect
Swarm says hello from helloswarm-kubernetes-2700449218-tmw68
Swarm says hello from helloswarm-kubernetes-2700449218-tmw68
```

As you can see, two unsuccessful attempts fail with a connection exception. The third time is the lucky charm, with successful responses coming back again.

In microservice architectures, the Circuit Breaker pattern is a popular choice for dealing with failures when calling services.

Using Hystrix as Circuit Breaker

The Spring Boot client can easily be changed to use Hystrix as the circuit breaker (we covered Hystrix in chapter 7, section 7.4.4). All it takes is adding camel-hystrix-starter to the Maven pom.xml file, and changing the Camel route, as shown in the following listing.

Listing 18.4 Using Hystrix Circuit Breaker to deal with failures when calling a Kubernetes service

```
from("timer:foo?period=2000")
  .hystrix()  ❶
```

❶ Hystrix EIP

```
    .to("netty4-http:http://helloswarm-kubernetes:8080")  ❷
```

❷ Calling Kubernetes service using its DNS name

```
  .onFallback()  ❸
```

❸ Fallback if the service call failed

```
      .transform().constant("Cannot call downstream service")
  .end()
  .log("${body}");
```

The Camel route is easy to understand. You've protected calling the downstream service in Kubernetes using Hystrix ❶. We've taken the opportunity to showcase another great feature in Kubernetes: you can call Kubernetes services using DNS. The hostname is simply the service name, so the WildFly Swarm service can be called using http://helloswarm-

kubernetes:8080 ❷. In case calling the downstream service fails, the Hystrix fallback ❸ is executed, which sets a constant message body that will be logged.

Hystrix will by default use a 1-second time-out when calling a downstream service. This threshold may often be too low, and you can configure a higher value. When using Spring Boot, you can also configure Camel Hystrix from the application.properties file, which is easy. To set a time out of 2 seconds, all you have to do is add the following line:

```
camel.hystrix.execution-timeout-in-milliseconds=2000
```

You can find this example in the chapter18/kubernetes/client-spring3 directory. To try this, you need to deploy the updated client:

```
mvn fabric8:deploy
```

At this point, you still have only one pod of the WildFly Swarm microservice running, so it's easy to provoke a failure by deleting the pod:

```
kubectl delete pod helloswarm-kubernetes-2700449218-tmw68
```

And you can see that the Spring Boot client should start to fail:

```
Swarm says hello from helloswarm-kubernetes-2700449218-tmw68
Cannot call downstream service
Cannot call downstream service
...
```

After a little while, the WildFly Swarm pod has been created again, and the client starts logging successful responses:

```
Swarm says hello from helloswarm-kubernetes-2700449218-1lvl3
```

Phew! That was a lot to take in. Let's recap what you learned.

WHAT DID YOU LEARN?

Building resilient and fault-tolerant microservices is a key requirement in distributed systems. You have to face the music and figure out how to design applications that can survive in an environment where failures will happen.

You learned that Kubernetes provides great support for this with services, replication controllers, and deployments that can help ensure that the cluster can self-heal. But developers still have to design their microservices with failures in mind. For example, microservices that provide services must implement some kind of readiness check allowing Kubernetes to route traffic only to containers that are ready to handle requests. Likewise, you learned that liveness checks must be implemented in your microservices so Kubernetes can constantly observe your applications and automatically restart or kill containers that are no longer alive.

Even with all this awesomeness from Kubernetes, the clients that are calling services must accept that failures can happen. When using Camel, you can use Camel's error handler or integration with Hystrix to elegantly handle those failures.

Unit and integration tests are a vital part of the lives of developers. You may wonder how you perform testing with Kubernetes. That's a good question, so let's spend a moment to cover the basics of testing Kubernetes.

18.6 Testing Camel microservices on Kubernetes

This section shows you how to unit-test Camel on Kubernetes. You're going to use the Arquillian testing framework. (Arquillian was also used in chapter 9.) More precisely, you'll use the Arquillian Cube (http://arquillian.org/arquillian-cube) extension, which provides capabilities for managing and running tests on Docker and Kubernetes.

In this section, you'll first add Arquillian Cube to your project. Then you'll write a basic unit test and run the test in Kubernetes.

18.6.1 Setting up Arquillian Cube

You'll use the Camel microservice that runs on Spring Boot, which is the simplest project. This example is located in the chapter18/kubernetes/client-spring-test directory.

You add Arquillian Cube to any Java project via Maven by importing the arquillian-cube-bom BOM to the dependency management of your Maven pom.xml file. Then add the following four Maven dependencies: arquillian-junit-standalone, arquillian-cube-kubernetes, assertj-core, and kubernetes-assertions. The last two dependencies aren't from Arquillian but make writing assertions using Kubernetes concepts much easier (you'll see this in a moment). The last thing to do is to create an arquillian.xml file in the src/main/resources directory with the following content:

```
<arquillian xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="http://jboss.org/schema/arquillian"
            xsi:schemaLocation="http://jboss.org/schema/arquillian
    http://jboss.org/schema/arquillian/arquillian_1_0.xsd">
  <extension qualifier="kubernetes">
  </extension>
</arquillian>
```

You do this to turn on the Kubernetes extension.

18.6.2 Writing a basic unit test using Arquillian Cube

The following listing shows a unit test that we highly recommend you run. The test checks that the application (pod) can start and be ready for a stable period of time (10 seconds).

Listing 18.5 Testing that the pod can start and run for a stable period of time

```
@RunWith(Arquillian.class)
@RequiresKubernetes    1
```

1 Test requires Kubernetes

```
public class PodStartedKT {

  @ArquillianResource
  KubernetesClient client;    ❷
```

❷ Injects Kubernetes client

```
  @Test
  public void testPodStarted() throws Exception {
    assertThat(client).deployments().pods().isPodReadyForPeriod();    ❸
```

❸ Asserts pod can deploy and be ready

```
  }
}
```

As you can see, the test has only a few lines of source code. The unit test uses Arquillian with Kubernetes ❶. Notice that you name the class `PodStarterKT` with the suffix `KT` to denote that this is a Kubernetes Test. `KubernetesClient` is injected as a resource ❷, which is being used in the test method that's only one line of code. That single line of code ❸ asserts that the deployment of the pod is ready for a stable period (10 seconds).

This is a powerful test because it tests that your application can be deployed, can start, and can run on Kubernetes.

18.6.3 Running Arquillian Cube tests on Kubernetes

You can run this test from a command line:

```
eval $(minikube docker-env)
cd chapter18/kubernetes/client-spring-test
mvn install -P kubernetes
```

Notice that you must specify `-P kubernetes` as an argument. This allows you to build and run regular unit tests locally without Kubernetes separated from running the Kubernetes test. You can find more details about how this is done in the pom.xml file of the example.

When the test runs, it logs to the console a bit of activity that happens and then completes with the following:

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 34.841 s
- in com.camelinaction.PodStartedKT
Destroying Session:a83be678-55d1-4888-b3a1-845bc0aef455
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

Here you can see that one test ran and completed with success in about 34 seconds. In that time, Arquillian Cube first uses the `KubernetesClient` to create a new temporary namespace in Kubernetes. Using a temporary namespace, you ensure that the unit test runs in a clean environment and doesn't cause side effects to any other pods and services running in the Kubernetes cluster. Then the application is deployed, and the test waits for the pod to become ready and stay ready for at least 10 seconds to be considered stable. When this happens, the test is complete and the pod is undeployed, and the temporary namespace is deleted.

Okay, that's impressive, but maybe you don't believe us. Let's, as an example, make a mistake in our Camel application on purpose and rerun the tests. In `HelloRoute`, change the route to use a nonexistent Camel component, which should cause Camel to fail on startup; therefore, we'd expect the pod to not become ready. We change the route from using `timer` to `timer2` as the component name:

```
from("timer2:foo?period=2000")
```

And then rerun the tests:

```
mvn install -P kubernetes
```

This time, the test takes longer to complete and should fail:

```
Tests in error:
  PodStartedKT.com.camelinaction.PodStartedKT
Tests run: 1, Failures: 0, Errors: 1, Skipped: 0
```

When a test fails, Arquillian Cube will grab the log from the failed pod and output it in the command shell. You can scroll up and find the error reported from Camel about the unknown component:

```
Failed to resolve endpoint: timer2://foo?period=2000 due to: No componen
found with scheme: timer2
```

This was a quick introduction to writing a basic unit test for Kubernetes. There's one more test we'd like to show you: a test that calls a Kubernetes service.

18.6.4 Writing a unit test that calls a Kubernetes service

In this section, you're going to write a unit test that calls the Hello service from the WildFly Swarm Camel example. You're going to use Arquillian Cube again, which you'll set up the same way as previously covered. The source code for the unit test is provided in the following listing.

Listing 18.6 Unit test that calls service running inside Kubernetes

```
@RunWith(Arquillian.class)
@RequiresKubernetes    ❶
```

❶ Test requires Kubernetes

```
public class HelloSwarmKT {

  @ArquillianResource    ❷
```

❷ Injects Kubernetes client

```
  KubernetesClient client;    ❷

  @Named("helloswarm-kubernetes")    ❸
```

❸

Injects URL to Kubernetes service to be called

```
  @PortForward    ❸
  @ArquillianResource    ❸
  URL url;    ❸

    @Test
    public void testCallService() throws Exception {
      OkHttpClient okHttpClient = new OkHttpClient();
      Request request = new Request.Builder().get().url(url).build();

    Response response = okHttpClient.newCall(request).execute();    ❹
```

❹

Uses OkHttpClient to call service

```
      assertEquals(200, response.code());

      String body = response.body().string();
    assertTrue(body.contains("Swarm says hello from"));    ❺
```

❺

Asserts response is as expected

```
    }
  }
```

The test requires running on Kubernetes, which is declared using the `@RequiresKubernetes` annotation on the class ❶. The test will use the `KubernetesClient` that's being injected ❷. The service you want to call in

Kubernetes is then declared with a couple of annotations ❸. The `@Named` annotation specifies the name of the service, and `@PortForward` ensures that the service is accessible from the JVM running the unit tests into the running Kubernetes cluster via Kubernetes port forwarding.

The service is then called using plain HTTP and the `OkHttpClient` library. Notice that this library has no knowledge of Kubernetes but uses the injected URL as is. Because of the port forwarding, the `OkHttpClient` is able to call ❹ into the running Kubernetes cluster and receive the response, which is then asserted as expected ❺.

You can try this example from the source code as follows:

```
eval $(minikube docker-env)
cd chapter18/kubernetes/hello-swarm-test
mvn install -P kubernetes
```

The test should complete successfully. As before, you can make a change to the source code to provoke an error and see what happens. For example, you can try to change the response generated in `HelloBean` to say goodbye instead of hello:

```
public String sayHello() throws Exception {
  return "Swarm says goodbye from " + InetAddressUtil.getLocalHostName()
}
```

And then rerun the tests again:

```
mvn install -P kubernetes
```

This should fail with an assertion error because the test expects the response to say hello and not goodbye.

Speaking of goodbye, that's all we have to say about testing Kubernetes. The last section is a mixed bag of some great Kubernetes-related projects.

18.7 Introducing fabric8, Helm, and OpenShift

This section describes projects related to Kubernetes that are worth keeping an eye on. The first project is fabric8, which you've already used

through its Maven tooling, such as the Docker and fabric8 Maven plugins. We will give you a brief overview and history of the fabric8 project and encourage you to take a closer look once in a while to see what the fabric8 team has done of late.

Linux users who are accustomed to installing software from package managers such as yum or apt will be pleased to know that Helm is their answer in the realm of Kubernetes. Before ending this chapter, we'll look at OpenShift, a platform built on top of Kubernetes, and see the additional capabilities that it brings to the table.

18.7.1 fabric8

The fabric8 project (https://fabric8.io) has evolved over the years. The project started as a platform to manage a cluster of Apache Karaf or JBoss Fuse application servers. This was before Docker or Kubernetes existed, when we had to build our own solutions with pieces from Java such as Apache ZooKeeper, Maven, and OSGi. Docker and Kubernetes changed all that, and fabric8 v2 was completely redesigned to be a platform on top of Kubernetes.

Today, fabric8 is a pure upstream project with a high degree of creativity that enables new ideas to be born and attempted. The good parts of fabric8 are then harvested and bought into other projects and communities such as Kubernetes, OpenShift, OpenShift.io, Apache Camel, and Spring Cloud. For example, fabric8 was a first mover on a better Java developer story on top of Kubernetes. Other successes are the fabric8-maven-plugin and the Kubernetes Java client used by various tooling and projects to make it easier for Java developers to work with Kubernetes. The continuous deployment effort from fabric8 found its way into the OpenShift project.

Moving on to the next project. The purpose of Helm is to make installing applications on Kubernetes easy.

18.7.2 Kubernetes Helm

Kubernetes Helm is like an OS package manager (yum or apt in Linux), but for installing and managing applications in Kubernetes. Helm provides a command-line tool with the surprising name of helm, and a

server component named Tiller that runs as a pod inside Kubernetes. Helm application packages are called charts.

A *chart* is just a YAML file with overall details about the application. Kubernetes resources are embedded in the chart, and these resources can be templated. In other words, they can be parameterized, so you can install an application and take in parameters from the command line such as usernames, passwords, and database names.

This makes it easier to install well-known applications such as MySQL, PostgreSQL, WordPress, and others. The Helm community maintains a growing list of charts at https://github.com/kubernetes/charts.

Let's try this.

Installing Helm

To install Helm, you can follow the instructions at the Helm website: https://github.com/kubernetes/helm. There are downloads for macOS, Linux, and Windows.

After installing Helm, you need to install Tiller in the Kubernetes cluster, and you can easily do this with the helm CLI tool:

```
helm init
```

Then wait a little while for the Tiller pod to be ready. If you can't find the Tiller pods, it's because they're not installed in the default namespace but in the Kubernetes system:

```
$ kubectl get pod -n kube-system
NAME                              READY   STATUS    RESTARTS   AGE
kube-addon-manager-minikube       1/1     Running   2          1d
kube-dns-v20-x1hf8                3/3     Running   6          1d
kubernetes-dashboard-tx751        1/1     Running   2          1d
tiller-deploy-3210613906-f936w    1/1     Running   0          1h
```

After Helm is installed, you can use the CLI to search for applications to install. For example, type this to find any MySQL applications:

```
helm search mysql
```

To install MySQL, type the following:

```
helm install --name my-great-database stable/mysql
```

During the installation, Helm will output further instructions, such as how to get generated passwords and URLs to the installed application. You can delete what you installed at any time by deleting the chosen name:

```
helm delete my-great-database
```

Installing your own application using Helm

You can try to install the two Camel microservices using Helm. Because fabric8-maven-plugin generates Helm charts, you can easily install the example. A prerequisite is that the Docker image must have been built and published to the Kubernetes Docker repository. This can be done as follows:

```
cd chapter18/kubernetes/hello-swarm2
mvn install
```

You can now install the application using Helm:

```
helm install target/helloswarm-kubernetes-2.0-SNAPSHOT-helm.tar.gz
```

And likewise, you can install the Spring-Boot client:

```
cd chapter18/kubernetes/client-spring3
mvn install
helm install target/spring-kubernetes-2.0-SNAPSHOT-helm.tar.gz
```

Notice that the Helm charts a tar.gz file, which you can publish to a Maven repository. Helm also makes it possible to build your own Helm chart repositories—for example, all the applications you use in your company.

The fabric8 project provides a chart repository that you can add using the following:

```
helm repo add fabric8 https://fabric8.io/helm
```

Helm is a promising package manager for Kubernetes, and we recommend taking a look at it.

Platforms built on top of Kubernetes

Kubernetes is a great container platform that's good at orchestrating and running containers at scale. As mentioned, Kubernetes was created by Google, based on its experience of running containers for over 10 years. And as such, Kubernetes is operation focused, and not as accessible for developers and system administrations from traditional enterprises. Something more is needed. That's why several companies are building platforms on top of Kubernetes that provide a better experience for developers and system administrators. Those platforms are known as *platforms as a service* (PaaS). One of these platforms is OpenShift by Red Hat.

18.7.3 OpenShift

Red Hat OpenShift is a platform as a service built on top of Kubernetes. OpenShift comes in three offerings:

- *OpenShift Origin*—Open source upstream project that's free to use
- *OpenShift Enterprise*—Commercial enterprise-grade product by Red Hat that runs on premises or in any of the big cloud providers such as Amazon, Azure, or Google
- *OpenShift Online*—Online multitenant cloud hosted by Red Hat

OpenShift comes with a rich set of features and functionality out of the box. For example, a feature-rich web console makes using the platform better for both developers and system administrators.

OpenShift provides powerful user-management capabilities, making it possible to specify what each user can and can't do. Each user is given access to the projects they work on (a *project* is a Kubernetes namespace). A user can act on only those resources that reside in the projects the user has access to. Access to projects is granted by cluster administrators.

OpenShift also allows you to integrate to existing user-management systems that organizations have, such as Microsoft Active Directory or any other LDAP servers.

Building images from source

One of the most appreciated features of OpenShift is its ability to build, deploy, and run applications straight from source code. With Kubernetes, users have to build their Docker images themselves (as you did in this chapter with fabric8-maven-plugin and with Docker installed locally on your computer). OpenShift can pull the source code from a code repository such as GitHub and build it as a Docker image within the cluster. This can also be seen as a more secure option, because you'd be in full control of how your images are built, as it's all done within the platform. It can be a security concern to let developers run arbitrary Docker containers they've found on the internet.

OpenShift can integrate with web hooks. When new code is pushed in the code repository, a new build is triggered that, when the Docker image has been successfully built, can trigger a rolling upgrade of pods running with same image tag.

Figure 18.9 is a screenshot of the OpenShift web console with a rolling upgrade in progress.

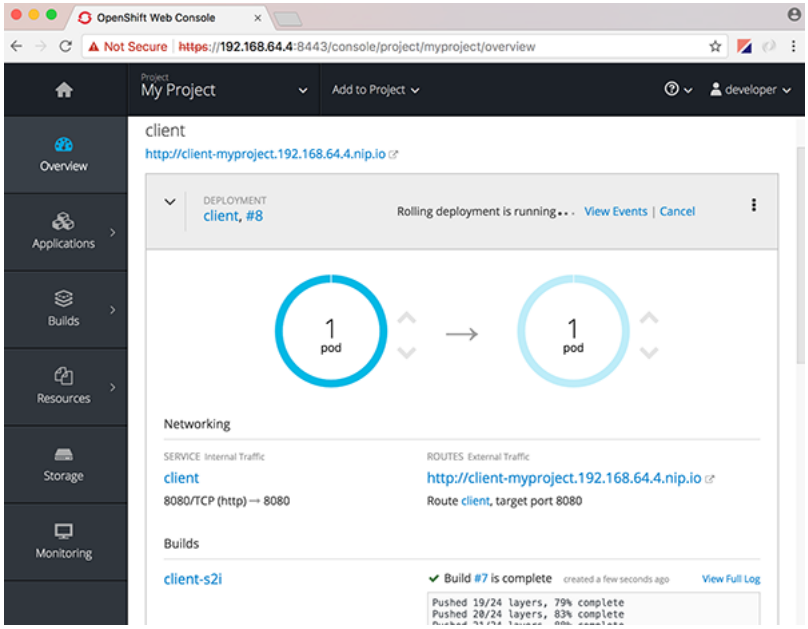

Figure 18.9 OpenShift web console. A source code change is pushed to GitHub, which triggers a build running in the cluster, which on success pushes an updated Docker image to the OpenShift Image Repository. The updated image then triggers a rolling upgrade of all pods that run this image, which is what is in progress on the screen. The old image is currently running and waiting for a new pod to spin up with the updated image.

Continuous delivery is also provided out of the box; you can customize your pipelines to your business processes. For example, before changes can go into production, an approval process is triggered, and a manager must accept the change before the pipeline continues.

Trying OpenShift

If you want to try OpenShift, you can install it locally using Minishift, which is equivalent to Minikube. Another way of trying OpenShift is by accessing OpenShift Online, which is a hosted platform that offers a free plan with numerous pods and resources. You can purchase additional resources if needed.

If you're going down the path of OpenShift, we highly recommend that you download the free ebook *OpenShift for Developers* (the link is provided in this chapter's summary), which has more details about the differences between Kubernetes and OpenShift and ways to get started developing on this platform.

18.8 Summary and best practices

This chapter is an introduction and hands-on guide to getting started with Camel in the world of Docker and Kubernetes. There's so much more to this new world of containers that warrants several books on its own. This chapter taught you how to get your Camel and Java applications running

on Kubernetes. Although Docker is only four years old (founded in 2013), the concept of Linux containers dates back decades. Still, using containers has only recently started to become more mainstream. As developers, we have to adapt and learn the skills required to stay relevant in our industry. We encourage you to start playing with containers when you get a chance.

The takeaways from this chapter are as follows:

- *Learn about containers*—The concept of containers and their impacts on the way we package, deploy, and run our applications at scale are here to stay. Climb onboard and learn the skill set for this new world.
- *Camel, containers, and microservices for the win*—Camel runs great on containers and as a microservice library. You can use Camel in popular microservice frameworks such as Spring Boot or WildFly Swarm.
- *Learn the Kubernetes concepts*—Kubernetes is the third-generation cluster platform from Google, which has more than a decade of experience running everything in containers. Kubernetes is a goldmine of great concepts for a distributed system.
- *Learn using Minikube*—Get started with Kubernetes on your own computer using Minikube. Learn how to build and run your Java applications in this local cluster. Don't be afraid, because if something goes wrong, you can always stop and delete Minikube and start from scratch.
- *Use fabric8-maven-plugin*—Developers using Maven should use this plugin to help them build, run, and debug their Java applications running in Kubernetes.
- *Design for failure*—Failures will happen in a distributed system. Build your Java applications with this in mind. Use the readiness and liveness probes from Kubernetes to make your applications more fault-tolerant, and use Kubernetes services for network communications.

We've covered a lot in this chapter but certainly didn't cover everything. When using Kubernetes (or any other container-based platform), there are many more things to consider in a microservices environment that we can't cover in this book. We don't want to leave you in despair, so we've listed some things you should consider here:

- *Logging and metrics*—Make your applications observable so a centralized platform-monitoring solution can capture logs and metrics. We covered monitoring and management in chapter 16. PaaS platforms such as OpenShift come with such functionality out of the box.
- *Tracing*—The more you break your applications into individual microservices, the more moving parts you have. You need tooling to help you see the big picture. For example, use message tracing such as Zipkin or OpenTracing so you can correlate a business transaction with other data from logging, metrics, and response times.
- *Continuous delivery*—When you have many more microservices than before with traditional architecture, your existing manual deployment process won't scale. Teams that own and operate their own microservices also need a way of deploying without intervention from a centralized operations team. For this, you need to use continuous deployment, which can automate deployment of applications into every environment in the cluster.
- *Configuration management*—Your microservices should be configurable—for example, to configure sensitive login credentials, or to turn on features for A/B testing. In Kubernetes, you can use *secrets* and *configmaps* to configure your running containers.
- *Enterprises should consider using a PaaS*—Companies looking to adopt Kubernetes may benefit from using a full-fledged PaaS platform to provide functionality that enterprises need and expect. Using vanilla Kubernetes often requires you to find and integrate needed third-party add-ons that you can otherwise spend on delivering value to your business.

Here are a few books and other resources that we found valuable when we were learning about Docker and Kubernetes:

- "The Decline of Java Application Servers when Using Docker Containers" by James Strachan (https://blog.fabric8.io/the-decline-of-java-application-servers-when-using-docker-containers-edbe032e1f30#.1n6jphq5y)
- *Kubernetes in Action* by Marko Luksa (Manning, 2017)
- *Kubernetes Patterns* (https://leanpub.com/k8spatterns/) by Bilgin Ibryam and Roland Huss (Leanpub, 2018)

- *Microservices for Java Developers* by Christian Posta (free book can be downloaded from https://developers.redhat.com/promotions/microservices-for-javadevelopers/)
- *OpenShift for Developers* by Grant Shipley and Graham Dumpleton (free book can be downloaded from www.openshift.com/promotions/for-developers.html)

We'll now leave the world of containers and talk about something completely different. Chapter 19 is dedicated to Camel tooling. We'll cover what we believe are the greatest Camel tools you can find; these come either out of the box from Apache Camel or from third-party projects and vendors.