KUBERNETES

Getting Started with Kubernetes

Automate the deployment, scaling, and management of containerized applications and services.

by Jesse Butler

Today, monolithic applications are considered an antipattern and, for most use cases, the cloud is the deployment platform of choice. These two changes equate to far more than booting virtual machines on other people's computers. Effectively leveraging the power and scalability of the cloud means departing from yesterday's monoliths and adopting new architectures and development practices.

A microservices architecture is the emerging standard for delivering applications and services in the cloud. Applications are broken down into loosely coupled discrete programs, each of which has a specific responsibility. Adopting this sort of architecture allows teams to work more independently as they push out their specific features—without being tied to a cumbersome whole-organization roadmap. In addition, with discrete software components, testing can be simplified and deployments streamlined.

Microservices adoption comes with its own set of challenges. Creating and deploying a few services on a virtual machine is a first step, but how do you manage the full software lifecycle? Container adoption has been primarily driven by this need. Using containers addresses several concerns for a microservices architecture, such as the following:

- Application software is decoupled from the host environment, providing great portability.
- Containers are lightweight and fairly transparent, thereby enabling scalability.
- Software is packaged along with its dependencies.

Given these benefits, containers are an excellent choice for the packaging and deployment of microservices. But containers are not magic. Under the covers, it's all still software, and you need a way to deploy, manage, and maintain your containers at scale. Where developers once had a single monolith to monitor and maintain, they now might have dozens or hundreds of services. This is where Kubernetes comes into play.

Kubernetes is an open source platform for automating the deployment, scaling, and management of containerized applications and services. It was developed in response to the challenges of deploying and managing large fleets of containers at Google, which open sourced the project and donated it to the Cloud Native Computing Foundation (CNCF). That foundation fosters the cloud native computing ecosystem. Kubernetes was the first graduated project for CNCF, and it became one of the fastest growing open source projects in history. Kubernetes now has more than 2,300 contributors and has been widely adopted by companies large and small, including half of the Fortune 100.

Getting Started with Kubernetes

How to get started? Kubernetes has a large ecosystem of supporting projects that have sprung up around it. The landscape can be daunting, and looking for answers to simple questions can lead you down a rabbit hole, which can easily make you feel like you're woefully behind. However, the first few steps down this path are simple, and from there you can explore more-advanced concepts as your needs dictate. In this article, I demonstrate how to:

- Set up a local development environment with Docker and Kubernetes
- Create a simple Java microservice with Helidon
- Build the microservice into a container image with Docker
- Deploy the microservice on a local Kubernetes cluster
- Scale the microservice up and down on the cluster

To follow this tutorial, you will need to have some tools installed locally:

- Docker 18.02 or later
- Kubernetes 1.7.4 or later
- JDK 8 or later
- Maven 3.5 or later

You can install the latest version of each of these requirements on macOS, Linux, or Windows.

If you don't have Docker installed already, refer to the *Getting Started with Docker guide* and follow the instructions for your platform. You'll want to be familiar with Docker basics.

You can use any Kubernetes cluster available to you, but use Minikube to follow along with this article. Minikube runs a single-node Kubernetes cluster inside a virtual machine on your local system, giving you just enough to get started. Follow the Minikube installation documentation if you don't have Minikube installed already.

In Kubernetes, a service is an abstraction that defines a way to access a pod or a set of pods.

Now that you have Docker ready to go and can spin up a local Kubernetes cluster with Minikube, you will need an example microservice to work with. Inspired by the Helidon article on microservices frameworks in the March/April issue of *Java Magazine*, I use Helidon in this tutorial to create a simple microservice.

Build a Basic Microservice

Get started quickly with Helidon by creating a new project that uses the Helidon quickstart Maven archetype. The following will get you up and running with a basic starter project:

```
$ mvn archetype:generate -DinteractiveMode=false \
    -DarchetypeGroupId=io.helidon.archetypes \
    -DarchetypeArtifactId=helidon-quickstart-se \
    -DarchetypeVersion=1.0.1 \
    -DgroupId=io.helidon.examples \
    -DartifactId=helidon-quickstart-se \
    -Dpackage=io.helidon.examples.quickstart.se
```

Change into the `helidon-quickstart-se` directory and build the service:

```
$ cd helidon-quickstart-se
$ mvn package
```

That's it—you now have a working example of a microservice. The project will have built an application JAR file for your sample microservice. Run it to check that everything is working properly:

```
$ java -jar ./target/helidon-quickstart-se.jar
[DEBUG] (main) Using Console logging
2019.03.20 12:52:46 INFO io.helidon.webserver.NettyW
started: [id: 0xbdfca94d, L:/0:0:0:0:0:0:0:0:8080]
WEB server is up! http://localhost:8080/greet
```

Use curl in another shell and put the service through its paces:

```
$ curl -X GET http://localhost:8080/greet
{"message":"Hello World!"}
$ curl -X GET http://localhost:8080/greet/Mary
{"message":"Hello Mary!"}
$ curl -X PUT -H "Content-Type: application/json" -d
http://localhost:8080/greet/greeting
$ curl -X GET http://localhost:8080/greet/Maria
{"message":"Hola Maria!"}
```

This isn't a very exciting service, but it serves the purpose of an example to build a container with. Before you carry on to Kubernetes, build a Docker image of your microservice. Helidon provides an example Dockerfile; you can build your image simply via

```
docker build -t helidon-quickstart-se target
```

You can now see what Kubernetes can do for you.

Moving to Kubernetes

With Docker, you can build container images, create containers, and manage images and containers locally. Kubernetes comes into play when it is time to deploy containers in production and at scale.

In Kubernetes, containers are deployed in groups of related containers called *pods*. A pod is a deployable unit and may contain more than one container. For example, a pod might contain two containers: one running a web server and the other being a logging service for the server. Later, you will create a pod for your microservice and it will have just one container: an instance of your `helidon-quickstart-se` image.

We thank reader Deepak Vohra for a correction to this paragraph.

Part of the role of Kubernetes is to ensure your application services are up and running. You describe what should be run and monitored by defining a *deployment*. Kubernetes will monitor the health of pods within a deployment, and in the event they fail or become unresponsive, it will redeploy the pods as needed.

Let's start by testing your local Kubernetes cluster with a simple example. First, to get your local cluster spun up, invoke the `minikube` start command. This command provides a running status of its progress. Wait for the completion message before moving on.

```
$ minikube start
  minikube v0.35.0 on darwin (amd64)
  ...
  Done! Thank you for using minikube!
```

The Minikube installation comes with `kubectl`, the primary interface to Kubernetes. Like the Docker client, it is a powerful multipurpose tool for

interacting with all aspects of your cluster and the containers deployed to it.

Use `kubectl` create to create a simple deployment with a prebaked "hello world" example. Then use `kubectl` get to show your deployment and the pods within it. You'll see something like this:

```
$ kubectl create deployment hello-node \
  --image=gcr.io/hello-minikube-zero-install/hello
deployment.apps/hello-node created
$ kubectl get deployments
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
hello-node   0/1     1            0           27s
$ kubectl get pods
NAME                          READY   STATUS    RES
hello-node-64c578bdf8-5b7jm   1/1     Running   0
```

By default, Kubernetes assigns a pod an internal IP address that is accessible only from within the cluster. To enable external access to the containers running within a pod, you will expose the pod as a *service*. In Kubernetes, a service is an abstraction that defines a way to access a pod or a set of pods.

Create a simple `LoadBalancer` service with `kubectl expose`. This will enable external access to your service through a load balancer.

```
$ kubectl expose deployment hello-node --type=LoadBa
service/hello-node exposed
$ kubectl get services
NAME         TYPE           CLUSTER-IP      EXTERNAL-
hello-node   LoadBalancer   10.104.108.47   <pending>
kubernetes   ClusterIP      10.96.0.1       <none>
```

[The output includes other fields that are omitted to fit on this page. This is also true of some subsequent output listings. —*Ed.*]

On a cloud provider or other managed Kubernetes platform, this action would result in the allocation of a load balancer resource, and its IP address would be shown in the `EXTERNAL-IP` column. In Minikube, use the service command, which will open a browser and show you that the service is working:

```
$ minikube service hello-node
```

With your sanity test done, you can tear things down and clean up. To do this, use the `kubectl delete` command:

```
$ kubectl delete service hello-node
service "hello-node" deleted
$ kubectl delete deployment hello-node
deployment.extensions "hello-node" deleted
```

Now that you have a local Kubernetes cluster and know that it is working, it's time to put your own image to work.

Deploying to Kubernetes

In the previous section, you created a simple deployment with the command-line arguments to `kubectl create`. Typically, you'll need to describe your deployments in more detail, and for that you can pass a YAML file to `kubectl`. This step allows you to define all aspects of your Kubernetes deployments. Another benefit of defining your deployments in YAML is that the files can be kept in source control along with your other project source code.

The Helidon starter project includes some boilerplate configuration for both a deployment and a service in the `target/app.yaml` file. Open this file in your favorite editor and let's examine its contents.

```
$ cat target/app.yaml
kind: Service
apiVersion: v1
metadata:
  name: helidon-quickstart-se
  labels:
    app: helidon-quickstart-se
spec:
  type: NodePort
  selector:
    app: helidon-quickstart-se
  ports:
  - port: 8080
    targetPort: 8080
    name: http
---
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: helidon-quickstart-se
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: helidon-quickstart-se
        version: v1
    spec:
      containers:
      - name: helidon-quickstart-se
        image: helidon-quickstart-se
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
---
```

This content might appear at first to be just a wall of metadata, but by reading through it, you can see that it defines a deployment that will consist of a pod with a `helidon-quickstart-se` container in it and a service that makes it available externally. Notice the service uses an `app` by the same name as the deployment's `app` label.

Note the image name in the deployment specification calls for your local image name: `helidon-quickstart-se`. When you deploy this in Kubernetes, the runtime will look for this image locally first. You *are* using a local image, in that Docker is running on your local machine. But the issue is that Minikube is running its own instance of Docker locally in its virtual machine. The image you built earlier will not be found there.

Minikube has a handy solution for this issue: the `docker-env` command. This command prints a set of shell environment variables that will direct a Docker client to use the Docker server where Minikube is running. Simply invoke this command inside an `eval` on Linux, UNIX, or Mac (consult the documentation for the Windows equivalent), and it will set the variables in your current shell environment.

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

Now when you invoke the `docker` client, it will connect to Docker running in the Minikube virtual machine. Note that this will be configured only in your current shell session; it's not a permanent change.

Now you can build the image as you did above, but this time it will be built in Minikube's virtual machine, and the resulting image will be local to it:

```
$ docker build -t helidon-quickstart-se target
Sending build context to Docker daemon  5.877MB
```

```
Step 1/5 : FROM openjdk:8-jre-slim
8-jre-slim: Pulling from library/openjdk
f7e2b70d04ae: Pull complete
05d40fc3cf34: Pull complete
b235bdb95dc9: Pull complete
9a9ecf5ba38f: Pull complete
91327716c461: Pull complete
Digest: sha256:...
Status: Downloaded newer image for openjdk:8-jre-sl
 ---> bafe4a0f3a02
Step 2/5 : RUN mkdir /app
 ---> Running in ec2d3dad6e73
Removing intermediate container ec2d3dad6e73
 ---> a091fb56d8c5
Step 3/5 : COPY libs /app/libs
 ---> a8a9ec8475ac
Step 4/5 : COPY helidon-quickstart-se.jar /app
 ---> b49c72bbfa4c
Step 5/5 : CMD ["java", "-jar", "/app/helidon-quicks
 ---> Running in 4a332d65a10d
Removing intermediate container 4a332d65a10d
 ---> 248aaf1a5246
Successfully built 248aaf1a5246
Successfully tagged helidon-quickstart-se:latest
```

With your image built local to your Kubernetes cluster, you can now create your deployment and service:

```
$ kubectl create -f target/app.yaml
service/helidon-quickstart-se created
deployment.extensions/helidon-quickstart-se created

$ kubectl get pods
NAME                                    READY   STA
helidon-quickstart-se-786bd599ff-n874p   1/1     Ru
$ kubectl get service
NAME             TYPE       CLUSTER-IP    EXTERNAL-
helidon-quick... NodePort   10.100.20.26  <none>
kubernetes       ClusterIP  10.96.0.1     <none>
```

There is one last thing to note before you test your newly deployed Kubernetes service. Earlier, you created a service with type `LoadBalancer`. You can access your pod through the service's external IP address, which is configured on a load balancer. This is one type of service providing external access to services running within your cluster. Another type is `NodePort`, which this service is using. A node port exposes the service on a mapped port across all of the cluster nodes.

Minikube once again comes through with a handy command. You can use the service command again, this time with the `--url` option to retrieve a service access URL. Invoke this command and then test your service with the URL it returns:

```
$ minikube service helidon-quickstart-se —url
http://192.168.99.101:31803
$ curl -X GET http://192.168.99.101:31803/greet
{"message":"Hello World!"}
```

You have now deployed a microservice to Kubernetes. Fantastic! Before you head off to do amazing things with containers, let's examine a few more basics.

Keeping an Eye on Things

When you ran your service at the command line, you saw the output of the web server starting up. This output is also captured by the container runtime and can be seen with the `kubectl logs` command. If things don't appear to be running correctly, this is a good place to check first.

```
$ kubectl logs helidon-quickstart-se-786bd599ff-n87
[DEBUG] (main) Using Console logging
```

```
2019.03.23 01:00:53 INFO io.helidon.webserver.NettyW
  Thread[nioEventLoopGroup-2-1,10,main]: Channel '@
  started: [id: 0x9f01de18, L:/0.0.0.0:8080]
WEB server is up! http://localhost:8080/greet
```

This command is especially useful when you are doing development and working with a local cluster. At scale, there are several ways to collect, store, and make use of log data. Open source projects as well as fully automated commercial offerings exist, and an online search will show many options to explore.

Scaling to Meet Demand

Suppose your service is part of an application and is responsible for saying “Hello” to users. Your team anticipates a spike in use tomorrow, so you will scale up your service to handle more users. Scaling your deployments up and down is a core feature of Kubernetes. To scale, simply modify the number of replicas defined in your deployment specification and apply the changes via `kubectl apply`.

Edit your `target/app.yaml` file and apply the changes now. Start by scaling the number of replicas up from one to five.

```
$ grep replicas target/app.yaml
replicas: 5
$ kubectl apply -f target/app.yaml
service/helidon-quickstart-se unchanged
deployment.extensions/helidon-quickstart-se configu
```

You should now see five pods deployed where there was only one before. Note that because Kubernetes configuration is declarative, only the changes that are required are committed to the cluster. In this case, four pods are added to the deployment.

```
$ kubectl get pods

NAME                                     READY   STA
helidon-quickstart-se-786bd599ff-5gm29   1/1     Ru
helidon-quickstart-se-786bd599ff-fkg8g   1/1     Ru
helidon-quickstart-se-786bd599ff-g7945   1/1     Ru
helidon-quickstart-se-786bd599ff-h6c5n   1/1     Ru
helidon-quickstart-se-786bd599ff-n874p   1/1     Ru
```

You can just as easily scale your deployment back down; once again edit your deployment specification and apply the change:

```
$ grep replicas target/app.yaml
replicas: 2
$ kubectl apply -f target/app.yaml
service/helidon-quickstart-se unchanged
deployment.extensions/helidon-quickstart-se configu
$ kubectl get pods
NAME                                     READY   STA
helidon-quickstart-se-786bd599ff-h6c5n   1/1     Ru
helidon-quickstart-se-786bd599ff-n874p   1/1     Ru
```

There are more-advanced scaling features in Kubernetes, such as the Horizontal Pod Autoscaler, as well as features in managed cloud platforms to automatically scale up node resources on demand.

Next Steps

While it is beyond the scope of this article, there are several more-advanced Kubernetes features that are now within your grasp. The Kubernetes documentation is a great resource to start with.

When you are done working with your service, you can remove it from your cluster by using `kubectl delete`:

```
$ kubectl delete service helidon-quickstart-se
service "helidon-quickstart-se" deleted
$ kubectl delete deployment helidon-quickstart-se
deployment.extensions "helidon-quickstart-se" delete
```

Along with the extensive documentation, there are several free online training options to take advantage of. The Kubernetes community is very welcoming and helpful as well. In addition, a great source for learning, ideas, and support can be found in the Kubernetes Slack channel. Issues and pull requests are always welcome in the project on GitHub.

Conclusion

Containers are a powerful abstraction for microservices deployment. They allow you to decouple your service from the host environment, making it portable and scalable. As you can see from this brief introduction, Kubernetes makes containers manageable at scale.

Of course, these examples are just the beginning. Dig into the documentation and learn about more-advanced concepts. Welcome to Kubernetes development!

Also in This Issue

Jesse Butler

Jesse Butler is a Developer Advocate for Cloud Native technologies in the Oracle Cloud. Before joining the OCI team, he spent several years in platform and OS development.

Share this Page