Kubernetes in practice

Brian Storti

Welcome!

Thanks for checking out this sample of Kubernetes in Practice!

There are excerpts from three chapters included in this sample. We will start by deploying our first application with Kubernetes, then we will talk about how Pods work, and how to make our applications more reliable with Deployments.

If you get stuck at any point, don't hesitate to email me at brian@brianstorti.com or send me a message on twitter.

If you want the full book, you can get it at kubernetesinpractice.com. Enjoy!

Kubernetes in Practice

Brian Storti

Contents

Deploying our first application

Before we start talking about all the things you can do with Kubernetes, let's run a quick example so you can see how things work in practice. Don't worry if you don't understand everything we are doing, I just want to show you that getting an application to run on Kubernetes is not that complicated.

As we will discuss later, we usually work with Kubernetes in a declarative way, that is, instead of telling it what to do, we send it a manifest (as a `yaml` file) describing our desired state. In this case, we want to run `nginx`, so the manifest would be something like this:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx-container
    image: nginx
```

We'll look into these manifest files in more detail later, so don't worry about that for now. To send the manifest to Kubernetes we will use the `kubectl` command line tool, that you should already have installed.

Save this file as `nginx.yaml`, and then run

```
$ kubectl apply -f nginx.yaml
```

You should see a message saying the `nginx` pod was created. If you now run `kubectl get pods`, you will see our application is being created:

```
$ kubectl get pods

# NAME        READY   STATUS              RESTARTS
# nginx       0/1     ContainerCreating   0
```

The `ContainerCreating` status means Kubernetes is downloading the `nginx` image so we can run it locally. After a few seconds, if you run this command again, you will see the pod is now in the `Running` state.

And that's all we need to run an application with Kubernetes! Not that complicated, right?

But just seeing a `Running` state is not good enough, let's try to actually see this application in our browser to make sure it's running fine.

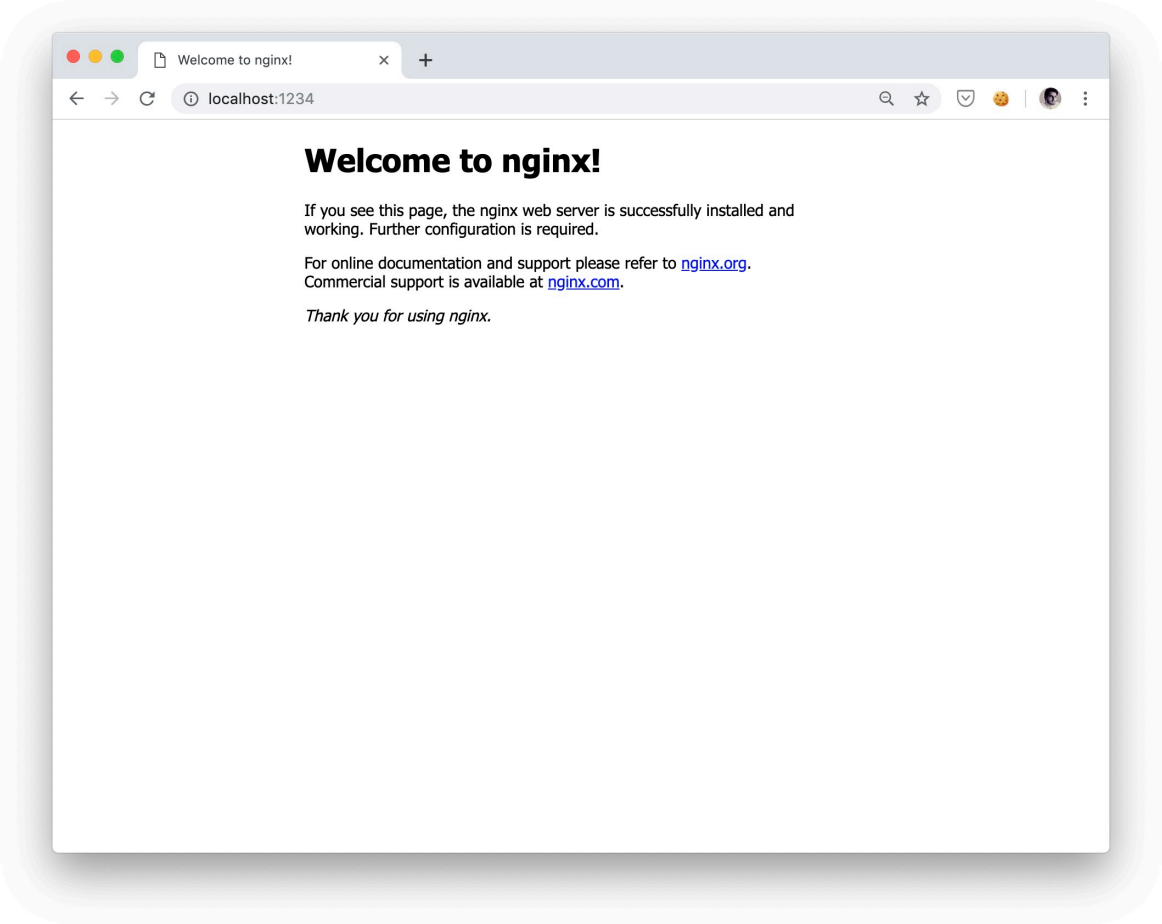
We'll later learn different ways to expose our applications so they can be accessed both by other applications running in our Kubernetes cluster and by our users, but for now let's just forward the requests for `localhost` to this application:

```
$ kubectl port-forward nginx 1234:80
```

We again used `kubectl`, but now calling the `port-forward` command. It will take the name of our application, that is `nginx`, and forward the requests

received locally on port `1234` to the container's port `80` (that's the default port `nginx` is listening to).

Now if you try to access `http://localhost:1234` in your browser, you should see the `nginx` welcome page.



Let's recap what we did:

1. We created a manifest file called `nginx.yaml` saying we wanted to run the `nginx` docker image
2. We sent this manifest to Kubernetes using `kubectl`
3. After we confirmed the application was running, we used `kubectl`

again to forward request received on port `1234` to our application.

And that's it, really. It took us less than 10 lines of `yaml` to run an application, as what's cool about this is that it's not at all different from what we would do if we were running Kubernetes in a cloud provider, like AWS, Google Cloud or Azure. The only thing that would change is that `kubectl` would be connected to a different cluster (instead of our local Kubernetes instance).

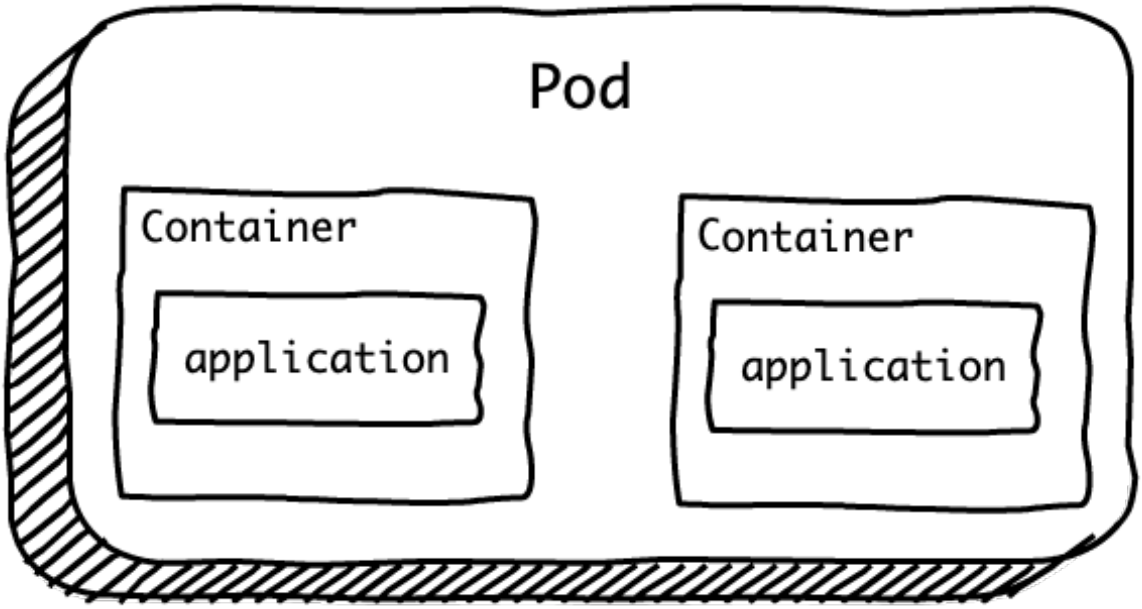
Of course, this is a very simple example and we are not doing anything super useful, but the principles used here will be applicable to almost everything we do in Kubernetes. We will write a manifest defining what we want, then we will give this manifest to Kubernetes, and it will do its best to make our desired state (e.g. to have an `nginx` container running) become reality. That is Kubernetes' *modus operandi*, it runs this infinite reconciliation loop checking what's our *desired* state and changing the *actual* state to match it, so we just need to learn how to speak its language to tell exactly what we want.

So let's get started!

Pods

Introduction

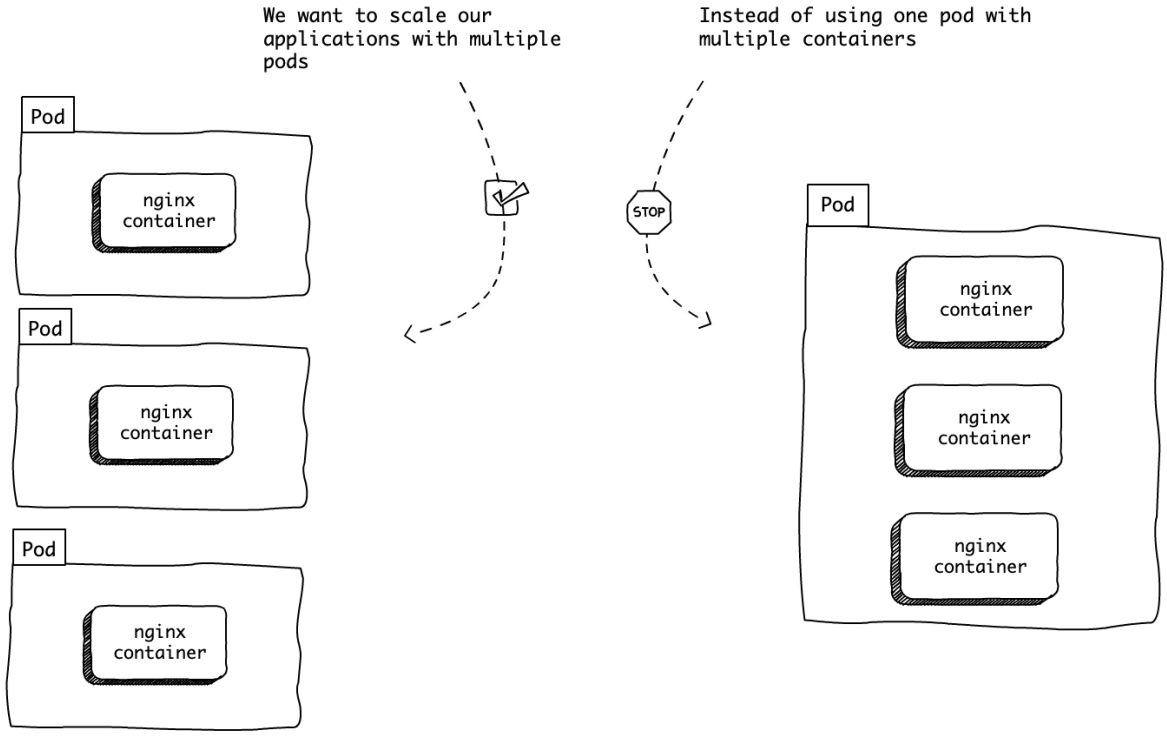
Pods are where our applications run. It's the basic building block and the atomic unit of scheduling in Kubernetes.
Each pod will include one or more containers and every time we run a container in Kubernetes, it will be inside a pod.



Pods are the atomic unit of scheduling

It is important to understand that a pod is the atomic unit of scheduling in Kubernetes, so if we want to run, say, 10 replicas of our application, we would create 10 pods, instead of creating 1 pod with 10 containers. We will

see more examples of that in practice when we start talking about deployments.



If you remember from a previous example, we had a manifest file like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx-container
    image: nginx
```

You will notice we have a `kind` key, saying that this manifest is defining a

Pod. Then we have the `metadata`, where we define a unique name for this pod (`nginx`), and a `spec` section, where we define what we want to run in this pod. In our case, we are running a container called `nginx-container`, and saying that the docker image used for this container is called `nginx`.

Most manifest files will follow a similar format. There are several other keys that can be defined, but this is the minimum we need to have a running pod.

Where does the image come from?

In this manifest we are defining `nginx` as the docker image name to run, but we never tell Kubernetes where it should look for this image.

The same way we can store code repositories in services like Github or Gitlab, we can store images in a docker registry (and there are several options available). When we don't specify which registry we want to use, Kubernetes will assume we mean DockerHub.

If we want to be more explicit we can change our manifest to use the full image path:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx-container
```

```
    image: registry.hub.docker.com/library/nginx
```

Or, if you are using a different service to store your images (say, ECR), we can define its full path as well:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx-container
    image: 579849551917.dkr.ecr.us-east-1.amazonaws.com/nginx
```

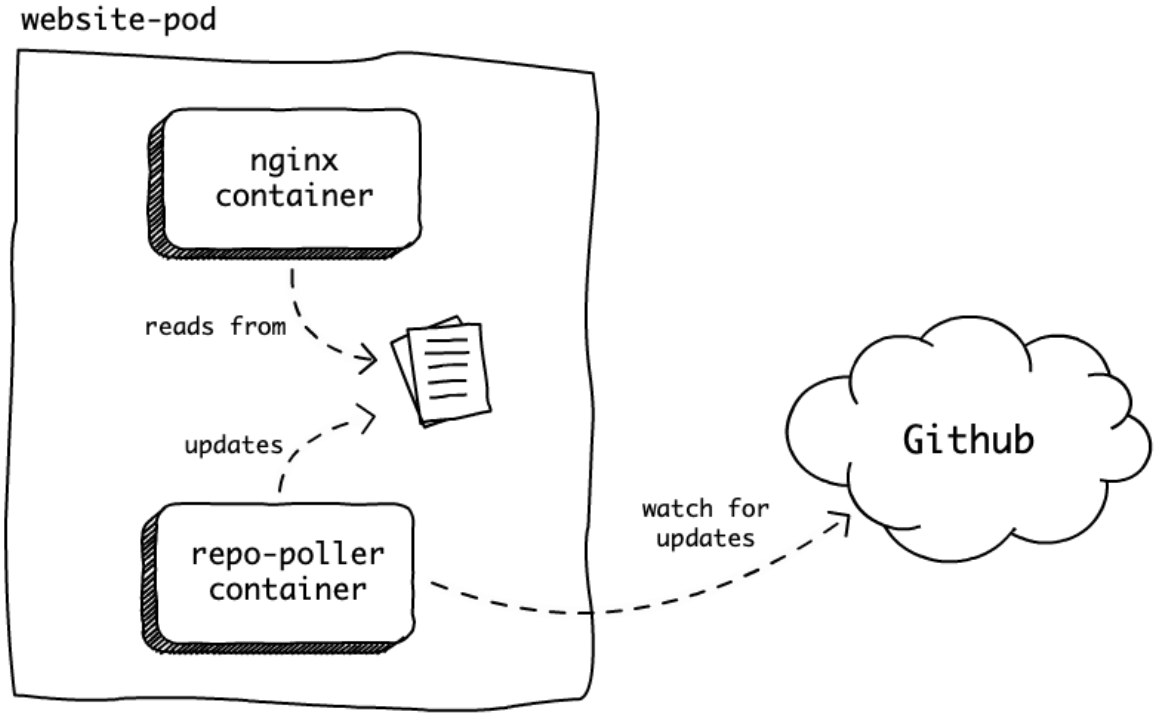
Multi-container Pods

At first glance a pod seems pretty similar to a container, but the main difference is that we can have multiple containers running inside a single pod, so we can think of a pod as a way to group containers that cooperate to do something.

In most cases, though, especially for simple applications like the ones we are running here, we will have only one container per pod.

When we run more than one container in a single pod, it's usually to support the primary application. For example, we could have our primary `nginx` con-

tainer running alongside another container that periodically pulls a `github` repository to update the website `nginx` is serving.



In this case, the primary container is `nginx`, that is serving our website, and the container that is getting the code from github is there just to help this pod serve its main function.

Playing with running pods

After we have a pod running, we can interact with it in a few different ways. Let's see a few examples.

If you don't have the pod from our previous example running, copy the man-

ifest to a file called `nginx.yaml` and run

```
$ kubectl apply -f nginx.yaml
```

To access the application from your browser, run

```
$ kubectl port-forward nginx 1234:80
```

Streaming logs

Our application is receiving requests, so it would be useful to see everything it's logging to `stdout`.

We can do that directly with the `kubectl logs` command:

```
$ kubectl logs --follow nginx
               ^        ^
               |        |
                ------------- tells kubectl to keep
                       |     streaming the logs live,
                       |     instead of just
                       |     printing and exiting
                       |
                        ---- the name of the pod we want
                             to see the logs
```

Now if you try to access `http://localhost:1234` again, you should see the logs showing up here.

Executing commands

We can also execute arbitrary commands in a container running in our pod. It's like opening an `ssh` connection to a machine where our application is running and being able to inspect it. Let's start with a simple example, where we just run the `ls` command in our `nginx` container:

```
$ kubectl exec nginx ls
# bin
# boot
# dev
# etc
# home
# ...
```

We call the `exec` command, passing the name of the pod and the command we want to run, and the output is printed in our terminal.

We can also run an interactive session using the `-it` flags. For example, we can start a `bash` session in this container:

```
$ kubectl exec -it nginx bash

root@nginx:/#
```

And now you should be inside the container! And again, it doesn't matter where this container is running, if it's in your local machine or in an `EC2` instance in AWS, the way you interact with it through `kubectl` is exactly the

same.

Let's try to change something in this container to see what happens:

```
$ root@nginx:/# echo "it works!" > /usr/share/nginx/html/index.html
```

When we access `http://localhost:1234`, `nginx` is serving the static html present in this location (`/usr/share/nginx/html/index.html`), so we are just overriding this html file with the string `"it works!"`, and now if you try refreshing the page you will see this change.

This is certainly not something you will want to do in a production environment (also because as soon as the pod is restarted our changes are lost) but it can be useful to poke around, debug issues and understand why a container behaves the way it does.

Killing pods

Lastly, we can kill our pod, either because we don't need it anymore or to simulate a failure. We can do that in two different ways.

First, killing it by name:

```
$ kubectl delete pod nginx
# pod "nginx" deleted
```

Or by sending the same manifest file we used to create the pod:

```
$ kubectl delete -f nginx.yaml
# pod "nginx" deleted
```

And now if you try to list your pods, `nginx` should be gone:

```
$ kubectl get pods
# No resources found.
```

Although that's what we expect from this example, it also exposes a problem we have: If for some reason our application crashes (and it will crash, eventually), it's not automatically rescheduled.

For that reason, we'll usually not create pods directly, as we did here, but instead use a higher-level object called `Deployment` to create and manage our pods, as we will see in the next chapter.

It's your turn

Almost everything in Kubernetes will work around pods, so it's very important to understand what they are and how to run and interact with them. I encourage you to try to run a pod yourself.

In the examples so far we have been using `nginx`, so a good exercise would be to try to run `Apache`, that is another web server that works similarly. Here's a list of things you could do:

- Write a pod manifest file to run the apache container (the docker image

is called `httpd`).

- Use `kubectl` to apply this manifest and see it running.
- Use `kubectl port-forward` to send requests from `localhost:1234` to the container's port `80`.
- Confirm you can see Apache's default page saying "It works!" when you access `localhost:1234`.
- Enter the container and change the contents of the file `/usr/local/apache2/htdocs/index.html` so when you refresh the page you can see your changes.

If you can do that, well done! You already understand the main concept we will be working with. From now on almost everything we will do is to ensure these pods are running the way they should.

Recap

- A pod is the atomic unit of scheduling in Kubernetes. It's where our containers run.
- We can have multiple containers running in the same pod, but we usually have one primary container and the others just helping the primary.
- We don't scale applications by creating a pod with multiple containers, but by creating multiple pods.
- We can interact with running pods through `kubectl`.
- Pods are not rescheduled automatically when they die.

Deployments

Introduction

We saw that when a pod dies, either because we manually kill it or because something unexpected happens, Kubernetes will not automatically reschedule it.

That's one of the reasons why we will almost never run pods directly. We will almost always want to manage pods with another Kubernetes resource called `Deployment`. Other than making sure our pods are rescheduled when they die, deployments will also help us with several other things:

- We can use deployments to scale our applications, increasing/decreasing the number of replicas we have running (so far we have only run `1` replica of our application)
- Deployments can handle the rollout of new versions of our application, so we can go from `v1` to `v2` without any downtime
- And they also allow us to easily rollback bad releases, as well as preventing bad releases from going through altogether in some cases

Before we start talking about how Deployments work, let's take a look at the application we will run. So far we have only used the `nginx` image because it was super simple and convenient to demonstrate how to run an application in Kubernetes. Now we are going to run our own application so we can more easily control different versions and introduce the behavior we want to test

our use cases.

I wrote this very simple application in Ruby, but you can change it if you want and use any other tool you prefer. As long as it's packaged as a docker image, everything will work the same way.

Here's the application code:

```
# app.rb
require "sinatra"

set :bind, "0.0.0.0"

get "*" do
  "[v1] Hello, Kubernetes!\n"
end
```

As you can see, there's not much going on. It's using `Sinatra`, that is a Ruby library that provides a DSL to create web applications. In this case, we are accepting requests to any path and returning the string `"[v1] Hello, Kubernetes!"`.

To package this application in a docker image, we can use a `Dockerfile` like this one:

```
FROM ruby:2.6.1-alpine3.9

RUN apk add curl && gem install sinatra
COPY app.rb .
```

```
ENTRYPOINT ["ruby", "app.rb"]
```

Here we define that our base image is an official Ruby image, install `sinatra`, copy the `app.rb` file (that has the code we just saw), and define the command `ruby app.rb` as the image's entrypoint.

And that's all the code we need for now.

You can push this image to your DockerHub account if you have one, or, if you prefer, you can just use the image from my repository. To use your own image just replace everywhere you see `brianstorti` with your account:

```
# Build the image
$ docker build . -t brianstorti/hellok8s:v1

# And push it to DockerHub
$ docker push brianstorti/hellok8s:v1
```

After the image is uploaded, we are ready to use it in our manifests!

Defining our Deployment manifest

Just like we defined a manifest for our `nginx` pod, we need to define a manifest for a deployment, and they actually look fairly similar:

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: hellok8s
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hellok8s
  template:
    metadata:
      labels:
        app: hellok8s
    spec:
      containers:
      - image: brianstorti/hellok8s:v1
        name: hellok8s-container
```

Let's go through all the fields in this manifest to understand what's happening. First, we tell Kubernetes what `kind` of object we are defining here. In our case, this is a `Deployment`.

In the `metadata.name` field we define a unique name for our deployment. This can be anything that helps you identify what this deployment is managing.

In the `spec` section is where we define what this deployment will do. First, we define how many `replicas` we want. That is, how many exact copies of

this pod we want to run. For now we're running just 1 replica, but we will play with that field in a bit.

The `selector` section is probably the only thing that's not super intuitive right away. What we are doing in this field is telling Kubernetes that this deployment is managing all the pods that have a label called `app` with the value `hellok8s`. This is what links deployments to pods. Labels are simply key-value pairs that you define for your pods, and that's what is used to find all the pods that a Deployment needs to look after.

Then we define the `template` section, that's the definition of pods we want to run. Here we are defining a pod with the label `app: hellok8s` (to create the link with the deployment), and saying this pod will run the `brianstorti/hellok8s:v1` docker image.

This can be a bit daunting the first time you see, but as you start using and writing more manifests it will become natural, I promise.

Okay, enough talking, let's see this in action. Save this file as `deployment.yaml` and apply it to your cluster:

```
$ kubectl apply -f deployment.yaml
# deployment.apps/hellok8s created
```

Now if you inspect the deployments you have running, you should see our new `hellok8s` there:

```
$ kubectl get deployments
```

```
# NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
# hellok8s    1         1         1            1
```

And if you inspect your pods, you will see that a new pod was also created by this deployment.

```
$ kubectl get pods

# NAME                          READY   STATUS    RESTARTS
# hellok8s-6678f66cb8-42jtr     1/1     Running   0
```

So there are a few things to notice here.

First, in the deployments output you see a couple of columns to indicate the state of this deployment. We said we wanted only 1 replica of this pod, so that's what we have in the `DESIRED` column. In the `CURRENT` column we have the actual number of pods that we have running at this moment, and Kubernetes will always try to match it to our desired state. In this case, it also says 1, so things look fine. We'll check what the `UP-TO-DATE` and the `AVAILABLE` columns mean in a little bit.

From the pods output, you will notice that the pod name now has a somewhat random suffix (`"-6678f66cb8-42jtr"` in this example, but it will be different on your machine). It needs to have this suffix because it's being managed by our deployment, and pods can come and go (as we will soon confirm), and we can also have multiple replicas for this pod, so they can't all have the same name. The thing to keep in mind is that the name of your

pods are not stable and they can change for various reasons.

This pod will behave exactly the same as the ones we created before, so all the things we could do with the `nginx` pod we created directly, we can also do with this one. For example, let's try to use `port-forward` again to send it a request:

```
# replace the pod name to what you have running locally
$ kubectl port-forward hellok8s-6678f66cb8-42jtr 1234:4567
```

And if you try to access `http://localhost:1234` in your browser you should see our app's response: `"[v1] Hello, Kubernetes!"`.

Cool, we see this deployment is just creating a pod for us, and that's what we already had before when we created the pod directly, so let's see what makes deployments so special.

Restarting failed pods

The problem we noticed with pods is that they were not automatically rescheduled when they died, so let's try to kill this pod now to see what happens.

```
$ kubectl get pods
# NAME                          READY   STATUS    RESTARTS
# hellok8s-6678f66cb8-42jtr     1/1     Running   0
```

```
$ kubectl delete pod hellok8s-6678f66cb8-42jtr
# pod "hellok8s-6678f66cb8-42jtr" deleted

$ kubectl get pods
# NAME                          READY   STATUS    RESTARTS
# hellok8s-6678f66cb8-8nqf2     1/1     Running   0
```

So that's nice, we see that as soon the deployment notices our pod died, it starts a new one. That goes back to the reconciliation loop we talked about, Kubernetes is always trying to ensure our desired state matches our current state, and if it doesn't (like when the actual number of pods we had running went from 1 to 0 because we killed it), it will take the necessary actions to go back to a stable state, that in this case means starting a new pod.

Scaling up our application

Another cool thing that deployments can do is to scale up and down the number of replicas we have running. Right now we are running a single container for our application. Changing that is just a matter of updating our manifest file with our new desired number of replica and sending that to Kubernetes:

```
# deployment.yaml

apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: hellok8s
spec:
+ replicas: 10
  selector:
    matchLabels:
      app: hellok8s
  template:
    metadata:
      labels:
        app: hellok8s
    spec:
      containers:
      - image: brianstorti/hellok8s:v1
        name: hellok8s-container
```

The manifest is exactly the same, except for the `replicas` field that we changed from 1 to 10. When we apply this change, we should see a bunch of new containers being created:

```
$ kubectl apply -f deployment.yaml
# deployment.apps/hellok8s configured
```

Kubernetes notices this deployment already exists, so it just configures it to use 10 replicas instead of 1.

```
$ kubectl get deployments
# NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
# hellok8s   10        10        10           1
```

Inspecting the deployment we see that we now have 10 replicas, but only 1 is available. That's because the other 9 containers are still being created, as we can see by inspecting the running pods:

```
$ kubectl get pods
# NAME                        READY   STATUS
# hellok8s-6678f66cb8-8nqf2   1/1     Running
# hellok8s-6678f66cb8-6r7fb   0/1     Pending
# hellok8s-6678f66cb8-7bg4s   0/1     Pending
# hellok8s-6678f66cb8-96xh5   0/1     Pending
# hellok8s-6678f66cb8-cmb4j   0/1     ContainerCreating
# hellok8s-6678f66cb8-h5tg4   0/1     ContainerCreating
# hellok8s-6678f66cb8-j2b5n   0/1     Pending
# hellok8s-6678f66cb8-l5hzw   0/1     Pending
# hellok8s-6678f66cb8-r9bzd   0/1     ContainerCreating
# hellok8s-6678f66cb8-wl4bb   0/1     ContainerCreating
```

After a few short seconds the number of available pods should go to 10 as well, and all the pods will be running.

```
$ kubectl get deployments
# NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
# hellok8s   10        10        10           10
```

```
$ kubectl get pods
# NAME                          READY   STATUS    RESTARTS
# hellok8s-6678f66cb8-8nqf2     1/1     Running   0
# hellok8s-6678f66cb8-6r7fb     1/1     Running   0
# hellok8s-6678f66cb8-7bg4s     1/1     Running   0
# hellok8s-6678f66cb8-96xh5     1/1     Running   0
# hellok8s-6678f66cb8-cmb4j     1/1     Running   0
# hellok8s-6678f66cb8-h5tg4     1/1     Running   0
# hellok8s-6678f66cb8-j2b5n     1/1     Running   0
# hellok8s-6678f66cb8-l5hzw     1/1     Running   0
# hellok8s-6678f66cb8-r9bzd     1/1     Running   0
# hellok8s-6678f66cb8-wl4bb     1/1     Running   0
```

And there we have it, 10 replicas of our application running.

A note about how containers are scheduled

Right now, as we are running Kubernetes locally, we have only 1 worker node running our applications, so all these pods will run on this node. In a production environment, where we will likely have several worker nodes in our cluster, Kubernetes will try to schedule these pods across different nodes so even if one of the nodes fails our application will still be up.

Scaling it down

You can probably imagine what we need to do scale the number of pods down, now. Just change the deployment manifest to match the number of replicas you want and Kubernetes will make sure to terminate the pods for us.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hellok8s
spec:
+ replicas: 2
  selector:
    matchLabels:
      app: hellok8s
  template:
    metadata:
      labels:
        app: hellok8s
    spec:
      containers:
      - image: brianstorti/hellok8s:v1
        name: hellok8s-container
```

```
$ kubectl apply -f deployment.yaml
# deployment.apps/hellok8s configured
```

```
$ kubectl get pods
# NAME                          READY   STATUS
# hellok8s-6678f66cb8-8nqf2     1/1     Running
# hellok8s-6678f66cb8-cmb4j     1/1     Running
# hellok8s-6678f66cb8-6r7fb     1/1     Terminating
# hellok8s-6678f66cb8-7bg4s     1/1     Terminating
# hellok8s-6678f66cb8-96xh5     1/1     Terminating
# hellok8s-6678f66cb8-h5tg4     1/1     Terminating
# hellok8s-6678f66cb8-j2b5n     1/1     Terminating
# hellok8s-6678f66cb8-l5hzw     1/1     Terminating
# hellok8s-6678f66cb8-r9bzd     1/1     Terminating
# hellok8s-6678f66cb8-wl4bb     1/1     Terminating
```

And after a few seconds you should see the 8 exceeding pods gone.

Rolling out releases

Another thing that deployments can help us with is rolling out new versions of our application. Let's first create a new version of our service to test that out. Here's the code for our v2:

```
get "*" do
  "[v2] Hello, Kubernetes!\n"
end
```

Then we build and push this image to DockerHub again to be able to start

using it in our manifest.

```
$ docker build . -t brianstorti/hellok8s:v2
$ docker push brianstorti/hellok8s:v2
```

Now we have the tags `v1` and `v2` for this same image to start playing with.

Releasing this new version is as easy as updating the manifest file to point to the version we want to use.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hellok8s
spec:
  replicas: 2
  selector:
    matchLabels:
      app: hellok8s
  template:
    metadata:
      labels:
        app: hellok8s
    spec:
      containers:
-     - image: brianstorti/hellok8s:v1
+     - image: brianstorti/hellok8s:v2
```

```
        name: hellok8s-container
```

Again, the only thing we had to change was the image that this deployment is using. Now if you apply this manifest and keep watching the pods we have running, you will see Kubernetes starting new pods that use the `v2` image and terminating the old pods

> *You can use the* `--watch` *flag to watch changes to a command output. For example:* `kubectl get pods --watch` *will print a new line for every change in its output. Sometimes it can be hard to see what's happening with the format of this output, so I personally prefer the* `watch` *command that will keep rerunning a given command every few seconds to show the latest outputs:*
>
> `$ watch kubectl get pods`
>
> *You may need to install* `watch` *in your system to be able to use it.*

```
$ kubectl apply -f deployment.yaml
# deployment.apps/hellok8s configured

$ kubectl get pods
# You should see new pods being created
# and the old ones being terminated.
```

After Kubernetes finishes running, you can try to access the service again to make sure you have `v2` running:

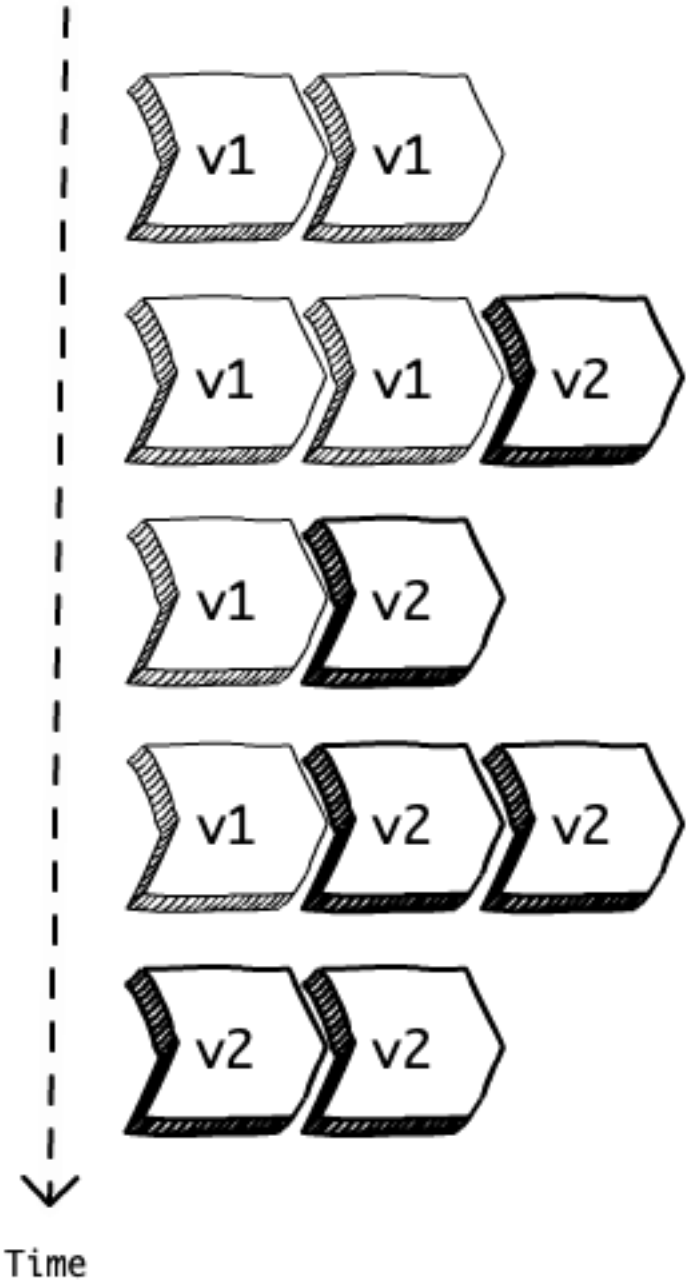
```
$ kubectl get pods
# NAME                          READY   STATUS
# hellok8s-6678f66cb8-52zt9     1/1     Running
# hellok8s-6678f66cb8-nxphs     1/1     Running

# make sure you replace the pod name
$ kubectl port-forward hellok8s-6678f66cb8-52zt9 1234:4567

$ curl http://localhost:1234
# [v2] Hello, Kubernetes!
```

Cool, it works! And that seems pretty simple, but let's take a moment to think about all the work Kubernetes is doing for us.

The kind of release that was done is called a `RollingUpdate`, which means we first create one pod with the new version, and, after it's running, we terminate one pod running the previous versions, and we keep doing that until all the pods running are using the desired version.



Let's say we have 10 pods running v1 and we wanted to manually release v2. We would need to manually start a new pod, wait until it was up and running, then terminate one of the pods still running v1, and repeat that process 10

times.

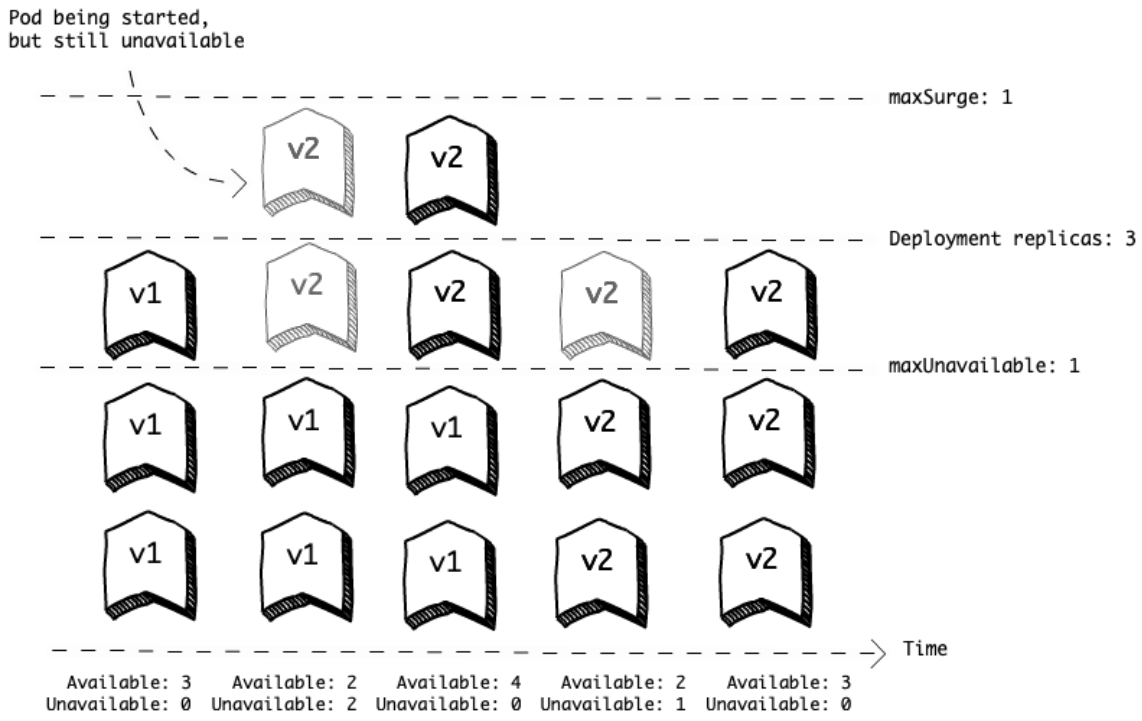
Controlling the rollout rate

In this case we have only 2 replicas, so rolling out a new version doesn't take too long. But imagine if we had, say, 100 replicas running and we wanted to rollout a new version in the same way, adding one new pod with the new version, waiting until it's ready, and removing one pod with the old version. It would take forever!

There are two properties we can use to define how fast our rollout will happen: `maxSurge` and `maxUnavailable`.

The `maxSurge` property will define how many pods we can have exceeding our desired `replica` count (specified in the deployment manifest), and `maxUnavailable` defines how many pods we can have below this count. These properties can be defined as an absolute number (e.g. `10`) or as a percentage (e.g. `20%`). The default value for both is `25%`.

Here's an example of how that would work, assuming a Deployment with 3 replicas and a `maxSurge` and `maxUnavailable` of `1` .



Kubernetes will ensure that at any point in time during this rollout we will have a minimum of 2 (`desired - maxUnavailable`) and maximum of 4 (`desired + maxSurge`) replicas.

To change the default value (25%) for these properties, you can define this in your manifest:

```
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hellok8s
spec:
+  strategy:
```

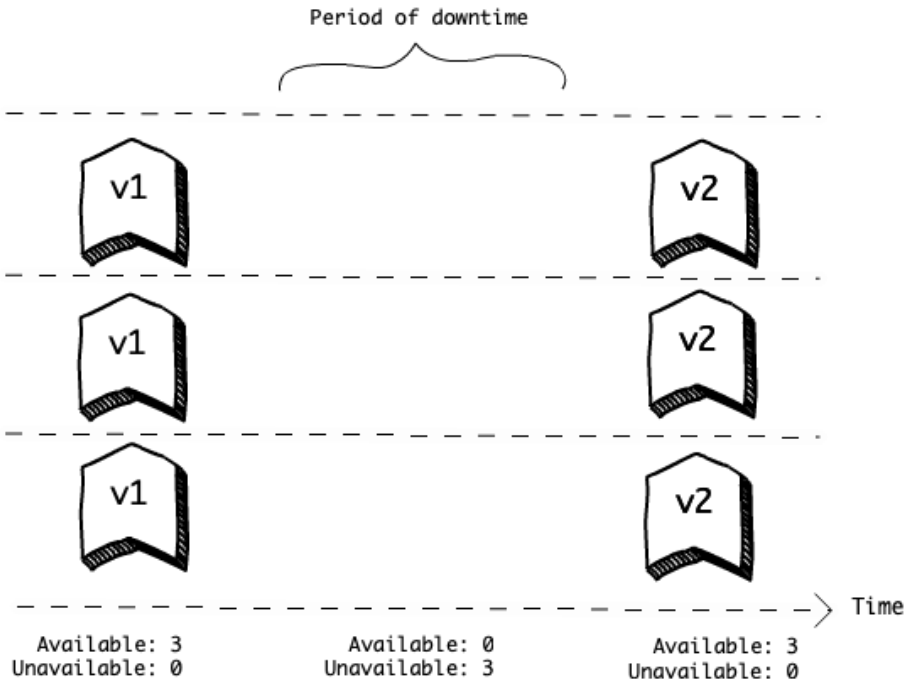
```
+    rollingUpdate:
+      maxSurge: 1
+      maxUnavailable: 1
  replicas: 3
  selector:
    matchLabels:
      app: hellok8s
  template:
    metadata:
      labels:
        app: hellok8s
    spec:
      containers:
      - image: brianstorti/hellok8s:v2
        name: hellok8s-container
```

Using a different rollout strategy

As you may have noticed, when we are using the `rollinUpdate` strategy (which is the default) we will have, for a period of time, both versions of our application (`v1` and `v2`) running in parallel. If for some reason you don't want that to happen, Deployments can be configured with e different strategy called `Recreate`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hellok8s2
spec:
+  strategy:
+    type: Recreate
  replicas: 3
  selector:
    matchLabels:
      app: hellok8s
  template:
    metadata:
      labels:
        app: hellok8s
    spec:
      containers:
      - image: brianstorti/hellok8s:v2
        name: hellok8s-container
```

And what will happen is that all your pods will be terminated before pods using the new version are created (at the cost of having a period of downtime while the new pods are being created)[^1]:



Recap

- Deployments are high level resources that will manage pods for us.
- They can help us scale our application up and down by creating and deleting replicas of our pods.
- We can use two different strategies to rollout new versions of our applications: `Recreate` and `RollingUpdate`.
- The `Recreate` strategy will guarantee we don't have different versions running at the same time, at the cost of requiring a short downtime.
- `RollingUpdate` is the default strategy and will gradually create pods that use the new version while terminating pods running the previous versions.

- We can use `maxSurge` and `maxUnavailable` to control the rollout rate.

Thank you!

Thanks for reading this sample from *Kubernetes in Practice.* I hope you enjoyed it! If you want the full book, you can get it at kubernetesinpractice.com.

If you have any questions, feel free to email me at brian@brianstorti.com or send me a message on twitter.

Have a great day!