Kubernetes

Raúl Estrada

Octubre 2020

Pods

Pods are the most basic element of the Kubernetes API. A Pod basically is a set of containers that work together in order to provide a service or part of it. The concept of Pod is something that can be misleading. The fact that we can run several containers working together suggests that we should be sticking the frontend and backend of our application on a single pod as they work together. Even though we can do this, it is a practice that I would strongly suggest you avoid. The reason for this is that by bundling together the frontend and the backend, we are losing a lot of flexibility that Kubernetes is providing us with, such as autoscaling, load balancing, or canary deployments.

In general, pods contain a single container and it is, by far, the most common use case, but there are few legitimate use cases for multi-container pods:

- Cache and cache warmer
- Precalculating and serving HTML pages
- File upload and file processing

As you can see, all of these are activities that are strongly coupled together, but if the feeling is that the containers within a pod are working toward different tasks (such as backend and frontend), it might be worth placing them in different Pods.

There are two options for communication between containers inside a pod:

- Filesystem
- Local network interface

As Pods are indivisible elements running on a single machine, volumes mounted in all the containers of a pod are shared: files created in a container within a pod can be accessed from other containers mounting the same volume.

The local network interface or loopback is what we commonly know as localhost. Containers inside a pod share the same network interface; therefore, they can communicate via localhost (or `127.0.0.1`) on the exposed ports.

Deploying a pod

As mentioned earlier, Kubernetes relies heavily on **Yet Another Markup Language** (**YAML**) files to configure API elements. In order to deploy a pod, we need to create a yaml file, but first, just create a folder called **deployments**, where we are going to create all the descriptors that we will be created on this section. Create a file called `pod.yaml` (or `pod.yml`) with the following content:

Copy

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    name: nginx
spec:
  containers:
  - name: nginx
     image: nginx
     ports:
      - containerPort: 80
     resources:
       requests:
         memory: "64Mi"
         cpu: "250m"
```

As you can see, the preceding `yaml` is fairly descriptive, but some points need clarification:

- `apiVersion` : This is the version of the Kubernetes API that we are going to use to define our resource (in this case, pod). Kuberentes is a living project that evolves very quickly. The version is the mechanism used to avoid deprecating resources with new releases. In general, Kuberentes works with three branches: alpha, beta, and stable. In the preceding case, we are using the stable version. More information can be found at https://kubernetes.io/docs/concepts/overview/kubernetes-api/.
- `metadata` : In this section, we are defining one of the most powerful discovery mechanisms that I have ever seen: the pattern matching. The section label, specifically, will be used later on to expose pods with certain labels to the outer world.
- `spec` : This is where we define our container. In this case, we are deploying an `nginx` instance so that we can easily see how everything works without focusing too much on the application itself. As expected, the image and the exposed port have been specified. We have also defined the CPU and memory limitations for this Pod, so we prevent an outbreak in resource consumption (note that the YAML file is requesting the resources; they might not be available so the pod will operate with lower profile resources).

This is the simplest configuration for an item that we can create in Kubernetes. Now it's time to deploy the resource in our cluster:

```
kubectl apply -f pod.yml
```

This will produce an output similar to the following one:

```
pod "nginx" created.
```

Disclaimer: there are several ways of creating a resource, but in this book, I will use `apply` as much as possible. Another possibility would be to use `create` :

```
kubectl create -f pod.yml
```

The advantage that `apply` has over create is that apply does a three-way diff between the previous version, the current version, and the changes that you want to apply and decides how is best to update the resource. This is letting Kubernetes do what it does best: automate container orchestration.

With create, Kubernetes does not save the state of the resource, and if we want to run apply afterward in order to gracefully change the state of a resource, a warning is produced:

Copy

```
Warning: kubectl apply should be used on resource created by either kubectl create --save-config or kubectl
pod "nginx" configured
```

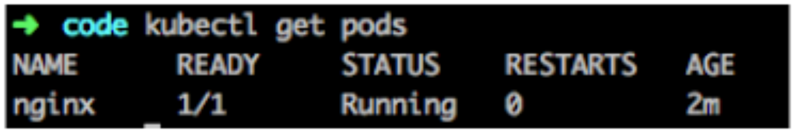
This means that we can push our system to an unstable state for few seconds, which might not be acceptable depending on your use case.

Once we have applied our YAML file, we can use `kubectl` to see what is going on in Kubernetes. Execute the following command:

Copy

```
kubectl get pods
```

This will output our pod:



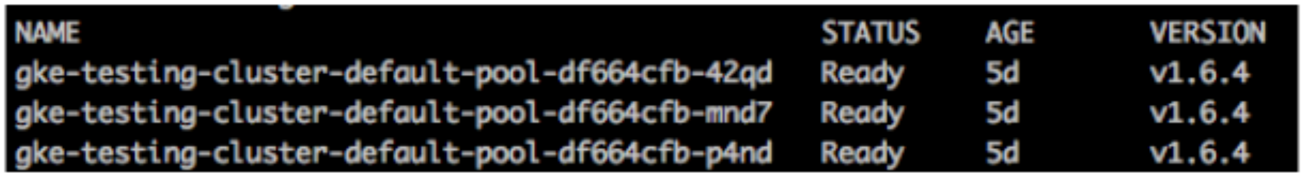
```
➜  code kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
nginx     1/1       Running   0          2m
```

We can do this for other elements of our cluster, such as the nodes:

Copy

```
kubectl get nodes
```

And this will output the following:



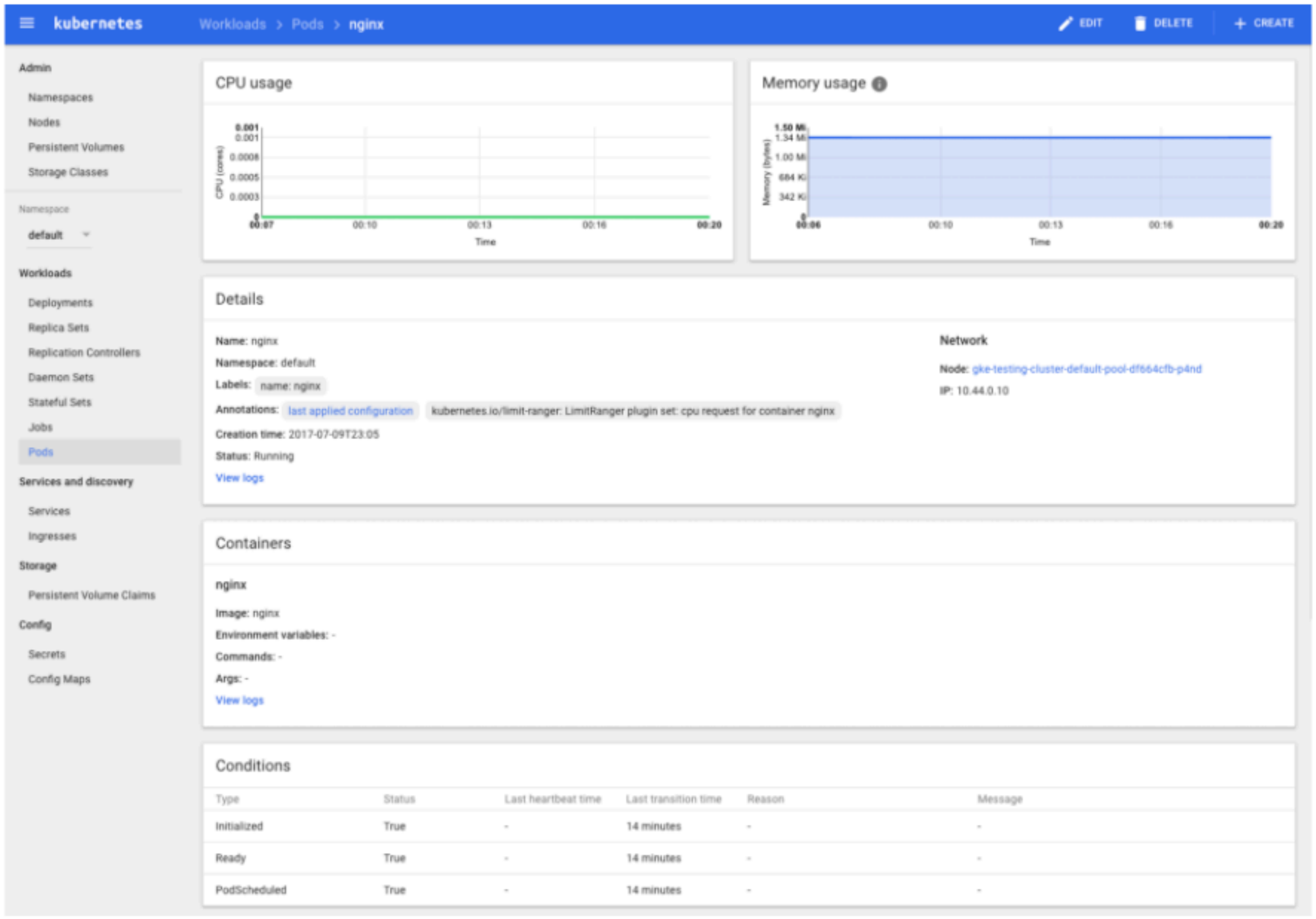
```
NAME                                         STATUS    AGE       VERSION
gke-testing-cluster-default-pool-df664cfb-42qd   Ready     5d        v1.6.4
gke-testing-cluster-default-pool-df664cfb-mnd7   Ready     5d        v1.6.4
gke-testing-cluster-default-pool-df664cfb-p4nd   Ready     5d        v1.6.4
```

The `kubectl get` works for all the workflows in Kubernetes and the majority of the API objects.

Another way of seeing what is going on in Kubernetes is using the dashboard. Now that we have created a pod, open the dashboard at `http://localhost:8001/ui` and navigate to the pods section on the left-hand side.

> **Note**
>
> Remember that in order to access the dashboard, first, you need to execute `kubectl proxy` on a Terminal.

There; you will see the list of the current deployed pods, in this case, just `nginx`. Click on it and the screen should look very similar to what is shown here:



Details

Name: nginx
Namespace: default
Labels: name: nginx
Annotations: last applied configuration kubernetes.io/limit-ranger: LimitRanger plugin set: cpu request for container nginx
Creation time: 2017-07-09T23:05
Status: Running
View logs

Network
Node: gke-testing-cluster-default-pool-df664cfb-p4nd
IP: 10.44.0.10

Containers

nginx
Image: nginx
Environment variables: -
Commands: -
Args: -
View logs

Conditions

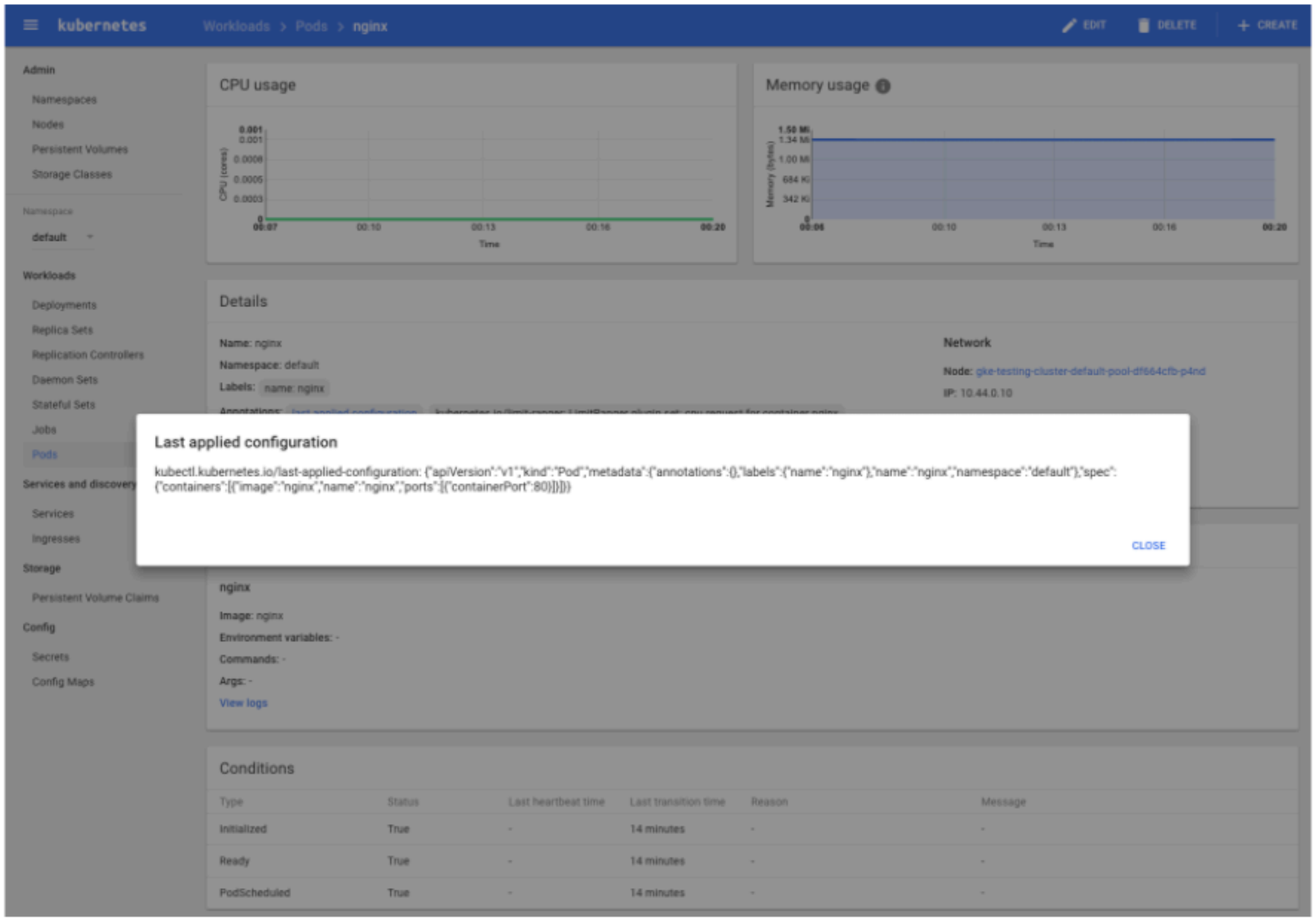
Type	Status	Last heartbeat time	Last transition time	Reason	Message
Initialized	True	-	14 minutes	-	-
Ready	True	-	14 minutes	-	-
PodScheduled	True	-	14 minutes	-	-

Here, we get a ton of information, from the memory and CPU that the pod is consuming to the node where it is running and a few other valuable items, such as the annotations applied to the pod. We can get this using the ' `describe` ' command of `kubectl` , as follows:

Copy

```
kubectl describe pod nginx
```

Annotations are a new concept and are the data around our API element, in this case, our pod. If you click on L`ast applied configuration` in the Details section, you can see the data from the YAML file, as shown in the following screenshot:



Details

Name: nginx
Namespace: default
Labels: name: nginx

Network
Node: gke-testing-cluster-default-pool-df664cfb-p4nd
IP: 10.44.0.10

Last applied configuration

```
kubectl.kubernetes.io/last-applied-configuration: {"apiVersion":"v1","kind":"Pod","metadata":{"annotations":{},"labels":{"name":"nginx"},"name":"nginx","namespace":"default"},"spec":
{"containers":[{"image":"nginx","name":"nginx","ports":[{"containerPort":80}]}]}}
```

CLOSE

nginx
Image: nginx
Environment variables: -
Commands: -
Args: -
View logs

Conditions

Type	Status	Last heartbeat time	Last transition time	Reason	Message
Initialized	True	-	14 minutes	-	-
Ready	True	-	14 minutes	-	-
PodScheduled	True	-	14 minutes	-	-

And this relates to the three-way diff that was explained earlier and is used by Kubernetes to decide the best way of upgrading a resource without getting into an inconsistent state.

As of now, our pod is running in Kubernetes but is not connected to the outer world; therefore, there is no way to open a browser and navigate to the `nginx` home page from outside the cluster. One thing that we can do is open a remote session to a bash Terminal in the container inside the pod in a manner similar to what we would do with Docker:

Copy

```
kubectl exec -it nginx bash
```

And we are in. Effectively, we have gained access to a root terminal inside our container and we can execute any command. We will use this functionality later on.

Once we have seen how pods work, you might have a few questions abound what Kubernetes is supposed to do:

- How can we scale pods?
- How can we roll out new versions of an application?
- How can we access our application?

We will answer all these questions, but first, we need to know other 'building blocks'.

Replica Sets

So far, we know how to deploy applications in pods. The sole concept of pod is very powerful, but it lacks robustness. It is actually impossible to define scaling policies or even make sure that the pods remain alive if something happens (such as a node going down). This might be okay in some situations, but here is an interesting question. If we are biting the bullet on the overhead of maintaining a Kubernetes cluster, why don't we take the benefits of it?

In order to do that, we need to work with **Replica Sets**. A Replica Set is like a traffic cop in a road full of pods: they make sure that the traffic flows and everything works without crashing and moving the pods around so that we make the best use of the road (our cluster, in this case).

Replica Sets are actually an update of a much older item: the Replication Controller. The reason for the upgrade is the labeling and selecting of resources, which we will see visit when we dive deep into the API item called Service.

Let's take a look at a Replica Set:

Copy

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
   name: nginx-rs
spec:
   replicas: 3
   template:
      metadata:
         labels:
            app: nginx
            tier: frontend
      spec:
         containers:
         - name: nginx
            image: nginx
            resources:
               requests:
                  cpu: 256m
                  memory: 100Mi
           ports:
```

Again, this a YAML file that is basically fairly easy to understand but might require some explanation:

- In this case, we have used the extensions API on the version `v1beta1` . If you remember from the pod section (previously), Kubernetes has three branches: stable, alpha, and beta. The complete reference can be found in the official documentation, and it is very likely to change often as Kubernetes is a vibrant and always evolving project.
- In the spec section is where the important things happen: we have defined a set of labels for the Replica Set, but we have also defined a pod (in this case, with a single container) and specified that we want three instances of it (replicas: three).

Simple and effective. Now we have defined a resource called Replica Set, which allows us to deploy a pod and keep it alive as per configuration.

Let's test it:

```
kubectl apply -f replicaset.yml
```

Once the command returns, we should see the following message:

```
replicaset "nginx-rs" created
```

Let's verify it using `kubectl` :

Copy

```
kubectl get replicaset nginx-rs
```

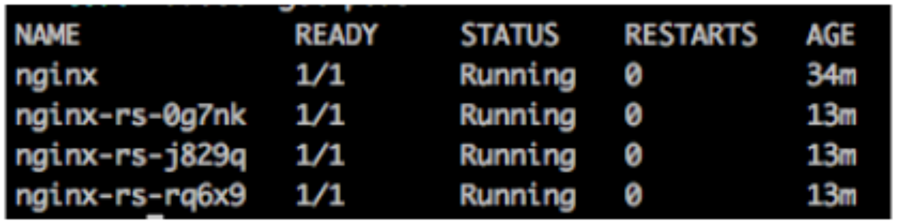
As the output of the preceding command, you should see the Replica Set explaining that there are three desired pods, three actually deployed, and three ready. Note the difference between current and ready: a pod might be deployed but still not ready to process requests.

We have specified that our `replicaset` should keep three pods alive. Let's verify this:

Copy

```
kubectl get pods
```

No surprises here: our `replicaset` has created three pods, as shown in the following screenshot:



```
NAME            READY     STATUS    RESTARTS   AGE
nginx           1/1       Running   0          34m
nginx-rs-0g7nk  1/1       Running   0          13m
nginx-rs-j829q  1/1       Running   0          13m
nginx-rs-rq6x9  1/1       Running   0          13m
```

We have four pods:

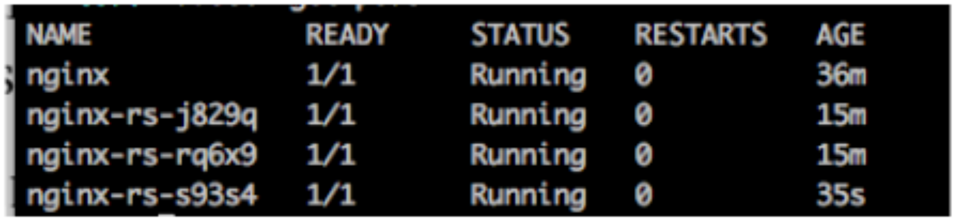
- One created in the preceding section
- Three created by the Replica Set

Let's kill one of the pods and see what happens:

Copy

```
kubectl delete pod nginx-rs-0g7nk
```

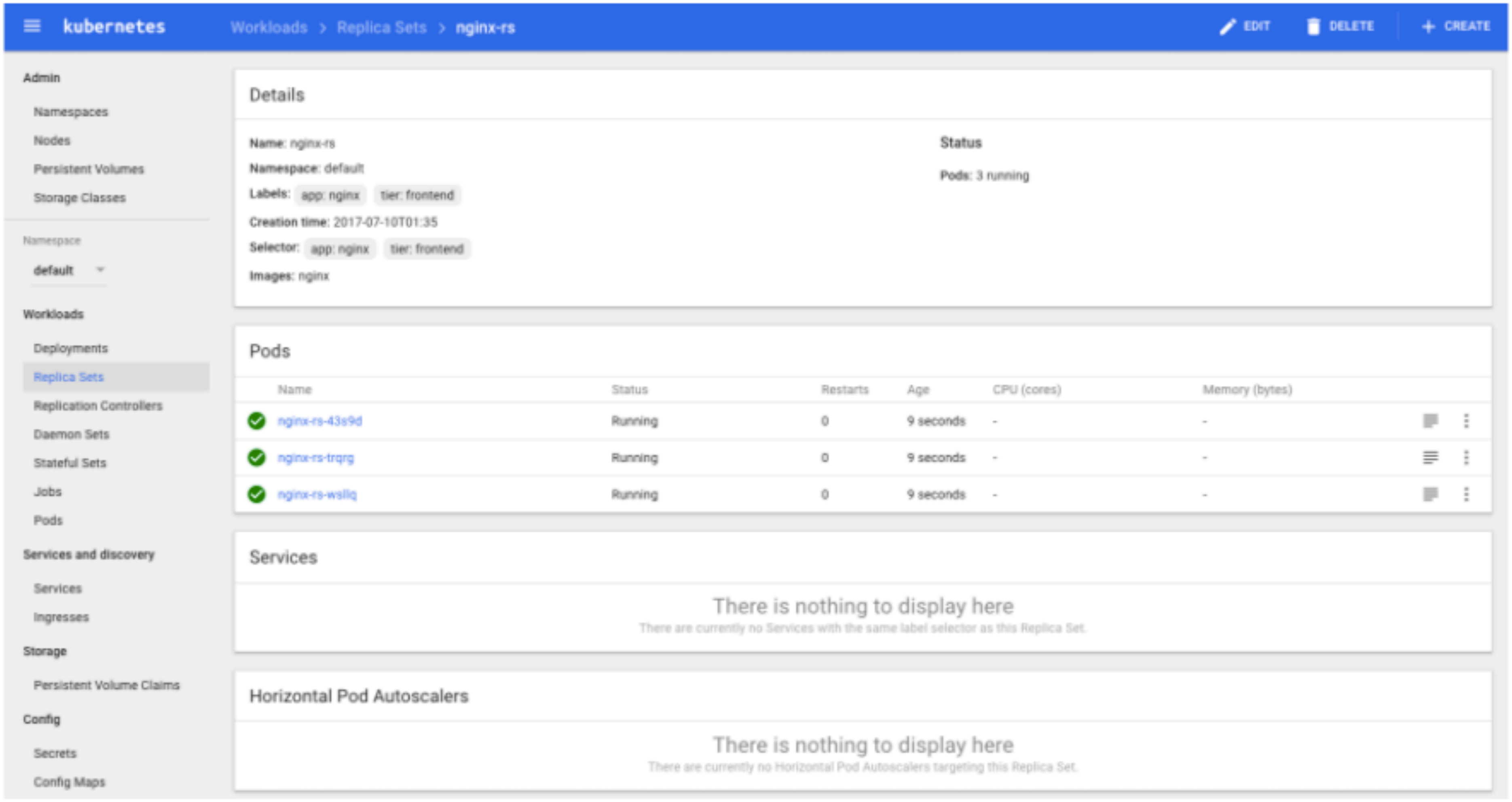
And now, query how many pods are running:



```
NAME             READY     STATUS    RESTARTS   AGE
nginx            1/1       Running   0          36m
nginx-rs-j829q   1/1       Running   0          15m
nginx-rs-rq6x9   1/1       Running   0          15m
nginx-rs-s93s4   1/1       Running   0          35s
```

Bingo! Our `replicaset` has created a new pod (you can see which one in the AGE column). This is immensely powerful. We have gone from a world where a pod (an application) being killed wakes you up at 4 a.m. in the morning to take action to a world where when one of our application dies, Kubernetes revives it for us.

Let's take a look at what happened in the dashboard:



Details

Name: nginx-rs
Namespace: default
Labels: app: nginx tier: frontend
Creation time: 2017-07-10T01:35
Selector: app: nginx tier: frontend
Images: nginx

Status
Pods: 3 running

Pods

	Name	Status	Restarts	Age	CPU (cores)	Memory (bytes)
✓	nginx-rs-43s9d	Running	0	9 seconds	-	-
✓	nginx-rs-trqrg	Running	0	9 seconds	-	-
✓	nginx-rs-wsllq	Running	0	9 seconds	-	-

Services

There is nothing to display here
There are currently no Services with the same label selector as this Replica Set.

Horizontal Pod Autoscalers

There is nothing to display here
There are currently no Horizontal Pod Autoscalers targeting this Replica Set.

As you can expect, the Replica Set has created the pods for you. You can try to kill them from the interface as well (the period icon to the very right of every pod will allow you to do that), but the Replica Set will re-spawn them for you.

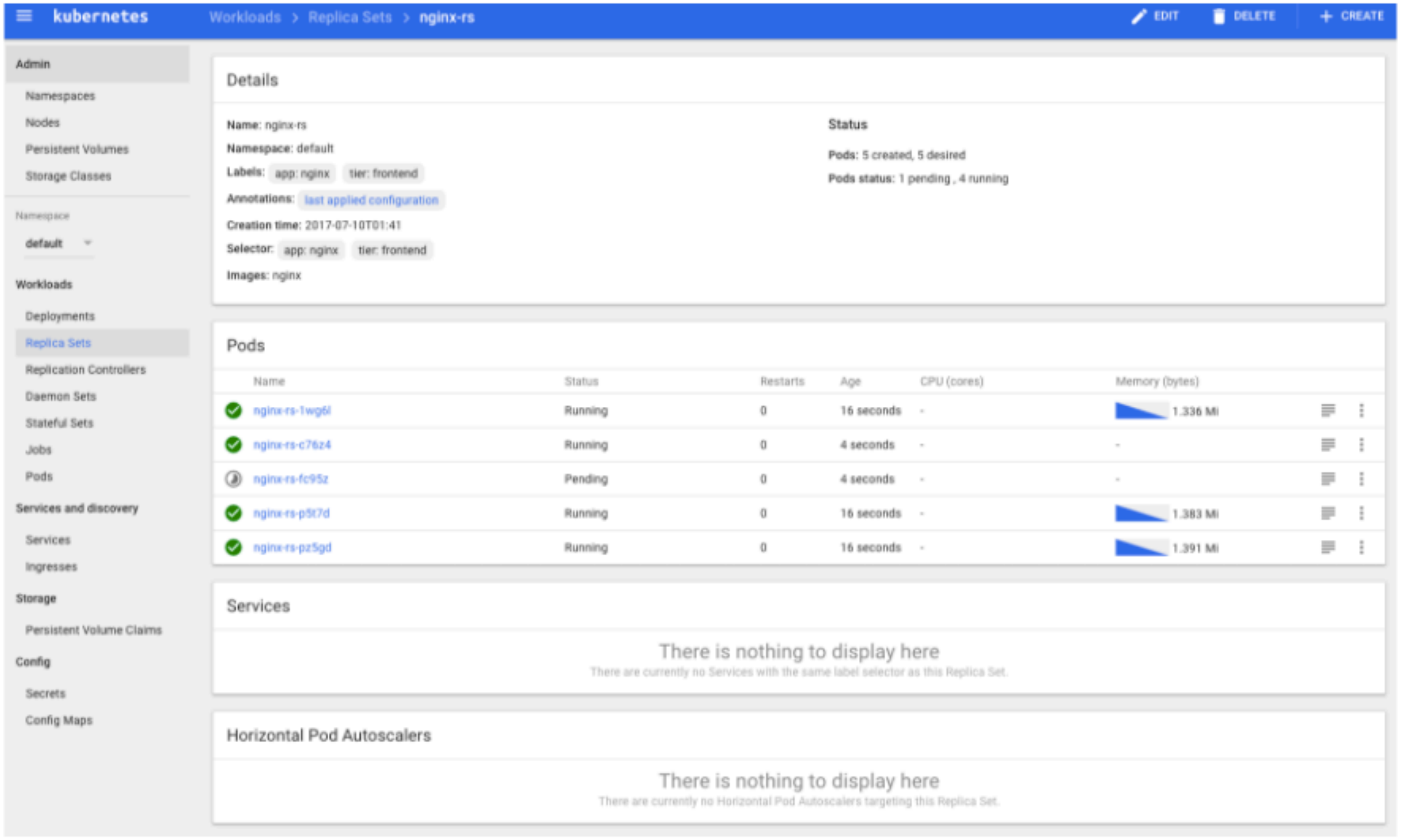
Now we are going to do something that might look like it's from out of this world: we are going to scale our application with a single command, but first, edit `replicaset.yml` and change the `replicas` field from three to five.

Save the file and execute this:

Copy

```
kubectl apply -f replicaset.yml
```

Now take a look at the dashboard again:



Details

Name: nginx-rs
Namespace: default
Labels: app: nginx tier: frontend
Annotations: last applied configuration
Creation time: 2017-07-10T01:41
Selector: app: nginx tier: frontend
Images: nginx

Status
Pods: 5 created, 5 desired
Pods status: 1 pending , 4 running

Pods

Name	Status	Restarts	Age	CPU (cores)	Memory (bytes)
nginx-rs-1wg6l	Running	0	16 seconds	-	1.336 Mi
nginx-rs-c76z4	Running	0	4 seconds	-	-
nginx-rs-fc95z	Pending	0	4 seconds	-	-
nginx-rs-p5t7d	Running	0	16 seconds	-	1.383 Mi
nginx-rs-pz5gd	Running	0	16 seconds	-	1.391 Mi

Services

There is nothing to display here
There are currently no Services with the same label selector as this Replica Set.

Horizontal Pod Autoscalers

There is nothing to display here
There are currently no Horizontal Pod Autoscalers targeting this Replica Set.

As you can see, Kubernetes is creating pods for us following the instructions of the Replica Set, `nginx-rs` . In the preceding sreenshot, we can see one pod whose icon is not green, and that is because its status is **`Pending`**, but after a few seconds, the status becomes **`Ready`**, just like any other pod.

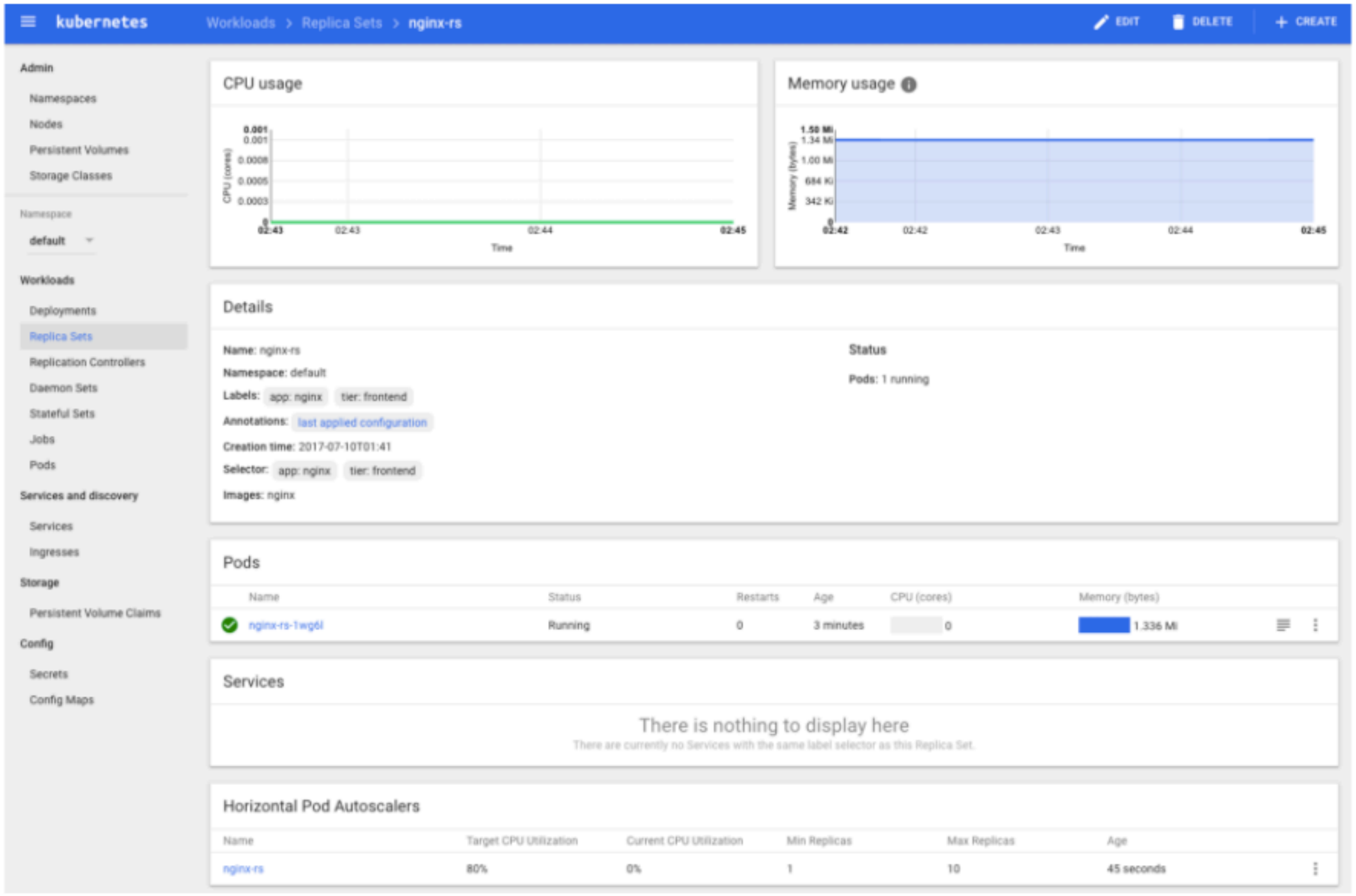
This is also very powerful, but there is a catch: who scales the application if the load spike happens at 4 a.m. in the morning? Well, Kubernetes provides a solution for this: **`Horizontal Pod Autoscalers`**.

Let's execute the following command:

Copy

```
kubectl autoscale replicaset nginx-rs --max=10
```

With the preceding command, we have specified that Kubernetes should attach a `Horizontal Pod Autoscalers` to our Replica Set. If you browse the Replica Set in the dashboard again, the situation has changed dramatically:



Details

Name: nginx-rs
Namespace: default
Labels: app: nginx tier: frontend
Annotations: last applied configuration
Creation time: 2017-07-10T01:41
Selector: app: nginx tier: frontend
Images: nginx

Status

Pods: 1 running

Pods

Name	Status	Restarts	Age	CPU (cores)	Memory (bytes)
nginx-rs-1wg6l	Running	0	3 minutes	0	1.336 Mi

Services

There is nothing to display here
There are currently no Services with the same label selector as this Replica Set.

Horizontal Pod Autoscalers

Name	Target CPU Utilization	Current CPU Utilization	Min Replicas	Max Replicas	Age
nginx-rs	80%	0%	1	10	45 seconds

Let's explain what happened here:

- We have attached an **`Horizontal Pod Autoscalers`** to our Replica Set: minimum `1` pod, maximum `10` , and the trigger for creating or destroying pods is the CPU utilization going over `80%` on a given pod.
- The Replica Set has scaled down to one pod because there is no load on the system, but it will scale back to up to 10 nodes if required and stay there for as long as the burst of requests is going on and scale back to the minimum required resources.

Now this is actually the dream of any sysadmin: no-hassle autoscaling and self-healing infrastructure. As you can see, Kubernetes starts making sense altogether, but there is one thing disturbing in the autoscaler part. It was a command that we ran in the terminal, but it is captured nowhere. So how can we keep track of our infrastructure (yes, an Horizontal Pod Autoscaler is part of the infrastructure)?

Well, there is an alternative; we can create a YAML file that describes our Horizontal Pod Autoscaler:

Copy

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
   name: nginx-hpa
spec:
   maxReplicas: 10
   minReplicas: 1
   scaleTargetRef:
      kind: ReplicaSet
      name: nginx-rs
   targetCPUUtilizationPercentage: 80
```

First, from the dashboard, remove `HorizontalPodAutoscaler` created from the previous example. Then, write the preceding content into a file called `horizontalpodautoscaler.yml` and run the following command:

```
kubectl apply -f horizontalpodautoscaler.yml
```

This should have the same effect as the `autoscale` command but with two obvious benefits:

- We can control more parameters, such as the name of the HPA, or add metadata to it, such as labels
- We keep our infrastructure as code within reach so we know what is going on

The second point is extremely important: we are in the age of the infrastructure as code and Kubernetes leverages this powerful concept in order to provide traceability and readability. Later on, in Chapter 8, **Release Management – Continuous Delivery**, you will learn how to create a continuous delivery pipeline with Kubernetes in a very easy way that works on 90% of the software projects.

Once the preceding command returns, we can check on the dashboard and see that effectively, our Replica Set has attached an Horizontal Pod Autoscaler as per our configuration.