

Kubernetes Advanced

Raúl Estrada

Octubre 20200

Deployments

Even though the Replica Set is a very powerful concept, there is one part of it that we have not talked about: what happens when we apply a new configuration to a Replica Set in order to upgrade our applications? How does it handle the fact that we want to keep our application alive 100% of the time without service interruption?

Well, the answer is simple: it doesn't. If you apply a new configuration to a Replica Set with a new version of the image, the Replica Set will destroy all the Pods and create newer ones without any guaranteed order or control. In order to ensure that our application is always up with a guaranteed minimum amount of resources (Pods), we need to use Deployments.

First, take a look at what a deployment looks like:

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 0
      maxSurge: 1
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          resources:
            requests:
              cpu: 256m
              memory: 100Mi
          ports:
            - containerPort: 80
```

As you can see, it is very similar to a Replica Set, but there is a new section: strategy. In strategy, we are defining how our `rollout` is going to work, and we have two options:

- `RollingUpdate`
- `Recreate`

RollingUpdate is the default option as it seems the most versatile in modern 24/7 applications: It coordinates two replica sets and starts shutting down pods from the old replica set at the same time that it is creating them in the new Replica Set. This is very powerful because it ensures that our application always stays up. Kubernetes decides what is best to coordinate the pods' rescheduling, but you can influence this decision with two parameters:



`maxUnavailable`



`maxSurge`

The first one defines how many pods we can loose from our Replica Set in order to perform a `rollout` . As an example, if our Replica Set has three replicas, a `rollout` with the `maxUnavailable` value of `1` will allow Kubernetes to transition to the new Replica Set with only two pods in the status `Ready` at some point. In this example, `maxUnavailable` is `0` ; therefore, Kubernetes will always keep three pods alive.

`MaxSurge` is similar to `maxUnavailable`, but it goes the other way around: it defines how many pods above the replicas can be scheduled by Kubernetes. In the preceding example, with three replicas with `maxSurge` set on `1` , the maximum amount of pods at a given time in our `rollout` will be `4` .

Playing with these two parameters as well as the replicas' number, we can achieve quite interesting effects. For example, by specifying three replicas with `maxSurge 1` and `maxUnavailable 1`, we are forcing Kubernetes to move the pods one by one in a very conservative way: we might have four pods during the `rollout`, but we will never go below three available pods.

Coming back to the strategies, Recreate basically destroys all the pods and creates them again with the new configuration without taking uptime into account. This might be indicated in some scenarios, but I would strongly suggest that you use `RollingUpdate` when possible (pretty much always) as it leads to smoother deployments.

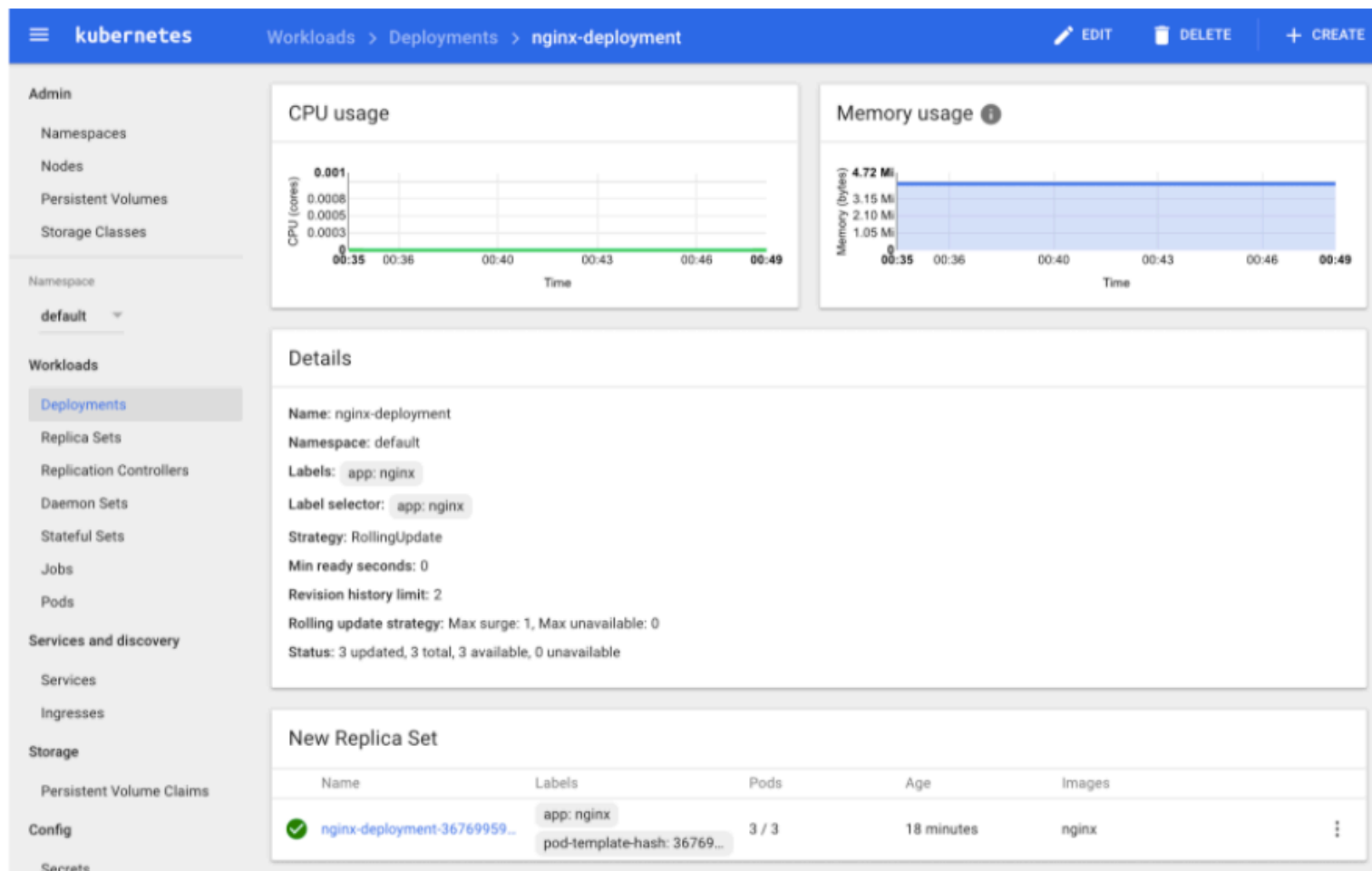
It is also possible to attach a Horizontal Pod Autoscaler to a Deployment in the same way that we would do with a Replica Set.

Let's test our deployment. Create a file called `deployment.yml` and apply it to our cluster:

```
kubectl apply -f deployment.yml --record
```

Copy

Once the command returns, we can go to the Kubernetes dashboard (`localhost:8001/ui` with the proxy active) and check what happened in the **Deployments** section in the menu on the left-hand side:



We have a new **Deployment** called **nginx-deployment**, which has created a Replica Set that also contains the specified pods. In the preceding command, we have passed a new parameter: **--record**. This saves the command in the **rollout** history of our deployment so that we can query the **rollout** history of a given deployment to see the changes applied to it. In this case, just execute the following:

```
kubectl rollout history deployment/nginx-deployment
```

Copy

This will show you all the actions that altered the status of a deployment called `nginx-deployment` . Now, let's execute some change:

```
kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
```

Copy

We have used `kubectl` to change the version of the `nginx` container back to version 1.9.1 (`kubectl` is very versatile; the official documentation offers shortcuts for pretty much everything), and a few things happened. The first one is that a new Replica Set has been created and the pods have been moved over to it from the old replica set. We can verify this in the **Replica Sets** section of the menu on the left-hand side of the dashboard:

The screenshot shows the Kubernetes dashboard interface. The top navigation bar is blue with the 'kubernetes' logo and a '+ CREATE' button. Below the bar, a breadcrumb trail reads 'Workloads > Replica Sets'. The left sidebar contains a menu with sections: 'Admin' (Namespaces, Nodes, Persistent Volumes, Storage Classes), 'Namespace' (default), and 'Workloads' (Deployments, **Replica Sets**, Replication Controllers, Daemon Sets, Stateful Sets, Jobs, Pods). The 'Replica Sets' section is active, displaying a table of Replica Sets.

Name	Labels	Pods	Age	Images	
nginx-deployment-321290875	app: nginx pod-template-hash: 32129...	3 / 3	2 minutes	nginx:1.9.1	⋮
nginx-deployment-36769959...	app: nginx pod-template-hash: 36769...	0 / 0	32 minutes	nginx	⋮

As you can see, the old replica set has 0 pods, whereas the new one that took over has three pods. This all happened without you noticing it, but it is a very clever workflow with a lot of work from the Kubernetes community and the companies behind it.

The second thing that happened was that we have a new entry in our rollout history. Let's check it out:

```
kubectl rollout history deployment/nginx-deployment
```

Copy

Which one should produce an output similar to the following one:

```
deployments "nginx-deployment"
REVISION    CHANGE-CAUSE
1           kubectl apply --filename=deployment.yml --record=true
2           kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
```

Now we have two entries that describe the changes applied to our deployment.

If you have been into IT for few years, by now, you have reached the conclusion that a rollback strategy is always necessary because bugs flowing into production are the reality no matter how good our QA is. I am a big fan of building the systems in a way that deployments are unimportant events (from a technical point of view), as shown with Kuberentes, and the engineers always have an easy way out if things start to fail in production. Deployments offer an easy rollback if something goes wrong:

```
kubectl rollout undo deployment/nginx-deployment
```

Copy

Execute the preceding and browse back to the dashboard on the **Replica Sets** section again:

kubernetes

Workloads > Replica Sets

+ CREATE

Admin

Namespaces

Nodes

Persistent Volumes

Storage Classes

Namespace

default

Workloads

Deployments

Replica Sets

Replication Controllers

Daemon Sets

Stateful Sets

Jobs

Pods

Replica Sets

Name	Labels	Pods	Age	Images	
<div>✓</div> nginx-deployment-321290875	<div>app: nginx</div> <div>pod-template-hash: 32129...</div>	0 / 0	10 minutes	nginx:1.9.1	⋮
<div>✓</div> nginx-deployment-36769959...	<div>app: nginx</div> <div>pod-template-hash: 36769...</div>	3 / 3	40 minutes	nginx	⋮

That's right. In a matter of seconds, we have gone from instability (a broken build) to the safety of the old known version without interrupting the service and without involving half of the IT department: a simple command brings back the stability to the system. The rollback command has a few configurations, and we can even select the revision where we want to jump to.

This is how powerful Kubernetes is and this is how simple our life becomes by using Kubernetes as the middleware of our enterprise: a modern CD pipeline assembled in a few lines of configuration that works in the same way in all the companies in the world by facilitating command `rollouts` and rollbacks. That's it...simple and efficient.

Right now, it feels like we know enough to move our applications to Kubernetes, but there is one thing missing. So far, up until now, we have just run predefined containers that are not exposed to the outer world. In short, there is no way to reach our application from outside the cluster. You are going to learn how to do that in the next section.

Services

Up until now, we were able to deploy containers into Kubernetes and keep them alive by making use of pods, Replica Sets, and Horizontal Pods Autoscalers as well as Deployments, but so far, you have not learned how to expose applications to the outer world or make use of service discovery and balancing within Kubernetes.

Services are responsible for all of the above. A Service in Kubernetes is not an element as we are used to it. A Service is an abstract concept used to give entity to a group of pods through pattern matching and expose them to different channels via the same interface: a set of labels attached to a Pod that get matched against a selector (another set of labels and rules) in order to group them.

First, let's create a service on top of the deployment created in the previous section:

Copy

```
kind: Service
apiVersion: v1
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

Easy and straightforward, but there's one detail: the selector section has a hidden message for us. The selectors are the mechanisms that Kubernetes uses to connect components via pattern matching algorithms. Let's explain what pattern matching is. In the preceding Service, we are specifying that we want to select all the Pods that have a label with the `app` key and the `nginx` value. If you go back to the previous section, you'll understand our deployment has these labels in the pod specification. This is a match; therefore, our service will select these pods. We can check this by browsing in the dashboard in the **Services** section and clicking on `nginx-service`, but first, you need to create the `service`:

```
kubectl apply -f service.yml
```

[Copy](#)

Then, check out the dashboard:

Details

Name: nginx-service

Namespace: default

Creation time: 2017-07-12T21:56

Label selector: app: nginx

Type: ClusterIP

Connection

Cluster IP: 10.47.245.73

Internal endpoints: nginx-service:80 TCP
nginx-service:0 TCP

Pods

Name	Status	Restarts	Age	CPU (cores)	Memory (bytes)		
✓ nginx-deployment-36769959...	Running	0	a day	<div></div> 0	<div></div> 1.355 Mi	☰	⋮
✓ nginx-deployment-36769959...	Running	0	a day	<div></div> 0	<div></div> 1.395 Mi	☰	⋮
✓ nginx-deployment-36769959...	Running	0	a day	<div></div> 0	<div></div> 1.391 Mi	☰	⋮

This screen has a lot of interesting information. The first piece of information is that the service has an IP: this IP is denominated as `clusterIP`. Basically, it is an IP within the cluster that can be reached by our pods and other elements in Kubernetes. There is also a field called `Type`, which allows us to choose the service type. There are three types:

- `ClusterIP`
- `NodePort`
- `LoadBalancer`

`ClusterIP` is what we just created and explained.

NodePort is another type of service that is rarely used in Cloud but is very common on premises. It allocates a port on all the nodes to expose our application. This allows Kubernetes to define the ingress of the traffic into our pods. This is challenging for two reasons:

- It generates extra traffic in our internal network as the nodes need to forward the traffic across to reach the pods (imagine a cluster of 100 nodes that has an app with only three pods, it is very unlikely to hit the node that is running one of them).
- The ports are allocated randomly so you need to query the Kubernetes API to know the allocated port.

`LoadBalancer` is the jewel in the crown here. When you create a service of type `LoadBalancer`, a cloud load balancer is provisioned so that the client applications hit the load balancer that redirects the traffic into the correct nodes. As you can imagine, for a cloud environment where infrastructure is created and destroyed in matter of seconds, this is the ideal situation.

Coming back to the previous screenshot, we can see another piece of interesting information: the internal endpoints. This is the service discovery mechanism that Kubernetes is using to locate our applications. What we have done here is connect the pods of our application to a name: `nginx-service`. From now on, no matter what happens, the only thing that our apps need to know in order to reach our `nginx` pods is that there is a service called `nginx` that knows how to locate them.

In order to test this, we are going to run an instance of a container called `busybox`, which is basically the Swiss army knife of command-line tools. Run the following command:

```
kubectl run -i --tty busybox --image=busybox --restart=Never -- sh
```

Copy

The preceding command will present us with a shell inside the container called `busybox` running in a pod so we are inside the Kubernetes cluster and, more importantly, inside the network so that we can see what is going on. Be aware that the preceding command runs just a pod: no deployment or replica set is created, so once you exit the shell, the pod is finalized and resources are destroyed.

Once we get the prompt inside `busybox` , run the following command:

Copy

```
nslookup nginx-service
```

This should return something similar to the following:

Copy

```
Server: 10.47.240.10
Address 1: 10.47.240.10 kube-dns.kube-system.svc.cluster.local

Name: nginx-service
Address 1: 10.47.245.73 nginx-service.default.svc.cluster.local
```

Okay, what happened here? When we created a service, we assigned a name to it: `nginx-service`. This name has been used to register it in an internal DNS for service discovery. As mentioned earlier, the DNS service is running on Kubernetes and is reachable from all the Pods so that it is a centralised repository of common knowledge. There is another way that the Kubernetes engineers have created in order to carry on with the service discovery: the environment variables. In the same prompt, run the following command:

```
env
```

Copy

This command outputs all the environment variables, but there are few that are relevant to our recently defined service:

Copy

```
NGINX_SERVICE_PORT_80_TCP_ADDR=10.47.245.73
NGINX_SERVICE_PORT_80_TCP_PORT=80
NGINX_SERVICE_PORT_80_TCP_PROTO=tcp
NGINX_SERVICE_SERVICE_PORT=80
NGINX_SERVICE_PORT=tcp://10.47.245.73:80
NGINX_SERVICE_PORT_80_TCP=tcp://10.47.245.73:80
NGINX_SERVICE_SERVICE_HOST=10.47.245.73
```

These variables, injected by Kubernetes at creation time, define where the applications can find our service. There is one problem with this approach: the environment variables are injected at creation time, so if our service changes during the life cycle of our pods, these variables become obsolete and the pod has to be restarted in order to inject the new values.

All this magic happens through the selector mechanism on Kubernetes. In this case, we have used the equal selector: a label must match in order for a pod (or an object in general) to be selected. There are quite a few options, and at the time of writing this, this is still evolving. If you want to learn more about selectors, here is the official documentation: <https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/>.

As you can see, services are used in Kubernetes to glue our applications together. Connecting applications with services allows us to build systems based on microservices by coupling REST endpoints in the API with the name of the service that we want to reach on the DNS.

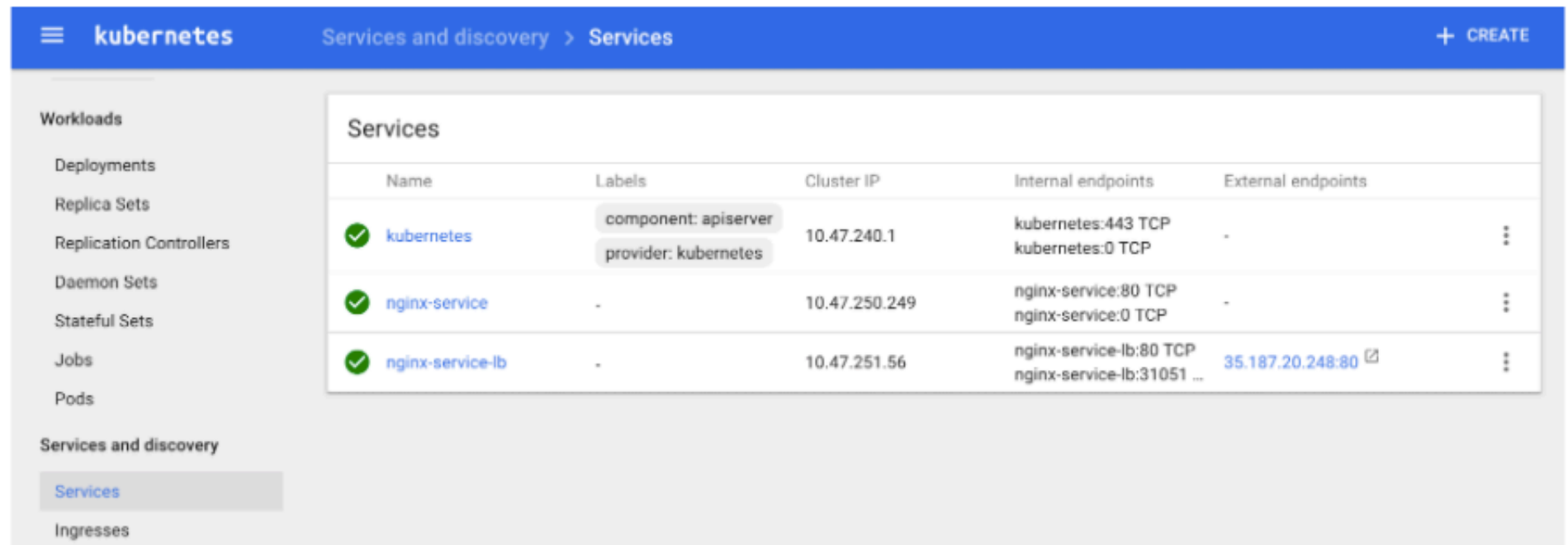
Up until now, you have learned how to expose our applications to the rest of our cluster, but how do we expose our applications to the outer world? You have also learned that there is a type of service that can be used for this:








LoadBalancer . Let's take a look at the following definition:

Copy

```
kind: Service
apiVersion: v1
metadata:
  name: nginx-service
spec:
  type: LoadBalancer
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

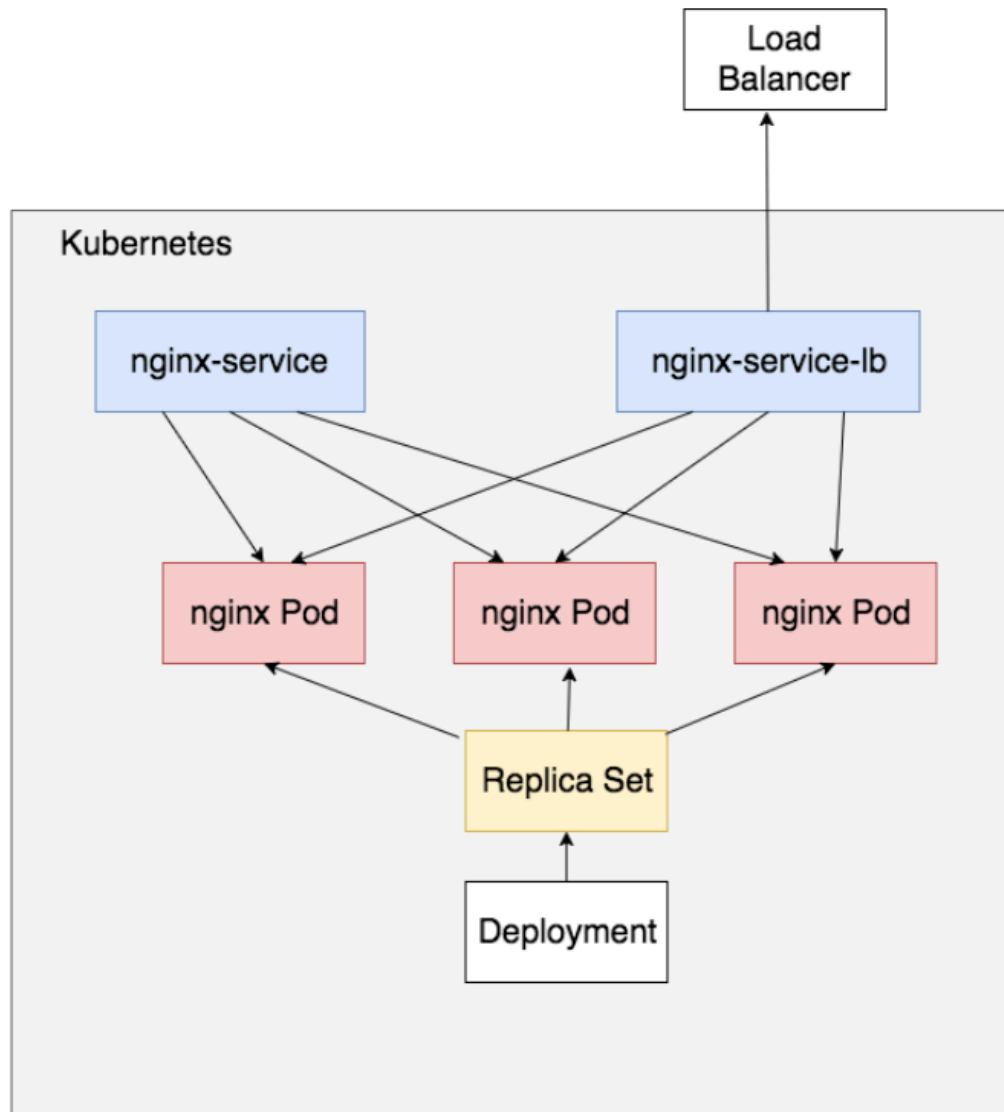
There is one change in the preceding definition: the service type is now `LoadBalancer`. The best way to explain what this causes is by going to the `Services` section of the dashboard:



Services					
Name	Labels	Cluster IP	Internal endpoints	External endpoints	
 kubernetes	<code>component: apiserver</code> <code>provider: kubernetes</code>	10.47.240.1	kubernetes:443 TCP kubernetes:0 TCP	-	
 nginx-service	-	10.47.250.249	nginx-service:80 TCP nginx-service:0 TCP	-	
 nginx-service-lb	-	10.47.251.56	nginx-service-lb:80 TCP nginx-service-lb:31051 ...	35.187.20.248:80 	

As you can see, our newly created service got assigned an external endpoint. If you browse it, bingo! The `nginx` default page is rendered.

We have created two services, `nginx-service` and `nginx-service-lb`, of the type `ClusterIP` and `LoadBalancer`, respectively, which both point to the same pods that belong to a deployment and are managed through a replica set. This can be a bit confusing, but the following diagram will explain it better:



Other Building Blocks

In the previous sections, you learned the basics needed to deploy applications into Kubernetes successfully. The API objects that we visited are as follows:

- Pod
- ReplicaSet
- Deployment
- Service

In Kubernetes, there are many other building blocks that can be used to build more advanced applications; every few months, the Kubernetes engineers add new elements to improve or add functionality.

One example of these additions is the ReplicaSet that was designed to replace another item called ReplicationController. The main difference between the ReplicationController and the ReplicaSet is that the latter one has a more advance semantics label selection for the Pods that were recently re-engineered in Kubernetes.

As a new product, Kuberentes is constantly changing (in fact, it is possible that by the time that you read this book, the core elements might have changed), so the engineers try to keep the compatibility across different versions so that people are not urged to upgrade in a short period of time.

Other examples of more advanced building blocks are the following:

- DaemonSet
- PetSets
- Jobs and CronJobs
- CronJobs

In order to go in deep to the full stack in Kubernetes, we would need a full book (or more!). Let's visit some of them.

Daemon Sets

Daemon Sets are an API element used to **ensure that a Pod is running in all (or some) nodes**. One of the assumptions in Kubernetes is that the pod should not worry about which node is being run, but that said, there might be a situation where we want to ensure that we run at least one pod on each node for a number of reasons:

- Collect logs
- Check the hardware
- Monitoring

In order to do that, Kubernetes provides an API element called Daemon Set. Through a combination of labels and selectors, we can define something called **affinity**, which can be used to run our pods on certain nodes (we might have specific hardware requirements that only a few nodes are able to provide so that we can use tags and selectors to provide a hint to the pods to relocate to certain nodes).

Daemon Sets have several ways to be contacted, from the DNS through a headless service (a service that works as a load balancer instead of having a cluster IP assigned) to the node IP, but Daemon Sets work best when they are the initiators of the communication: something happens (an event) and a Daemon Set sends an event with information about that event (for example, a node is running low on space).

PetSets

PetSets are an interesting concept within Kubernetes: they are strong named resources whose naming is supposed to stay the same for a long term. As of now, a pod does not have a strong entity within a Kubernetes cluster: you need to create a service in order to locate a pod as they are ephemeral. Kubernetes can reschedule them at any time without prior notice for changing their name, as we have seen before. If you have a deployment running in Kubernetes and kill one of the pods, its name changes from (for example) **pod-xyz** to **pod-abc** in an unpredictable way. so we cannot know which names to use in our application to connect to them beforehand.

When working with a Pet Set, this changes completely. A pet set has an ordinal order, so it is easy to guess the name of the pod. Let's say that we have deployed a Pet Set called `mysql`, which defines pods running a MySQL server. If we have three replicas, the naming will be as follows:

- `mysql-0`
- `mysql-1`
- `mysql-2`

So, we can bake this knowledge in our application to reach them. This is suboptimal but good enough: we are still coupling services by name (DNS service discovery has this limitation), but it works in all cases and is a sacrifice that is worth paying for because in return, we get a lot of flexibility. The ideal situation in service discovery is where our system does not need to know even the name of the application carrying the work: just throw the message into the ether (the network) and the appropriated server will pick it up and respond accordingly.

Pet Sets have been replaced in later versions of Kubernetes with another item called **Stateful Set**. The Stateful Set is an improvement over the Pet Set mainly in how Kubernetes manages the **master knowledge to avoid a split brain situation**: where two different elements think that they are in control.

Jobs

A **Job** in Kubernetes is basically an element that spawns the defined number of pods and waits for them to finish before completing its life cycle. It is very useful when there is a need to run a one-off task, such as rotating logs or migrating data across databases.

Cron jobs have the same concept as Jobs, but they get triggered by time instead of a one-off process.

Both in combination are very powerful tools to keep any system running. If you think about how we rotate logs without Kubernetes via ssh, it is quite risky: there is no control (by default) over who is doing what, and usually, there is no review process in the ssh operations carried by an individual.

With this approach, it is possible to create a Job and get other engineers to review it before running it for extra safety.

Secrets and configuration management

On Docker in general, as of today, secrets are being passed into containers via environment variables. This is very insecure: first, there is no control over who can access what, and second, environment variables are not designed to act as secrets and a good amount of commercial software (and open source) outputs them into the standard output as part of bootstrapping. Needless to say, that's rather inconvenient.

Kubernetes has solved this problem quite gracefully: instead of passing an environment variable to our container, a volume is mounted with the secret on a file (or several) ready to be consumed.

By default, Kubernetes injects a few secrets related to the cluster into our containers so that they can interact with the API and so on, but it is also possible to create your own secrets.

There are two ways to create secrets:

- Using `kubectl`
- Defining an API element of type secret and using `kubectl` to deploy it

The first way is fairly straightforward. Create a folder called **secrets** in your current work folder and execute the following commands inside it:

```
echo -n "This is a secret" > ./secret1.txt  
echo -n "This is another secret" > ./secret2.txt
```

Copy

This creates two files with two strings (simple strings as of now). Now it is time to create the secret in Kubernetes using **kubectl** :

```
kubectl create secret generic my-secrets --from-file=./secret1.txt --from-file=./secret2.txt
```

Copy

And that's it. Once we are done, we can query the secrets using `kubectl` :

```
kubectl get secrets
```

Copy

This, in my case, returns two secrets:

- A service account token injected by the cluster
- My newly created secret (`my-secrets`)

The second way of creating a secret is by defining it in a `yaml` file and deploying it via `kubectl` . Take a look at the following definition:

Copy

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret-yaml
type: Opaque
data:
  secret1: VGhpcyBpcyBhIHNlY3JldA==
  secret2: VGhpcyBpcyBhbm90aGVyIHNlY3JldA==
```

First, the values for `secret1` and `secret2`, seem to be encrypted, but they are not; they are just encoded in `base64` :

Copy

```
echo -n "This is a secret" | base64  
echo -n "This is another secret" | base64
```

This will return the values that you can see here. The type of the secret is Opaque, which is the default type of secret, and the rest seems fairly straightforward. Now create the secret with `kubectl` (save the preceding content in a file called `secret.yml`):

Copy

```
kubectl create -f secret.yml
```

And that's it. If you query the secrets again, note that there should be a new one called `my-secret-yaml`. It is also possible to list and see the secrets in the dashboard on the **Secrets** link in the menu on left-hand side.

Now it is time to use them. In order to use the secret, two things need to be done:

- Claim the secret as a volume
- Mount the volume from the secret

Let's take a look at a **Pod** using a secret:

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "test-secrets",
    "namespace": "default"
  },
  "spec": {
    "containers": [{
      "name": "pod-with-secret",
      "image": "nginx",
      "volumeMounts": [{
        "name": "secrets",
        "mountPath": "/secrets",
        "readOnly": true
      }]
    }],
    "volumes": [{
      "name": "secrets",
      "secret": {
        "secretName": "my-secret"
      }
    }]
  }
}
```

So, you have learned a new thing here: `kubectl` also understands JSON. If you don't like YAML, it is possible to write your definitions in JSON without any side-effects.

Now, looking at the JSON file, we can see how first, the secret is declared as a volume and then how the secret is mounted in the path/secrets.

If you want to verify this, just run a command in your container to check it:

Copy

```
kubectl exec -it test-secrets ls /secrets
```

This should list the two files that we have created, `secret1.txt` and `secret2.txt`, containing the data that we have also specified.

Kubernetes- moving on

In this chapter, you learned enough to run simple applications in Kubernetes, but even though we cannot claim ourselves to be experts, we got the head start in becoming experts. Kubernetes is a project that evolves at the speed of light, and the best thing that you can do to keep yourself updated is follow the project on GitHub at <https://github.com/kubernetes>.

The Kubernetes community is very responsive with issues raised by the users and are also very keen on getting people to contribute to the source code and documentation.

If you keep working with Kubernetes, some help will be required. The official documentation is quite complete, and even though it feels like it needs a reshuffle sometimes, it is usually enough to keep you going.

The best way that I've found to learn Kubernetes is by experimenting in Minikube (or a test cluster) before jumping into a bigger commitment.