START KUBERNETES

The Beginner's Guide to Kubernetes

PETER JAUSOVEC

www.startkubernetes.com

Start Kubernetes

Peter Jausovec

Version 1.0.0

Table of Contents

What do I need to start with Kubernetes?

This book is a mix of theoretical explanations and practical examples. You'll get the most out of this book if you follow along with the practical examples. To do that, you will need the following tools:

- Docker Desktop
- Access to a Kubernetes cluster, see Which Kubernetes cluster should I use?
- Kubernetes CLI (`kubectl`)

For some sections, you might also need other tools that I mention in the specific chapters.

Which Kubernetes cluster should I use?

You have multiple choices. The most 'real-world' option would be to get a Kubernetes cluster from one cloud provider. However, for numerous reasons, that might not be an option for everyone.

The next best option is to run a Kubernetes cluster on your computer. Assuming you have some memory and CPU to spare, you can use one of the following tools to run a single-node Kubernetes cluster on your computer:

- Docker Desktop
- kind
- Minikube

You could go with any of the above options. Creating Kubernetes ReplicaSets, Deployments, and Pods works with any of them. You can also create Kubernetes Services. However, things get a bit complicated when you're trying to use a LoadBalancer service type.

With the cloud-managed cluster, creating a LoadBalancer service type creates an actual instance of the load balancer, and you would get an external/public IP address you can use to access your services.

The one solution from the above list closest to simulating the LoadBalancer service type is Docker Desktop. With Docker Desktop your service gets exposed on an external IP, `localhost`. You can access these services using both kind and Minikube as well; however, it requires you to run additional commands.

For that purpose, I'll be mostly using Docker Desktop. I'll specifically call out when I am using either Minikube or a cloud-managed cluster. You can follow the installation instructions for all options from their respective websites.

Kubernetes and contexts

After you've installed one of these tools, make sure you download the Kubernetes CLI. Kubernetes CLI is a single binary called `kubectl`, and it allows you to run commands against your cluster. To make sure everything is working correctly, you can run `kubectl get nodes` to list all nodes in the Kubernetes cluster. The output from the command should look like this:

```
$ kubectl get nodes
NAME             STATUS   ROLES    AGE   VERSION
docker-desktop   Ready    master   63d   v1.16.6-beta.0
```

You can also check that the context is set correctly to `docker-desktop`. Kubernetes uses a configuration file called `config` to find the information it needs to connect to the cluster. Kubernetes CLI reads this file from your home folder - for example `$HOME/.kube/config`. Context is an element inside that config file, and it contains a reference to the cluster, namespace, and the user. If you're accessing or running a single cluster, you will only have one context in the config file. However, you can have multiple contexts defined that point to different clusters.

Using the `kubectl config` command, you can view these contexts and switch between them. You can run the `current-context` command to view the current context:

```
$ kubectl config current-context
docker-desktop
```

There are other commands such as `use-context`, `set-context`, `view-contexts`, etc. I prefer to use a tool called `kubectx`. This tool allows you to switch between different Kubernetes contexts quickly. For example, if I have three clusters (contexts) set in the config file, running `kubectx` outputs this:

```
$ kubectx
docker-desktop
peterj-cluster
minikube
```

The tool highlights the currently selected context when you run the command. To switch to the `minikube` context, I can run: `kubectx minikube`.

The equivalent commands you can run with `kubectl` would be `kubectl config get-contexts` to view all contexts, and `kubectl config use-context minikube` to switch the context.

Before you continue, make sure your context is set to `docker-desktop` if you're using Docker for Mac/Windows or `minikube` if you're using Minikube.

Let's get started with your journey to Kubernetes and cloud-native world with the container orchestration!

What is container orchestration?

Containers are everywhere these days. People use tools such as Docker for packaging anything from applications to databases. With the growing popularity of microservice architecture and moving away from the monolithic applications, a monolith application is now a collection of multiple smaller services.

Managing a single application has its issues and challenges, let alone managing tens of smaller services that have to work together. You need a way to automate and manage your deployments, figure out how to scale individual services, use the network, connect them, and so on.

The container orchestration can help you do this. Container orchestration can help you manage the lifecycles of your containers. Using a container orchestration system allows you to do the following:

- Provision and deploy containers based on available resources
- Perform health monitoring on containers
- Load balancing and service discovery
- Allocate resources between different containers
- Scaling the containers up and down

A couple of examples of container orchestrators are Marathon, Docker Swarm and the one discussed in this course, Kubernetes.

Kubernetes is an open-source project and one of the popular choices for cluster management and scheduling container-centric workloads. You can use Kubernetes to run your containers, do zero-downtime deployments where you can update your application without impacting your users, and bunch of other cool stuff.

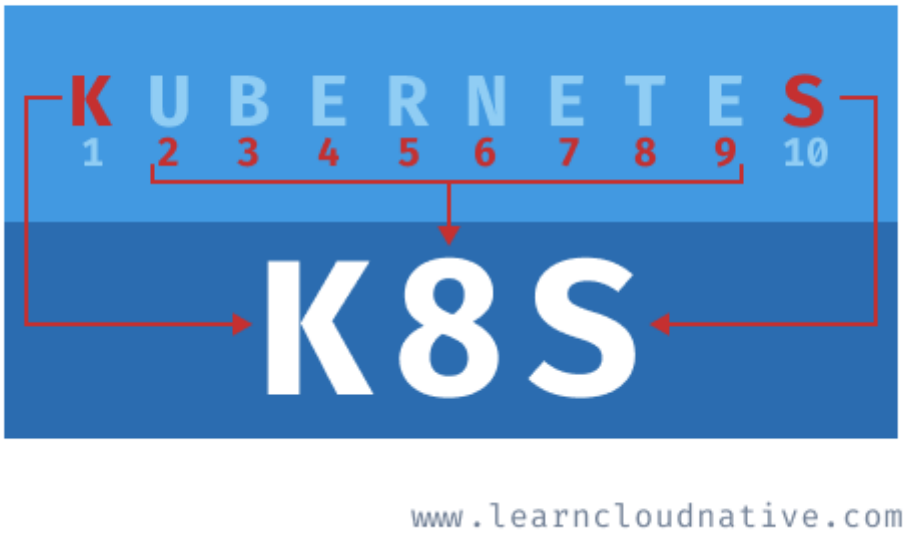


Figure 1. Kubernetes Numeronym

NOTE	Frequently, people refer to Kubernetes as "K8S". K8S is a **numeronym** for Kubernetes. The first (K) and the last letter (S) are the first, and the last letters in the word Kubernetes, and 8 is the number of characters between those two letters. Other popular numeronyms are "i18n" for internationalization or "a11y" for accessibility.

What is the difference Kubernetes and Docker?

Using Docker, you can package your application. This package is called an **image** or a **Docker image**. You can think of an image as a template. Using Docker, you can create **containers** from your images. For example, if your Docker image contains a Go binary or a Java application, then the container is a running instance of that application. If you want to learn more about Docker, check out the Beginners Guide to Docker.

Kubernetes, on the other hand, is a container orchestration tool that knows how to manage Docker (and other) containers. Kubernetes uses higher-level constructs such as Pods to wrap Docker (or other) containers and gives you the ability to manage them.

Kubernetes vs. Docker Swarm?

Docker Swarm is a container orchestration tool, just like Kubernetes is. You can use it to manage Docker containers. Using Swarm, you can connect multiple Docker hosts into a virtual host. You can then use Docker CLI to talk to multiple hosts at once and run Docker containers on it.

Kubernetes architecture

A Kubernetes cluster is a set of physical or virtual machines and other infrastructure resources needed to run your containerized applications. Each machine in a Kubernetes cluster is called a **node**. There are two types of node in each Kubernetes cluster:

- Master node(s): this node hosts the Kubernetes control plane and manages the cluster
- Worker node(s): runs your containerized applications

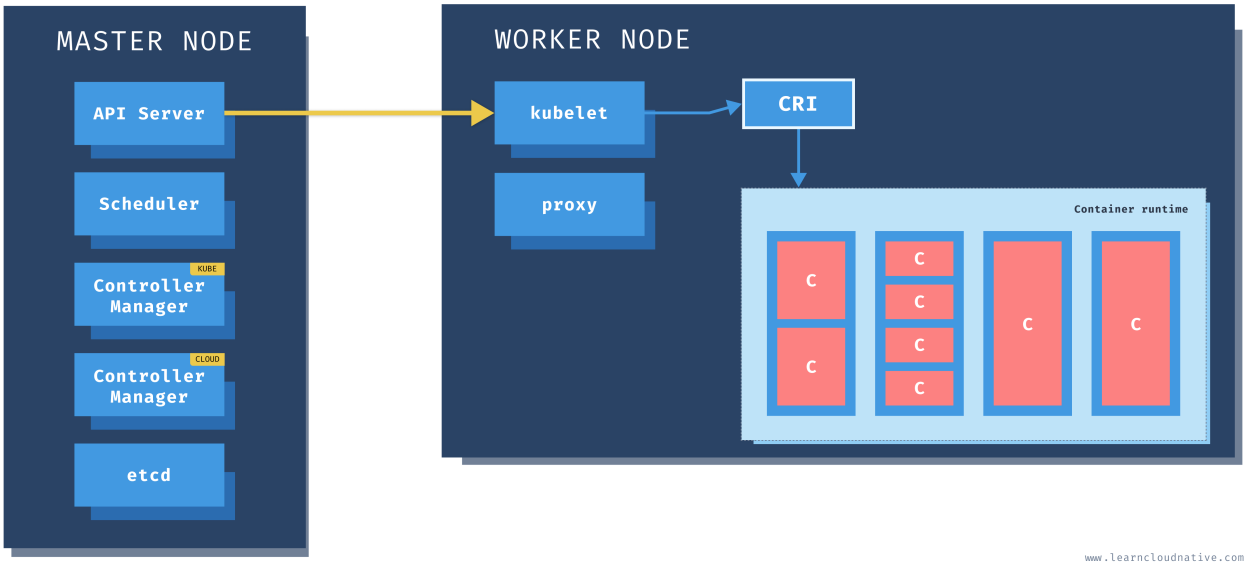


Figure 2. Kubernetes Architecture

Master nodes

One of the main components on the master node is called the **API server**. The API server is the endpoint that Kubernetes CLI (`kubectl`) talks to when you're creating Kubernetes resources or managing the cluster.

The **scheduler component** works with the API server to schedule the applications or workloads on to the worker nodes. It also knows about resources available on the nodes and the resources requested by the workloads. Using this information, it can decide on which worker nodes your workloads end up.

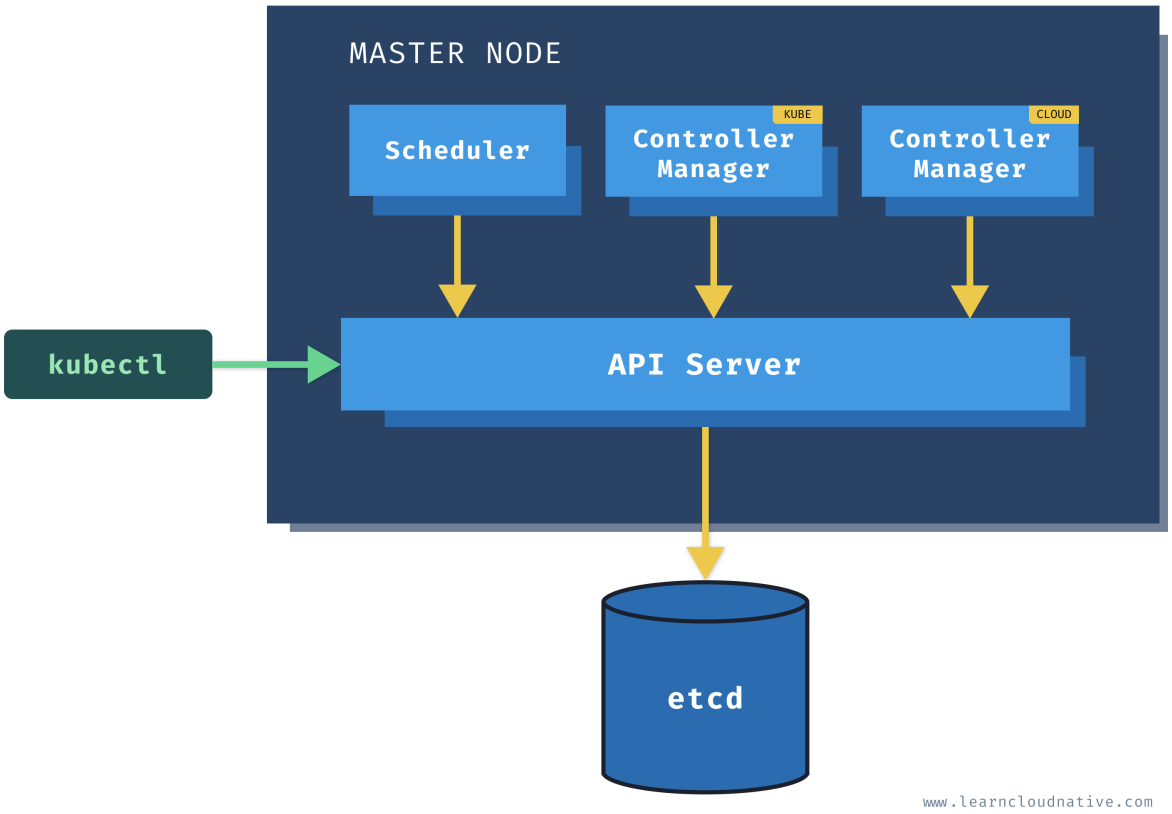


Figure 3. Kubernetes Master Node

There are two types of controller managers running on master nodes.

The **kube controller manager** runs multiple controller processes. These controllers watch the state of the cluster and try to reconcile the current state of the cluster (e.g., "5 running replicas of workload A") with the desired state (e.g. "I want ten running replicas of workload A"). The controllers include a node controller, replication controller, endpoints controller, service account and token controllers.

The **cloud controller manager** runs controllers specific to the cloud provider and can manage resources outside of your cluster. This controller only runs if your Kubernetes cluster is running in the cloud. If you're running Kubernetes cluster on your computer, this controller won't be running. The purpose of this controller is for the cluster to talk to the cloud providers to manage the nodes, load balancers, or routes.

Finally, etcd is a distributed key-value store. Kubernetes stores the state of the cluster and API in the etcd.

Worker nodes

Just like on the master node, worker nodes have different components running as well. The first one is **kubelet**. This service runs on each worker node, and its job is to manage the container. It makes sure containers are running and healthy, and it connects back to the control plane. Kubelet talks to the API server, and it is responsible for managing resources on the node it's running on.

When you add a new worker node to the cluster, the kubelet introduces itself to the API server and provides its resources (e.g., "I have X CPU and Y memory"). Then, it the API server if there are any containers to run. You can think of the kubelet as a worker node manager.

Kubelet uses the container runtime interface (CRI) to talk to the container runtime. The container runtime is responsible for working with the containers. In addition to Docker, Kubernetes also supports other container runtimes, such as containerd or cri-o.

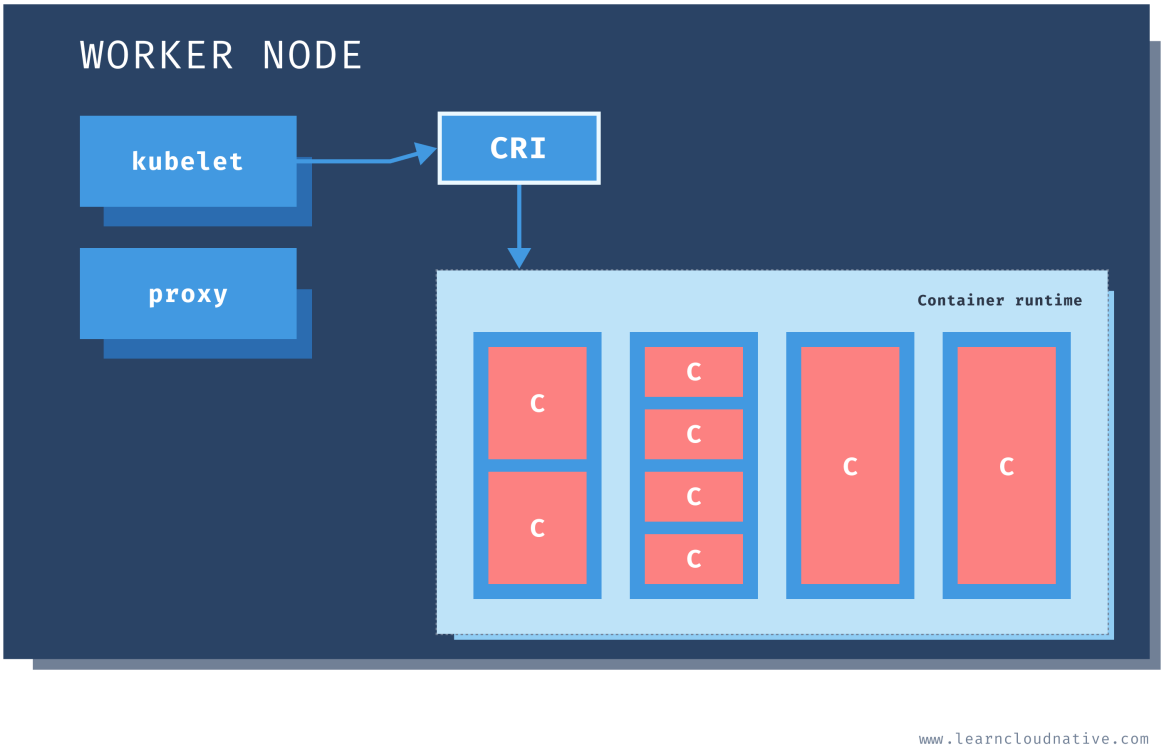


Figure 4. Kubernetes Worker Node

The containers are running inside pods, represented by the blue rectangles in the above figure (containers are the red rectangles inside each pod). A pod is the smallest deployable unit you can create, schedule, and manage on a Kubernetes cluster. A pod is a logical collection of containers that make up your application. The containers running inside the same pod also share the network and storage space.

Each worker node also has a proxy that acts as a network proxy and a load balancer for workloads

running on the worker nodes. Through these proxies, the external load balancer redirects client requests to containers running inside the pod.

Kubernetes Resources

The Kubernetes API defines many objects called resources, such as namespaces, pods, services, secrets, config maps, etc.

Of course, you can also define your custom resources using the custom resource definition or CRD.

After you've configured Kubernetes CLI and your cluster, you should try and run `kubectl api-resources` command. It will list all defined resources - there will be a lot of them.

Resources in Kubernetes can be defined using YAML. People commonly use YAML (YAML Ain't Markup Language) for configuration files. It is a superset of JSON format, which means you can also use JSON to describe Kubernetes resources.

Every Kubernetes resource has an `apiVersion` and `kind` fields to describe which version of the Kubernetes API you're using when creating the resource (for example, `apps/v1`) and what kind of a resource you are creating (for example, `Deployment`, `Pod`, `Service`, etc.).

The `metadata` includes the data that can help to identify the resource you are creating. Commonly found fields in the metadata section include the `name` (for example `mydeployment`) and the `namespace` where Kubernetes creats the resource.

Other fields you can have in the metadata section are `labels` and `annotations`. Kubernetes also adds a couple of fields automatically after creating the resource (such as `creationTimestamp`, for example).

Labels and selectors

You can use labels (key/value pairs) to label resources in Kubernetes. They are used to organize, query, and select objects and attach identifying metadata to them. You can specify labels at creation time or add, remove, or update them later after you've created the resource.

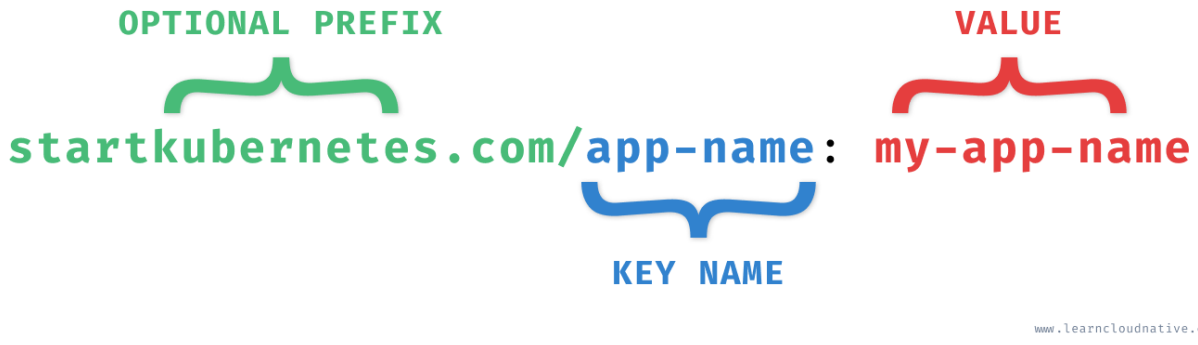


Figure 5. Kubernetes Labels

The labels have two parts: the key name and the value. The key name can have an optional prefix and the name, separated by a slash (`/`).

The `startkuberenetes.com` portion in the figure is the optional prefix, and the key name is `app-name`. The prefix, if specified, must be a series of DNS labels separated by dots (`.`). It can't be longer than 253 characters.

The key name portion is required and must be 63 characters or less. It has to start and end with an alphanumeric character. The key name is made of alphanumerical values and can include dashes (-), underscores (_), and dots (.).

Just like the key name, a valid label value must be 63 characters or less. It can be empty or being and end with an alphanumeric character and can include dashes (-), underscores (_), and dots (.).

Here's an example of a couple of valid labels on a Kubernetes resource:

```
metadata:
  labels:
    startkubernetes.com/app-name: my-application
    blog.startkubernetes.com/version: 1.0.0
    env: staging-us-west
    owner: ricky
    department: AB1
```

Selectors are used to query for a set of Kubernetes resources. For example, you could use a selector to identify all Kubernetes cluster objects with a label `env` set to `staging-us-west`. You could write that selector as `env = staging-us-west`. This selector is called an equality-based selector.

Selectors also support multiple requirements. For example, we could query for all resources with the `env` label set to `staging-us-west` and are not of version `1.0.0`. The requirements have to be separated by commas that act as a logical AND operator. We could write the above two requirements like this: `env = staging-us-west, blog.startkubernetes.com/version != 1.0.0`.

The second type of selector is called set-based selectors. These selectors allow filtering label keys based on a set of values. The following three operators are supported: `in`, `notin`, and `exists`. Here's an example:

```
env in (staging-us-west,staging-us-east)
owner notin (ricky, peter)
department
department!
```

The first example selects all objects with the `env` label set to either `staging-us-west` or `staging-us-east`. The second example uses the `notin` operator and selects all objects where the `owner` label values are not `ricky` or `peter`. The third example selects all objects with a `department` label set, regardless of it's value, and the last example selects all objects without the `department` label set.

Later on, we will see the practical use of labels and selectors when discussing how Kubernetes services know which pods to include in their load balancing pools.

Labeling resources is essential, so make sure you take your time to decide on the core set of labels you will use in your organization. Setting labels on all resources make it easier to do bulk operations against them later on.

Kubernetes provides a list of recommended labels that share a common prefix `app.kubernetes.io`:

Table 1. Recommended Labels

Name	Value	Description
`app.kubernetes.io/name`	my-app	Application name
`app.kubernetes.io/instance`	my-app-1122233	Identifying instance of the application
`app.kubernetes.io/version`	1.2.3	Application version
`app.kubernetes.io/component`	website	The name of the component
`app.kubernetes.io/part-of`	carts	The name of the higher-level application this component is part of
`app.kubernetes.io/managed-by`	helm	The tools used for managing the application

Additionally, you could create an maintain a list of your own labels:

- Project ID (`carts-project-123`)
- Owner (`Ricky`, `Peter`, or `team-a`, `team-b`, ...)
- Environment (`dev`, `test`, `staging`, `prod`})
- Release identifer (`release`-1.0.0 `)
- Tier (`backend`, `frontend`)

Annotations

Annotations are similar to labels as they also add metadata to Kubernetes objects. However, you don't use annotations for identifying and selecting objects. The annotations allow you to add non-identifying metadata to Kubernetes objects. Examples would be information needed for debugging, emails, or contact information of the on-call engineering team, port numbers or URL paths used by monitoring or logging systems, and any other information that might be used by the client tools or DevOps teams.

For example, annotations are used by ingress controllers to claim the Ingress resource (see Exposing multiple applications with Ingress).

Similar to labels, annotations are key/value pairs. The key name has two parts and the same rules apply as with the label key names.

However, the annotation values can be both structured or unstructured and can include characters that are not valid in the labels.

```
metadata:
  annotations:
    startkubernetes.com/debug-info: |
      { "debugTool": "sometoolname", "portNumber": "1234", "email":
"hello@example.com" }
```

The above example defines an annotation called `startkubernetes.com/debug-info` that contains a JSON string.

Working with Pods

Pods are probably one of the most common resources in Kubernetes. They are a collection of one or more containers. The containers within the Pod share the same network and storage. That means any containers within the same Pod can talk to each other through `localhost` and access the same Volumes.

You always design your Pods as temporary, disposable entities that can get deleted and rescheduled to run on different nodes. Whenever you or Kubernetes restart the Pod, you are also restarting containers within the Pod. We will discuss how to persist data between Pod restarts in Stateful Workloads, What are Volumes?, and Creating Persistent Volumes.

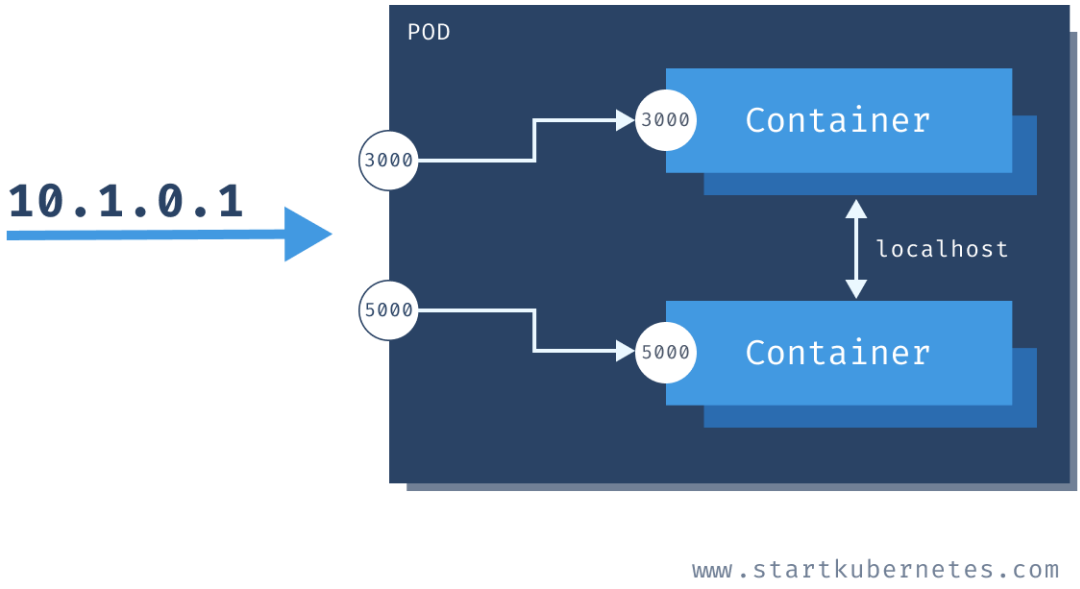


Figure 6. Kubernetes Pod

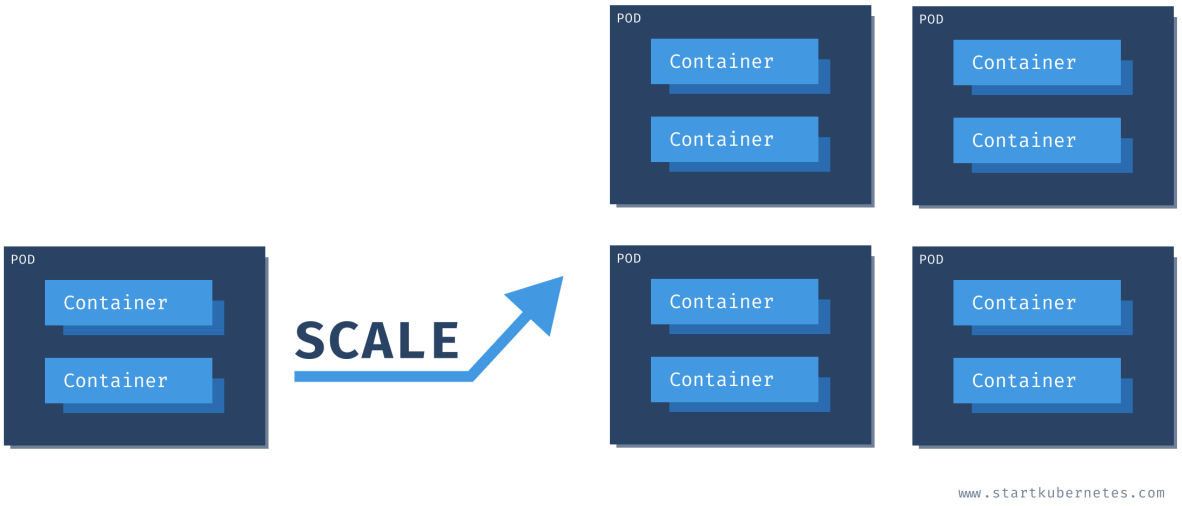
When created, each Pod gets assigned a unique IP address. The containers inside your Pod can listen to different ports. To access your containers, you can use the Pods' IP address. Using the example from the above figure, you could run `curl 10.1.0.1:3000` to talk to the one container and `curl 10.1.0.1:5000` to talk to the other container. However, if you wanted to talk between containers - for example, calling the top container from the bottom one, you could use `http://localhost:3000`.

If your Pod restarts, it will get a different IP address. Therefore, you cannot rely on the IP address. Talking to your Pods directly by the IP is not the right way to go.

An abstraction called a Kubernetes Service is what you can to communicate with your Pods. A Kubernetes Service gives you a stable IP address and DNS name. I'll talk about services later on.

Scaling Pods

All containers within the Pod get scaled together. The figure below shows how scaling from a single Pod to four Pods would look like. Note that you cannot scale individual containers within the Pods. The Pod is the unit of scale, which means that whenever you scale a Pod, you will scale all containers inside the Pod as well.



WARNING "Awesome! I can run my application and a database in the same Pod!!" No! Do not do that.

First, in most cases, your database will not scale at the same rate as your application. Remember, you're scaling a Pod and all containers inside that Pod, not just a single container.

Second, running a stateful workload in Kubernetes, such as a database, differs from running **stateless** workloads. For example, you need to ensure that data is persistent between Pod restarts and that the restarted Pods have the same network identity. You can use resources like persistent volumes and stateful sets to accomplish this. We will discuss running stateful workloads in Kubernetes in Stateful Workloads section.

Creating Pods

Usually, you shouldn't be creating Pods manually. You can do it, but you really should not. The reason being is that if the Pod crashes or if it gets deleted, it will be gone forever. That said, throughout this book, we will be creating Pods directly for the sake of simplicity and purposes of learning and explaining different concepts. However, if you're planning to run your applications inside Kubernetes, make sure you aren't creating Pods manually.

Let's look at how a single Pod can be defined using YAML.

ch3/pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-pod
  labels:
    app.kubernetes.io/name: hello
spec:
  containers:
    - name: hello-container
      image: busybox
      command: ["sh", "-c", "echo Hello from my container! && sleep 3600"]
```

In the first couple of lines, we define the kind of resource (`Pod`) and the metadata. The metadata includes the name of our Pod (`hello-pod`) and a set of labels that are simple key-value pairs (`app.kubernetes.io/name=hello`).

In the `spec` section, we are describing how the Pod should look. We will have a single container inside this Pod, called `hello-container`, and it will run the image called `busybox`. When the container starts, it executes the command defined in the `command` field.

To create the Pod, you can save the above YAML to a file called `pod.yaml`. Then, you can use Kubernetes CLI (`kubectl`) to create the Pod:

```
$ kubectl apply -f pod.yaml
pod/hello-pod created
```

Kubernetes responds with the resource type and the name it created. You can use `kubectl get pods` to get a list of all Pods running the `default` namespace of the cluster.

```
$ kubectl get pods
NAME        READY   STATUS    RESTARTS   AGE
hello-pod   1/1     Running   0          7s
```

You can use the `logs` command to see the output from the container running inside the Pod:

```
$ kubectl logs hello-pod
Hello from my container!
```

TIP

When you have multiple containers running inside the same Pod, you can use the `-c` flag to specify the container name whose logs you want to get. For example: `kubectl logs hello-pod -c hello-container`

If we delete this Pod using `kubectl delete pod hello-pod`, the Pod will be gone forever. There's nothing that would automatically restart it. If you run the `kubectl get pods` again, you will notice

the Pod is gone:

```
$ kubectl get pods
No resources found in default namespace.
```

Not automatically recreating the Pod is the opposite of the behavior you want. If you have your containers running in a Pod, you would want Kubernetes to reschedule and restart them if something goes wrong automatically.

To make sure the crashed Pods get restarted, you need a **controller** to manage the Pods' lifecycle. This controller automatically reschedules your Pod if it gets deleted or if something terrible happens (cluster nodes go down and Kubernetes need to evict the Pods).

Managing Pods with ReplicaSets

The job of a ReplicaSet is to maintain a stable number of Pod copies. The word **replicas** is often used to refer to the number of Pod copies. The ReplicaSet controller guarantees that a specified number of identical Pods (or replicas) is running . The number of replicas is controlled by the `replicas` field in the Pod resource definition.

Let’s say you start with a single Pod, and you want to scale to five Pods. The single Pod is the current state in the cluster, and the five Pods is the desired state. The ReplicaSet controller uses the current and desired state and goes to create four more Pods to meet the desired state (five pods). The ReplicaSet also keeps an eye on the Pods. If you scale the Pod up or down (add a Pod replica or remove a Pod replica), the ReplicaSet does the necessary to meet the desired number of replicas. To create the Pods, the ReplicaSet uses the Pod template that’s part of the resource definition.

But how does ReplicaSet know which Pods to watch and control?

To explain that, let’s look at how the ReplicaSet gets defined:

ch3/replicaset.yaml

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: hello
  labels:
    app.kubernetes.io/name: hello
spec:
  replicas: 5
  selector:
    matchLabels:
      app.kubernetes.io/name: hello
  template:
    metadata:
      labels:
        app.kubernetes.io/name: hello
    spec:
      containers:
        - name: hello-container
          image: busybox
          command: ['sh', '-c', 'echo Hello from my container! && sleep 3600']
```

Every Pod that's created by a ReplicaSet has a field called `metadata.ownerReferences`. This field specifies which ReplicaSet owns the Pod. When you delete a Pod, the ReplicaSet that owns it will know about it and act accordingly (i.e., re-creates the Pod).

The ReplicaSet also uses the `selector` object and `matchLabel` to check for any new Pods that it might own. If there's a new Pod that matches the selector labels and it doesn't have an owner reference, or the owner is not a controller (i.e., if we manually created a pod), the ReplicaSet will take it over and start controlling it.

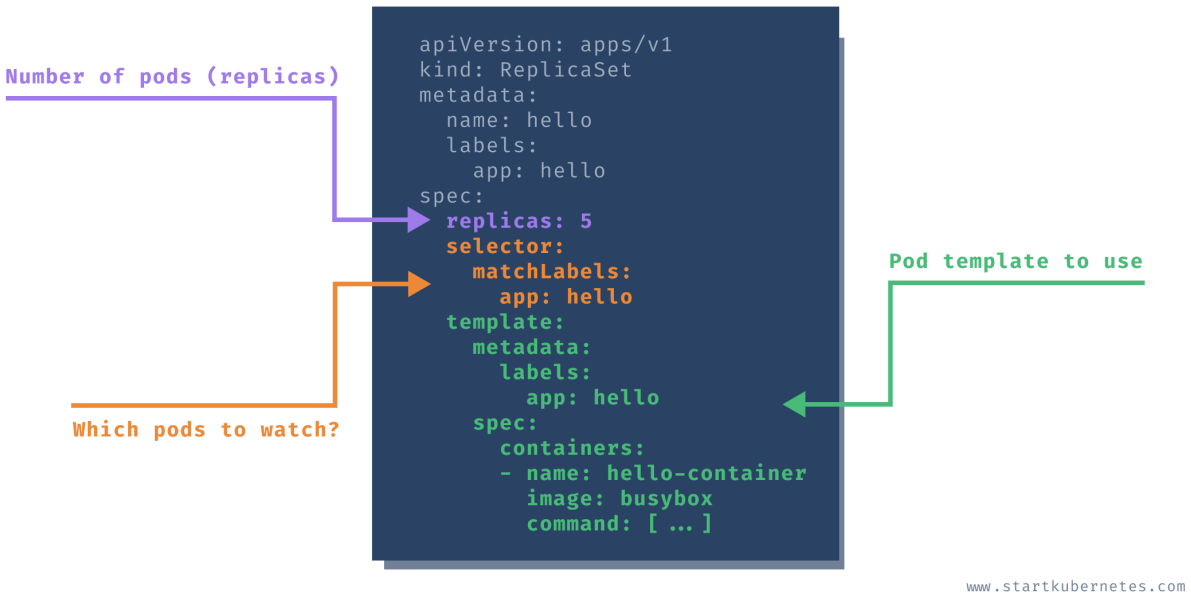


Figure 7. ReplicaSet Details

Let's save the above contents in `replicaset.yaml` file and run:

```
$ kubectl apply -f replicaset.yaml
replicaset.apps/hello created
```

You can view the ReplicaSet by running the following command:

```
$ kubectl get replicaset
NAME    DESIRED   CURRENT   READY   AGE
hello   5         5         5       30s
```

The command will show you the name of the ReplicaSet and the number of desired, current, and ready Pod replicas. If you list the Pods, you will notice that five Pods are running:

```
$ kubectl get po
NAME          READY   STATUS    RESTARTS   AGE
hello-dwx89   1/1     Running   0          31s
hello-fchvr   1/1     Running   0          31s
hello-fl6hd   1/1     Running   0          31s
hello-n667q   1/1     Running   0          31s
hello-rftkf   1/1     Running   0          31s
```

You can also list the Pods by their labels. For example, if you run `kubectl get po -l=app.kubernetes.io/name=hello`, you will get all Pods that have `app.kubernetes.io/name: hello` label set. The output includes the five Pods we created.

Let's also look at the `ownerReferences` field. We can use the `-o yaml` flag to get the YAML representation of any object in Kubernetes. Once we get the YAML, we will search for the `ownerReferences` string:

```
$ kubectl get po hello-dwx89 -o yaml | grep -A5 ownerReferences
...
  ownerReferences:
  - apiVersion: apps/v1
    blockOwnerDeletion: true
    controller: true
    kind: ReplicaSet
    name: hello
```

In the `ownerReferences`, Kubernetes sets the `name` to `hello`, and the `kind` to `ReplicaSet`. The name corresponds to the ReplicaSet that owns the Pod.

TIP	Notice how we used `po` in the command to refer to Pods? Some Kubernetes resources have short names you can use in place of the 'full names'. You can use `po` when you mean `pods` or `deploy` when you mean `deployment`. To get the full list of supported short names, you can run `kubectl api-resources`.

Another thing you will notice is how the Pods are named. Previously, where we were creating a single Pod directly, the name of the Pod was `hello-pod`, because that's what we specified in the YAML. When using the ReplicaSet, Pods are created using the combination of the ReplicaSet name (`hello`) and a semi-random string such as `dwx89`, `fchrv` and so on.

NOTE	Semi-random? Yes, Kubernetes maintainers removed the vowels, and numbers 0,1, and 3 to avoid creating 'bad words'.

Let's try and delete one of the Pods. To delete a resource from Kubernetes you use the `delete` keyword followed by the resource (e.g. `pod`) and the resource name `hello-dwx89`:

```
$ kubectl delete po hello-dwx89
pod "hello-dwx89" deleted
```

Once you've deleted the Pod, you can run `kubectl get pods` again. Did you notice something? There are still five Pods running.

```
$ kubectl get po
NAME          READY   STATUS    RESTARTS   AGE
hello-fchvr   1/1     Running   0          14m
hello-fl6hd   1/1     Running   0          14m
hello-n667q   1/1     Running   0          14m
hello-rftkf   1/1     Running   0          14m
hello-vctkh   1/1     Running   0          48s
```

If you look at the `AGE` column, you will notice four Pods created 14 minutes ago and one created more recently. ReplicaSet created this new Pod. When we deleted one Pod, the number of actual replicas decreased from five to four. The replica set controller detected that and created a new Pod to match the replicas' desired number (5).

Let's try something different now. We will manually create a Pod with labels that match the ReplicaSets selector labels (`app.kubernetes.io/name: hello`).

ch3/stray-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: stray-pod
  labels:
    app.kubernetes.io/name: hello
spec:
  containers:
    - name: stray-pod-container
      image: busybox
      command: ["sh", "-c", "echo Hello from stray pod! && sleep 3600"]
```

Save the above YAML in the `stray-pod.yaml` file, then create the Pod by running:

```
$ kubectl apply -f stray-pod.yaml
pod/stray-pod created
```

The Pod gets created, and all looks good. However, if you run `kubectl get pods` you will notice that the `stray-pod` has dissapeared. What happened?

The ReplicaSet makes sure only five Pod replicas are running. When we manually created the `stray-pod` with the label `app.kubernetes.io/name=hello` that matches the selector label on the ReplicaSet, the ReplicaSet took that new Pod for its own. Remember, manually created Pod didn't have the owner. With this new Pod under ReplicaSets' management, the number of replicas was six and not five, as stated in the ReplicaSet. Therefore, the ReplicaSet did what it's supposed to do; it deleted the new Pod to maintain the desired state of five replicas.

Zero-downtime updates?

I mentioned zero-downtime deployments and updates earlier. How can that be done using a replica set? Well, it you can't do it with a replica set. At least not in a zero-downtime manner.

Let's say we want to change the Docker image used in the original replica set from `busybox` to `busybox:1.31.1`. We could use `kubectl edit rs hello` to open the replica set YAML in the editor, then update the image value.

Once you save the changes - nothing will happen. Five Pods will still keep running as if nothing has happened. Let's check the image used by one of the Pods:

```
$ kubectl describe po hello-fchvr | grep image
  Normal  Pulling    14m        kubelet, docker-desktop  Pulling image "busybox"
  Normal  Pulled     13m        kubelet, docker-desktop  Successfully pulled image
"busybox"
```

Notice it's referencing the `busybox` image, but there's no sign of the `busybox:1.31.1` anywhere. Let's

see what happens if we delete this same Pod:

```
$ kubectl delete po hello-fchvr
pod "hello-fchvr" deleted
```

We already know that ReplicaSet will bring up a new Pod (`hello-q8fnl` in our case) to match the desired replica count from the previous test we did. If we run `describe` against the new Pod that came up, you will notice how the image is changed this time:

```
$ kubectl describe po hello-q8fnl | grep image
  Normal  Pulling    74s        kubelet, docker-desktop  Pulling image "busybox:1.31"
  Normal  Pulled     73s        kubelet, docker-desktop  Successfully pulled image
"busybox:1.31"
```

A similar thing will happen if we delete the other Pods that are still using the old image (`busybox`). The ReplicaSet would start new Pods, and this time, the Pods would use the new image `busybox:1.31.1`.

We can use another resource to manage the ReplicaSets, allowing us to update Pods in a controlled manner. Upon changing the image name, it can start Pods using the new image names in a controlled way. This resource is called a **Deployment**.

To delete all Pods, you need to delete the ReplicaSet by running: `kubectl delete rs hello`. `rs` is the short name for `replicaset`. If you list the Pods (`kubectl get po`) right after you issued the delete command, you will see the Pods getting terminated:

```
NAME          READY   STATUS        RESTARTS   AGE
hello-fchvr   1/1     Terminating   0          18m
hello-fl6hd   1/1     Terminating   0          18m
hello-n667q   1/1     Terminating   0          18m
hello-rftkf   1/1     Terminating   0          18m
hello-vctkh   1/1     Terminating   0          7m39s
```

Once the replica set terminates all Pods, they will be gone, and so will be the ReplicaSet.

Creating Deployments

A deployment resource is a wrapper around the ReplicaSet that allows doing controlled updates to your Pods. For example, if you want to update image names for all Pods, you can edit the Pod template, and the deployment controller will re-create Pods with the new image.

If we continue with the same example as we used before, this is how a Deployment would look like:

ch3/deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello
  labels:
    app.kubernetes.io/name: hello
spec:
  replicas: 5
  selector:
    matchLabels:
      app.kubernetes.io/name: hello
  template:
    metadata:
      labels:
        app.kubernetes.io/name: hello
    spec:
      containers:
        - name: hello-container
          image: busybox
          command: ["sh", "-c", "echo Hello from my container! && sleep 3600"]
```

The YAML for Kubernetes Deployment looks almost the same as for a ReplicaSet. There's the replica count, the selector labels, and the Pod template.

Save the above YAML contents in `deployment.yaml` and create the deployment:

```
$ kubectl apply -f deployment.yaml --record
deployment.apps/hello created
```

NOTE Why the `--record` flag? Using this flag, we are telling Kubernetes to store the command we executed in the annotation called `kubernetes.io/change-cause`. Record flag is useful to track the changes or commands that you executed when the deployment was updated. You will see this in action later on when we do rollouts.

To list all deployments, we can use the `get` command:

```
$ kubectl get deployment
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
hello   5/5     5            5           2m8s
```

The output is the same as when we were listing the ReplicaSets. When we create the deployment, controller also creates a ReplicaSet:

```
$ kubectl get rs
NAME               DESIRED   CURRENT   READY   AGE
hello-6fcbc8bc84   5         5         5       3m17s
```

Notice how the ReplicaSet name has the random string at the end. Finally, let's list the Pods:

```
$ kubectl get po
NAME                     READY   STATUS    RESTARTS   AGE
hello-6fcbc8bc84-27s2s   1/1     Running   0          4m2s
hello-6fcbc8bc84-49852   1/1     Running   0          4m1s
hello-6fcbc8bc84-7tpvs   1/1     Running   0          4m2s
hello-6fcbc8bc84-h7jwd   1/1     Running   0          4m1s
hello-6fcbc8bc84-prvpq   1/1     Running   0          4m2s
```

When we created a ReplicaSet previously, the Pods got named like this: `hello-fchvr`. However, this time, the Pod names are a bit longer - `hello-6fcbc8bc84-27s2s`. The random middle section in the name `6fcbc8bc84` corresponds to the random section of the ReplicaSet name, and the Pod names get created by combining the deployment name, ReplicaSet name, and a random string.

Figure 8. Deployment, ReplicaSet, and Pod naming

Just like before, if we delete one of the Pods, the Deployment and ReplicaSet will make sure awalys to maintain the number of desired replicas:

```
$ kubectl delete po hello-6fcbc8bc84-27s2s
pod "hello-6fcbc8bc84-27s2s" deleted

$ kubectl get po
NAME                     READY   STATUS    RESTARTS   AGE
hello-6fcbc8bc84-49852   1/1     Running   0          46m
hello-6fcbc8bc84-58q7l   1/1     Running   0          15s
hello-6fcbc8bc84-7tpvs   1/1     Running   0          46m
hello-6fcbc8bc84-h7jwd   1/1     Running   0          46m
hello-6fcbc8bc84-prvpq   1/1     Running   0          46m
```

How to scale the Pods up or down?

There's a handy command in Kubernetes CLI called `scale`. Using this command, we can scale up (or down) the number of Pods controlled by the Deployment or a ReplicaSet.

Let's scale the Pods down to three replicas:

```
$ kubectl scale deployment hello --replicas=3
deployment.apps/hello scaled

$ kubectl get po
NAME                     READY   STATUS    RESTARTS   AGE
hello-6fcbc8bc84-49852   1/1     Running   0          48m
hello-6fcbc8bc84-7tpvs   1/1     Running   0          48m
hello-6fcbc8bc84-h7jwd   1/1     Running   0          48m
```

Similarly, we can increase the number of replicas back to five, and ReplicaSet will create the Pods.

```
$ kubectl scale deployment hello --replicas=5
deployment.apps/hello scaled

$ kubectl get po
NAME                     READY   STATUS    RESTARTS   AGE
hello-6fcbc8bc84-49852   1/1     Running   0          49m
hello-6fcbc8bc84-7tpvs   1/1     Running   0          49m
hello-6fcbc8bc84-h7jwd   1/1     Running   0          49m
hello-6fcbc8bc84-kmmzh   1/1     Running   0          6s
hello-6fcbc8bc84-wfh8c   1/1     Running   0          6s
```

Updating the Pod templates

When we were using a ReplicaSet we noticed that ReplicaSet did not automatically restart the Pods when we updated the image name. However, Deployment can do this.

Let's use the `set image` command to update the image in the Pod templates from `busybox` to `busybox:1.31.1`.

> **TIP** You can use the `set` command to update the parts of the Pod template, such as image name, environment variables, resources, and a couple of others.

```
$ kubectl set image deployment hello hello-container=busybox:1.31.1 --record
deployment.apps/hello image updated
```

If you run the `kubectl get pods` right after you execute the `set` command, you might see something like this:

```
$ kubectl get po
NAME                     READY   STATUS        RESTARTS   AGE
hello-6fcbc8bc84-49852   1/1     Terminating   0          57m
hello-6fcbc8bc84-7tpvs   0/1     Terminating   0          57m
hello-6fcbc8bc84-h7jwd   1/1     Terminating   0          57m
hello-6fcbc8bc84-kmmzh   0/1     Terminating   0          7m15s
hello-84f59c54cd-8khwj   1/1     Running       0          36s
hello-84f59c54cd-fzcf2   1/1     Running       0          32s
hello-84f59c54cd-k947l   1/1     Running       0          33s
hello-84f59c54cd-r8cv7   1/1     Running       0          36s
hello-84f59c54cd-xd4hb   1/1     Running       0          35s
```

The controller terminated the Pods and has started five new Pods. Notice something else in the Pod names? The ReplicaSet section looks different, right? That's because Deployment scaled down the Pods controlled by the previous ReplicaSet and create a new ReplicaSet that uses the latest image we defined.

Remember that `--record` flag we set? We can now use `rollout history` command to view the previous rollouts.

```
$ kubectl rollout history deploy hello
deployment.apps/hello
REVISION  CHANGE-CAUSE
1         kubectl apply --filename=deployment.yaml --record=true
2         kubectl set image deployment hello hello-container=busybox:1.31.1 --record
=true
```

The history command shows all revisions you made to the deployment. The first revision is when we initially created the resource and the second one is when we updated the image.

Let's say we rolled out this new image version, but for some reason, we want to go back to the previous state. Using the `rollout` command, we can also roll back to an earlier revision of the resource.

To roll back, you can use the `rollout undo` command, like this:

```
$ kubectl rollout undo deploy hello
deployment.apps/hello rolled back
```

With the undo command, we rolled back the changes to the previous revision, which is the original state we were in before we updated the image:

```
$ kubectl rollout history deploy hello
deployment.apps/hello
REVISION  CHANGE-CAUSE
2         kubectl set image deployment hello hello-container=busybox:1.31.1 --record
=true
3         kubectl apply --filename=deployment.yaml --record=true
```

Let's remove the deployment by running:

```
$ kubectl delete deploy hello
deployment.apps "hello" deleted
```

Deployment strategies

You might have wondered what logic or strategy Deployment controller used to bring up new Pods or terminate the old ones when we scaled the deployments up and down and updated the image names.

There are two different strategies used by deployments to replace old Pods with new ones. The **Recreate** strategy and the **RollingUpdate** strategy. The latter is the default strategy.

Here's a way to explain the differences between the two strategies. Imagine you work at a bank, and your job is to manage the tellers and ensure there's always someone working that can handle customer requests. Since this is an imaginary bank, let's say you have 10 desks available, and at the moment, five tellers are working.

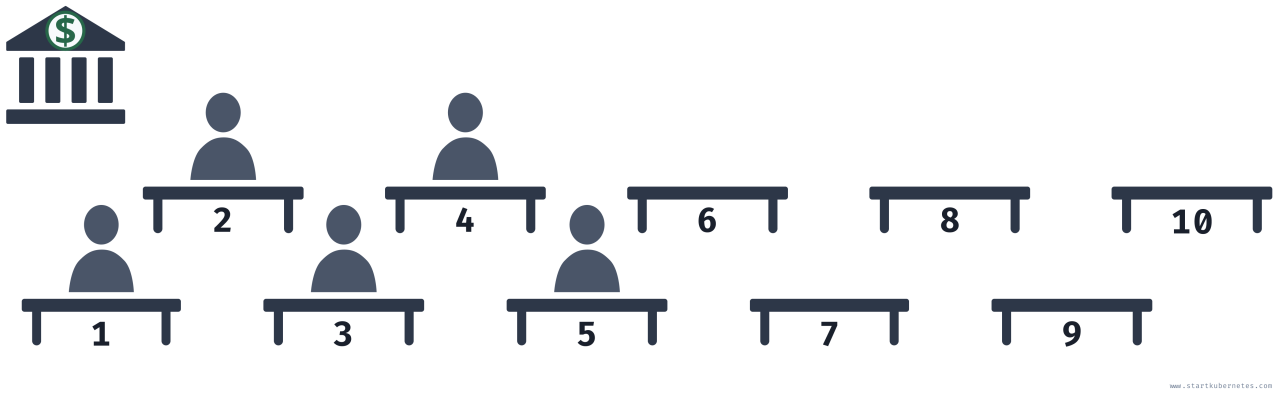


Figure 9. Five Bank Tellers at Work

Time for a shift change! The current tellers have to leave their desks and let the new shift come in to take over.

One way you can do that is to tell the current tellers to stop working. They will put up the "Closed" sign in their booth, pack up their stuff, get off their seats, and leave. Only once all of them have left their seats, the new tellers can come in, sit down, unpack their stuff, and start working.

Figure 10. New Shift Waiting

Using this strategy, there will be downtime where you won't be able to serve any customers. As shown in the figure above, you might have one teller working, and once they pack up, it will take time for the new tellers to sit down and start their work. This is how the **Recreate** strategy works.

The **Recreate** strategy terminates all existing (old) Pods (shift change happens), and only when they are all terminated (they leave their booths), it starts creating the new ones (new shift comes in).

Using a different strategy, you can keep serving all of your customers while the shift is changing. Instead of waiting for all tellers to stop working first, you can utilize the empty booths and put your new shift to work right away. That means you might have more than five booths working at the same time during the shift change.

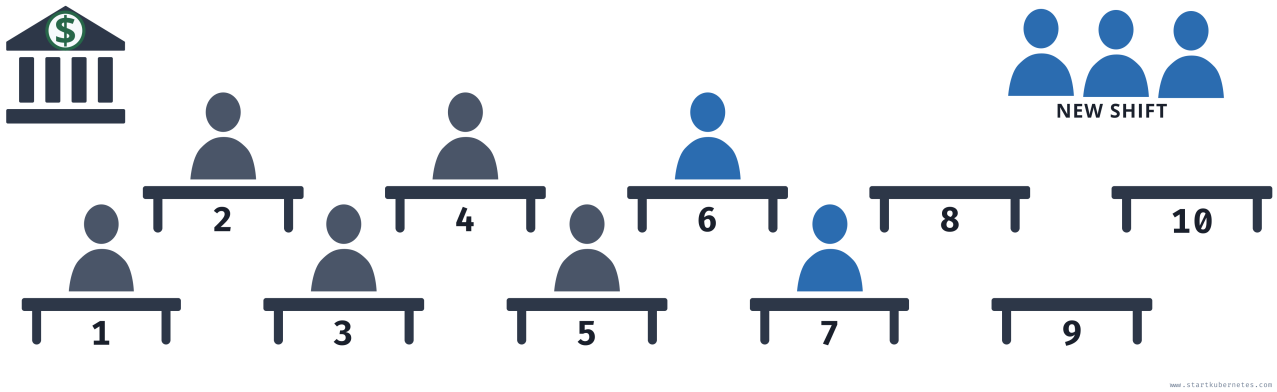


Figure 11. Seven Tellers Working

As soon as you have seven tellers working, for example (five from the old shift, two from the new shift), more tellers from the old shift can start packing up, and new tellers can start replacing them. You could also say that you always want at least three tellers working during the shift change, and you can also accommodate more than five tellers working at the same time.

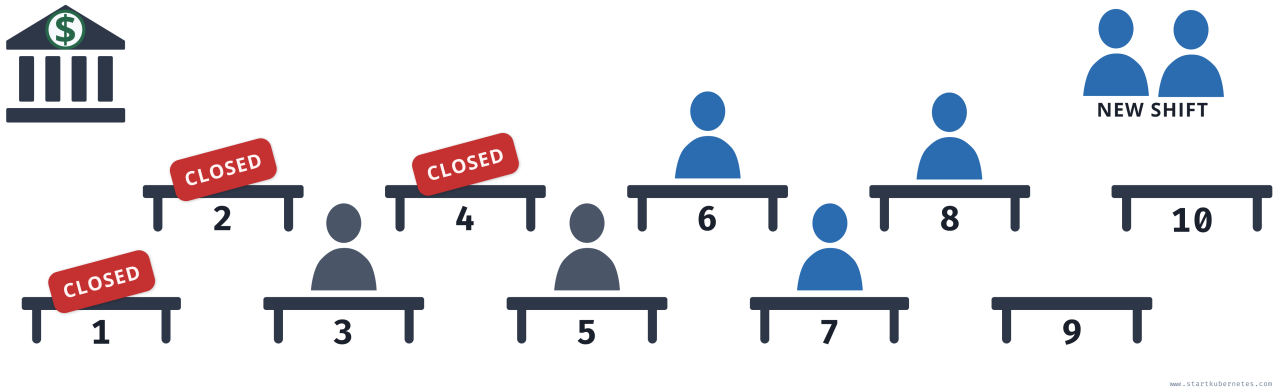


Figure 12. Mix of Tellers Working

This is how the **RollingUpdate** strategy works. The `maxUnavailable` and `maxSurge` settings specify the maximum number of Pods that can be unavailable and the maximum number of old and new Pods that can be running at the same time.

Recreate strategy

Let's create a deployment that uses the recreate strategy - notice the highlighted lines show where we specified the strategy.

ch3/deployment-recreate.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello
  labels:
    app.kubernetes.io/name: hello
spec:
  replicas: 5
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app.kubernetes.io/name: hello
  template:
    metadata:
      labels:
        app.kubernetes.io/name: hello
    spec:
      containers:
        - name: hello-container
          image: busybox
          command: ["sh", "-c", "echo Hello from my container! && sleep 3600"]
```

Copy the above YAML to `deployment-recreate.yaml` file and create the deployment:

```
kubectl apply -f deployment-recreate.yaml
```

To see the recreate strategy in action, we will need a way to watch the changes that are happening to the Pods as we update the image version, for example.

You can open a second terminal window and use the `--watch` flag when listing all Pods - the `--watch` flag will keep the command running, and any changes to the Pods are written to the screen.

```
kubectl get pods --watch
```

From the first terminal window, let's update the Docker image from `busybox` to `busybox:1.31.1` by running:

```
kubectl set image deployment hello hello-container=busybox:1.31.1
```

The second terminal window's output where we are watching the Pods should look like the one below. Note that I have added the line breaks between the groups.

```
NAME                     READY   STATUS    RESTARTS   AGE
hello-6fcbc8bc84-jpm64   1/1     Running   0          54m
hello-6fcbc8bc84-wsw6s   1/1     Running   0          54m
hello-6fcbc8bc84-wwpk2   1/1     Running   0          54m
hello-6fcbc8bc84-z2dqv   1/1     Running   0          54m
hello-6fcbc8bc84-zpc5q   1/1     Running   0          54m

hello-6fcbc8bc84-z2dqv   1/1     Terminating   0          56m
hello-6fcbc8bc84-wwpk2   1/1     Terminating   0          56m
hello-6fcbc8bc84-wsw6s   1/1     Terminating   0          56m
hello-6fcbc8bc84-jpm64   1/1     Terminating   0          56m
hello-6fcbc8bc84-zpc5q   1/1     Terminating   0          56m
hello-6fcbc8bc84-wsw6s   0/1     Terminating   0          56m
hello-6fcbc8bc84-z2dqv   0/1     Terminating   0          56m
hello-6fcbc8bc84-zpc5q   0/1     Terminating   0          56m
hello-6fcbc8bc84-jpm64   0/1     Terminating   0          56m
hello-6fcbc8bc84-wwpk2   0/1     Terminating   0          56m
hello-6fcbc8bc84-z2dqv   0/1     Terminating   0          56m

hello-84f59c54cd-77hpt   0/1     Pending       0          0s
hello-84f59c54cd-77hpt   0/1     Pending       0          0s
hello-84f59c54cd-9st7n   0/1     Pending       0          0s
hello-84f59c54cd-lxqrn   0/1     Pending       0          0s
hello-84f59c54cd-9st7n   0/1     Pending       0          0s
hello-84f59c54cd-lxqrn   0/1     Pending       0          0s
hello-84f59c54cd-z9s5s   0/1     Pending       0          0s
hello-84f59c54cd-8f2pt   0/1     Pending       0          0s
hello-84f59c54cd-77hpt   0/1     ContainerCreating   0          0s
hello-84f59c54cd-z9s5s   0/1     Pending             0          0s
hello-84f59c54cd-8f2pt   0/1     Pending             0          0s
hello-84f59c54cd-9st7n   0/1     ContainerCreating   0          1s
hello-84f59c54cd-lxqrn   0/1     ContainerCreating   0          1s
hello-84f59c54cd-z9s5s   0/1     ContainerCreating   0          1s
hello-84f59c54cd-8f2pt   0/1     ContainerCreating   0          1s
hello-84f59c54cd-77hpt   1/1     Running             0          3s
hello-84f59c54cd-lxqrn   1/1     Running             0          4s
hello-84f59c54cd-9st7n   1/1     Running             0          5s
hello-84f59c54cd-8f2pt   1/1     Running             0          6s
hello-84f59c54cd-z9s5s   1/1     Running             0          7s
```

The first couple of lines show all five Pods running. Right at the first empty line above (I added that for clarity), we ran the set image command. The controller terminated all Pods first. Once they were terminated (second empty line in the output above), the controller created the new Pods.

The apparent downside of this strategy is that once controller terminates old Pods and stastarts up the new ones, there are no running Pods to handle any traffic, which means that there will be downtime. Make sure you delete the deployment `kubectl delete deploy hello` before continuing. You can also press CTRL+C to stop running the `--watch` command from the second terminal window (keep the window open as we will use it again shortly).

Rolling update strategy

The second strategy called **RollingUpdate** does the rollout in a more controlled way. There are two settings you can tweak to control the process: `maxUnavailable` and `maxSurge`. Both of these settings are optional and have the default values set - 25% for both settings.

The `maxUnavailable` setting specifies the maximum number of Pods that can be unavailable during the rollout process. You can set it to an actual number or a percentage of desired Pods.

Let's say `maxUnavailable` is set to 40%. When the update starts, the old ReplicaSet is scaled down to 60%. As soon as new Pods are started and ready, the old ReplicaSet is scaled down again and the new ReplicaSet is scaled up. This happens in such a way that the total number of available Pods (old and new, since we are scaling up and down) is always at least 60%.

The `maxSurge` setting specifies the maximum number of Pods that can be created *over* the desired number of Pods. If we use the same percentage as before (40%), the new ReplicaSet is scaled up right away when the rollout starts. The new ReplicaSet will be scaled up in such a way that it does not exceed 140% of desired Pods. As old Pods get killed, the new ReplicaSet scales up again, making sure it never goes over the 140% of desired Pods.

Let's create the deployment again, but this time we will use the `RollingUpdate` strategy.

ch3/deployment-rolling.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello
  labels:
    app.kubernetes.io/name: hello
spec:
  replicas: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 40%
      maxSurge: 40%
  selector:
    matchLabels:
      app.kubernetes.io/name: hello
  template:
    metadata:
      labels:
        app.kubernetes.io/name: hello
    spec:
      containers:
        - name: hello-container
          image: busybox
          command: ["sh", "-c", "echo Hello from my container! && sleep 3600"]
```

Save the contents to `deployment-rolling.yaml` and create the deployment:

```
$ kubectl apply -f deployment-rolling.yaml
deployment.apps/hello created
```

Let's do the same we did before, run the `kubectl get po --watch` from the second terminal window to start watching the Pods.

```
kubectl set image deployment hello hello-container=busybox:1.31.1
```

This time, you will notice that the new ReplicaSet is scaled up right away and the old ReplicaSet is scaled down at the same time:

```
$ kubectl get po --watch
NAME                     READY   STATUS    RESTARTS   AGE
hello-6fcbc8bc84-4xnt7   1/1     Running   0          37s
hello-6fcbc8bc84-bpsxj   1/1     Running   0          37s
hello-6fcbc8bc84-dx4cg   1/1     Running   0          37s
hello-6fcbc8bc84-fx7ll   1/1     Running   0          37s
hello-6fcbc8bc84-fxsp5   1/1     Running   0          37s
hello-6fcbc8bc84-jhb29   1/1     Running   0          37s
hello-6fcbc8bc84-k8dh9   1/1     Running   0          37s
hello-6fcbc8bc84-qlt2q   1/1     Running   0          37s
hello-6fcbc8bc84-wx4v7   1/1     Running   0          37s
hello-6fcbc8bc84-xkr4x   1/1     Running   0          37s

hello-84f59c54cd-ztfg4   0/1     Pending   0          0s
hello-84f59c54cd-ztfg4   0/1     Pending   0          0s
hello-84f59c54cd-mtwcc   0/1     Pending   0          0s
hello-84f59c54cd-x7rww   0/1     Pending   0          0s

hello-6fcbc8bc84-dx4cg   1/1     Terminating   0          46s
hello-6fcbc8bc84-fx7ll   1/1     Terminating   0          46s
hello-6fcbc8bc84-bpsxj   1/1     Terminating   0          46s
hello-6fcbc8bc84-jhb29   1/1     Terminating   0          46s
...
```

Using the rolling strategy, you can keep a percentage of Pods running at all times while you're doing updates. That means that there will be no downtime for your users.

Make sure you delete the deployment before continuing:

```
kubectl delete deploy hello
```

Accessing and exposing Pods with Services

Pods are supposed to be temporary or short-lived. Once they crash, they are gone, and the ReplicaSet ensures to bring up a new Pod to maintain the desired number of replicas.

Let's say we are running a web frontend in a container within Pods. Each Pod gets a unique IP address. However, due to their temporary nature, we cannot rely on that IP address.

Let's create a deployment that runs a web frontend:

ch3/web-frontend.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-frontend
  labels:
    app.kubernetes.io/name: web-frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: web-frontend
  template:
    metadata:
      labels:
        app.kubernetes.io/name: web-frontend
    spec:
      containers:
        - name: web-frontend-container
          image: learncloudnative/helloworld:0.1.0
          ports:
            - containerPort: 3000
```

Comparing this deployment to the previous one, you will notice we changed the resource names and the image we are using.

One new thing we added to the deployment is the `ports` section. Using the `containerPort` field, we set the port number website server listens on. The `learncloudnative/helloworld:0.1.0` is a simple Node.js Express application.

Save the above YAML in `web-frontend.yaml` and create the deployment:

```
$ kubectl apply -f web-frontend.yaml
deployment.apps/web-frontend created
```

Run `kubectl get pods` to ensure Pod is up and running and then get the logs from the container:

```
$ kubectl get po
NAME                           READY   STATUS    RESTARTS   AGE
web-frontend-68f784d855-rdt97   1/1     Running   0          65s

$ kubectl logs web-frontend-68f784d855-rdt97

> helloworld@1.0.0 start /app
> node server.js

Listening on port 3000
```

From the logs, you will notice that the container is listening on port `3000`. If we set the output flag to gives up the wide output (`-o wide`), you'll notice the Pods' IP address - `10.244.0.170`:

```
$ kubectl get po -o wide
NAME                           READY   STATUS    RESTARTS   AGE   IP           NODE
NOMINATED NODE   READINESS GATES
web-frontend-68f784d855-rdt97   1/1     Running   0          15s   172.17.0.4
minikube   <none>           <none>
```

If we delete this Pod, a new one will take its' place, and it will get a brand new IP address as well:

```
$ kubectl delete po web-frontend-68f784d855-rdt97
pod "web-frontend-68f784d855-rdt97" deleted

$ kubectl get po -o wide
NAME                           READY   STATUS    RESTARTS   AGE   IP           NODE
NOMINATED NODE   READINESS GATES
web-frontend-68f784d855-8c76m   1/1     Running   0          15s   172.17.0.5
minikube   <none>           <none>
```

Similarly, if we scale up the deployment to four Pods, we will four different IP addresses:

```
$ kubectl scale deploy web-frontend --replicas=4
deployment.apps/web-frontend scaled

$ kubectl get pods -o wide
NAME                            READY   STATUS    RESTARTS   AGE     IP           NODE
NOMINATED NODE   READINESS GATES
web-frontend-68f784d855-8c76m   1/1     Running   0          5m23s   172.17.0.5
minikube   <none>           <none>
web-frontend-68f784d855-jrqq4   1/1     Running   0          18s     172.17.0.6
minikube   <none>           <none>
web-frontend-68f784d855-mftl6   1/1     Running   0          18s     172.17.0.7
minikube   <none>           <none>
web-frontend-68f784d855-stfqj   1/1     Running   0          18s     172.17.0.8
minikube   <none>           <none>
```

How to access the Pods without a service?

If you try to send a request to one of those IP addresses, it's not going to work:

```
$ curl -v 172.17.0.5:3000
*   Trying 172.17.0.5...
* TCP_NODELAY set
* Connection failed
* connect to 172.17.0.5 port 3000 failed: Network is unreachable
* Failed to connect to 172.17.0.5 port 3000: Network is unreachable
* Closing connection 0
curl: (7) Failed to connect to 172.17.0.5 port 3000: Network is unreachable
```

The Pods are running within the cluster, and that IP address is only accessible from within the cluster.

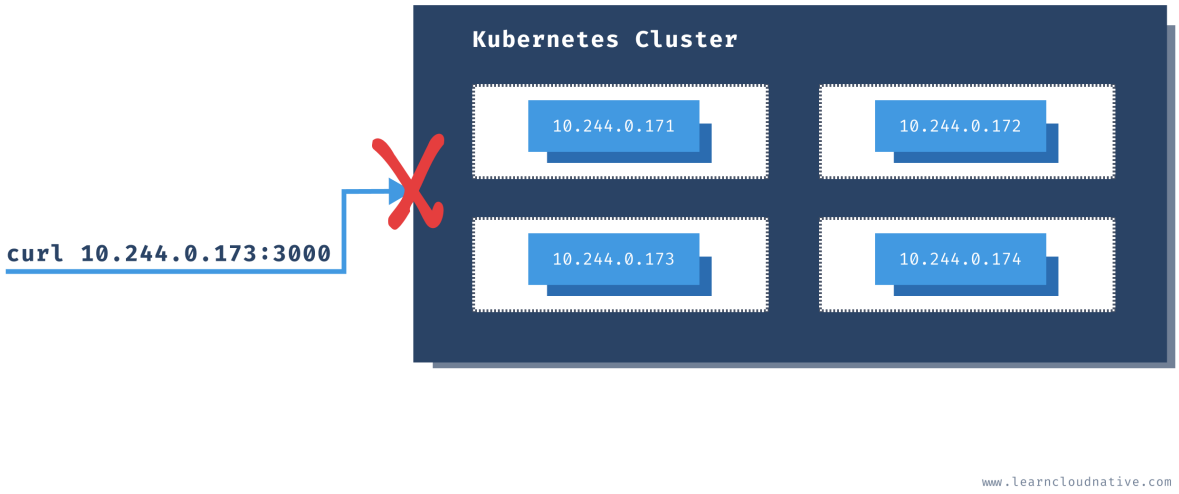


Figure 13. Can't Access Pods from Outside

For the testing purposes, you can run a pod inside the cluster and then get shell access to that Pod. Yes, that is possible!

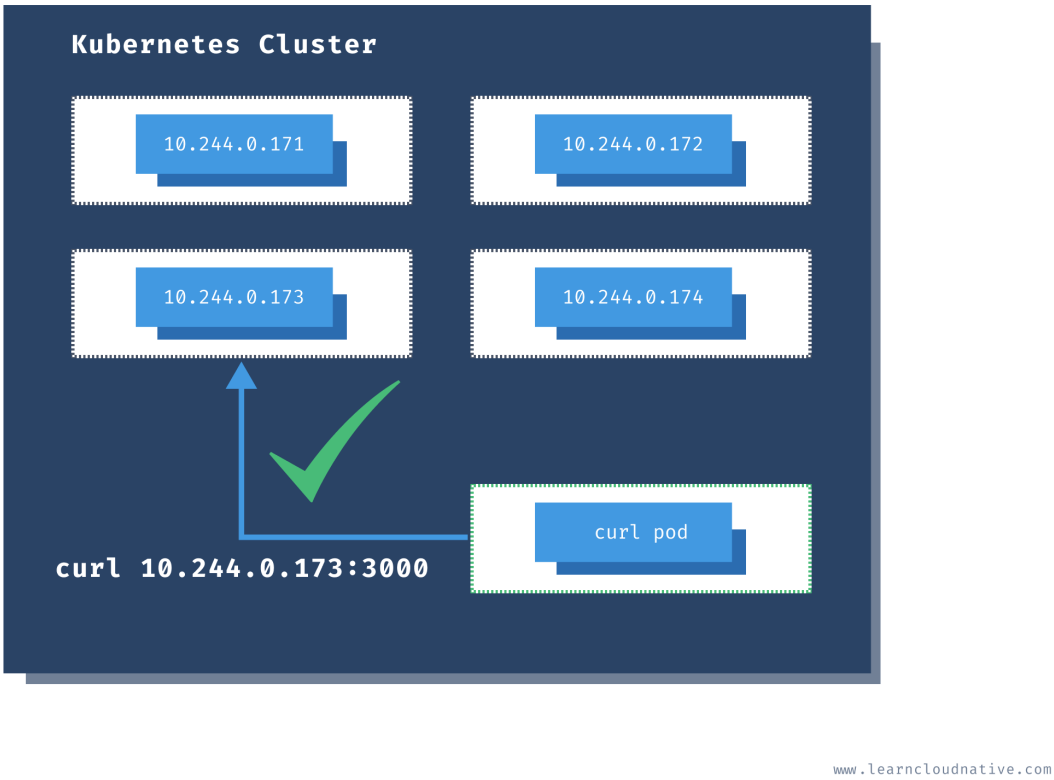


Figure 14. Accessing Pods from a Pod

The `radialbusyboxplus:curl` is the image I frequently run inside the cluster if I need to check something or debug things. Using the `-i` and `--tty` flags, we are want to allocate a terminal (`tty`), and that we want an interactive session so that we can run commands directly inside the container.

I usually name this Pod `curl`, but you can name it whatever you like:

```
$ kubectl run curl --image=radial/busyboxplus:curl -i --tty
If you dont see a command prompt, try pressing enter.
[ root@curl:/ ]$
```

Now that we have access to the the terminal running inside the container that' inside the cluster, we can run the same cURL command as before:

```
[ root@curl:/ ]$ curl -v 172.17.0.5:3000
> GET / HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 172.17.0.5:3000
> Accept: */*
>
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: text/html; charset=utf-8
< Content-Length: 111
< ETag: W/"6f-U4ut6Q03D1uC/sbzBDyZfMqFSh0"
< Date: Wed, 20 May 2020 22:10:49 GMT
< Connection: keep-alive
<
<link rel="stylesheet" type="text/css" href="css/style.css" />

<div class="container">
    Hello World!
</div>[ root@curl:/ ]$
```

This time, we get a response from the Pod! Make sure you run `exit` to return to your terminal. The `curl` Pod will continue to run and to reaccess it, you can use the `attach` command:

```
kubectl attach curl -c curl -i -t
```

TIP You can get a terminal session to any container running inside the cluster using the `attach` command.

Using a Kubernetes Service

The Kubernetes Service is an abstraction that gives us a way to reach the Pod IP's reliably.

The service controller (similar to the ReplicaSet controller) maintains a list of endpoints or the Pod IP addresses. The controller uses a selector and labels to watch the Pods. Whenever the controller creates or deletes a Pod that matches the selector, the controller adds or removes the Pods' IP address from the endpoints list.

Let's look at how would the Service look like for our website:

ch3/web-frontend-service.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: web-frontend
  labels:
    app.kubernetes.io/name: web-frontend
spec:
  selector:
    app.kubernetes.io/name: web-frontend
  ports:
    - port: 80
      name: http
      targetPort: 3000
```

The top portion of the YAML should already look familiar - except for the `kind` field, it's the same as we saw with the Pods, ReplicaSets, Deployments.

The highlighted `selector` section is where we define the labels that Service uses to query the Pods. If you go back to the Deployment YAML, you will notice that the Pods have this exact label set as well.

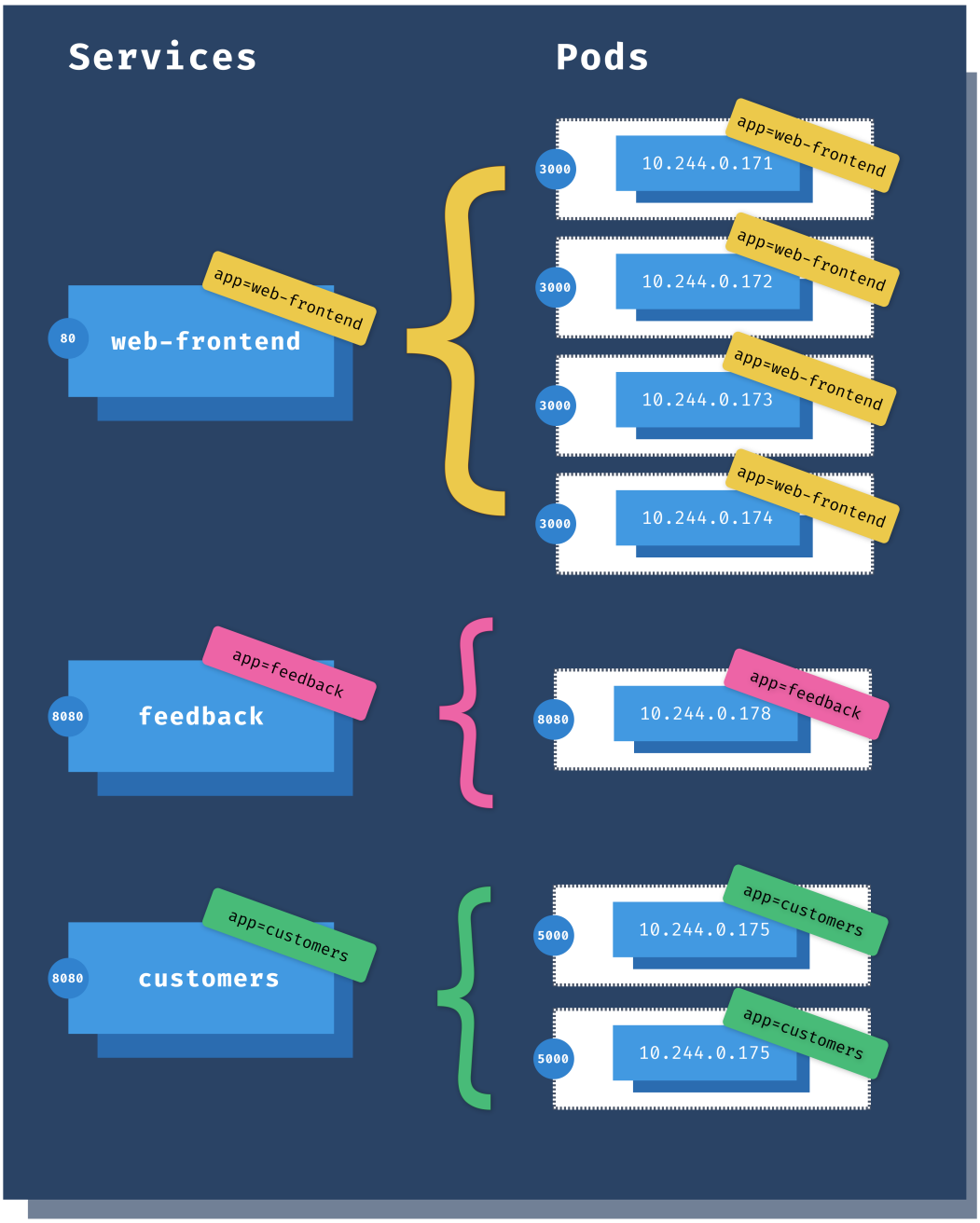


Figure 15. Kubernetes Services and Pods

Lastly, under the `ports` field, we are defining the port where the Service will be accessible on (`80`), and with the `targetPort`, we are telling the Service on which port it can access the Pods. The `targetPort` value matches the `containerPort` value in the Deployment YAML.

Save the YAML from above to `web-frontend-service.yaml` file and deploy it:

```
$ kubectl apply -f web-frontend-service.yaml
service/web-frontend created
```

To see the created service you can run the `get service` command:

```
$ kubectl get service
NAME           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
kubernetes     ClusterIP   10.96.0.1       <none>        443/TCP   7d
web-frontend   ClusterIP   10.100.221.29   <none>        80/TCP    24s
```

The `web-frontend` Service has an IP address that will not change (assuming you don't delete the Service), and you use it to access the underlying Pods reliably.

Let's attach to the `curl` container we started before and try to access the Pods through the Service:

```
$ kubectl attach curl -c curl -i -t
If you dont see a command prompt, try pressing enter.
[ root@curl:/ ]$
```

Since we set the service port to `80`, we can curl directly to the service IP and we will get back the same response as previously:

```
[ root@curl:/ ]$ curl 10.100.221.29
<link rel="stylesheet" type="text/css" href="css/style.css" />

<div class="container">
    Hello World!
</div>
```

Even though the Service IP address is stable and won't change, it would be much better to use a friendlier name to access the Service. Every Service you create in Kubernetes gets a DNS name assigned following this format `service-name.namespace.svc.cluster-domain.sample`.

So far, we deployed everything to the `default` namespace and the cluster domain is `cluster.local` we can access the service using `web-frontend.default.svc.cluster.local`:

```
[ root@curl:/ ]$ curl web-frontend.default.svc.cluster.local
<link rel="stylesheet" type="text/css" href="css/style.css" />

<div class="container">
    Hello World!
</div>
```

In addition to using the full name, you can also use just the service name, or service name and the namespace name:

```
</div>[ root@curl:/ ]$ curl web-frontend
<link rel="stylesheet" type="text/css" href="css/style.css" />

<div class="container">
    Hello World!
[ root@curl:/ ]$ curl web-frontend.default
<link rel="stylesheet" type="text/css" href="css/style.css" />

<div class="container">
    Hello World!
</div>
```

I would suggest you always use the Service name and the namespace name when making requests.

Using the Kubernetes proxy

Another way for accessing services that are only available inside of the cluster is through the proxy. The `kubectl proxy` command creates a gateway between your local computer (localhost) and the Kubernetes API server.

The proxy allows you to access the Kubernetes API as well as access the Kubernetes services. You should **NEVER** use this proxy to expose your service to the public. You should only use the proxy for debugging or troubleshooting.

Let's open a separate terminal window and run the following command to start the proxy server that will proxy requests from `localhost:8080` to the Kubernetes API inside the cluster:

```
kubectl proxy --port=8080
```

If you open `http://localhost:8080/` in your browser, you will see the list of all APIs from the Kubernetes API proxy:

```
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/",
    "/apis/admissionregistration.k8s.io",
    "/apis/admissionregistration.k8s.io/v1",
    "/apis/admissionregistration.k8s.io/v1beta1",
  ]
  ...
}
```

For example, if you want to see all Pods running in the cluster, you could navigate to `http://localhost:8080/api/v1/pods` or navigate to `http://localhost:8080/api/v1/namespaces` to see

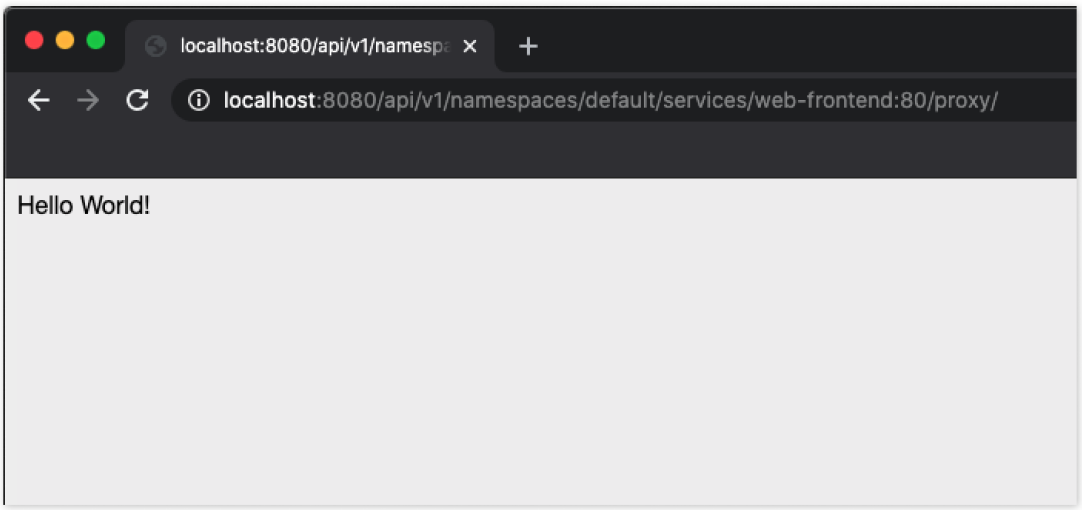
all namespaces.

Using this proxy, we can also access the `web-frontend` service we deployed. So, instead of running a Pod inside the cluster and make cURL requests to the service or Pods, you can create the proxy server and use the following URL to access the service:

```
http://localhost:8080/api/v1/namespaces/default/services/web-frontend:80/proxy/
```

NOTE The URL format to access a service is `PORT/api/v1/namespaces/[NAMESPACE]/services/[SERVICE_NAME]:[SERVICE_PORT]/proxy`. In addition to using the service port (e.g. `80`) you can also name your ports an use the port name instead (e.g. `http`).

Browsing to the URL above will render the simple HTML site with the `Hello World` message.



www.learncloudnative.com

Figure 16. Accessing the Service through Kubernetes Proxy

To stop the proxy, you can press `Ctrl` + `C` in the terminal window.

Viewing Service details

Using the `describe` command, you can describe an object in Kubernetes and look at its properties. For example, let's take a look at the details of the `web-frontend` Service we deployed:

```
$ kubectl describe svc web-frontend
Name:              web-frontend
Namespace:         default
Labels:            app.kubernetes.io/name=web-frontend
Annotations:       Selector:  app.kubernetes.io/name=web-frontend
Type:              ClusterIP
IP:                10.100.221.29
Port:              http  80/TCP
TargetPort:        3000/TCP
Endpoints:         172.17.0.4:3000,172.17.0.5:3000,172.17.0.6:3000 + 1 more...
Session Affinity:  None
Events:            <none>
```

This view gives us more information than the `get` command does - it shows the labels, selector, the service type, the IP, and the ports. Additionally, you will also notice the `Endpoints`. These IP addresses correspond to the IP addresses of the Pods.

You can also view endpoints using the `get endpoints` command:

```
$ kubectl get endpoints
NAME           ENDPOINTS                                                  AGE
kubernetes     192.168.64.5:8443                                          13m
web-frontend   172.17.0.4:3000,172.17.0.5:3000,172.17.0.6:3000 + 1 more...   66s
```

To see the controller that manages these endpoints in action, you can use the `--watch` flag to watch the endpoints like this:

```
$ kubectl get endpoints --watch
```

Then, in a separate terminal window, let's scale the deployment to one Pod:

```
$ kubectl scale deploy web-frontend --replicas=1
deployment.apps/web-frontend scaled
```

As soon as the deployment is scaled, you will notice how the endpoints get automatically updated. For example, this is how the output looks like when the deployment is scaled:

```
$ kubectl get endpoints -w
NAME           ENDPOINTS                                                  AGE
kubernetes     192.168.64.5:8443                                          14m
web-frontend   172.17.0.4:3000,172.17.0.5:3000,172.17.0.6:3000 + 1 more...   93s
web-frontend   172.17.0.4:3000,172.17.0.5:3000,172.17.0.7:3000            95s
web-frontend   172.17.0.5:3000                                            95s
```

Notice how it went from four Pods, then down to two and finally to one. If you scale the deployment

back to four Pods, you will see the endpoints list populated with new IPs.

Kubernetes service types

From the service description output we saw earlier, you might have noticed this line:

```
Type:              ClusterIP
```

Every Kubernetes Service has a type. If you don't provide a service type, the **ClusterIP** gets assigned by default. In addition to the ClusterIP, there are three other service types in Kubernetes. These are **NodePort**, **LoadBalancer**, and **ExternalName**.

Let's explain the differences between these service types.

ClusterIP

You would use the **ClusterIP** service type to access Pods from within the cluster through a cluster-internal IP address. In most cases, you will use this type of service for your applications running inside the cluster. Using the ClusterIP type, you can define the port you want your service to be listening on through the `ports` section in the YAML file.

Kubernetes assigns a cluster IP to the service. You can then access the service using the cluster IP address and the port you specified in the YAML.

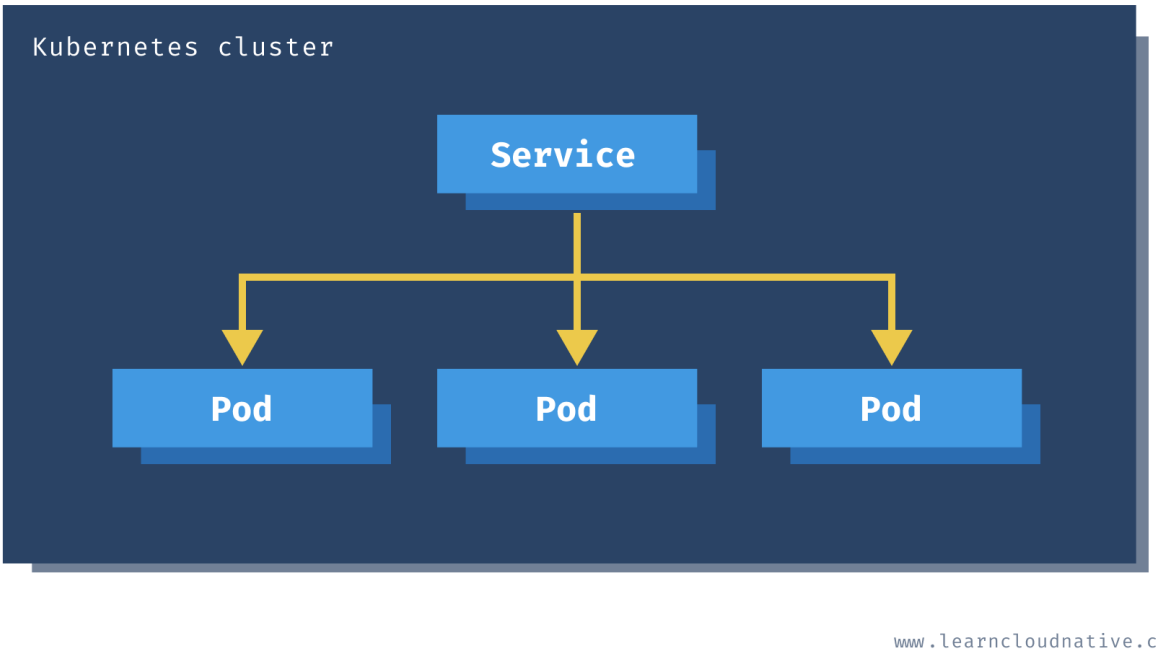


Figure 17. Cluster IP Service

NodePort

At some point, you will want to expose your services to the public and allow external traffic to enter your cluster. The **NodePort** service type opens a specific port on every worker node in your

cluster. Any traffic sent to the node IP and the port number reaches the Service and your Pods.

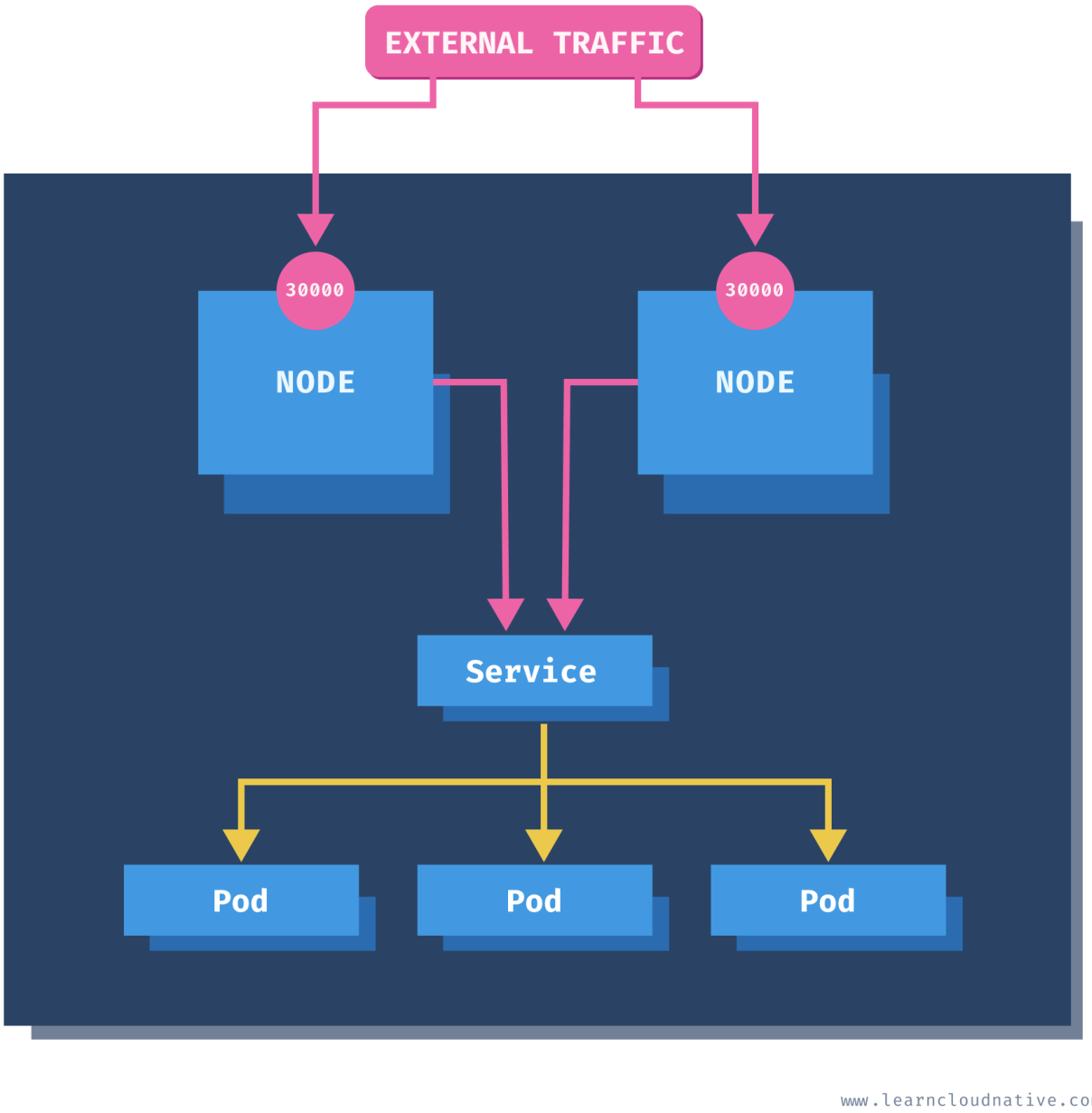


Figure 18. NodePort Service

For example, if you use the node IP and the port `30000` as shown in the figure above, you will access the Service and the Pods.

To create a NodePort Service, you need to specify the service type as shown in the listing below.

ch3/nodeport-service.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: web-frontend
  labels:
    app.kubernetes.io/name: web-frontend
spec:
  type: NodePort
  selector:
    app.kubernetes.io/name: web-frontend
  ports:
    - port: 80
      name: http
      targetPort: 3000
```

You can set additional field called `nodePort` under the `ports` section. However, it is a best practice to leave it out and let Kubernetes pick a port available on all nodes in the cluster. By default, Kubernetes allocates the node port between 30000 and 32767 (this is configurable in the API server).

You would use the NodePort type when you want to control the load balancing. You can expose your services via NodePort and then configure the load balancer to use the node IPs and node ports. Another scenario where you could use this is if you are migrating an existing solution to Kubernetes, for example. In that case, you'd probably already have an existing load balancer, and you could add the node IPs and node ports to the load balancing pool.

Let's delete the previous `web-frontend` service and create one that uses `NodePort`. To delete the previous service, run `kubectl delete svc web-frontend`. Then, copy the YAML contents above to the `nodeport-service.yaml` file and run `kubectl apply -f web-frontend-nodeport.yaml`:

ch3/nodeport-service.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: web-frontend
  labels:
    app.kubernetes.io/name: web-frontend
spec:
  type: NodePort
  selector:
    app.kubernetes.io/name: web-frontend
  ports:
    - port: 80
      name: http
      targetPort: 3000
```

Once the service is created, run `kubectl get svc` - you will notice the service type has changed and the port number is random as well:

```
$ kubectl get svc
NAME           TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes     ClusterIP   10.96.0.1        <none>        443/TCP        16m
web-frontend   NodePort    10.107.154.215   <none>        80:30417/TCP   4s
```

Now since we are using a local Kubernetes cluster (kind, Minikube, or Docker for Mac/Windows), demonstrating the NodePorts is a bit awkward. We have a single node, and the node IPs are private as well. When using a cloud-managed cluster you can set up the load balancer to access the same virtual network where your nodes are. Then you can configure it to access the node IPs through the node ports.

Let's get the internal node IP:

```
$ kubectl describe node | grep InternalIP
  InternalIP:  192.168.64.5
```

TIP If using Minikube, you can also run `minikube ip` to get the cluster's IP address.

Armed with this IP we can cURL to the clusters IP address and the node port (`30417`):

```
$ curl 192.168.64.5:30417
<link rel="stylesheet" type="text/css" href="css/style.css" />

<div class="container">
    Hello World!
</div>
```

If you are using Docker for Desktop, you can use `localhost` as the node address and the node port to access the service. If you open `http://localhost:30417` in your browser, you will be able to access the `web-frontend` service.

LoadBalancer

The LoadBalancer service type is the way to expose Kubernetes services to external traffic. If you are running a cloud-managed cluster and create the Service of the LoadBalancer type, the Kubernetes cloud controller creates an actual Load Balancer in your cloud account.

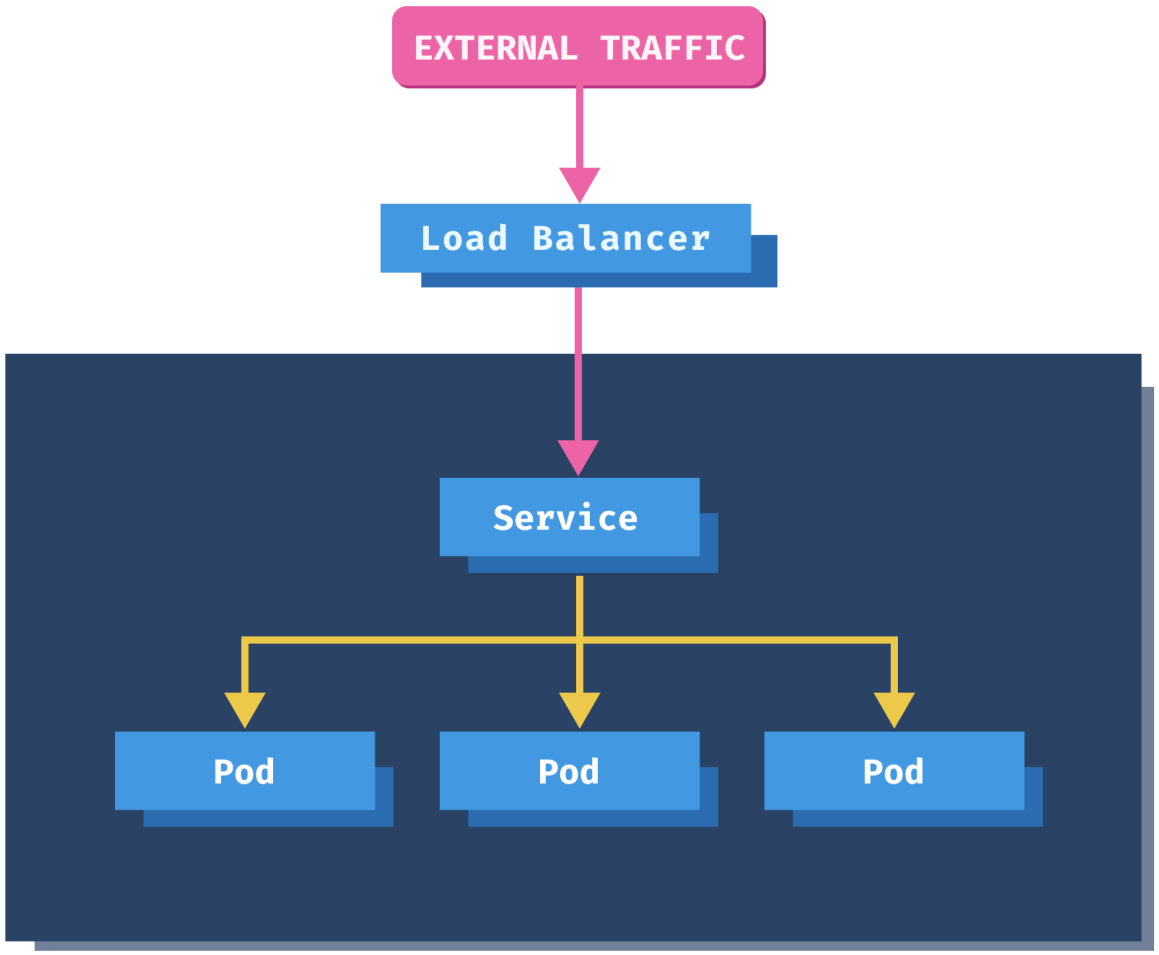


Figure 19. LoadBalancer Service

Let's delete the previous NodePort service with `kubectl delete svc web-frontend`.

ch3/lb-service.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: web-frontend
  labels:
    app.kubernetes.io/name: web-frontend
spec:
  type: LoadBalancer
  selector:
    app.kubernetes.io/name: web-frontend
  ports:
    - port: 80
      name: http
      targetPort: 3000
```

We can create a service that uses the LoadBalancer type using the above YAML. Run `kubectl apply`

`-f lb-service.yaml` to create the service.

If we look at the service now, you'll notice that the type has changed to LoadBalancer and the external IP address is pending:

```
$ kubectl get svc
NAME           TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes     ClusterIP      10.96.0.1       <none>        443/TCP        19m
web-frontend   LoadBalancer   10.106.30.168   <pending>     80:30962/TCP   4s
```

NOTE When using a cloud-managed Kubernetes cluster, the external IP would be a public, the external IP address you could use to access the service.

If you are using Docker Desktop, you can open `http://localhost` or `http://127.0.0.1` in your browser to see the website running inside the cluster.

If you are using Minikube, you can run the `minikube tunnel` command from a separate terminal window. Once the tunnel command is running, run `kubectl get svc` again to get an external IP address of the service:

```
$ kubectl get svc
NAME           TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)        AGE
kubernetes     ClusterIP      10.96.0.1       <none>          443/TCP        21m
web-frontend   LoadBalancer   10.106.30.168   10.106.30.168   80:30962/TCP   104s
```

You can now open `http://10.106.30.168` in your browser to access the service running inside the cluster. We will discuss how the Minikube tunnel command works in the Exposing multiple applications with Ingress section.

ExternalName

The **ExternalName** service type is a particular type of a service that does not use selectors. Instead, it uses DNS names.

Using the ExternalName, you can map a Kubernetes service to a DNS name. When you send a request to the Service, it returns the CNAME record with the value in `externalName` field instead of the service's cluster IP.

Here's an example of how ExternalName service would look like:

ch3/external-name.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: my-database
spec:
  type: ExternalName
  externalName: mydatabase.example.com
```

You could use the ExternalName service type when migrating workloads to Kubernetes, for example. You could keep your database running outside of the cluster and then use the `my-database` service to access it from the workloads running inside your cluster.

Exposing multiple applications with Ingress

You can use the Ingress resource to manage external access to the Services running inside your cluster. With the Ingress resource, you can define the rules on how the services inside the cluster can be accessed.

The Ingress resource on its own is useless. It's a collection of rules and paths, but it needs something to apply these rules to. That "something" is an **ingress controller**. The ingress controller acts as a gateway and routes the traffic based on the Ingress resource rules defined.

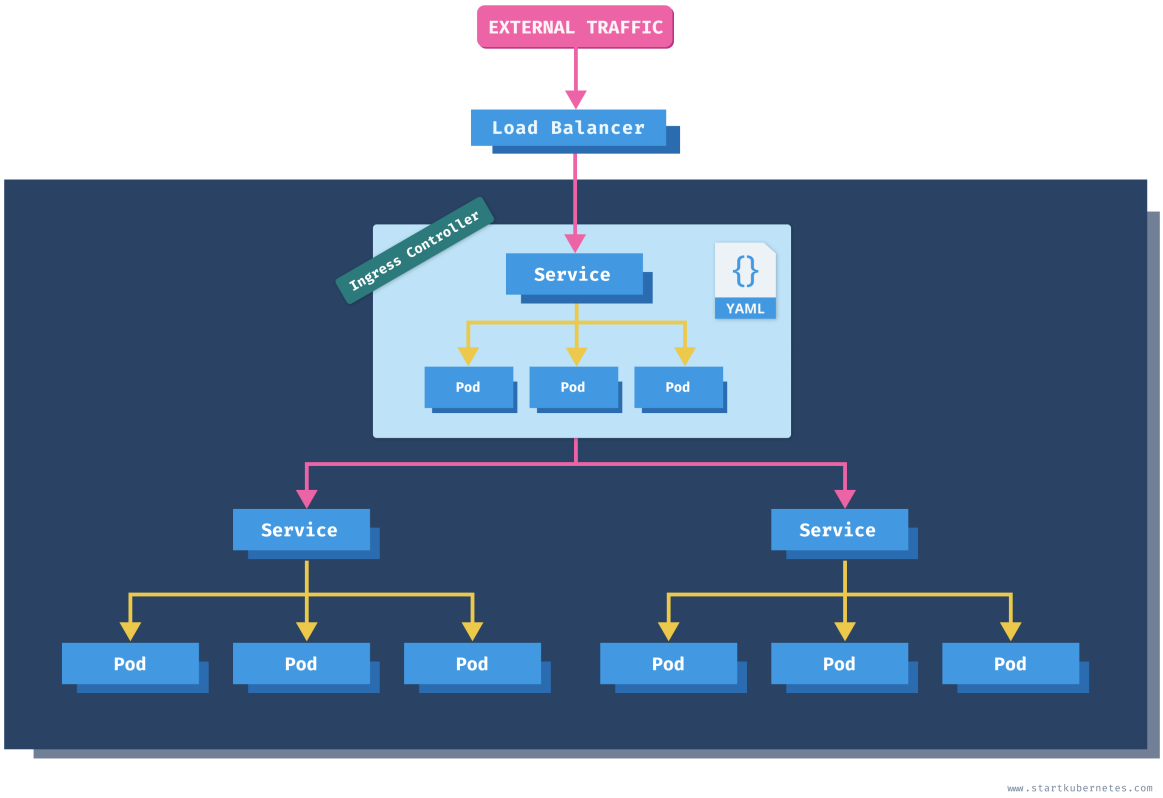


Figure 20. Kubernetes Ingress

An ingress controller is a collection of the following items:

- Kubernetes deployment running one or more Pods with containers running a gateway/proxy server such as NGINX, Ambassador, etc.
- Kubernetes service that exposes the ingress controller Pods
- Other supporting resources for the ingress controller (configuration maps, secrets, etc.)

NOTE

How about the load balancer? The load balancer is not necessarily part of the Ingress controller. The Kubernetes service used for the ingress controller can be of the LoadBalancer type, which triggers a load balancer's creation if using a cloud-managed Kubernetes cluster. It is a way for the traffic to enter your cluster and, subsequently the ingress controller that routes the traffic according to the rules.

The idea is that you can deploy the ingress controller, expose it on a public IP address, then use the Ingress resource to create the traffic rules. Here's how an Ingress resource would look like:

ch3/ingress-example.yaml

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-example
spec:
  rules:
    - host: example.com
      http:
        paths:
          - path: /blog
            backend:
              serviceName: my-blog-service
              servicePort: 5000
          - path: /music
            backend:
              serviceName: my-music-service
              servicePort: 8080
```

With these rules, we are routing traffic that comes into `example.com/blog` to a Kubernetes service `my-blog-service:5000`. Similarly, any traffic coming to `example.com/music` goes to a Kubernetes service `my-music-service:8080`.

NOTE

The ingress resource will also contain one or more annotations to configure the Ingress controller. The annotations and options you can configure will depend on the ingress controller you're using.

Let's say we want to run two websites in our cluster - the first one will be a simple Hello World website, and the second one will be a Daily Dog Picture website that shows a random dog picture.

Assuming you have your cluster up and running, let's create the deployments for these two websites.

ch3/helloworld-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
  labels:
    app.kubernetes.io/name: hello-world
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: hello-world
  template:
    metadata:
      labels:
        app.kubernetes.io/name: hello-world
    spec:
      containers:
        - name: hello-world-container
          image: learncloudnative/helloworld:0.1.0
          ports:
            - containerPort: 3000
```

Run `kubectl apply -f helloworld-deployment.yaml` to create the `hello-world` deployment. Next, we will deploy the Daily Dog Picture website.

ch3/dogpic-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dogpic-web
  labels:
    app.kubernetes.io/name: dogpic-web
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: dogpic-web
  template:
    metadata:
      labels:
        app.kubernetes.io/name: dogpic-web
    spec:
      containers:
        - name: dogpic-container
          image: learncloudnative/dogpic-service:0.1.0
          ports:
            - containerPort: 3000
```

Run `kubectl apply -f dogpic-deployment.yaml` to deploy the Daily Dog Picture website. Make sure both pods from both deployments are up and running:

```
$ kubectl get pods
NAME                           READY   STATUS    RESTARTS   AGE
dogpic-web-559f4bb5db-dlrks    1/1     Running   0          24m
hello-world-5fd44c56d7-d8g4j   1/1     Running   0          29m
```

We still need to deploy the Kubernetes Services for both of these deployments. The services will be of default, `ClusterIP` type, so there's no need to set `type` field explicitly. You can refer to Kubernetes service types for the explanation of the `ClusterIP` service.

ch3/services.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: dogpic-service
  labels:
    app.kubernetes.io/name: dogpic-web
spec:
  selector:
    app.kubernetes.io/name: dogpic-web
  ports:
    - port: 3000
      name: http
---
kind: Service
apiVersion: v1
metadata:
  name: hello-world
  labels:
    app.kubernetes.io/name: hello-world
spec:
  selector:
    app.kubernetes.io/name: hello-world
  ports:
    - port: 3000
      name: http
```

Save the YAML above to `service.yaml` and then create the services by running `kubectl apply -f services.yaml`.

TIP You can use `---` as a separator in YAML files to deploy multiple resources from a single file.

Installing the Ambassador API gateway

Before we create the Ingress resource, we need to deploy an Ingress controller. The job of an ingress controller is to receive the incoming traffic and route it based on the rules defined in the Ingress resource.

You have multiple options you can go with for the Ingress controller. Some of the gateways and proxies you could use are:

- Ambassador
- NGINX
- HAProxy
- Traefik

You can find the list of other controllers in the Kubernetes ingress controller documentation.

In this example, I'll be using the open-source version of the Ambassador API gateway.

NOTE	"I heard ABC/XZY/DEF is much better than GHI and JKL". Yep, that very well might be right. My purpose is to explain what an Ingress resource is and how it works. Some of the ingress controllers use their custom resources, instead of the default Kubernetes Ingress resource. That way, they can support more features than the default Ingress resource. I would encourage you to explore the available options and pick the one that works best for you.

To deploy the Ambassador API gateway, we will start by deploying the **custom resource definitions (CRDs)** the gateway uses:

```
$ kubectl apply -f https://www.getambassador.io/yaml/ambassador/ambassador-crds.yaml
customresourcedefinition.apiextensions.k8s.io/authservices.getambassador.io created
customresourcedefinition.apiextensions.k8s.io/consulresolvers.getambassador.io created
customresourcedefinition.apiextensions.k8s.io/hosts.getambassador.io created
customresourcedefinition.apiextensions.k8s.io/kubernetesendpointresolvers.getambassado
r.io created
customresourcedefinition.apiextensions.k8s.io/kubernetesserviceresolvers.getambassador
.io created
customresourcedefinition.apiextensions.k8s.io/logservices.getambassador.io created
customresourcedefinition.apiextensions.k8s.io/mappings.getambassador.io created
customresourcedefinition.apiextensions.k8s.io/modules.getambassador.io created
customresourcedefinition.apiextensions.k8s.io/ratelimitservices.getambassador.io
created
customresourcedefinition.apiextensions.k8s.io/tcpmappings.getambassador.io created
customresourcedefinition.apiextensions.k8s.io/tlscontexts.getambassador.io created
customresourcedefinition.apiextensions.k8s.io/tracingservices.getambassador.io created
```

What is the difference between `create` and `apply`?

It's a difference between imperative management (`create`) and declarative management (`apply`).

Using the `create` command, you are telling Kubernetes which resources to create or delete. With `apply`, you are telling Kubernetes how you want your resources to look. You don't define operations to be taken as you would with `create` or `delete`. You are letting Kubernetes detect the operations for each object. Let's say you used the `create` command and create a deployment with image `image:123`. If you want to change the image in the deployment to `image:999` you won't be able to use the `create` command as the deployment already exists. You'd have to delete the deployment first, then create it again. Using the `apply` command, you don't need to delete the deployment. The apply command will 'apply' the desired changes to an existing resource (i.e., update the image name in our case). You can use both approaches in production. Using the declarative approach, Kubernetes determines the changes needed for each object. The object retains any configuration changes made with the declarative approach. If you're using the imperative approach, the changes made previously will be gone as you will replace it. On the other hand, the declarative approach can be harder to debug because the resulting object is not necessarily the same as in the file you applied.

The next step is to create the Ambassador deployment (`ambassador`) and other resources needed to run the API gateway:

```
$ kubectl apply -f https://www.getambassador.io/yaml/ambassador/ambassador-rbac.yaml
service/ambassador-admin created
clusterrole.rbac.authorization.k8s.io/ambassador created
serviceaccount/ambassador created
clusterrolebinding.rbac.authorization.k8s.io/ambassador created
deployment.apps/ambassador created
```

NOTE

RBAC stands for Role-Based Access Control, and it is a way of controlling access to resources based on the roles. For example, using RBAC, you can create roles called `admin` and `normaluser`, and then allow admin role access to everything and `normaluser` only access to certain namespaces or control if they can create or only view resources. You can read more about the RBAC in Using Role-Based Access Control (RBAC)

Let's see the resources we created when we deployed the Ambassador API gateway:

```
$ kubectl get deploy
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
ambassador    3/3     3            3           30m
dogpic-web    1/1     1            1           2d
hello-world   1/1     1            1           2d

$ kubectl get svc
NAME               TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
ambassador-admin   NodePort    10.107.45.225    <none>        8877:31524/TCP   30m
dogpic-service     ClusterIP   10.110.213.161   <none>        3000/TCP         48m
hello-world        ClusterIP   10.109.157.27    <none>        3000/TCP         48m
kubernetes         ClusterIP   10.96.0.1        <none>        443/TCP          66d
```

The default installation creates three Ambassador Pods and the `ambassador-admin` service.

We need to separately create a LoadBalancer service that will route traffic to the `ambassador` Pods.

ch3/ambassador-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: ambassador
spec:
  type: LoadBalancer
  externalTrafficPolicy: Local
  ports:
    - port: 80
      targetPort: 8080
  selector:
    service: ambassador
```

Create the load balancer service for Ambassador, by running `kubectl apply -f ambassador-service.yaml`.

If you list the services again, you will notice the `ambassador` service doesn't have an IP address in the EXTERNAL-IP column. This is because we are running a cluster locally. If we used a cloud-managed cluster, this would create an actual load balancer instance in our cloud account, and we would get a public/private IP address we could use to access the services.

```
$ kubectl get svc
NAME               TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
ambassador         LoadBalancer   10.109.103.63    <pending>     80:32004/TCP     10d
ambassador-admin   NodePort       10.107.45.225    <none>        8877:31524/TCP   10d
dogpic-service     ClusterIP      10.110.213.161   <none>        3000/TCP         10d
hello-world        ClusterIP      10.109.157.27    <none>        3000/TCP         10d
kubernetes         ClusterIP      10.96.0.1        <none>        443/TCP          77d
```

With Minikube, you can access the `NodePort` services using the combination of the cluster IP and the port number (e.g., 32004 or 31524). The command `minikube ip` gives you the clusters' IP address (`192.168.64.3` in this case). You could use that IP and the NodePort, for example, `32004` for the `ambassador` service, and access the service.

An even better approach is to use the `minikube service` command and have Minikube open the correct IP and port number. Try and run the following command:

```
$ minikube service ambassador
|-----------|------------|-------------|---------------------------|
| NAMESPACE |    NAME    | TARGET PORT |            URL            |
|-----------|------------|-------------|---------------------------|
| default   | ambassador |             | http://192.168.64.3:32004 |
|-----------|------------|-------------|---------------------------|
  Opening service default/ambassador in default browser...
```

The page won't render because we haven't created any Ingress rules yet. However, you can try and navigate to `http://192.168.64.3:32004/ambassador/v0/diag` to open the Ambassador diagnostics page.

You could open any other service that's of type `NodePort` using the same command.

However, we want to use the LoadBalancer service type and a completely different IP address, so we don't have to deal with the cluster IP or the node ports. You can use the `tunnel` command to create a route to all services deployed with the LoadBalancer type.

Since this command has to be running, open a separate terminal window and run `minikube tunnel`:

```
$ minikube tunnel
Status:
        machine: minikube
        pid: 50383
        route: 10.96.0.0/12 -> 192.168.64.3
        minikube: Running
        services: [ambassador]
    errors:
                minikube: no errors
                router: no errors
                loadbalancer emulator: no errors
```

WARNING Minikube `tunnel` command needs admin privileges and you might get prompted for a password.

The tunnel command creates a network route on your computer to the cluster's service CIDR (Classless Inter-Domain Routing). The `10.96.0.0/12` CIDR includes IPs starting from `10.96.0.0` to `10.111.255.255`. This network route uses the cluster's IP address (`192.168.64.3`) as a gateway. You can also get the Minikube clusters' IP address by running `minikube ip` command.

Let's list the services again, and this time the `ambassador` service will get an actual IP address that falls in the CIDR from the tunnel command:

```
$ kubectl get svc
NAME              TYPE           CLUSTER-IP       EXTERNAL-IP      PORT(S)
AGE
ambassador        LoadBalancer   10.102.244.196   10.102.244.196   80:30395/TCP
1h
ambassador-admin  NodePort       10.106.191.105   <none>           8877:32561/TCP
21h
dogpic-service    ClusterIP      10.104.72.244    <none>           3000/TCP
21h
hello-world       ClusterIP      10.108.178.113   <none>           3000/TCP
21h
kubernetes        ClusterIP      10.96.0.1        <none>           443/TCP
67d
```

Since we will be using the external IP address, let's store it in an environment variable, so we don't have to type it out each time:

```
$ export AMBASSADOR_LB=10.102.244.196
```

Now we can open the build-in Ambassador diagnostic web site by navigating to: `http://AMBASSADOR_LB/ambassador/v0/diag` (replace the AMBASSADOR_LB with the actual IP address).

Ambassador Diagnostic Overview

Ambassador version `1.5.0`
Hostname `ambassador-5d7684c8f7-zwxrp`
Cluster ID `51d7cd0f-c480-538a-be36-24054a568317`
Configuration from 2020-05-28 22:38:01.078650 — 42 minutes, 38 seconds ago
Envoy ready, last status report within the last second

Ambassador ID `default`
Ambassador namespace `default`

Current log level: info

Set Debug On | Set Debug Off

Ambassador configuration **issues found**

System	Specifics
✓ Error check passed	✓ No errors logged
✕ Mappings failed	✕ No Mappings are active
✕ TLS failed	✕ No TLSContexts are active

Ambassador Documentation

Getting Started	Using Ambassador
Ambassador Architecture	Configuration Reference
Running Ambassador	Single Sign-On with OAuth/OIDC
Statistics and Monitoring	Custom Filters
Troubleshooting Ambassador	Rate Limiting

Ambassador Resolvers In Use

Kind	Resolver
`KubernetesServiceResolver`	kubernetes-service

Ambassador Route Table

URL	Service	Weight
`http://localhost/ambassador/v0/check_ready`	`127.0.0.1:8877`	100.0%
`http://localhost/ambassador/v0/check_alive`	`127.0.0.1:8877`	100.0%
`http://localhost/ambassador/v0/`	`127.0.0.1:8877`	100.0%

Figure 21. Ambassador API Gateway Diagnostics

The Ambassador API gateway diagnostics page gives you an overview of the gateway. You could use this if you are running into any issues or if you need to debug something. Of course, you can also turn this diagnostics page off for any production scenarios.

Single service Ingress

Now that we have the ingress controller up and running, we can create an Ingress resource.

The simplest version of an Ingress resource is one without any rules. Ingress directs all traffic to the same backend service, regardless of the traffic origin.

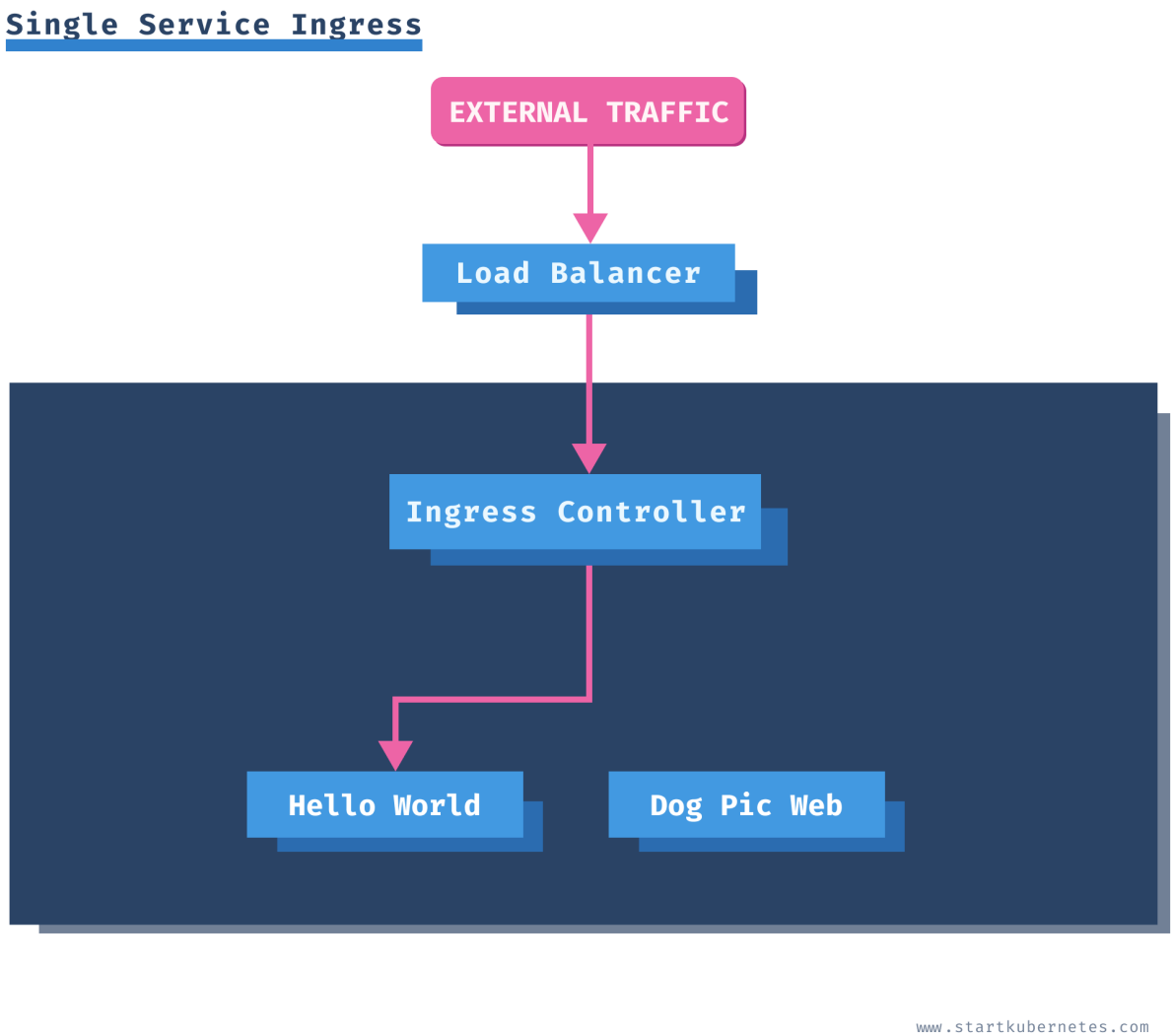


Figure 22. Single Service Ingress

Let's create an Ingress that only defines a backend service and doesn't have any rules.

ch3/single-service-ing.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: ambassador
  name: my-ingress
spec:
  defaultBackend:
    service:
      name: hello-world
      port:
        number: 3000
```

Save the YAML to `single-service-ing.yaml` and deploy the Ingress using `kubectl apply -f single-service-ing.yaml` command.

We used the following annotation `kubernetes.io/ingress/class: ambassador` in the above YAML. The Ambassador controller uses this annotation to claim the Ingress resource, and any traffic sent to the controller will be using the rules defined in the Ingress resource.

If you list the Ingress resources, you will see the created resource:

```
$ kubectl get ing
NAME         CLASS    HOSTS   ADDRESS   PORTS   AGE
my-ingress   <none>   *                 80      1h
```

The * in the HOSTS column means that there are no hosts defined. Later, when we define per-host rules, you will see those rules show up under the HOSTS column.

If you open the browser and navigate to the same IP as before (`http://AMBASSADOR_LB`), the Hello World website will show up.

Path based routing with Ingress

Since we want to expose two services through the Ingress, we need to write some rules. Using a path configuration, you can route traffic from one hostname to multiple services based on the URI path.

In this example, we want to route traffic from `http://AMBASSADOR_LB/hello` to the Hello World service and traffic from `http://AMBASSADOR_LB/dog` to Dog Pic Service.

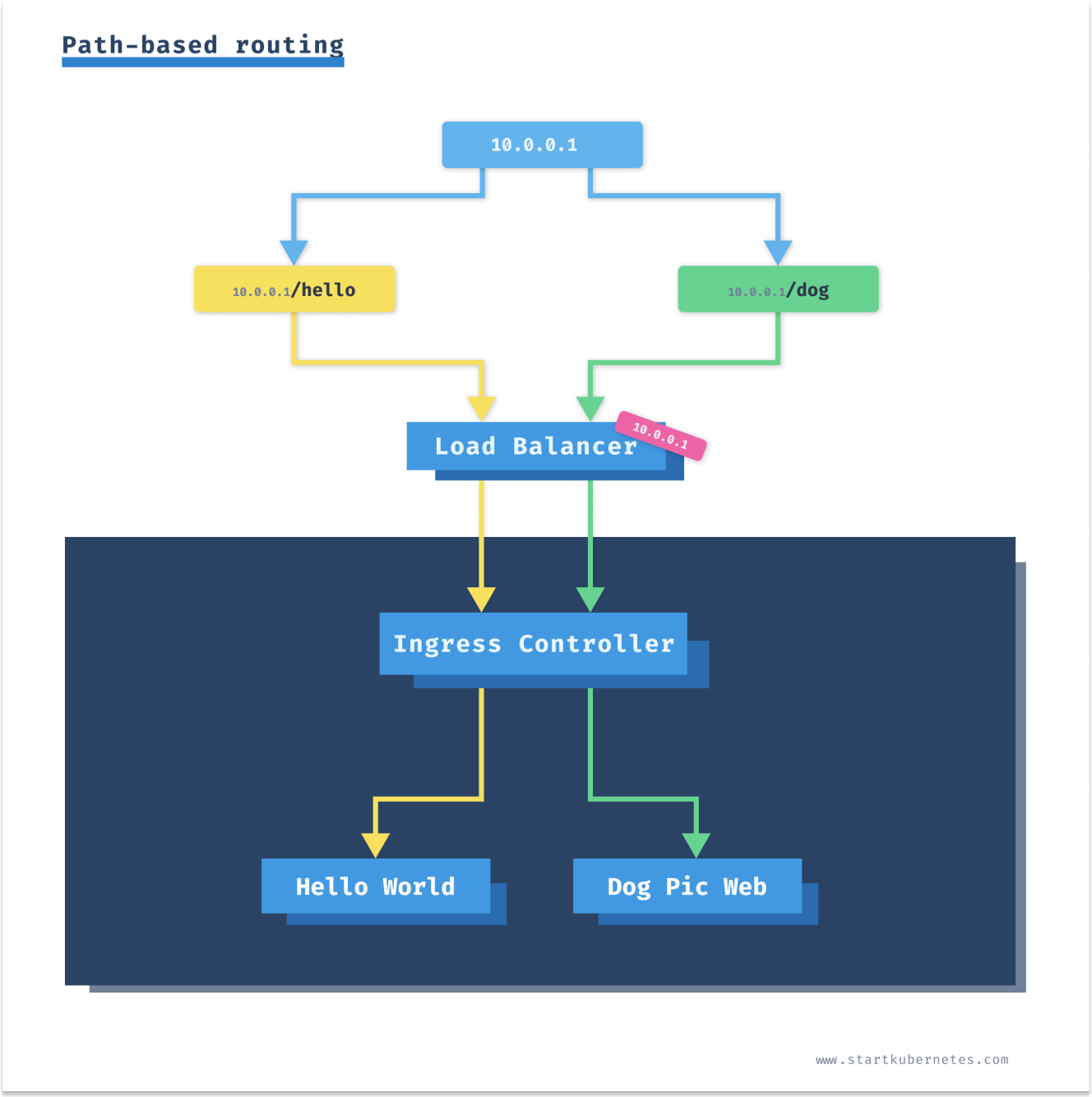


Figure 23. Path-based Routing with Ingress

To do that, we will define two rules in the Ingress resource:

ch3/path-ing.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: ambassador
  name: my-ingress
spec:
  rules:
    - http:
        paths:
          - path: /hello
            pathType: Prefix
            backend:
              service:
                name: hello-world
                port:
                  number: 3000
          - path: /dog
            pathType: Prefix
            backend:
              service:
                name: dogpic-service
                port:
                  number: 3000
```

Save the YAML to `path-ing.yaml` and create the ingress by running `kubectl apply -f path-ing.yaml`.

Let's look at the details of the created Ingress resource using the `describe` command:

```
$ kubect describe ing my-ingress
Name:             my-ingress
Namespace:        default
Address:
Default backend:  default-http-backend:80 (<error: endpoints "default-http-backend"
not found>)
Rules:
  Host        Path  Backends
  ----        ----  --------
  *
              /hello   hello-world:3000 (172.17.0.4:3000)
              /dog     dogpic-service:3000 (172.17.0.5:3000)
Annotations:  kubernetes.io/ingress.class: ambassador
Events:       <none>
```

Under the rules section, you will see the two paths we defined and the backends (service names).

If you navigate to `http://AMBASSADOR_LB/hello` the Hello World website will render, and if you

navigate to `http://AMBASSADOR_LB/dog` you will get the Dog Pic website.

Let's take this a step further. Wouldn't it be nice if we could type in `http://example.com/dog` instead of the IP address?

Using a hostname instead of an IP address

If we want to use a hostname, we will have to specify it in the Ingress resource, so the controller knows which hosts and where to direct the traffic.

ch3/hostname-ing.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: ambassador
  name: my-ingress
spec:
  rules:
    - host: example.com
      http:
        paths:
          - path: /hello
            pathType: Prefix
            backend:
              service:
                name: hello-world
                port:
                  number: 3000
          - path: /dog
            pathType: Prefix
            backend:
              service:
                name: dogpic-service
                port:
                  number: 3000
```

Save the above YAML to `hostname-ing.yaml` file and run `kubectl apply -f hostname-ing.yaml` to create the Ingress.

This time we defined a host name (`example.com`) and that will show up when you get the Ingress details:

```
$ kubectl describe ing my-ingress
Name:             my-ingress
Namespace:        default
Address:
Default backend:  default-http-backend:80 (<error: endpoints "default-http-backend"
not found>)
Rules:
  Host         Path  Backends
  ----         ----  --------
  example.com
               /hello   hello-world:3000 (172.17.0.4:3000)
               /dog     dogpic-service:3000 (172.17.0.5:3000)
Annotations:   kubernetes.io/ingress.class: ambassador
Events:        <none>
```

Notice how the Host column contains the actual host we defined.

If you try to navigate to the same Ambassador load balancer address as before (`http://AMBASSADOR_LB`), you will get an HTTP 404 error. This error is expected because we explicitly defined the host (`example.com`), but we haven't defined a default backend service - this is the service traffic gets routed to if none of the rules evaluate to true. We will see how to do that later on.

There are multiple ways you can access the IP address using a hostname.

The simplest way is to set a `Host` header when making a request from the terminal. For example:

```
$ curl -H "Host: example.com" http://$AMBASSADOR_LB/hello
<link rel="stylesheet" type="text/css" href="css/style.css" />

<div class="container">
    Hello World!
</div>
```

Setting the Host header works, but it would be much better if we could do the same through a browser.

I am using a browser extension called ModHeader[https://bewisse.com/modheader]. This extension allows you to set the same Host header in your browser.

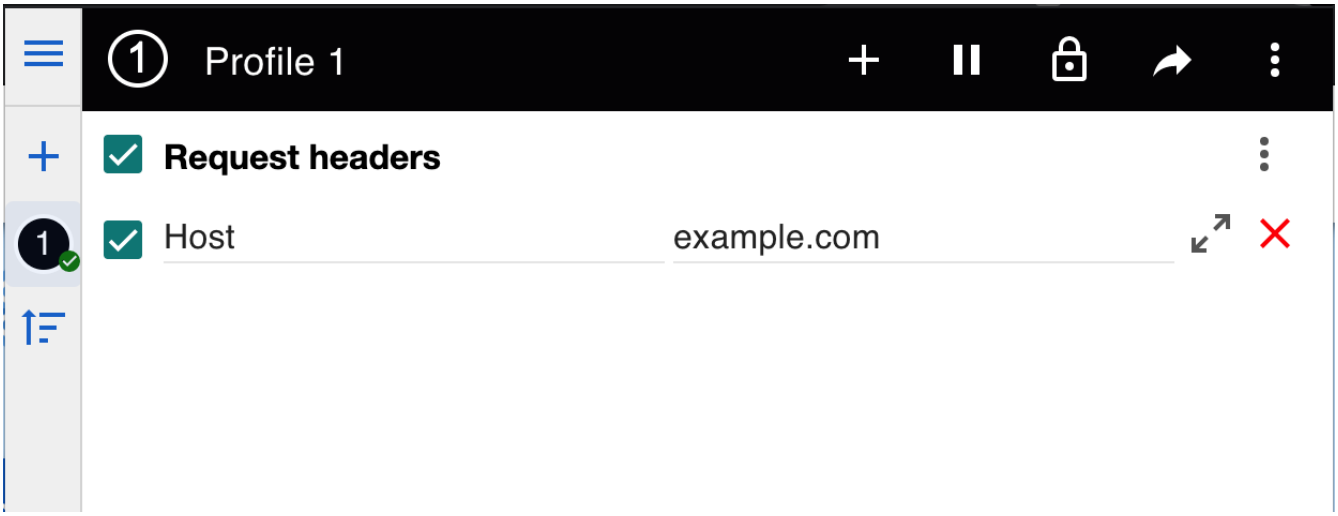


Figure 24. ModHeader Extension

If you navigate to `http://$AMBASSADOR_LB/hello` or `http://$AMBASSADOR_LB/dog` you will notice both web pages will load. This option works well, as you can load the page in the browser. However, it would be helpful to use the hostname e.g. `example.com/dog`, for example.

You can modify the `hosts` file on your computer that allows you to map hostnames to IP addresses. You can map the IP address ($AMBASSADOR_LB) to `example.com`.

Open the `/etc/hosts` file (or `%SystemRoot%\System32\drivers\etc\hosts` on Windows) and add the line mapping the hostname to an IP address. Make sure you use `sudo` or open the file as administrator on Windows.

```
$ sudo vim /etc/hosts
...
10.102.244.196 example.com
...
```

Save the file and if you navigate to `example.com/hello` or `example.com/dog` you will see both pages open. Make sure to uncheck/delete the header you have set with ModHeader.

Next, let's see how we can set a default backend that receives the traffic if Ingress controller can't match any of the rules.

Setting a default backend

In most cases, the default backend will be set by the Ingress controller. Some Ingress controllers automatically install a default backend service as well (NGINX, for example). Then, to configure the default backend, you can use either annotation or one of the custom resource definitions installed Ingress controller supports.

Since we don't want to dig into Ingress controllers' specifics, we will set the default backend directly in the Ingress resource. Ideally, you would be using your Ingress controller configuration and set the default backend there. To be completely honest, you might even just use the custom resources each Ingress controller supports, instead of the vanilla Kubernetes Ingress resource.

For this example we will set the default backend to the `hello-world` service. Here's the updated Ingress resource, with modified lines highlighted:

ch3/default-backend-ing.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: ambassador
  name: my-ingress
spec:
  defaultBackend:
    service:
      name: hello-world
      port:
        number: 3000
  rules:
    - host: example.com
      http:
        paths:
          - path: /hello
            pathType: Prefix
            backend:
              service:
                name: hello-world
                port:
                  number: 3000
          - path: /dog
            pathType: Prefix
            backend:
              service:
                name: dogpic-service
                port:
                  number: 3000
```

Save the above YAML to `default-backend-ing.yaml` and update the Ingress with `kubectl apply -f default-backend-ing.yaml`. If you describe the Ingress resource using the `describe` command, you will get a nice view of all rules and the default backend that we just set:

```
$ kubectl describe ing my-ingress
Name:             my-ingress
Namespace:        default
Address:
Default backend:  hello-world:3000 (172.17.0.8:3000)
Rules:
  Host         Path  Backends
  ----         ----  --------
  example.com
               /hello   hello-world:3000 (172.17.0.8:3000)
               /dog     dogpic-service:3000 (172.17.0.9:3000)
Annotations:   kubernetes.io/ingress.class: ambassador
Events:        <none>
```

If you open `http://example.com` you will notice that this time the Hello World web page will load. The `/hello` and `/dog` endpoints will still work the same way.

Name-based Ingress

Sometimes you don't want to use the fanout option with paths; instead, you want to route the traffic based on the subdomains. For example, routing `example.com` to one service, `dogs.example.com` to another, etc. For this example, we will try to set up the following rules:

Host name	**Kubernetes service**
example.com	hello-world:3000
dog.example	dogpic-service:3000

To create the above rules, we need to add two `host` entries under the Ingress resource's rules section. We define the paths and the backend service and port name we wnat to route the traffic to under each host entry.

ch3/name-ing.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: ambassador
  name: my-ingress
spec:
  rules:
    - host: example.com
      http:
        paths:
          - path: /hello
            pathType: Prefix
            backend:
              service:
                name: hello-world
                port:
                  number: 3000
    - host: dog.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: dogpic-service
                port:
                  number: 3000
```

Save the above YAML to `name-ing.yaml` and deploy it using `kubectl apply -f name-ing.yaml`.

Before we can try this out, we need to add the `dog.example.com` to the `hosts` file just like we did with the `example.com`. Open the `/etc/hosts` file (or `%SystemRoot%\System32\drivers\etc\hosts` on Windows) and add the line mapping the `dog.example.com` hostname to the IP address. Make sure you use `sudo` or open the file/terminal as an administrator on Windows.

```
$ sudo vim /etc/hosts
...
10.102.244.196 example.com
10.102.244.196 dog.example.com
...
```

NOTE

When using a real domain name, the entries we added to the `hosts` file would correspond to the DNS records at your domains registrar. With an A record, you can map a name (`example.com`) to a stable IP address. For `example.com`, you would create an A record that points to the external IP address. Another commonly used record is the CNAME record. You would use CNAME to map one name to another name. For example, to map `dog.mydomain.com` to `dog.example.com`, while `dog.example.com` uses an A record and maps to an IP. In the end, the `dog.mydomain.com` would resolve to the IP address, same as `dog.example.com`.

Save the file, open the browser, and navigate to `http://example.com`. You should see the response from the Hello World service as shown below.

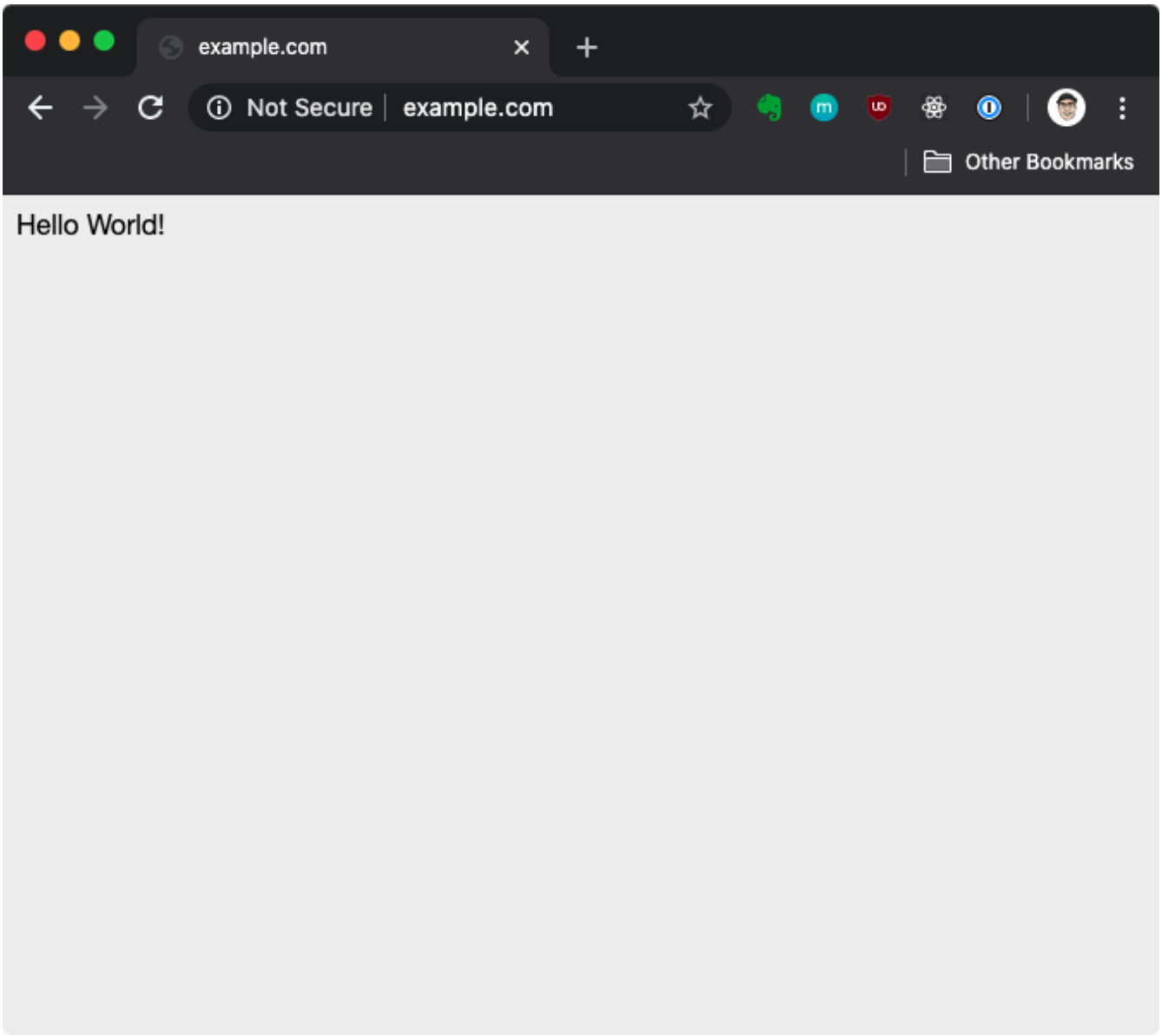


Figure 25. Hello World Website

Similarly, if you enter `http://dog.example.com` you will get the Dog Pic website.

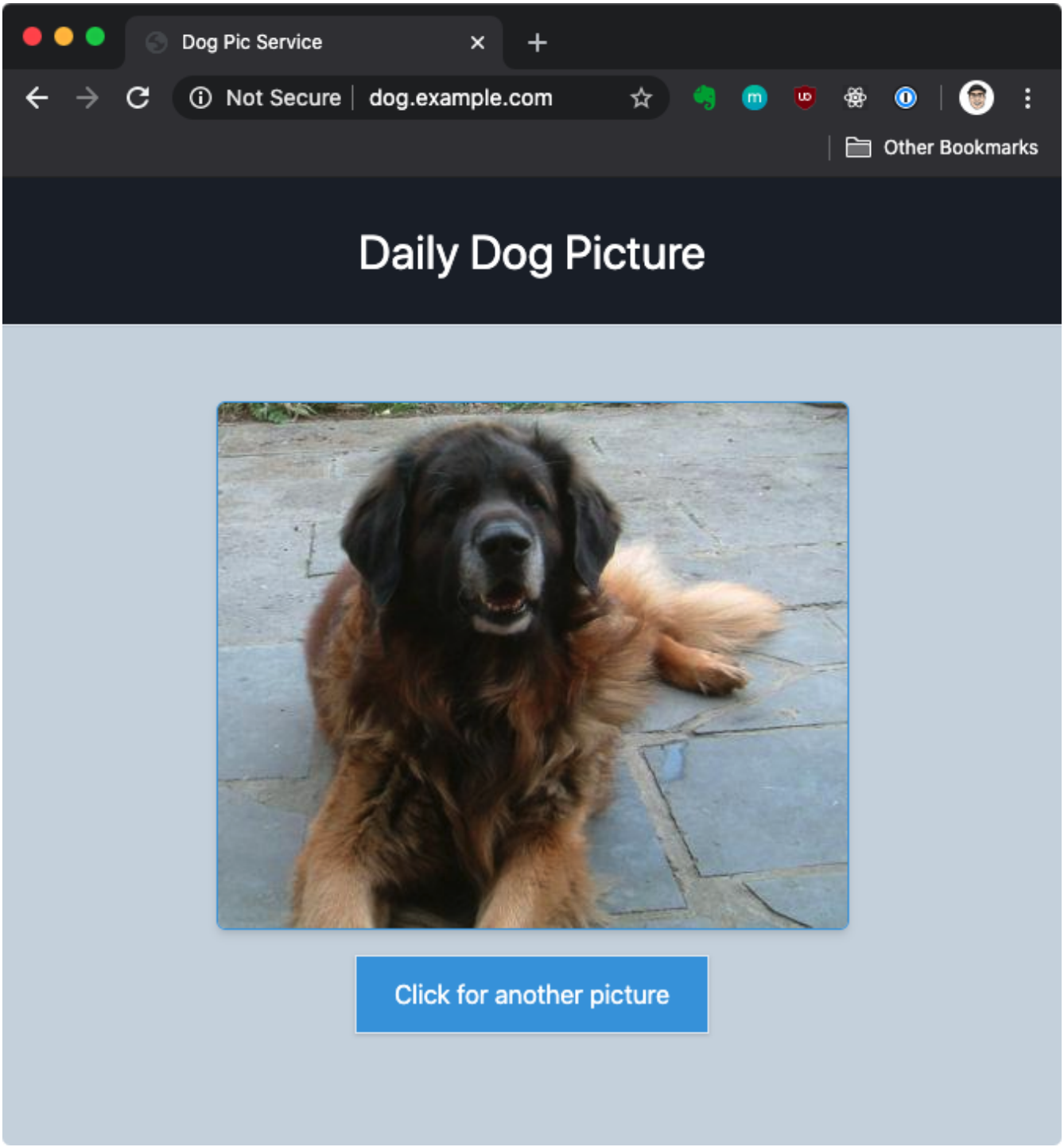


Figure 26. Dog Pic Service

Cleanup

You can delete the Service, Deployments, and Ingress using the `kubectl delete` command. For example, to delete the `dogpic-web` deployment, run:

```
$ kubectl delete deploy dogpic-web
deployment.apps "dogpic-web" deleted
```

To delete a different resource, replace the resource name (`deploy` in the above example) with `ingress` or `service`.

Another way of deleting the resources is to provide the YAML file you used to create the resource. For example, if you created the `dogpic-web` from a file called `dogpic.yaml` you can delete it like this:

```
$ kubectl delete -f dogpic.yaml
```

If you get completely stuck and can't delete something or delete too much (everyone has done that at some point), you can always just reset your cluster. If you're using Minikube, you can run `minikube delete` to delete the cluster and afterward run `minikube start` to get a fresh cluster. Similarly, you can reset the Kubernetes cluster from the Preferences menu when using Docker Desktop.

Other Ingress controller responsibilities

The job of an ingress controller or an ingress gateway is to proxy or "negotiate" the communication between the client and the server. The client is anyone making requests, and the server is the Kubernetes cluster or instead services running inside the cluster.

In addition to routing the incoming requests or exposing service APIs through a single endpoint, the ingress gateways do other tasks, such as rate-limiting, SSL termination, load balancing, authentication, circuit breaking, and more.

Organizing applications with namespaces

Up until now, we haven't talked about namespaces. Namespaces in Kubernetes provide a way to scope and group different Kubernetes resources. Each namespace can contain multiple resources, however, a single resource can only be in one namespace. The resource names need to be unique within a namespace, but not across the namespaces.

For example, you can only have one service called `customers` inside a namespace called `production`. However, you could create a `customers` service inside a namespace called `testing`. In thise case, the full name of the Service would be `customers.production` and `customers.testing`.

You will create most of the Kubernetes resources inside a namespace. However some resources are not in a namespace or are not namespaced. The example of a non-namespaced resource is the namespace itself because namespaces cannot be nested. Other examples of non-namespaced resources are Nodes, PersistentVolumes, CustomResourceDefinitions, and others.

TIP You can get the full list of non-namespaced resources in your cluster by running `kubectl api-resources --namespaced=false`. If you switch the flag value to `true`, you can list all namespaced resources.

Typically you would use multiple namespaces in clusters with many users spread across different projects or teams. You can deploy most of the off-the-shelf Kubernetes application using Helm package manager into separate namespaces. Later in the book, when we talk about resource quotas, we will show an example of how to divide the cluster resources between multiple users using namespaces.

For the namespaced resource, you specify the namespace in the metadata section of the resource:

```
...
metadata:
  name: hello
  namespace: mynamespace
...
```

If you don't provide the namespace, Kubernetes creates the resources in the default namespace. The default namespace is called `default`, however, that can be changed per Kubernetes context. For example, if you're working with multiple clusters, you might want to set different default namespace for your clusters.

Let's consider the following output:

```
$ kubectl config get-contexts
CURRENT   NAME                 CLUSTER          AUTHINFO
NAMESPACE
          docker-for-desktop   docker-desktop   docker-desktop
          kind-kind            kind-kind        kind-kind
          minikube             minikube         minikube
*         peterjk8s            peterjk8s        clusterUser_mykubecluster_peterjk8s
```

Notice the namespace column is empty because I haven't explicitly set namespaces for my contexts. If I create a resource without specifying a namespace, it will end up in the `default` namespace.

You can create namespaces the same way you create other Kubernetes resources. You can either define the namespace using YAML or use `kubectl`.

Let's create a namespace called `testing` using `kubectl`:

```
$ kubectl create namespace testing
namespace/testing created

$ kubectl get namespace
NAME              STATUS   AGE
default           Active   32d
kube-node-lease   Active   32d
kube-public       Active   32d
kube-system       Active   32d
testing           Active   3s
```

TIP Short name for `namespace` is `ns`, so you can save typing seven characters!

The YAML representation of a namespace is straightforward, compared to some of the other resources. You can get the YAML representation of any Kubernetes resource using the `--output yaml` flag when running the `get` command:

```
$ kubect get ns testing --output yaml
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: "2020-06-23T21:51:35Z"
  name: testing
  resourceVersion: "3750772"
  selfLink: /api/v1/namespaces/testing
  uid: 266e95ec-7de4-429c-a945-83d91a2a2296
spec:
  finalizers:
  - kubernetes
status:
  phase: Active
```

The above output contains the field values such as `creationTimestamp`, `resourceVersion`, `selfLink`, and others that Kubernetes adds when it creates the resource.

A cleaner way of getting the YAML representation of a resource is to add the `--dry-run` flag when creating the resource. Any command you execute with the `--dry-run` flag will not be persisted and won't have any side-effects. That means nothing will get created, deleted, or modified. However, this allows you to see how the resource would look like processed and persisted.

Let's combine the dry run flag and the output flag and try again. Note that I am using the short name for the output flag, `-o`:

```
$ kubectl create ns testing --dry-run=client -o yaml
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: null
  name: testing
spec: {}
status: {}
```

If we remove the empty and null fields, the YAML representation of the Namespace resource looks like this:

```
apiVersion: v1
kind: Namespace
metadata:
  name: testing
```

You can also specify a namespace with each `kubectl` command. For example, if you want to get all Pods inside the `kube-system` namespace, you can do that using the `--namespace` or `-n` flag:

```
$ kubectl get pods -n kube-system
NAME                               READY   STATUS    RESTARTS   AGE
coredns-66bff467f8-glmzm           1/1     Running   0          8d
coredns-66bff467f8-msvgx           1/1     Running   0          8d
etcd-minikube                      1/1     Running   0          8d
kube-apiserver-minikube            1/1     Running   0          8d
kube-controller-manager-minikube   1/1     Running   1          8d
kube-proxy-wqn8f                   1/1     Running   0          8d
kube-scheduler-minikube            1/1     Running   0          8d
storage-provisioner                1/1     Running   0          8d
```

If you don't explicitly provide a namespace, Kubernetes uses the `default` namespace. Similarly, you can use the flag called `--all-namespaces` or `-A` to list resources across all namespaces. For example, to list all Services in your cluster, regardless of the namespace, you would run:

```
$ kubectl get svc -A
NAMESPACE     NAME          TYPE          CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
default       kubernetes    ClusterIP     10.96.0.1       <none>        443/TCP
8d
default       web-frontend  LoadBalancer  10.106.30.168   <pending>     80:30962/TCP
8d
kube-system   kube-dns      ClusterIP     10.96.0.10      <none>
53/UDP,53/TCP,9153/TCP   8d
```

As mentioned in section Using a Kubernetes Service a namespace plays a role when resolving Services. Let's consider the following listing of services inside a cluster:

```
$ kubectl get svc -A
NAMESPACE     NAME          TYPE          CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
production    web-frontend  ClusterIP     10.96.0.1       <none>        80:30642/TCP
8d
testing       web-frontend  ClusterIP     10.96.0.2       <none>        80:30962/TCP
8d
```

We have two Services called `web-frontend`. One Service is in the `production` namespace and the other in the `testing` namespace.

To correctly reference one or the other, you need to use the full name of the service. For example `web-frontend.production.svc.cluster.local` for the service in the `production` namespace. If your application that lives inside the `production` namespace sends a request to `web-frontend`, that automatically resolves to the `web-frontend` service inside the `production` namespace. However, if you want to reach the `web-frontend` service from the `testing` namespace, you will have to use a fully qualified name, which is `web-frontend.testing.svc.cluster.local`. It is a good practice always to use fully qualified names, so there's no confusion on which service you are calling.

Jobs and CronJobs

The type of workloads we were deploying previously were all long-running applications - things like websites and services that keep on running continuously. If something went wrong, they get rescheduled and start running again.

The other type of workloads you often need to run are workloads that perform a particular task, and once the task is completed they stop running. An example of a workload like that would be doing a backup or generating daily reports. It does not make sense to keep the reporting workload running. It only needs to run when it's generating the report. Once the task generates the report it can go away. If the task fails, you can configured it to restart automatically.

Kubernetes features a resource called Job you can use to run such workloads. The Job resource can create one or more Pods and track the number of successful completions. The Job resource ensures that Pods are run to completion. You could achieve a similar behavior only with Pods, but then you'd have to be the manage the Pods' lifecycle in case it fails or gets rescheduled.

Let's run a Job that doesn't nothing but sleep for a minute:

ch3/sleep-job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: sleep-on-the-job
spec:
  template:
    metadata:
      labels:
        app.kubernetes.io/name: sleep-on-the-job
    spec:
      restartPolicy: Never
      containers:
        - name: sleep-container
          image: busybox
          args:
            - sleep
            - "60"
```

Save the above YAML in `sleep-job.yaml` and run `kubectl apply -f sleep-job.yaml` to create the Job.

NOTE

We are explicitly setting the `restartPolicy` to Never. The default value for `restartPolicy` is Always, however, the Job resource does not support that restart policy. The two supported values are Never and OnFailure. Setting one of these values prevents the container from being restarted when it finishes running.

You can list the Job the same way as any other resource. Notice how the output also shows the number of completions of the Job and duration of the Job.

```
$ kubectl get job
NAME              COMPLETIONS   DURATION   AGE
sleep-on-the-job   0/1           4s         4s
```

If you describe the Job you will notice in the Events section that controller creates a Pod to run the Job:

```
$ kubectl describe job sleep-on-the-job
...
Events:
  Type    Reason            Age   From            Message
  ----    ------            ----  ----            -------
  Normal  SuccessfulCreate  18s   job-controller  Created pod: sleep-on-the-job-f9ht8
...
```

Let's look at the Pod that was created by this Job. We will use the `--labels` flag to get all Pods with the label `app.kubernetes.io/name=sleep-on-the-job`:

```
$ kubectl get pods -l=app.kubernetes.io/name=sleep-on-the-job
NAME                     READY   STATUS    RESTARTS   AGE
sleep-on-the-job-f9ht8   1/1     Running   0          48s
```

After a minute, the Pod will stop running, however, it won't be deleted. If you run the same command as above, you will notice that the Pod is still around. Kubernetes keeps the Pod around so you can look at the logs, for example.

```
$ kubectl get pods -l=app.kubernetes.io/name=sleep-on-the-job
NAME                     READY   STATUS      RESTARTS   AGE
sleep-on-the-job-f9ht8   0/1     Completed   0          28m
```

Similarly happens with the Job resource. The resource stays around until you explicitly delete it. Deleting the Job also deletes the Pod. Notice how the number of completions now shows `1/1`, which means that the Job was completed successfully one time.

```
$ kubectl get job
NAME              COMPLETIONS   DURATION   AGE
sleep-on-the-job   1/1           63s        12m
```

If you're wondering if a job can be completed and executed multiple times, the answer is yes, it can. The Job can track the number of successful completions, and you can use that number as to control when the Job completes.

By default, Kubernetes sets the number of completions to 1, and you can change that by setting a different value to the `completions` field. Let's create a new Job called `three-sleeps-on-the-job` where

we set the completions to `3`. Setting it to `3` causes the Job's Pod to run three times sequentially:

ch3/three-sleeps.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: three-sleeps-on-the-job
spec:
  completions: 3
  template:
    metadata:
      labels:
        app.kubernetes.io/name: three-sleeps-on-the-job
    spec:
      restartPolicy: Never
      containers:
        - name: sleep-container
          image: busybox
          args:
            - sleep
            - "60"
```

Save the above YAML in `three-sleeps.yaml` and deploy it with `kubectl apply -f three-sleeps.yaml`.

If you look at Jobs now, you will notice the `COMPLETIONS` column for the latest Job shows `0/3`:

```
$ k get job
NAME                     COMPLETIONS   DURATION   AGE
sleep-on-the-job         1/1           63s        40m
three-sleeps-on-the-job  0/3           10s        10s
```

As soon as the first job finishes (60 seconds later), the column will be updated and will show `1/3` in the `COMPLETIONS` column. The Job executes Pods one after another, and Kubernetes marks the Job completed when there are three successful completions (i.e., three pods ran without any failures).

If you need to execute Pods in parallel, you can use the `parallelism` setting. The parallelism setting defines how many Pods you can run in parallel.

Let's take the previous example and set the `parallelism` value to two:

ch3/three-sleeps-parallel.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: three-sleeps-on-the-job-parallelism
spec:
  completions: 3
  parallelism: 2
  template:
    metadata:
      labels:
        app.kubernetes.io/name: three-sleeps-on-the-job-parallelism
    spec:
      restartPolicy: Never
      containers:
        - name: sleep-container
          image: busybox
          args:
            - sleep
            - "60"
```

Save the above YAML in `three-sleeps-parallel.yaml` and deploy it with `kubectl apply -f three-sleeps-parallel.yaml`.

If you list the Pods now, you will notice two Pods running at the same time:

```
$ kubectl get pods
NAME                                        READY   STATUS      RESTARTS   AGE
three-sleeps-on-the-job-parallelism-b8ckz   1/1     Running     0          8s
three-sleeps-on-the-job-parallelism-zcrzq   1/1     Running     0          8s
```

What happens if the Pods keep failing? Well, the Job will keep creating them and retrying based on the `backoffLimit` setting. The back-off limit is a setting on the spec that specifies the number of retries before Kubernetes considers the Job failed. Kubernetes sets the default value to `6` and re-creates them with an exponential back-off delay. The back-off delay means if the first Pod fails, the controller will wait for 10 seconds before recreating it. Then if it fails again, it will wait for 20 seconds and so on up until a total of six minutes of delay. Kubernetes resets the back-off delay either when you delete the Pod or when the Pod completes successfully.

In addition to the `backoffLimit`, you can also terminate a job using the `activeDeadlineSeconds`. The `activeDeadlineSeconds` represents how long a Job can run, regardless of how many Pods it creates. If we consider the previous example and set the `activeDeadlineSeconds` to 10, the Job will fail after 10 seconds. Kubernetes will terminate the Pods and set the Jobs' status to `DeadlineExceeded`.

Let’s look at this using an example:

ch3/failing-job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: failing-job
spec:
  completions: 3
  activeDeadlineSeconds: 20
  template:
    metadata:
      labels:
        app.kubernetes.io/name: failing-job
    spec:
      restartPolicy: Never
      containers:
        - name: sleep-container
          image: busybox
          args:
            - sleep
            - "60"
```

Save the above YAML in `failing-job.yaml` and deploy it with `kubectl apply -f failing-job.yaml`.

The Job will create a Pod, but after 20 seconds, the Pod will get terminated, and the Job will fail with the `DeadlineExceeded` reason:

```
$ kubectl describe job failing-job
...
Events:
  Type     Reason            Age   From            Message
  ----     ------            ----  ----            -------
  Normal   SuccessfulCreate  86s   job-controller  Created pod: failing-job-rq9ht
  Normal   SuccessfulDelete  66s   job-controller  Deleted pod: failing-job-rq9ht
  Warning  DeadlineExceeded  66s   job-controller  Job was active longer than
specified deadline
```

If you ran all these examples, you have probably noticed that Kubernetes does not automatically remove the Jobs and completed Pods. You can clean up the completed (or failed) Jobs by setting the `ttlSecondsAfterFinished` value. After a Job completes (or fails) the controller waits for the duration specified in the `ttlSecondsAfterFinished` field and then deletes the Job and Pods.

ch3/delete-job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: delete-job
spec:
  ttlSecondsAfterFinished: 30
  template:
    metadata:
      labels:
        app.kubernetes.io/name: delete-job
    spec:
      restartPolicy: Never
      containers:
        - name: sleep-container
          image: busybox
          args:
            - sleep
            - "10"
```

Save the above YAML in `delete-job.yaml` and deploy it with `kubectl apply -f delete-job.yaml`.

The above Job will complete in 10 seconds. After that, the controller will wait for an extra 30 seconds before deleting the Job and the Pod. If you want to delete the Job right after it finishes, you can set the `ttlSecondsAfterFinished` to 0.

Note that this feature is currently in alpha state. To use it, you need to enable alpha features in your cluster.

CronJobs

A CronJob is a type of a Job you can run at specified times and dates. The regular Jobs start running when you create them. There are essential settings that allow you to control how Kubernetes runs the Job. However, you cannot schedule the Job to run at a particular times or intervals. Running Jobs at particular times or intervals is what the CronJob allows you to do.

The CronJob resource uses a well-known cron format, made out of five fields that represent the time to execute the command.

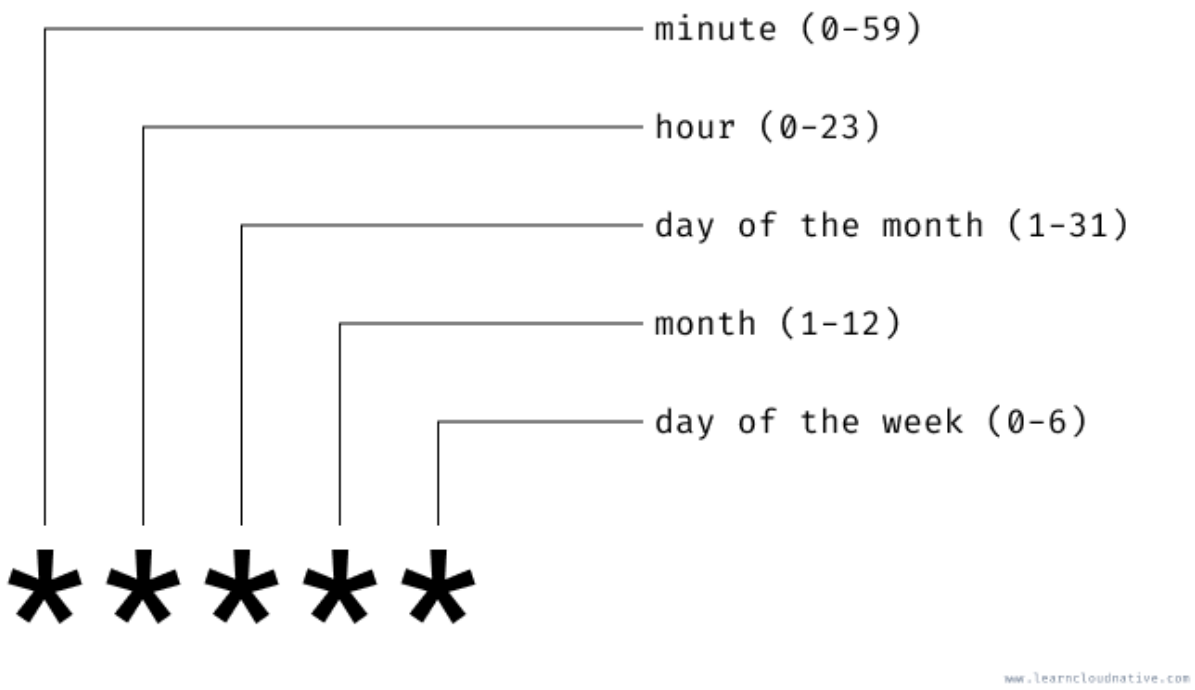


Figure 27. cron format

The cron format is out of scope for this book, but here are a couple of examples:

Example	Description
`* * * * *`	Runs every minute, every hour, day, month and day of the week)
`*/10 * * * *`	Runs every 10 minutes
`00 9,21 * * *`	Runs at 9 AM and 9 PM, every day, month, and day of the week
`00 7-17 * * *`	Run at the top of every hour, from 7 AM and 5 PM, every day, month, and day of the week
`00 7-17 * * 1-5`	Run at the top of every hour, from 7 AM and 5 PM, every day, month, but only during weekdays.
`0,15,30,45 * * * *`	Runs every 15 minutes of every hour, day, month, and day of the week

Let's create a CronJob that runs every minute, so that we can see it in action. You can set the cron format with the `schedule` field:

ch3/minute-cron.yaml

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: minute-cron
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        metadata:
          labels:
            app.kubernetes.io/name: minute-cron
        spec:
          restartPolicy: Never
          containers:
            - name: sleep-container
              image: busybox
              args:
                - sleep
                - "10"
```

Save the above YAML in `minute-cron.yaml` and deploy it with `kubectl apply -f minute-cron.yaml`.

Let's look at the CronJob by running `kubectl get cronjob` or using the short name `kubectl get cj`:

```
$ kubectl get cj
NAME          SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
minute-cron   */1 * * * *   False     0        <none>          3s
```

If you wait for a couple of minutes, you will see the CronJob automatically create the Pods every minute.

To suspend the CronJob you can edit the resource (e.g. `kubectl edit minute-cron`) and change the `suspend` field from `false` to `true`.

In addition to the CronJob schedule, you can also configure how CronJob deals with concurrent executions. Let's say you configured the CronJob to run every 5 minutes. The Pod starts, and for some reason, it runs for more than 5 minutes. What should the CronJob controller do in this case? Does it create the second Pod as per schedule or do nothing, since the previous Job is still executing?

You can control this behavior using the `concurrencyPolicy` setting. This setting has three possible values - `Forbid`, `Allow`, and `Replace`. The default value is `Allow`, and if the previous iteration hasn't completed when the new one is supposed to start, the allow setting will allow the second instance to run.

Setting the value to `Forbid` will do the opposite - the Pod that was supposed to start per schedule will

not start.

Finally, the `Replace` will stop the currently running Job and start a new one.

Another configuration value we need to mention here is the `startingDeadlineSeconds`. You would set this in cases where you don't want to start the job over the scheduled time. If the Job doesn't start at the scheduled time + the deadline, it will be marked as Failed.

Consider the following example:

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: ten-minute-cron
spec:
  schedule: "*/10 * * * *"
  startingDeadlineSeconds: 30
  ...
```

The Job needs to run every ten minutes. Let's say the first Job is supposed to start at 2 PM (2 hours, 0 minutes, and 0 seconds). The `startingDeadlineSeconds` says that if the Job doesn't begin by 2:00:30 the controller should mark it as failed.

Configuration

One of the factors from the Twelve-Factor App manifesto talks about application configuration, specifically about storing configuration in the environment, instead of hardcoding it inside your services.

In most cases, the application configuration will vary between different deployment environments. For example, when you're running your application in a testing or staging environment, it is highly likely that you want your application to use databases or other backing services from the same environment. You don't want the application in a testing environment to talk to your production database - that would be bad.

Another reason for separating code from your configuration is that you don't want to rebuild your application each time you change a configuration value. That would be a massive waste of time and resources. Instead, you should always deploy the same application or binary, but augment its behavior using a configuration specific to the environment application is running in.

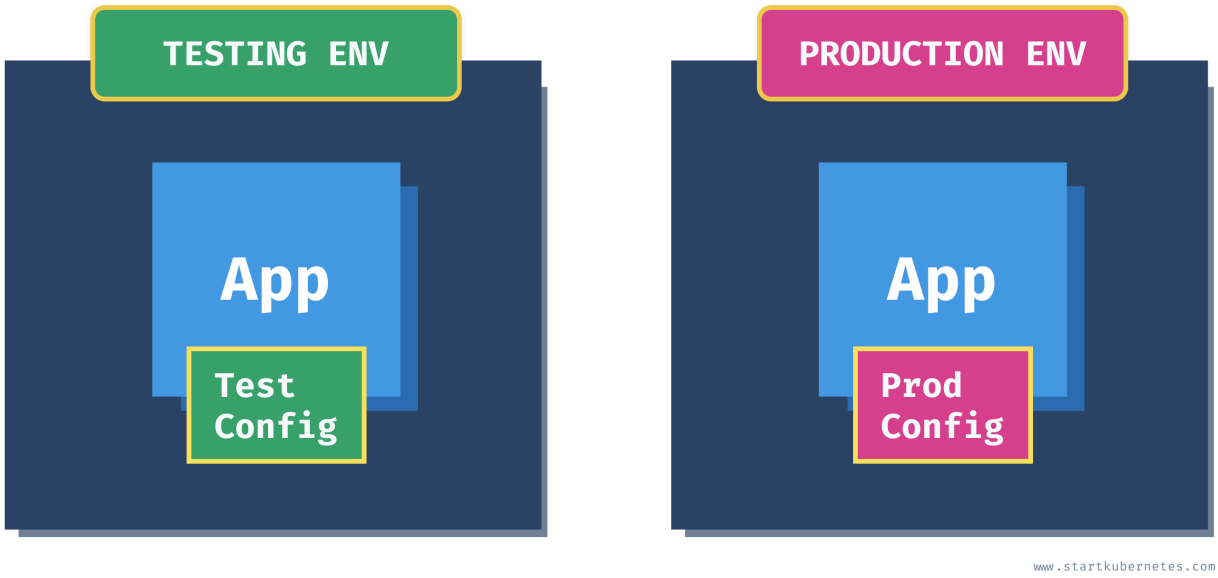


Figure 28. Configuration per Environment

Typically configuration is anything that your application needs to be able to start and run. Here are a couple of standard configuration settings:

- Connection strings to databases, queues, or other connection strings
- Credentials (any usernames, passwords, keys, certificates)
- Port numbers, dependent service names, and addresses

We can separate these configuration settings into two groups at a high-level: **non-sensitive** configuration values and **sensitive** configuration values.

The former contains anything that you don't consider to be sensitive information - things like port numbers, service names, or perhaps even connection strings, assuming they don't contain any usernames or passwords. The latter group includes sensitive information or secrets. Things like

passwords, API keys, application secrets, credentials, certificates, etc. Pretty much anything that can severely compromise your application or your system if this information is leaked.

Configuring application through arguments

The simplest way to configure the application running inside Kubernetes is to set the command and pass arguments to it - we have already done that in the previous chapter, where we talked about Pods. Here's an example we used:

ch4/hello-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-pod
  labels:
    app.kubernetes.io/name: hello
spec:
  containers:
    - name: hello-container
      image: busybox
      command: ["sh", "-c", "echo Hello from my container!"]
```

With the above YAML, we are invoking `sh` and then echoing the "Hello from my container!" message. Another way you can do this is to specify "echo" as the command and pass-in the message as an argument, just like in the YAML below:

ch4/hello-pod-args.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-pod
  labels:
    app.kubernetes.io/name: hello
spec:
  containers:
    - name: hello-container
      image: busybox
      command: ["echo"]
      args: ["Hello from my container!"]
```

In some cases (depending on how you defined your Dockerfile), you could even omit the command and specify the arguments. Here's a snippet from the Traefik deployment YAML and how the arguments are passed to the Traefik Docker image:

```
...
    spec:
      containers:
        - name: router
          image: traefik:v2.3
          args:
            - --entrypoints.web.Address=:8000
            - --entrypoints.websecure.Address=:4443
            - --providers.kubernetescrd
            - --accesslog=true
            - --accesslog.filepath=/var/log/traefik.log
            - --certificatesresolvers.myresolver.acme.tlschallenge
...
```

In the above case, the Dockerfile for the `traefik` image is using `ENTRYPOINT` command to point to the `traefik` binary inside the container. Therefore, you don't need to set the command name explicitly. Here's a snippet from the Traefiks' Dockerfile:

```
...
EXPOSE 80
VOLUME ["/tmp"]
ENTRYPOINT ["/traefik"]
```

Let's quickly look at what the difference between the CMD and ENTRYPOINT is.

Difference between CMD and ENTRYPOINT in Dockerfiles

With the `ENTRYPOINT` instruction, you can specify a command that Docker executes when the container starts. On the other hand, with `CMD` you can specify arguments that get passed to the `ENTRYPOINT` instruction. Let's consider the following Dockerfile:

ch4/Dockerfile

```
FROM alpine
RUN apk add curl

ENTRYPOINT ["/usr/bin/curl"]
CMD ["google.com"]
```

If you build this image (let's call it `curlalpine`) and then run it without any arguments, you will get back the response from google.com:

```
$ docker run -it curlalpine
<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="http://www.google.com/">here</A>.
</BODY></HTML>
```

If you run the same image with an argument `example.com`, the container will curl to the `example.com`:

```
$ docker run -it curlalpine example.com
<!doctype html>
<html>
<head>
    <title>Example Domain</title>
...
```

Now let's say we modified the Dockerfile, removed the ENTRYPOINT and invoked the `curl google.com` within the CMD instruction:

ch4/Dockerfile.cmd

```
FROM alpine
RUN apk add curl

CMD ["/usr/bin/curl", "google.com"]
```

Running the container without any arguments will invoke "curl google.com", just like before. However, if you pass in an argument (`example.com`, like we did before), you will get an error:

```
$ docker run -it curlcmd example.com
docker: Error response from daemon: OCI runtime create failed: container_linux.go:349:
starting container process caused "exec: \"example.com\": executable file not found in
$PATH": unknown.
ERRO[0000] error waiting for container: context canceled
```

The container is trying to run the argument, which in our case, doesn't make any sense (unless you have a binary called `example.com` in the container). If you replace `example.com` with a command that exists inside the container (let's say `ls`), the container will execute that command:

```
$ docker run -it curlcmd ls
bin    etc    lib    mnt    proc   run    srv    tmp    var
dev    home   media  opt    root   sbin   sys    usr
```

In addition to passing arguments to containers, Kubernetes also has dedicated resources to store

the configuration. A resource for storing configuration settings is called a ConfigMap, and the resource for storing secret configuration values is named Secret. Let's look at the ConfigMaps first.

Creating and using ConfigMaps

A ConfigMap resource can store configuration values (key-value pairs). These values can be consumed or 'mounted' to containers inside a Pod either as environment variables or files. Here's a YAML representation of a ConfigMap that stores single values ("8080", "Ricky") as well as values that look like fragments of a configuration file (e.g. "someVariable" and "anotherName" under the `settings.env` key):

ch4/simple-config.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: simple-config
  namespace: default
data:
  portNumber: "8080"
  name: Ricky
  settings.env: |
    someVariable=5
    anotherName=blah
```

Save the above YAML into a file called `simple-config.yaml` and create it by running `kubectl apply -f simple-config.yaml`.

To see the details of the ConfigMap you create, you can use the `describe` command:

```
$ kubectl describe cm simple-config
Name:         simple-config
Namespace:    default
Labels:       <none>
Annotations:
Data
====
name:
\----
Ricky
portNumber:
\----
8080
settings.env:
\----
someVariable=5
anotherName=blah

Events:  <none>
```

For Pod to use the values from the ConfigMap, both Pod and ConfigMap need to reside in the same namespace. You can mount a ConfigMap to your Pod and use the values in one of the following ways:

- As command-line arguments
- As environment variables
- As a file in a read-only-volume
- Through Kubernetes API that reads the ConfigMap

Let's create a Pod and show how to consume the values from a ConfigMap.

ch4/config-pod-1.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: config-pod
  labels:
    app.kubernetes.io/name: config-pod
spec:
  containers:
    - name: config
      image: busybox
      command: ["sh", "-c", "sleep 3600"]
      env:
        - name: FIRST_NAME
          valueFrom:
            configMapKeyRef:
              name: simple-config
              key: name
        - name: PORT_NUMBER
          valueFrom:
            configMapKeyRef:
              name: simple-config
              key: portNumber
```

We are using the `valueFrom` and `configMapKeyRef` to specify where the value for the environment variable is coming from. The environment variable name (e.g. `FIRST_NAME`) is and can be different from the key name stored in the ConfigMap (e.g. `name` or `portNumber`).

Save the above YAML in `config-pod-1.yaml` and create the Pod with `kubectl apply -f config-pod-1.yaml`. Once the Pod is running, let's get the terminal inside the container and look at the environment variables:

```
$ kubectl exec -it config-pod -- /bin/sh
/ # env
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBERNETES_SERVICE_PORT=443
HOSTNAME=config-pod
SHLVL=1
HOME=/root
PORT_NUMBER=8080
TERM=xterm
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_PROTO=tcp
FIRST_NAME=Ricky
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_SERVICE_HOST=10.96.0.1
PWD=/
```

Notice the `PORT_NUMBER` and `FIRST_NAME` environment variables values are coming from the ConfigMap.

How about the values under the `settings.env` key? Let's mount that as a file inside the Pod. Here's the updated YAML for the `config-pod`:

ch4/config-pod-2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: config-pod
  labels:
    app.kubernetes.io/name: config-pod
spec:
  containers:
    - name: config
      image: busybox
      command: ["sh", "-c", "sleep 3600"]
      env:
        - name: FIRST_NAME
          valueFrom:
            configMapKeyRef:
              name: simple-config
              key: name
        - name: PORT_NUMBER
          valueFrom:
            configMapKeyRef:
              name: simple-config
              key: portNumber
      volumeMounts:
        - name: config
          mountPath: "/config"
          readOnly: true
  volumes:
    - name: config
      configMap:
        name: simple-config
        items:
          - key: "settings.env"
            path: "local.env"
```

To bring in the `settings.env` from the ConfigMap, we create a Volume called `config` from the ConfigMap and mount the items (or, in our case, just one item called `settings.env`) to the path called `local.env`. Mounting creates a file `local.env` file containing the key-value pairs defined under the key `settings.env` from the ConfigMap. Using a `volumeMount` field, we mount the `config` volume under the `/config` folder inside the container.

Before deploying the above YAML, make sure you delete the previous Pod first, by running `kubectl delete pod config-pod`. Then, save the above YAML in `config-pod-2.yaml` and apply it with `kubectl apply -f config-pod-2.yaml`.

Once the Pod is deployed, let's use the `exec` command to get a shell inside the container and look in the `/config` folder:

```
$ kubectl exec -it config-pod -- /bin/sh
/ # cat config/local.env
someVariable=5
anotherName=blah
/ #
```

In the previous two examples, we were setting two configuration settings as environment variables and mounting the `settings.env` as a file from a volume. Instead of mounting configuration as environment variables, we can mount the whole ConfigMap as a Volume. In this case, each configuration setting name (e.g., `portNumber`, `name`) will be created as a file containing the respective value.

Once you've delete the previous pod (`kubectl delete po config-pod`), deploy the following YAML:

ch4/config-pod-3.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: config-pod
  labels:
    app.kubernetes.io/name: config-pod
spec:
  containers:
    - name: config
      image: busybox
      command: ["sh", "-c", "sleep 3600"]
      volumeMounts:
        - name: config
          mountPath: "/config"
          readOnly: true
  volumes:
    - name: config
      configMap:
        name: simple-config
```

The above YAML looks similar to what you've seen before. The only difference is that we aren't explicitly calling out any configuration settings - we are merely creating a volume (`config`) from a ConfigMap and then mounting that volume under `/config` folder inside the Pod.

If you look at the contents of the `/config` folder inside the container, you will notice that there's a separate file created for every configuration setting from the ConfigMap:

```
$ kubectl exec -it config-pod -- ls /config
name          portNumber    settings.env
```

Each of the files contains the value(s) set in the ConfigMap:

```
$ kubectl exec -it config-pod -- cat /config/name
Ricky
```

The mounted ConfigMaps are updated automatically. Let's edit the `simple-config` ConfigMap and change the `portNumber` from `8080` to `5000`. Run `kubectl edit cm simple-config` to launch the editor from the terminal and change the value of `portNumber` to `5000`. Save the changes and exit the editor.

Kubelet periodically checks if the ConfigMap was updated and sets the new value. If you check the value of the `/config/portNumber` file, you will notice that the value is updated to `5000`:

```
$ kubectl exec -it config-pod -- cat /config/portNumber
5000
```

Similarly, you can use `envFrom` field to automatically set all configuration settings from a ConfigMap as environment variables. Here's how the Pod YAML looks like in this case:

ch4/pod-env-prefix.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: config-pod
  labels:
    app.kubernetes.io/name: config-pod
spec:
  containers:
    - name: config
      image: busybox
      command: ["sh", "-c", "sleep 3600"]
      envFrom:
        - prefix: SIMPLE_CONFIG_
          configMapRef:
            name: simple-config
```

Instead of using the `env` field, you use the `envFrom`, specify the prefix you want to set to each entry, and reference the ConfigMap. If you don't set the prefix, the variable names will look like this:

```
portNumber
name
settings.env
```

With the prefix set to `SIMPLE_CONFIG_`, Kubernetes names the variables like this:

```
SIMPLE_CONFIG_portNumber
SIMPLE_CONFIG_name
SIMPLE_CONFIG_settings.env
```

The prefix is optional, and you don't have to set it. However, it might make sense to select it, especially if you have values coming from different ConfigMaps, and want to group them.

If you deploy the above YAML and run the `env` command in the container, you will see the environment variables being set with the prefix:

```
$ kubectl exec -it config-pod -- env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=config-pod
TERM=xterm
SIMPLE_CONFIG_name=Ricky
SIMPLE_CONFIG_portNumber=8080
SIMPLE_CONFIG_settings.env=someVariable=5
anotherName=blah

KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
KUBERNETES_SERVICE_HOST=10.96.0.1
HOME=/root
```

Up until now, you've been creating ConfigMaps through YAML. In some cases, though, that might not be practical. Think about having an environment variable file with 10+ key-value pairs - it doesn't make sense to re-type all of them to create the YAML for the ConfigMap. For this purpose, you can also use Kubernetes CLI to create the ConfigMaps.

Creating ConfigMaps using Kubernetes CLI

The Kubernetes CLI supports creating ConfigMaps in three different ways:

- Creating key-value pairs using `--from-literal` setting
- Creating key-value pairs from an environment file (list of `key=value` pairs) using `--from-env-file` setting
- Creating key-value pairs from the file name and file contents using `--from-file` setting

Let's look at the first option where you can create key-value pairs using the `--from-literal` setting:

```
$ kubectl create cm literal-config --from-literal=portNumber=8080 --from-literal
=firstName=Ricky
configmap/literal-config created

$ kubectl describe cm literal-config
Name:         literal-config
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
firstName:
\----
Ricky
portNumber:
\----
8080
Events:  <none>
```

You should use the `--from-env-file` setting if you are using `.env` files in your projects. Let's consider the following `local.env` file:

```
PORT=8080
SERVICE_URL=http://myservice
FIRST_NAME=Ricky
RETRIES=10
```

You could use `--from-literal` for each of the settings, but it's much easier to use `--from-env-file` command like this:

```
$ kubectl create cm env-config --from-env-file=local.env
configmap/env-config created

$ kubectl describe cm env-config
Name:         env-config
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
FIRST_NAME:
\----
Ricky
PORT:
\----
8080
RETRIES:
\----
10
SERVICE_URL:
\----
http://myservice
Events:  <none>
```

Finally, let's consider a scenario where you're storing configuration settings in a `config.json` file that looks like this:

```
{
  "portNumber": "5000",
  "firstName": "Ricky",
  "serviceUrl": "http://myservice"
}
```

It is fairly safe to assume that your applications consume the JSON configuration as is - so it wouldn't make sense to split the file into regular key-value pairs and mount them as environment variables or something like that. You need a single configuration item that has the JSON contents as its value. To do this, you can use the `--from-file` setting:

```
$ kubectl create cm fromfile-cm --from-file=config.json
configmap/fromfile-cm created

$ kubectl describe cm fromfile-cm
Name:         fromfile-cm
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
config.json:
\----
{
    "portNumber": "5000",
    "firstName": "Ricky",
    "serviceUrl": "http://myservice"
}

Events:  <none>
```

The `--from-file` setting will use the file name (`config.json`) as the configuration key, and the value will be the contents of the file. If you have multiple configuration files, you can also pass in the folder name to the `--from-file` setting. The CLI will iterate through the files in the folder and create the configuration items, just like a single file.

You can combine one of the file options with the literal option in a single command and create a ConfigMap from multiple sources. For example:

```
$ kubectl create cm combined-cm --from-file=config.json --from-literal=HELLO=world
```

Note that the command fails if you use more than one option that reads a file (e.g. `--from-file` and `--from-env-file`) together in the same command.

Storing secrets in Kubernetes

The Secret resource in Kubernetes is similar to ConfigMaps. Just like with ConfigMaps, you can use Secrets to store key-value pairs. You can mount them into your containers either as files in a volume or use them as environment variables.

The choice between using a ConfigMaps or a Secret is straightforward: if the value you're storing contains sensitive information (keys, passwords, credentials, etc.) store it in a Secret. Otherwise, store the values in a ConfigMap. The Secret resource values are stored encrypted in the etcd (etcd is the key-value store used by Kubernetes to store the objects and their state).

There are three types of Secrets you can create as shown in the table below.

Table 2. Secrets

Secret type	Description	Example
generic	Secret that can be created from a file, directory or a literal value	`kubectl create secret generic service-auth --from -literal=username=ricky --from -file=password=password -file.txt`
docker-registry	Secret for use with Docker registry. You can pass in `.dockercfg` file or manually specify values needed to authenticate with the Docker registry (`docker-server`, `docker-username`, `docker-password`, `docker-email`)	`kubectl create secret docker-registry my-docker-registry --docker -server=https://index.docker.io/v1/ --docker -username=username --docker -password=ILOVEPIZZA2`
tls	Secret that holds the public/private key pair.	`kubectlcreate tls my-tls-secret --cert=my/certs/tls.cert --key=/my/certs/tls.key`

Let's create a generic secret using `kubectl`:

```
$ kubectl create secret generic service-auth --from-literal=username=ricky --from
-literal=password=ILOVEPIZZA2
secret/service-auth created
```

You can now use the `describe` command to describe the Secret:

```
$ kubectl describe secret service-auth
Name:         service-auth
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
password:  11 bytes
username:  5 bytes
```

If you remember from earlier when you describe a ConfigMap, you could see the actual key values. Let's create a ConfigMap with the same two values and describe it:

```
$ kubectl create cm service-auth-cm --from-literal=username=ricky --from-literal
=password=ILOVEPIZZA2
configmap/service-auth-cm created

$ kubetl describe cm service-auth-cm
Name:         service-auth-cm
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
password:
\----
ILOVEPIZZA2
username:
\----
ricky
Events:  <none>
```

When describing the ConfigMap, you can see the actual key values (the username and password), while in the Secret you only know the size (11 and 5 bytes).

Let's also look at the YAML representation of the Secret to see how the values are stored:

```
$ kubectl get secret service-auth -o yaml
apiVersion: v1
data:
  password: SUxPVkVQSVpaQTI=
  username: cmlja3k=
....
```

>You can add the `-o yaml` flag to the `get` command to show the YAML representation of any resource in Kubernetes:

Kubernetes stores the values as a Base64-encoded string. Encoding the values like that allows you to store not just plain text, but also binary data. If you are creating a secret through YAML, you will have to Base64-encode all binary values. However, if you have any secrets that don't need to be Base64-encoded (i.e., non-binary values), you can provide them in plain text using the `stringData` field.

Here's the same Secret as before, but in this case, we are providing the `username` in plain text, while the password is still provided as Base64-encoded string:

ch4/svc-auth-2.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: service-auth-2
  namespace: default
stringData:
  username: ricky
data:
  password: SUxPVkVQSVpaQTI=
```

Save the above YAML in `svc-auth-2.yaml` file and create it using this command: `kubectl apply -f svc-auth-2.yaml`.

If you get the YAML representation, you will see that both values still end up being Base64-encoded and the field `stringData` is omitted (Kubernetes uses it to create the Base64-encode entries under the `data` field):

```
$ kubectl get secret service-auth-2 -o yaml
apiVersion: v1
data:
  password: SUxPVkVQSVpaQTI=
  username: cmlja3k=
kind: Secret
metadata:
....
```

Just like with ConfigMaps, Pods can consume Secrets through environment variables and volumes. When you use the Secret is a Pod, the values are stored as plain text. You don't have to Base64-decode the values in your containers.

Consuming Secrets as environment variables

Let's look at how you can consume the values from a Secret we created earlier as environment variables:

ch4/pod-secret.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-pod
  labels:
    app.kubernetes.io/name: secret-pod
spec:
  containers:
    - name: secret
      image: busybox
      command: ["sh", "-c", "sleep 3600"]
      env:
        - name: USERNAME
          valueFrom:
            secretKeyRef:
              name: service-auth
              key: username
        - name: PASSWORD
          valueFrom:
            secretKeyRef:
              name: service-auth
              key: password
```

Save the above YAML in `pod-secret.yaml` and deploy it using `kubectl apply -f pod-secret.yaml`.

If you compare the above to the example we did in the ConfigMaps section you won't see many differences. The difference is in the field name. When referencing a Secret, you use `secretKeyRef`, instead of `configMapKeyRef` when you reference a ConfigMap.

Once the Pod is running, let's look at the environment variables set in the container:

```
$ kubectl exec -it secret-pod -- env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=secret-pod
TERM=xterm
USERNAME=ricky
PASSWORD=ILOVEPIZZA2
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
KUBERNETES_SERVICE_HOST=10.96.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_PORT=443
HOME=/root
```

Note that we didn't have to do any Base64-encoding, and the Secret values show up encoded. To delete this Pod, run `kubectl delete pod secret-pod`.

Even though you can use secrets as environment variables, it is **not recommended** as it is much easier to expose them unintentionally. For example, you might be writing all environment variables to log files when your application crashes or when the application starts up. To be safe, you should instead use volumes to mount secrets into Pods.

Consuming Secrets from Volumes

Let's look at an example of how to mount the Secret values through files. Mounting Secrets as files is the preferred and more secure way of consuming secrets than using them through environment variables.

ch4/secret-pod-volume.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-pod
  labels:
    app.kubernetes.io/name: secret-pod
spec:
  containers:
    - name: secret
      image: busybox
      command: ["sh", "-c", "sleep 3600"]
      volumeMounts:
        - name: auth-secret
          mountPath: "/var/secrets"
          readOnly: true
  volumes:
    - name: auth-secret
      secret:
        secretName: service-auth
```

We define the volume holding the Secret similarly as we did the ConfigMap. We are using the `secret` field and then providing the `secretName`. Finally, we are mounting the volume using the `volumeMounts` field and specifying the volume name and the container's location where we want to store the secrets.

Save the above YAML in `secret-pod-volume.yaml` and create the Pod using `kubectl apply -f secret-pod-volume.yaml`.

If you run the `ls` command inside the container, you will notice two files: `password` and `username`. These two files contain the Secret values in plain-text.

```
$ kubectl exec -it secret-pod -- ls /var/secrets
password  username

$ kubectl exec -it secret-pod -- cat /var/secrets/username
ricky

$ kubectl exec -it secret-pod -- cat /var/secrets/password
ILOVEPIZZA2
```

Stateful Workloads

Running stateful workloads inside Kubernetes is different from running stateless services. The reason being is that the containers and Pods can get created and destroyed at any time. If any of the cluster nodes go down or a new node appears, Kubernetes needs to reschedule the Pods.

If you ran a stateful workload or a database in the same way you are running a stateless service, all of your data would be gone the first time your Pod restarts.

Therefore we need to store the data outside of the container. Storing the data outside ensures that nothing happens to it when the container restarts.

What are Volumes?

The Volume abstraction in Kubernetes solves this problem. The Volume lives as long as the Pod lives. If any of the containers within the Pod get restarted, Volume preserves the data. However, once you delete the Pod, the Volume gets deleted as well.

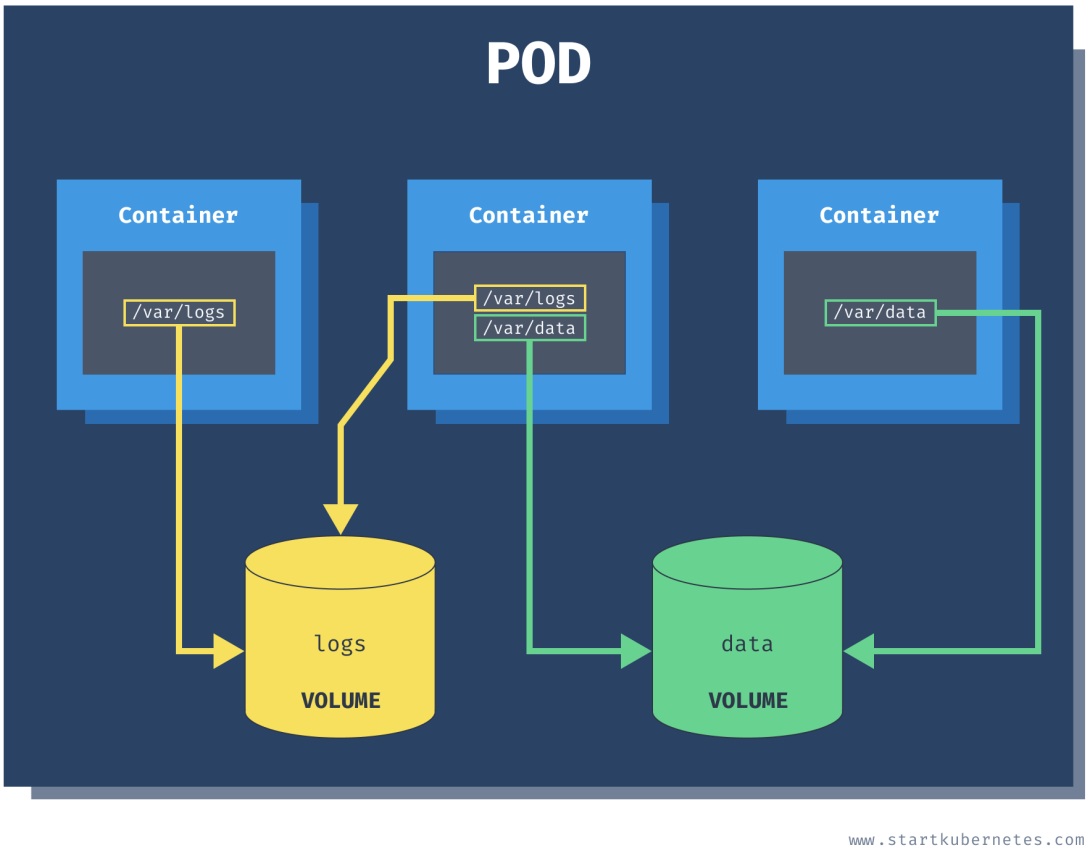


Figure 29. Volumes in a Pod

The Volume is just a folder that may or may not have any data in it. The folder is accessible to all containers in a pod. How this folder gets created and the backing storage is determined by the volume type.

The most basic volume type is an empty directory (`emptyDir`). When you create a Volume with the

`emptyDir` type, Kubernetes creates it when it assigns a Pod to a node. The Volume exists for as long as the Pod is running. As the name suggests, it is initially empty, but the containers can write and read from the Volume. Once you delete the Pod, Kubernetes deletes the Volume as well.

There are two parts to using the Volumes. The first one is the Volume definition. You can define the volumes in the Pod spec by specifying the volume name and the type (`emptyDir` in our case). The second part is mounting the Volume inside of the containers using the `volumeMounts` key. In each Pod you can use multiple different Volumes at the same time.

Inside the volume mount we refer to the Volume by name (`pod-storage`) and specifying which path we want to mount the Volume under (`/data/`).

ch5/empty-dir-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: empty-dir-pod
spec:
  containers:
    - name: alpine
      image: alpine
      args:
        - sleep
        - "120"
      volumeMounts:
        - name: pod-storage
          mountPath: /data/
  volumes:
    - name: pod-storage
      emptyDir: {}
```

Save the above YAML in `empty-dir-pod.yaml` and run `kubectl apply -f empty-dir.pod.yaml` to create the Pod.

Next, we are going to use the `kubectl exec` command to get a terminal inside the container:

```
$ kubectl exec -it empty-dir-pod -- /bin/sh
/ # ls
bin    dev    home   media  opt    root   sbin   sys    usr
data   etc    lib    mnt    proc   run    srv    tmp    var
```

If you run `ls` inside the container, you will notice the `data` folder. The `data` folder is mounted from the `pod-storage` Volume defined in the YAML.

Let's create a dummy file inside the `data` folder and wait for the container to restart (after 2 minutes) to prove that the data inside the `data` folder stays around.

From inside the container create a `hello.txt` file under the `data` folder:

```
echo "hello" >> data/hello.txt
```

You can type `exit` to exit the container. If you wait for 2 minutes, the container will automatically restart. To watch the container restart, run the `kubectl get po -w` command from a separate terminal window.

Once container restarts, you can check that the file `data/hello.txt` is still in the container:

```
$ kubectl exec -it empty-dir-pod -- /bin/sh
/ # ls data/hello.txt
data/hello.txt
/ # cat data/hello.txt
hello
/ #
```

Kubernetes stores the data on the host under the `/var/lib/kubelet/pods` folder. That folder contains a list of pod IDs, and inside each of those folders is the `volumes`. For example, here's how you can get the pod ID:

```
$ kubectl get po empty-dir-pod -o yaml | grep uid
  uid: 683533c0-34e1-4888-9b5f-4745bb6edced
```

Armed with the Pod ID, you can run `minikube ssh` to get a terminal inside the host Minikube uses to run Kubernetes. Once inside the host, you can find the `hello.txt` in the following folder:

```
$ sudo cat /var/lib/kubelet/pods/683533c0-34e1-4888-9b5f-
4745bb6edced/volumes/kubernetes.io~empty-dir/pod-storage/hello.txt
hello
```

If you are using Docker Desktop, you can run a privileged container and using `nsenter` run a shell inside all namespace of the process with id 1:

```
$ docker run -it --privileged --pid=host debian nsenter -t 1 -m -u -n -i sh
/ #
```

Once you get the terminal, the process is the same - navigate to the `/var/lib/kubelet/pods` folder and find the `hello.txt` just like you would if you're using Minikube.

Kubernetes supports a large variety of other volume types. Some of the types are generic, such as `emtpyDir` or `hostPath` (used for mounting folders from the nodes' filesystem). Other types are either used for **cloud-provider storage** (such as `azureFile`, `awsElasticBlockStore`, or `gcePersistentDisk`), **network storage** (`cephfs`, `cinder`, `csi`, `flocker`, ...), or for mounting Kubernetes resources into the Pods (`configMap`, `secret`).

Lastly, another particular type of Volumes are Persistent Volumes and Persistent Volume Claims we discuss in Persisting data with Persistent Volumes and Persistent Volume Claims.

The lack of the word "persistent" when talking about other volumes can be misleading. If you are using any cloud-provider storage volume types (`azureFile` or `awsElasticBlockStore`), the data will still be persisted. The persistent volume and persistent volume claims are just a way to abstract how Kubernetes provisions the storage.

For the full and up-to-date list of all volume types, check the Kubernetes Docs

Persisting data with Persistent Volumes and Persistent Volume Claims

A persistent volume (`PersistentVolume` or PV) is another volume type you can use to mount a persistent volume into a Pod.

Using a PV and persistent volume claim (`PersistentVolumeClaim` or PVC), you can claim a portion of durable storage (such as persistent disks from cloud providers) without knowing any details about the cloud environment and how that storage was provisioned or created.

If you think about this as a user or a developer, you only want to get a piece of durable storage to store your apps' data, but you don't necessarily care about the data's location.

You can create PVs in two different ways:

1. Provisioned by cluster administrators
2. Dynamically provisioned using Storage Classes

Once the persistent volume is created (either by the administrator or dynamically using a storage class), you need a way to request it or claim it. You use the persistent volume claim (`PersistentVolumeClaim` or PVC) to request or claim the volume. Inside the PVC, you can request a volume of a specific size and different access modes. For example, you might have multiple persistent volumes available - one optimized for heavy reads, other optimized for writes, etc.

When you create a PVC, a Kubernetes controller will try to match the PVC with a requested PV and bind them together. In case the matching PV does not exist, the claim remains unbound. As soon as the PV is available again, the PVC will get bound to it.

Storage classes

A storage class (`StorageClass`) is a resource that describes a type of storage. A cluster administrator can create multiple storage classes with different provisioners and properties. Here's an example of a StorageClass that uses an Azure Disk as its provisioner:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: azure-disk-slow
provisioner: kubernetes.io/azure-disk
parameters:
  skuName: Standard_LRS
  location: westus
  storageAccount: [storage-account-name]
```

If `azure-disk-slow` storage class is requested as part of the persistent volume claim, Kubernetes will use the storage class information to either use the existing storage account or create a new one.

The parameters for each type of storage class depend on the provisioner. For Azure Disk provisioner, we set the `skuName`, `location`, and `storageAccount`. The Glusterfs provisioner might require us to set parameters such as URL cluster ID, secret name, etc.

A Kubernetes cluster running with Docker Desktop uses a `hostpath` storage class provisioner. You can get a list of all storage classes in your cluster by running the following command:

```
# Docker Desktop
$ kubectl get storageclass
NAME                 PROVISIONER          AGE
hostpath (default)   docker.io/hostpath   44h
```

Similarly, a cluster running with Minikube has the following storage class:

```
# Minikube
$ kubectl get storageclass
NAME                 PROVISIONER                RECLAIMPOLICY   VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION   AGE
standard (default)   k8s.io/minikube-hostpath   Delete          Immediate
false                  2m49s
```

You will also notice the word **default** next to the `hostpath` storage class name. You can mark a storage class as default, so when, for example, you request some storage using a PVC without specifying the storage class, Kubernetes uses the default storage class.

Creating Persistent Volumes

Let's start by creating a persistent volume using the hostPath volume plugin. In the resource, we define the storage capacity (5 gibibytes), the access mode (`ReadWriteOnce`), and a host path of `/mnt/data`. This means that we will allocate 5Gi on the node in the folder `/mnt/data`, and the Volume can only be mounted as read-write by a single node.

There are three different access modes we can choose from:

Table 3. Access Modes

Access mode	Short name	Description
ReadWriteOnce	RWO	The volume can be mounted as read-write by a single node only
ReadOnlyMany	ROX	The volume can be mounted read-only by many nodes
ReadWriteMany	RWX	The volume can be mounted as read-write by many nodes

Note that not all volume plugins support all access modes. For example, the `hostPath` plugin only supports the `ReadWriteOnce` access mode, and so does the Azure Disk plugin. However, Azure File, Glusterfs, and a couple of other plugins support all three access modes.

Let's save the YAML below in `local-pv.yaml` and run `kubectl apply -f local-pv.yaml` to create the PersistentVolume.

ch5/local-pv.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv-volume
spec:
  storageClassName: manual
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

You can view the created persistent volume by running the following command:

```
$ kubectl get pv
NAME              CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM
STORAGECLASS   REASON   AGE
local-pv-volume   5Gi        RWO            Retain           Available
manual                  16s
```

The reclaim policy for the volume (`Retain` in the above case) specifies that Kubernetes retains the data, and it's up to the user/administrator to reclaim the space manually. The other two options are `Recycle` where volume contents are deleted, and `Delete` that applies to cloud-provider backed storage where the storage resource is deleted.

Now that we have the persistent volume, we can create a persistent volume claim to request storage. Let's look at the YAML first and then explain the different fields:

ch5/local-pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: local-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Using the above claim we request 1Gb of storage using the `manual` storage class name and RWO access mode. Surprise, surprise, this exactly matches the persistent volume we create before.

Save the above YAML in `local-pvc.yaml` and run `kubectl apply -f local-pvc.yaml` to create the PersistentVolumeClaim.

If we look at the status of the persistent volume and run the `kubectl get pv` command, you will notice the `STATUS` column shows the `Bound` status and the `CLAIM` column shows the name of the claim:

```
$ kubectl get pv
NAME              CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
STORAGECLASS   REASON   AGE
local-pv-volume   5Gi        RWO            Retain           Bound    default/local-
pv-claim   manual                  14m
```

You can also see the Volume name and status by looking at the PVC:

```
$ kubectl get pvc
NAME             STATUS   VOLUME            CAPACITY   ACCESS MODES   STORAGECLASS
AGE
local-pv-claim   Bound    local-pv-volume   5Gi        RWO            manual
19s
```

Before we create a Pod that consumes this Volume, let's create a file in the `/mnt/data` folder on the host.

If you're using Docker Desktop, you can use the command below to get into the virtual machine (host):

```
docker run -it --privileged --pid=host debian nsenter -t 1 -m -u -n -i sh
```

If you're using Minikube, the equivalent command is `minikube ssh`.

Once inside the host, let's start by creating the `/mnt/data` folder. If you're using Docker Desktop, create the folder under the `/containers/services/docker/rootfs` folder. If using Minikube, just create the `/mnt/data` folder.

The data folder will have an `index.html` file inside. In the Pod, we will run an Nginx container that will show that `index.html` file.

```
# create the folder (Docker Desktop)
mkdir -p /containers/services/docker/rootfs/mnt/data

# create the folder (Minikube)
sudo mkdir -p /mnt/data

# create index.html (Docker Desktop)
echo "Hello from Storage!" >> /containers/services/docker/rootfs/mnt/data/index.html

# create index.html (Minikube)
sudo sh -c "echo 'Hello from Storage!' > /mnt/data/index.html"

# exit the shell
exit
```

With the volume populated, let's create a Pod that consumes the volume:

ch5/pv-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pv-pod
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - name: http
          containerPort: 80
      volumeMounts:
        - name: pv-storage
          mountPath: "/usr/share/nginx/html"
  volumes:
    - name: pv-storage
      persistentVolumeClaim:
        claimName: local-pv-claim
```

We refer to the persistent volume claim under `volumes`, not to the underlying volume. Inside the `containers` field, we are then referencing the volume (`pv-storage`) and mounting it under `/usr/share/nginx/html` (this is the folder Nginx uses by default).

Save the above YAML in `pv-pod.yaml` file and create it by running `kubectl apply -f pv-pod.yaml`.

To check if the volume mount worked correctly, we are going to use the `port-forward` command to forward port `5000` on the local machine to port `80` on the container. Open a separate terminal window and run:

```
$ kubectl port-forward po/pv-pod 5000:80
Forwarding from 127.0.0.1:5000 -> 80
Forwarding from [::1]:5000 -> 80
```

You can either open a browser and navigate to `http://localhost:5000` or run `curl http://localhost:5000` from another terminal window. You should get back the contents of the `index.html` file we created earlier:

```
$ curl http://localhost:5000
Hello from Storage!
```

You can clean up everything by deleting the Pod, PVC, and then the PV:

```
$ kubectl delete po pv-pod
$ kubectl delete pv local-pv-volume
$ kubectl delete pvc local-pv-claim
```

Finally, you can also delete the contents of the `/mnt/data` folder inside the host VM.

Using Storage Class for dynamic provisioning

In the previous examples, we manually created a PersistentVolume first (using `manual` storage class), before we could claim the storage and then use it.

Both Minikube and Docker Desktop have a default storage class set - in both cases, they use the `hostPath` provisioner.

Let's try and create a persistent volume claim, but this time we won't specify the storage class field at all:

ch5/default-pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: default-pv-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Save the above YAML in `default-pvc.yaml` and run `kubectl apply -f default-pvc.yaml` to create the persistent volume claim that uses the default storage class.

If you list the PVC, you will notice that it was bound to a volume called `pvc-bf222497-024e-4ff1-a29f-1f4d3a441607` using `hostpath` storage class (this is the default storage class name if using Docker Desktop).

```
$ kubectl get pvc
NAME               STATUS   VOLUME                                     CAPACITY
ACCESS MODES   STORAGECLASS   AGE
default-pv-claim   Bound    pvc-d1cb8b49-bb9b-499b-baca-3631b40a44bf   1Gi        RWO
hostpath       3s
```

You can also list the PV to see the persistent volume that was created dynamically by the storage class:

```
$ kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY
STATUS   CLAIM                      STORAGECLASS   REASON   AGE
pvc-d1cb8b49-bb9b-499b-baca-3631b40a44bf   1Gi        RWO            Delete
Bound    default/default-pv-claim   hostpath                90s
```

On Docker Desktop the volume gets created under `/var/lib/k8s-pvs/[pvc-name]/[pv-name]` folder on the host VM, where the PVC name in our case is `default-pv-claim` and the PV name is that randomly generate volume name (`pvc-d1cb8b49-bb9b-499b-baca-3631b40a44bf`).

So, let's get a shell inside the VM and create an `index.html` file, just like we did before:

```
# Get a shell inside host VM
docker run -it --privileged --pid=host debian nsenter -t 1 -m -u -n -i sh

# Create index.html file
echo "Hello Dynamic Volume!" >> /var/lib/k8s-pvs/default-pv-claim/pvc-d1cb8b49-bb9b-
499b-baca-3631b40a44bf/index.html

# exit the VM
exit
```

Finally, let's create the Pod - the resource looks exactly the same as before, the only difference is the PVC name:

ch5/default-pv-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pv-pod
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - name: http
          containerPort: 80
      volumeMounts:
        - name: pv-storage
          mountPath: "/usr/share/nginx/html"
  volumes:
    - name: pv-storage
      persistentVolumeClaim:
        claimName: default-pv-claim
```

Save the above YAML in `default-pv-pod.yaml` and run `kubectl apply -f default-pv-pod.yaml` to create the Pod.

When the Pod starts running, use `port-forward` to forward local port `5000` to the port `80` on the container:

```
$ kubectl port-forward po/pv-pod 5000:80
Forwarding from 127.0.0.1:5000 -> 80
Forwarding from [::1]:5000 -> 80
```

Just like before, you can open `http://localhost:5000` in the browser or run the curl command to get back the response, contents of the `index.html` file:

```
$ curl http://localhost:5000
Hello Dynamic Volume!
```

If you delete the Pod and then the PVC, you will notice that the Persistent Volume gets deleted automatically. The automatic deletion is due to the reclaim policy on the default storage class. The value is set to `Delete`, and Kubernetes controller automatically deletes the volume once the claim is gone.

Running stateful workloads with StatefulSets

Using PersistentVolumes and PersistentVolumeClaims, you could run multiple Pod replicas using a ReplicaSet. These replicas have different names and IP addresses, but other than that, they are the same.

Let's consider the following snippet from one of the previous examples of a Pod and a PVC:

```
...
spec:
  containers:
      ...
      volumeMounts:
        - name: pv-storage
          mountPath: "/usr/share/nginx/html"
  volumes:
    - name: pv-storage
      persistentVolumeClaim:
        claimName: local-pv-claim
....
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: local-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

The Volume and the persistent volume claim reference are both defined in the Pod spec, the template ReplicaSet uses to create the new Pods. If you have a single Pod, it will reference a single PVC and a single PV.

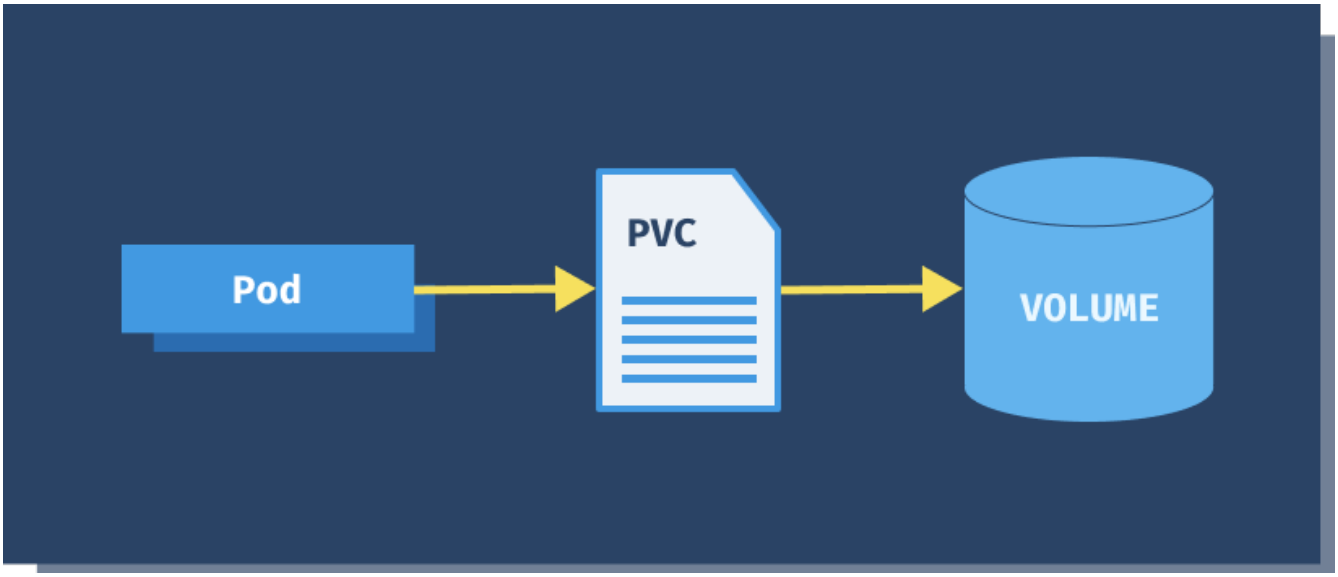


Figure 30. Single Pod with a PVC and PV

When you scale the Pods, each newly created Pod gets a different name and an IP address. Since you include PV and PVC in the Pod template, all replicas use the same PVC and the same PV as shown in the figure below.

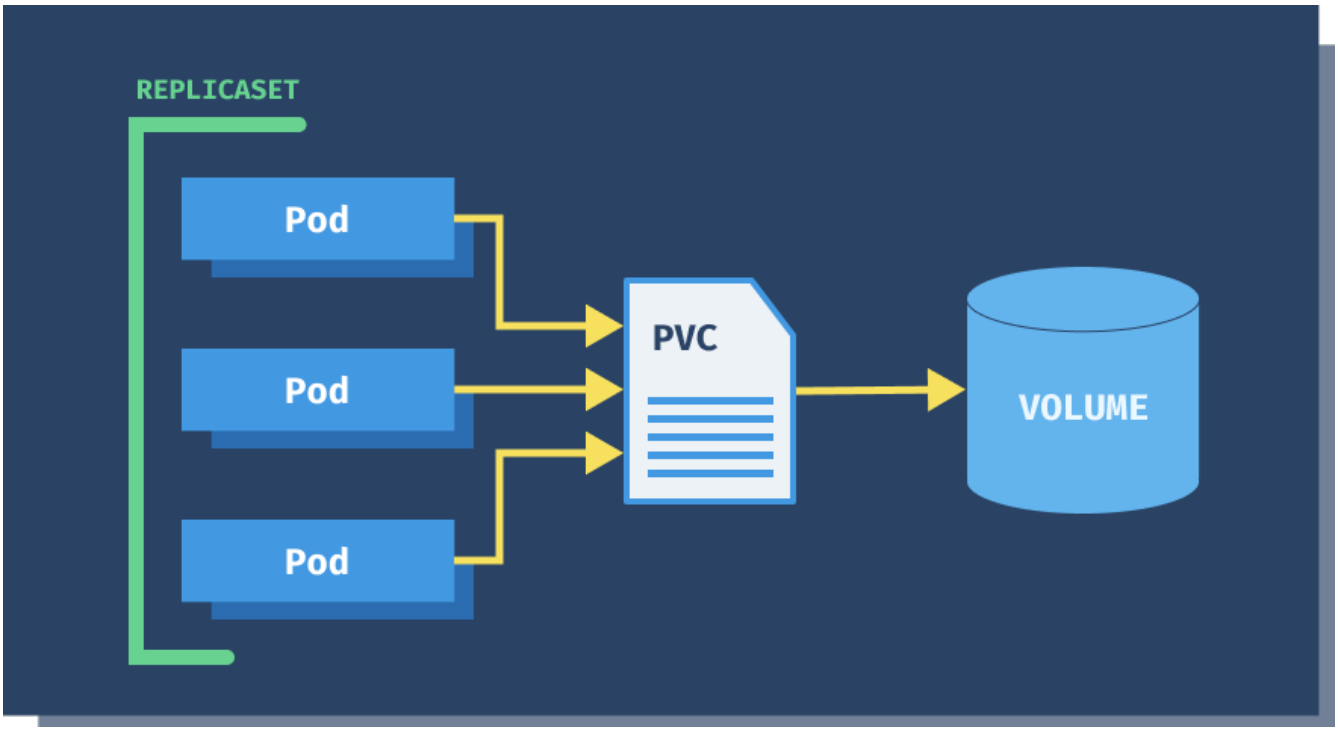


Figure 31. Multiple Pods with a Single PVC and PV

Because you defined the reference to the PVC in the Pod template, you can't make each replica use its persistent volume claim. For that reason, you can't use a single ReplicaSet to run a distributed data store where each Pod has its storage.

There are a couple of workarounds, such as creating the Pods manually and have them use separate PVC in that way, however, you don't want to manually manage the Pods' lifecycle (remember that any manually created Pod will not get automatically restarted when or if it crashes). Another workaround would be to use multiple ReplicaSets - one for each Pod. This solution could work; however, it is painful to maintain.

In addition to have a separate storage per Pod, you also need to ensure the Pods have a stable and persistent identity. Stable identity means if the Pod is restarted, it comes back with the same identifier (the same name and the same IP address). The reason for stable identity is that you often need to address a specific replicate, especially when running a storage system. That is complicated to do if you don't have stable identifiers.

A resource called a **StatefulSet** can help. You can use StatefulSets for applications that require a stable name and state. The main difference between a StatefulSet and a ReplicaSet is that when Pods created by a ReplicaSet are rescheduled, they get a new name and a new IP address. On the other hand, the StatefulSet ensures that Pod keeps the same name and the same IP address even when it gets rescheduled.

StatefulSets also allow you to run multiple Pod replicas. These replicas are not the same - the can have their own set of volume claims or, more specifically volume claim templates. The replicas don't have random names. Instead, each Pod gets assigned an index.

We have mentioned earlier that in some scenarios, you want to be able to address a specific replica from the StatefulSet. You can do that by creating a headless Kubernetes service.

In the next example, we will show how to run MongoDB using a StatefulSet. As the first step, we will create a headless service called `mongodb`. We need to create this first because we will reference the service name in the StatefulSet.

ch5/mongodb-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: mongodb
  labels:
    app: mongodb
spec:
  clusterIP: None
  selector:
    app: mongodb
  ports:
    - port: 27017
      targetPort: 27017
```

Save the above YAML in `mongodb-svc.yaml` and create the Service by running `kubectl apply -f mongodb-svc.yaml`. If you list the Services, you will notice the `mongodb` Service does not have an IP address set:

```
$ kubectl get svc
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)     AGE
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP     47h
mongodb      ClusterIP   None         <none>        27017/TCP   1s
```

Next, we will create the following StatefulSet:

ch5/mongodb-statefulset.yaml

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongodb
spec:
  serviceName: mongodb
  replicas: 1
  selector:
    matchLabels:
      app: mongodb
  template:
    metadata:
      labels:
        app: mongodb
        selector: mongodb
    spec:
      containers:
        - name: mongodb
          image: mongo:4.0.17
          ports:
            - containerPort: 27017
          volumeMounts:
            - name: pvc
              mountPath: /data/db
  volumeClaimTemplates:
    - metadata:
        name: pvc
      spec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 1Gi
```

Looking at the YAML above, you will notice that it looks very similar to the ReplicaSet. The critical part is the `volumeClaimTemplates` section. That's the section where we defined the PersistentVolumeClaim. When the StatefulSet needs to create a new Pod replica, it uses that template also to create a PersistentVolume and a PersistentVolumeClaim resource for each Pod, as shown in the figure below.

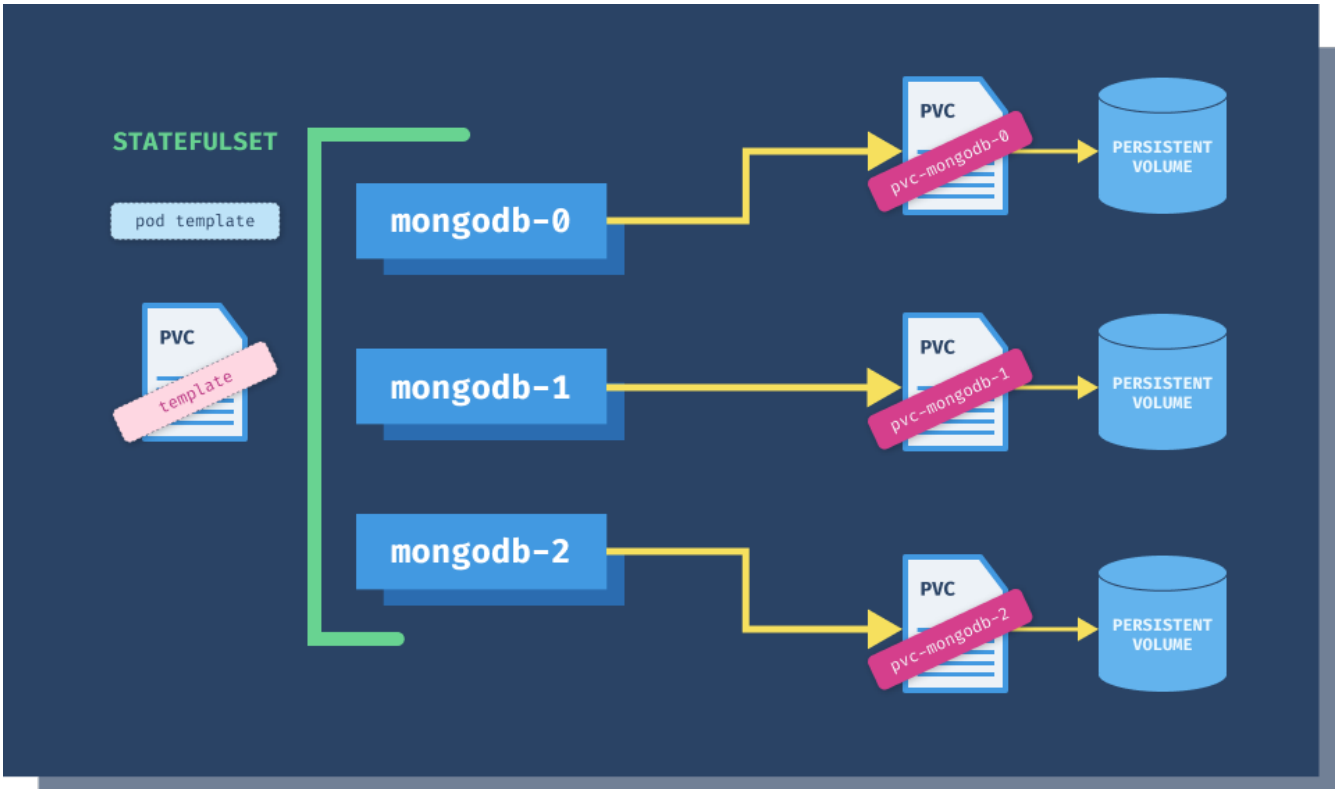


Figure 32. Pods Created by StatefulSet

Save the above YAML in `mongodb-statefulset.yaml` and create the StatefulSet by running `kubectl apply -f mongodb-statefulset.yaml`.

If you list the Pods, you will notice you created a single Pod named `mongodb-0`:

```
$ kubectl get pods
NAME        READY   STATUS    RESTARTS   AGE
mongodb-0   1/1     Running   0          29m
```

Similarly, let's list the PersistentVolumes and PersistentVolumeClaims:

```
$ kubectl get pv,pvc
NAME                                                        CAPACITY   ACCESS MODES
RECLAIM POLICY   STATUS   CLAIM                   STORAGECLASS   REASON   AGE
persistentvolume/pvc-275f7731-0f10-4b33-b99a-3cc95d0be30a   1Gi        RWO
Delete           Bound    default/pvc-mongodb-0   standard                31m

NAME                                  STATUS   VOLUME
CAPACITY   ACCESS MODES   STORAGECLASS   AGE
persistentvolumeclaim/pvc-mongodb-0   Bound    pvc-275f7731-0f10-4b33-b99a-
3cc95d0be30a   1Gi        RWO            standard       31m
```

Notice the naming of both resources - the PVC uses the name (`pvc`) we provided under the `volumeClaimTemplates`, and it appends the name of the Pod (`mongodb-0`) to it. Kubernetes prefixes the PersistentVolume name with the PVC name (`pvc`). The rest of the name is a unique identifier.

Let's see what happens if we scale the StatefulSet to 3 Pods:

```
$ kubectl scale statefulset mongodb --replicas=3
statefulset.apps/mongodb scaled
```

If you watch the Pods as Kubernetes creates them (`kubectl get pods -w`) you will notice they are created in order - the replica named `mongodb-1` is created first, and after that replica `mongodb-2` is created.

A StatefulSet stores the name of the pod in label called `statefulset.kubernetes.io/pod-name`. You can see these labels if you run `kubectl get po --show-labels`:

```
$ kubectl get po --show-labels
NAME        READY   STATUS    RESTARTS   AGE     LABELS
mongodb-0   1/1     Running   0          44m     statefulset.kubernetes.io/pod-name
=mongodb-0
mongodb-1   1/1     Running   0          2m34s   statefulset.kubernetes.io/pod-name
=mongodb-1
mongodb-2   1/1     Running   0          2m30s   statefulset.kubernetes.io/pod-name
=mongodb-2
```

These Pods have other labels (`selector`, `app`, `controller-revision-hash`). I removed them from the above output for better readability.

Using this label, you could create a service to target a specific replica. Remember when we created a headless service? Let's see how we can use it to access a particular replica.

We have 3 Pods running, and they are named `mongodb-0`, `mongodb-1`, and `mongodb-2`. We also have a headless service called `mongodb`. To access each instance, we use the following format `[podName].[serviceName]`. For example, to access `mongodb-1` we can use `mongodb-1.mongodb` or `mongodb-1.mongodb.default.svc.cluster.local`, where `default` is the namespace where the Pods (and Service) reside, and `cluster.local` is the local cluster domain.

Let's try accessing these instances. We will run a `mongo` container inside the cluster and use that container to access the `mongodb-0`,`mongodb-1`, and `mongodb-2` instances.

Run the following command to get create a Pod called `mongo-shell` and run a `/bin/bash` inside it:

```
$ kubectl run -it mongo-shell --image=mongo:4.0.17 --rm -- /bin/bash
If you don\'t see a command prompt, try pressing enter.
root@mongo-shell: # /
```

The container has the `mongo` shell installed, and we can use this binary to connect to the MongoDB instances. Let's try and connect to the `mongodb-0` instance.

To do that, we will use the `mongodb-0.mongodb` name - remember that this only works because we are running the `mongo-shell` container inside of the cluster.

```
root@mongo-shell:/# mongo mongodb-0.mongodb
MongoDB shell version v4.0.17
connecting to: mongodb://mongodb-0.mongodb:27017/test?gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("7972f9c4-f04e-42fe-aa15-e63adfaea2eb") }
MongoDB server version: 4.0.17
Welcome to the MongoDB shell.
....
...
>
```

The shell will automatically connect us to the collection called `test`. Let's insert a dummy entry into that collection so that we can prove the MongoDB instances are each storing the data in a different volume.

Run the command below to insert a single entry with the name field set to `first MongoDB instance`:

```
> db.test.insert({ name: "first MongoDB instance" })
WriteResult({ "nInserted" : 1 })
```

You can also run `db.test.find()` to list all documents from the `test` collection (there will only be one in there):

```
> db.test.find()
{ "_id" : ObjectId("5f540f288a4810beb9d9237a"), "name" : "first MongoDB instance" }
```

Let's connect to the second instance now. You can type `exit` to exit the first instance, and get back to the `mongo-shell` container. Just like before, we will use the `mongo` binary to connect to a different instance:

```
$ root@mongo-shell:/# mongo mongodb-1.mongodb
```

Once connected, you can look at the contents of the `test` collection and, as expected, there will be 0 documents in the collection:

```
> db.test.find()
>
```

We can create a different document here:

```
> db.test.insert({ name: "second MongoDB instance" })
WriteResult({ "nInserted" : 1 })
```

Let's see if the data is persistent when we delete the `mongodb-1` Pod. Open a separate terminal

window and delete the `mongodb-1` Pod:

```
$ kubectl delete po mongodb-1
pod "mongodb-1" deleted
```

As soon as the Pod is deleted, StatefulSet will re-create it again - it will use the same name, PVC and PV. If you run the `db.test.find()` from the mongo shell in the first terminal window you will notice that it contains the same data we inserted earlier right after it reconnects to the Pod:

```
> db.test.find()
2020-09-05T22:29:43.690+0000 I NETWORK  [js] trying reconnect to mongodb-
1.mongodb:27017 failed
2020-09-05T22:29:43.693+0000 I NETWORK  [js] reconnect mongodb-1.mongodb:27017 ok
{ "_id" : ObjectId("5f5411034901f8a94bfdcc7c"), "name" : "second MongoDB instance" }
```

Organizing Containers

Init containers

Init containers allow you to separate your application from the initialization logic and provide a way to run the initialization tasks such as setting up permissions, database schemas, or seeding data for the main application, etc. The init containers may also include any tools or binaries that you don't want to have in your primary container image due to security reasons.

The init containers are executed in a sequence before your primary or application containers start. On the other hand, any application containers have a non-deterministic startup order, so you can't use them for the initialization type of work.

The figure below shows the execution flow of the init containers and the application containers.

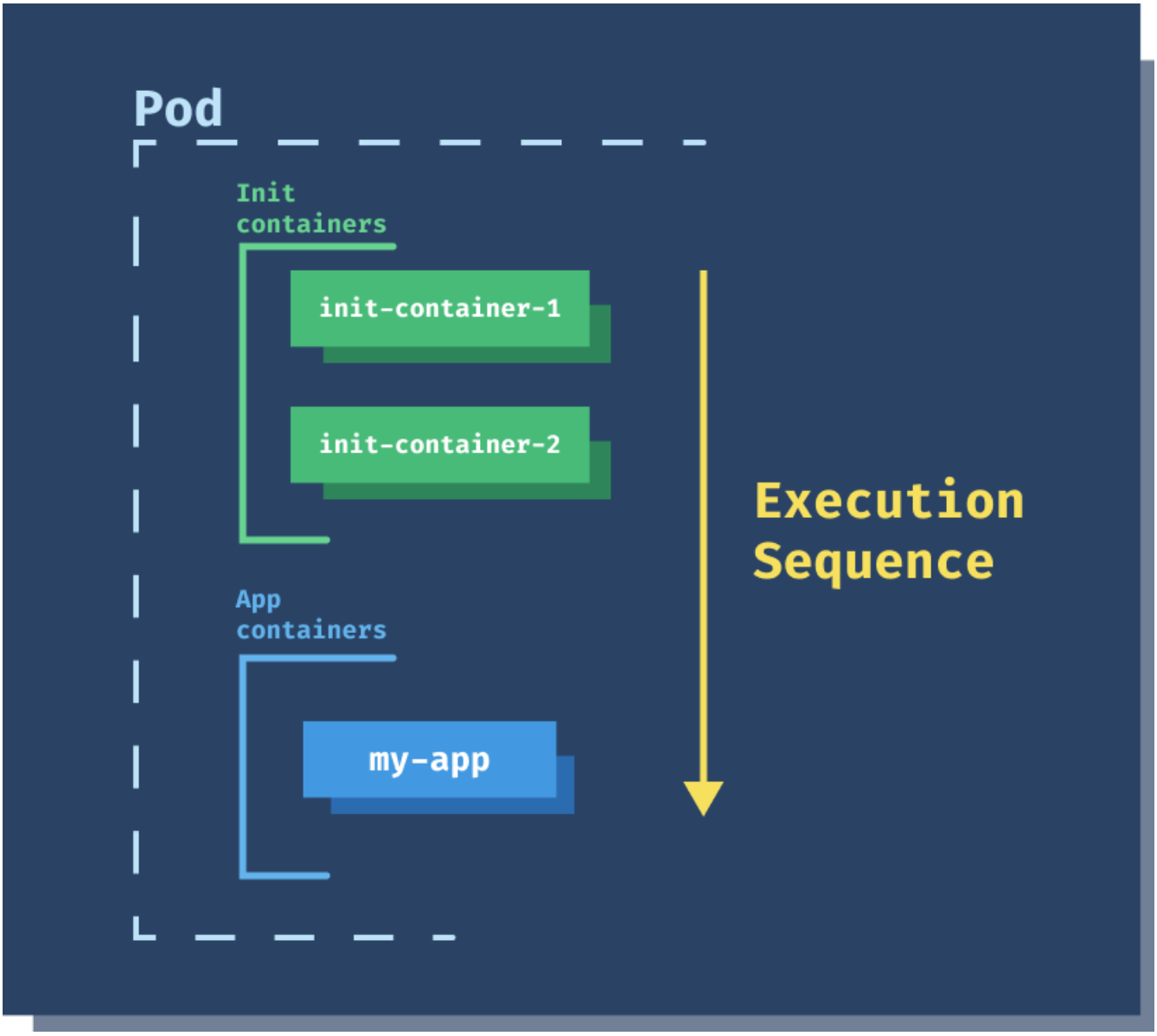


Figure 33. Init Containers

The application containers will wait for the init containers to complete successfully before starting. If the init containers fail, the Pod is restarted (assuming we didn't set the restart policy to

`RestartNever`), which causes the init containers to run again. When designing your init containers, make sure they are idempotent, to run multiple times without issues. For example, if you're seeding the database, check if it already contains the records before re-inserting them again.

Since init containers are part of the same Pod, they share the volumes, network, security settings, and resource limits, just like any other container in the Pod.

Let's look at an example where we use an init container to clone a GitHub repository to a shared volume between all containers. The Github repo contains a single `index.html`. Once the repo is cloned and the init container has executed, the primary container running the Nginx server can use `index.html` from the shared volume and serve it.

You define the init containers under the `spec` using the `initContainers` field, while you define the application containers under the `containers` field. We define an `emptyDir` volume and mount it into both the init and application container. When the init container starts, it will run the `git clone` command and clone the repository into the `/usr/share/nginx/html` folder. This folder is the default folder Nginx serves the HTML pages from, so when the application container starts, we will be able to access the HTML page we cloned.

ch6/init-container.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: website
spec:
  initContainers:
    - name: clone-repo
      image: alpine/git
      command:
        - git
        - clone
        - --progress
        - https://github.com/peterj/simple-http-page.git
        - /usr/share/nginx/html
      volumeMounts:
        - name: web
          mountPath: "/usr/share/nginx/html"
  containers:
    - name: nginx
      image: nginx
      ports:
        - name: http
          containerPort: 80
      volumeMounts:
        - name: web
          mountPath: "/usr/share/nginx/html"
  volumes:
    - name: web
      emptyDir: {}
```

Save the above YAML to `init-container.yaml` and create the Pod using `kubectl apply -f init-container.yaml`.

If you run `kubectl get pods` right after the above command, you should see the status of the init container:

```
$ kubectl get po
NAME      READY   STATUS     RESTARTS   AGE
website   0/1     Init:0/1   0          1s
```

The number `0/1` indicates a total of 1 init containers, and 0 containers have been completed so far. In case the init container fails, the status changes to `Init:Error` or `Init:CrashLoopBackOff` if the container fails repeatedly.

You can also look at the events using the `describe` command to see what happened:

```
Normal  Scheduled  19s   default-scheduler  Successfully assigned default/website to
minikube
Normal  Pulling    18s   kubelet, minikube  Pulling image "alpine/git"
Normal  Pulled     17s   kubelet, minikube  Successfully pulled image "alpine/git"
Normal  Created    17s   kubelet, minikube  Created container clone-repo
Normal  Started    16s   kubelet, minikube  Started container clone-repo
Normal  Pulling    15s   kubelet, minikube  Pulling image "nginx"
Normal  Pulled     13s   kubelet, minikube  Successfully pulled image "nginx"
Normal  Created    13s   kubelet, minikube  Created container nginx
Normal  Started    13s   kubelet, minikube  Started container nginx
```

You will notice as soon as Kubernetes schedules the Pod, the first Docker image is pulled (`alpine/git`), and the init container (`clone-repo`) is created and started. Once that's completed (the container cloned the repo) the main application container (`nginx`) starts.

Additionally, you can also use the `logs` command to get the logs from the init container by specifying the container name using the `-c` flag:

```
$ kubectl logs website -c clone-repo
Cloning into '/usr/share/nginx/html'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (6/6), done.
```

Finally, to actually see the static HTML page can use `port-forward` to forward the local port to the port `80` on the container:

```
$ kubectl port-forward pod/website 8000:80
Forwarding from 127.0.0.1:8000 -> 80
Forwarding from [::1]:8000 -> 80
```

You can now open your browser at `http://localhost:8000` to open the static page as shown in figure below.

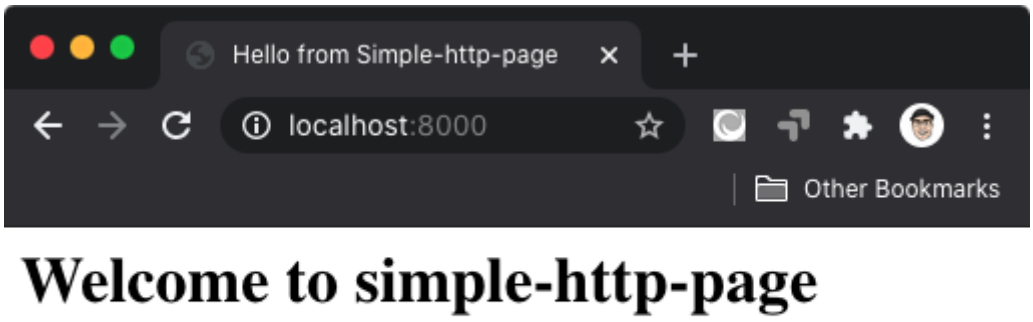


Figure 34. Static HTML from Github repo

Lastly, delete the Pod by running `kubectl delete po website`.

Sidecar container pattern

The sidecar container aims to add or augment an existing container's functionality without changing the container. In comparison to the init container, we discussed previously, the sidecar container starts and runs simultaneously as your application container. The sidecar is just a second container you have in your container list, and the startup order is not guaranteed.

Probably one of the most popular implementations of the sidecar container is in Istio service mesh. The sidecar container (an Envoy proxy) is running next to the application container and intercepting inbound and outbound requests. In this scenario, the sidecar adds the functionality to the existing container and allows the operator to do traffic routing, failure injection, and other features.

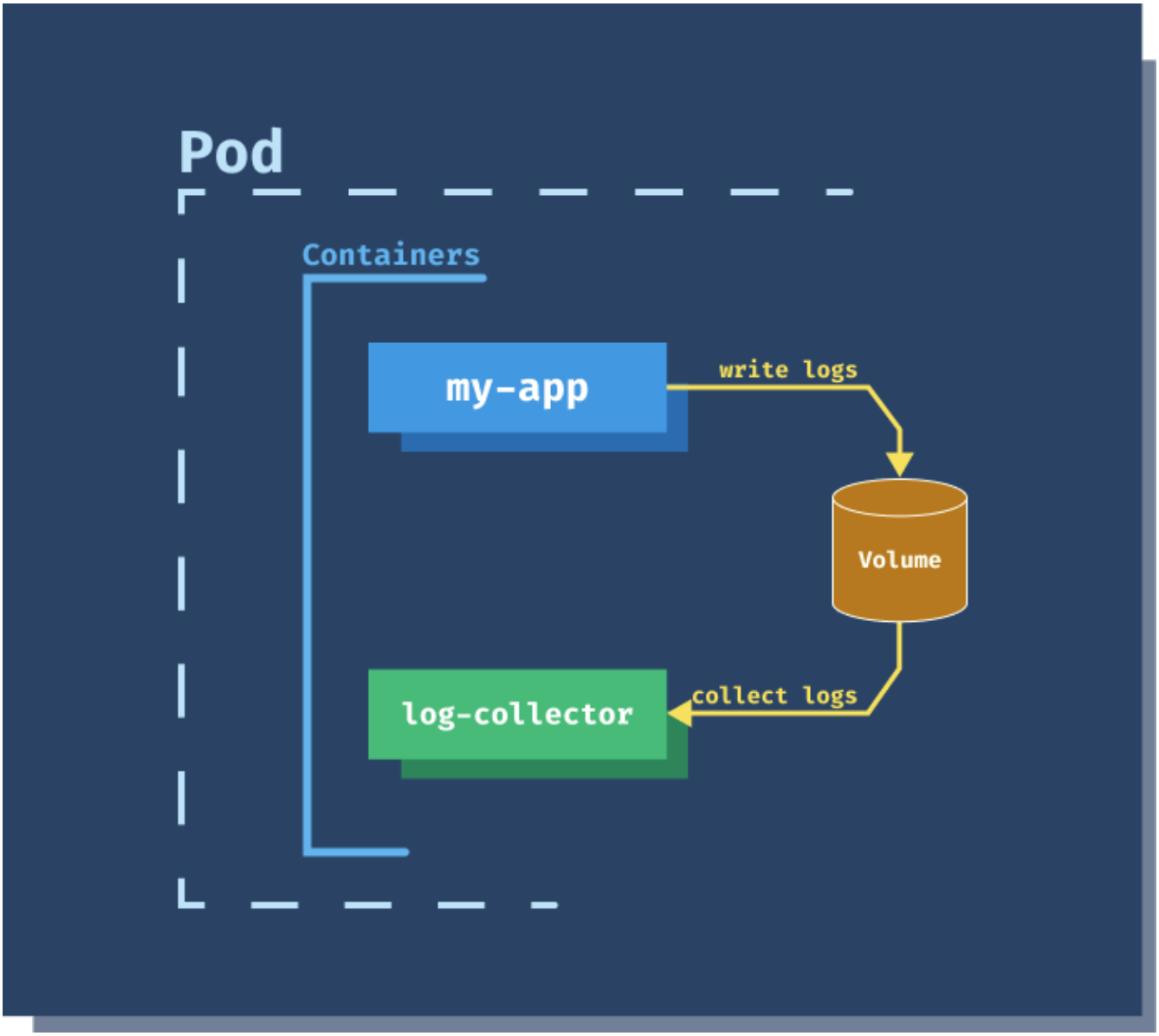


Figure 35. Sidecar Pattern

A simpler idea might be having a sidecar container (`log-collector`) that collects and stores application container's logs. That way, as an application developer, you don't need to worry about collecting and storing logs. You only need to write logs to a location (a volume, shared between the containers) where the sidecar container can collect them and send them to further processing or archive them.

If we continue with the example we used for the init container; we could create a sidecar container that periodically updates runs `git pull` and updates the repository. For this to work, we will keep the init container to do the initial clone, and a sidecar container that will periodically (every 60 seconds for example) check and pull the repository changes.

To try this out, make sure you fork the original repository (`https://github.com/peterj/simple-http-page.git`) and use your fork in the YAML below.

ch6/sidecar-container.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: website
spec:
  initContainers:
    - name: clone-repo
      image: alpine/git
      command:
        - git
        - clone
        - --progress
        - https://github.com/peterj/simple-http-page.git
        - /usr/share/nginx/html
      volumeMounts:
        - name: web
          mountPath: "/usr/share/nginx/html"
  containers:
    - name: nginx
      image: nginx
      ports:
        - name: http
          containerPort: 80
      volumeMounts:
        - name: web
          mountPath: "/usr/share/nginx/html"
    - name: refresh
      image: alpine/git
      command:
        - sh
        - -c
        - watch -n 60 git pull
      workingDir: /usr/share/nginx/html
      volumeMounts:
        - name: web
          mountPath: "/usr/share/nginx/html"
  volumes:
    - name: web
      emptyDir: {}
```

We added a container called `refresh` to the YAML above. It uses the `alpine/git` image, the same image as the init container, and runs the `watch -n 60 git pull` command.

NOTE The `watch` command periodically executes a command. In our case, it executes `git pull` command and updates the local repository every 60 seconds.

Another field we haven't mentioned before is `workingDir`. This field will set the working directory for the container. We are setting it to `/usr/share/nginx/html` as that's where we originally cloned the

repo to using the init container.

Save the above YAML to `sidecar-container.yaml` and create the Pod using `kubectl apply -f sidecar-container.yaml`.

If you run `kubectl get pods` once the init container has executed, you will notice the `READY` column now shows `2/2`. These numbers tell you right away that this Pod has a total of two containers, and both of them are ready:

```
$ kubectl get po
NAME      READY   STATUS    RESTARTS   AGE
website   2/2     Running   0          3m39s
```

If you set up the port forward to the Pod using `kubectl port-forward pod/website 8000:80` command and open the browser to `http://localhost:8000`, you will see the same webpage as before.

We can open a separate terminal window and watch the logs from the `refresh` container inside the `website` Pod:

```
$ kubectl logs website -c refresh -f

Every 60.0s: git pull
Already up to date.
```

The `watch` command is running, and the response from the last `git pull` command was `Already up to date`. Let's make a change to the `index.html` in the repository you forked.

I added `<div>` element and here's how the updated `index.html` file looks like:

ch6/index.html

```
<html>
  <head>
    <title>Hello from Simple-http-page</title>
  </head>
  <body>
    <h1>Welcome to simple-http-page</h1>
    <div>Hello!</div>
  </body>
</html>
```

Next, you need to stage this and commit it to the `master` branch. The easiest way to do that is from the Github's webpage. Open the `index.html` on Github (I am opening `https://github.com/peterj/simple-http-page/blob/master/index.html`, but you should replace my username `peterj` with your username or the organization you forked the repo to) and click the pencil icon to edit the file (see the figure below).

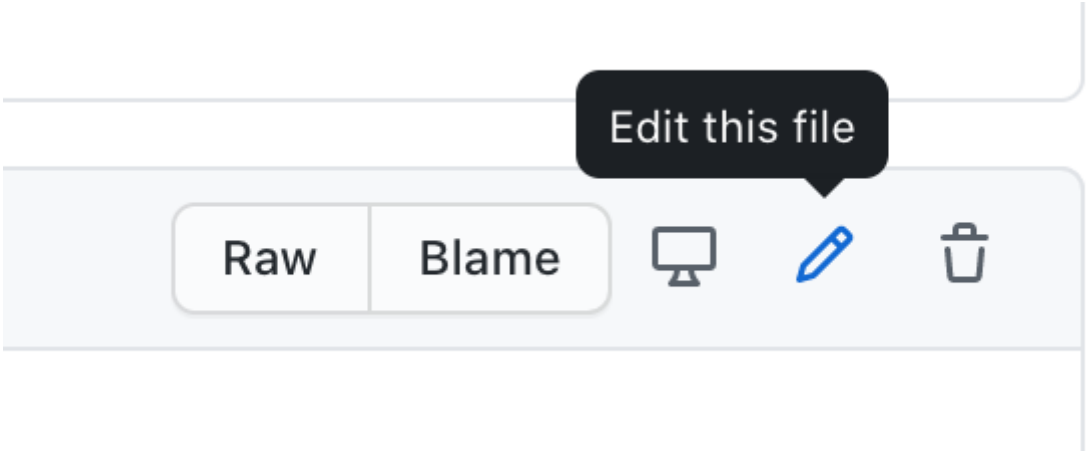


Figure 36. Editing index.html on Github

Make the change to the `index.html` file and click the **Commit changes** button to commit them to the branch. Next, watch the output from the `refresh` container, and you should see the output like this:

```
Every 60.0s: git pull

From https://github.com/peterj/simple-http-page
   f804d4c..ad75286  master     -> origin/master
Updating f804d4c..ad75286
Fast-forward
 index.html | 1 +
 1 file changed, 1 insertion(+)
```

The above output indicates changes to the repository. Git pulls the updated file to the shared volume. Finally, refresh your browser where you have `http://localhost:8000` opened, and you will notice the changes on the page:

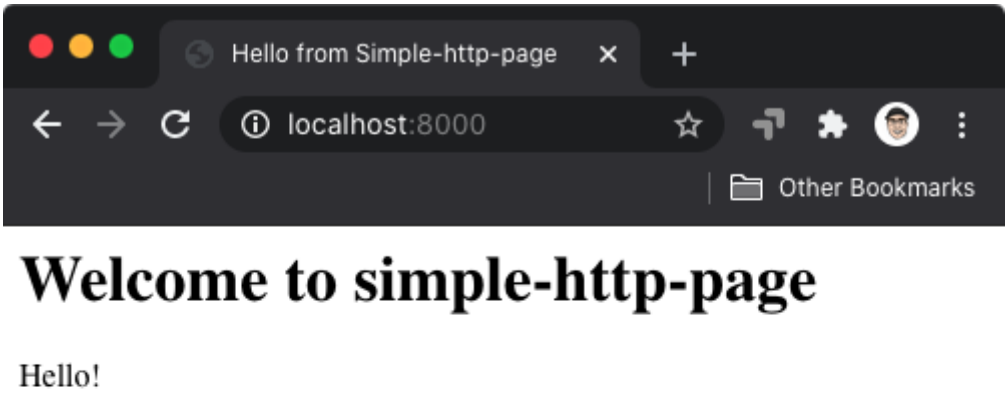


Figure 37. Updated index.html page

You can make more changes, and each time, the page will get updated within 60 seconds. You can delete the Pod by running `kubectl delete po website`.

Ambassador container pattern

The ambassador container pattern aims to hide the primary container's complexity and provide a unified interface through which the primary container can access services outside of the Pod.

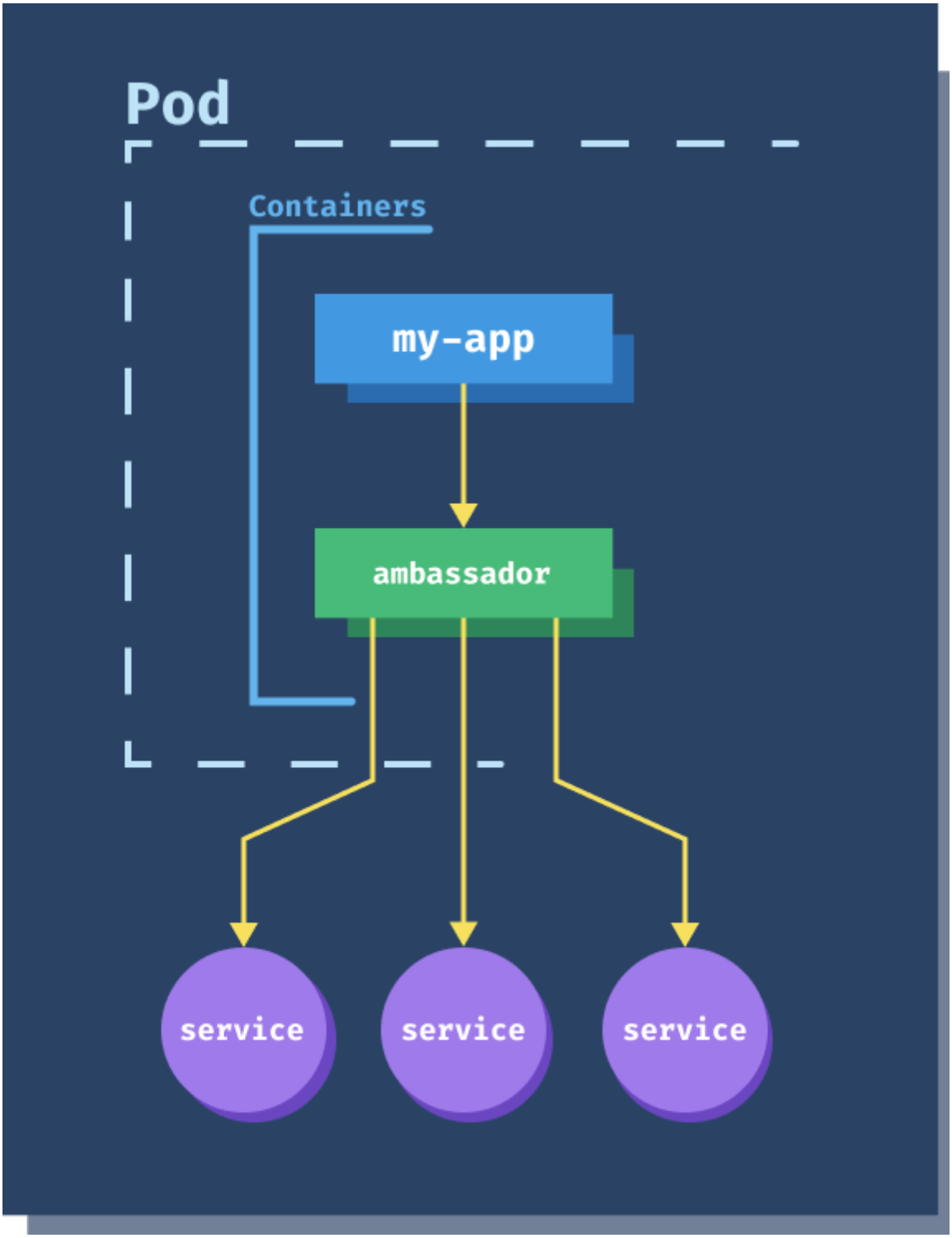


Figure 38. Ambassador Pattern

These outside or external services might present different interfaces and have other APIs. Instead of writing the code inside the main container that can deal with these external services' multiple interfaces and APIs, you implement it in the ambassador container. The ambassador container knows how to talk to and interpret responses from different endpoints and pass them to the main container. The main container only needs to know how to talk to the ambassador container. You can then re-use the ambassador container with any other container that needs to talk to these services while maintaining the same internal interface.

Another example would be where your main containers need to make calls to a protected API. You could design your ambassador container to handle the authentication with the protected API. Your

main container will make calls to the ambassador container. The ambassador will attach any needed authentication information to the request and make an authenticated request to the external service.

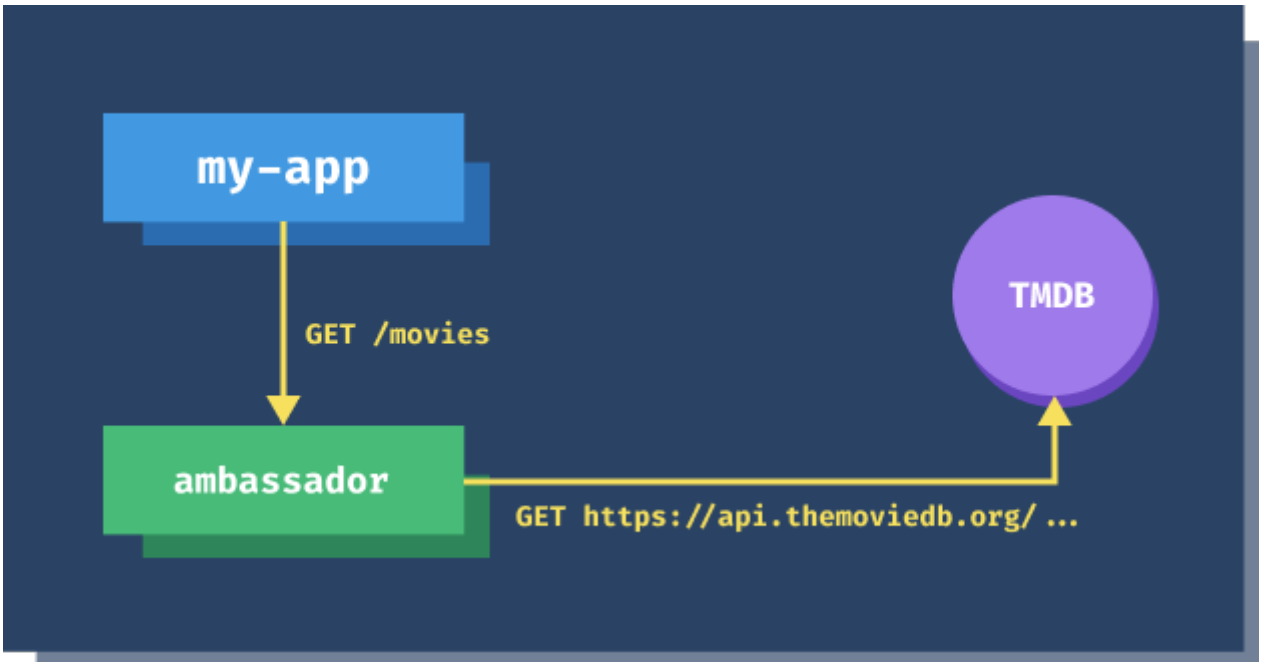


Figure 39. Calls Through Ambassador

To demonstrate how the ambassador pattern works, we will use The Movie DB (TMBD)[https://www.themoviedb.org/]. Head over to the website and register (it's free) to get an API key.

The Movie DB website offers a REST API where you can get information about the movies. We have implemented an ambassador container that listens on path `/movies`, and whenever it receives a request, it will make an authenticated request to the API of The Movie DB.

Here's the snippet from the code of the ambassador container:

```
func TheMovieDBServer(w http.ResponseWriter, r *http.Request) {
    apiKey := os.Getenv("API_KEY")
    resp, err := http.Get(fmt.Sprintf
("https://api.themoviedb.org/3/discover/movie?api_key=%s", apiKey))
    // ...
    // Return the response
}
```

We will read the `API_KEY` environment variable and then make a GET request to the URL. Note if you try to request to URL without the API key, you'll get the following error:

```
$ curl https://api.themoviedb.org/3/discover/movie
{"status_code":7,"status_message":"Invalid API key: You must be granted a valid key."
,"success":false}
```

I have pushed the ambassador's Docker image to `startkubernetes/ambassador:0.1.0`.

Just like with the sidecar container, the ambassador container is just another container that's running in the Pod. We will test the ambassador container by calling `curl` from the main container.

Here's how the YAML file looks like:

ch6/ambassador-container.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: themoviedb
spec:
  containers:
    - name: main
      image: radial/busyboxplus:curl
      args:
        - sleep
        - "600"
    - name: ambassador
      image: startkubernetes/ambassador:0.1.0
      env:
        - name: API_KEY
          valueFrom:
            secretKeyRef:
              name: themoviedb
              key: apikey
      ports:
        - name: http
          containerPort: 8080
```

Before we can create the Pod, we need to create a Secret with the API key. Let's do that first:

```
$ kubectl create secret generic themoviedb --from-literal=apikey=<INSERT YOUR API KEY
HERE>
secret/themoviedb created
```

You can now store the Pod YAML in `ambassador-container.yaml` file and create it with `kubectl apply -f ambassador-container.yaml`.

When Kubernetes creates the Pod (you can use `kubectl get po` to see the status), you can use the `exec` command to run the `curl` command inside the `main` container:

```
$ kubectl exec -it themoviedb -c main -- curl localhost:8080/movies

{"page":1,"total_results":10000,"total_pages":500,"results":[{"popularity":2068.491,"v
ote_count":
...
```

Since containers within the same Pod share the network, we can make a request against `localhost:8080`, which corresponds to the port on the ambassador container.

You could imagine running an application or a web server in the main container, and instead of making requests to the `api.themoviedb.org` directly, you are making requests to the ambassador container.

Similarly, if you had any other service that needed access to the `api.themoviedb.org` you could add the ambassador container to the Pod and solve access like that.

Adapter container pattern

For the ambassador container pattern, we said that it hides outside services' complexity and provides a unified interface to the main container. The adapter container pattern does the opposite. It provides a unified interface to the external services.

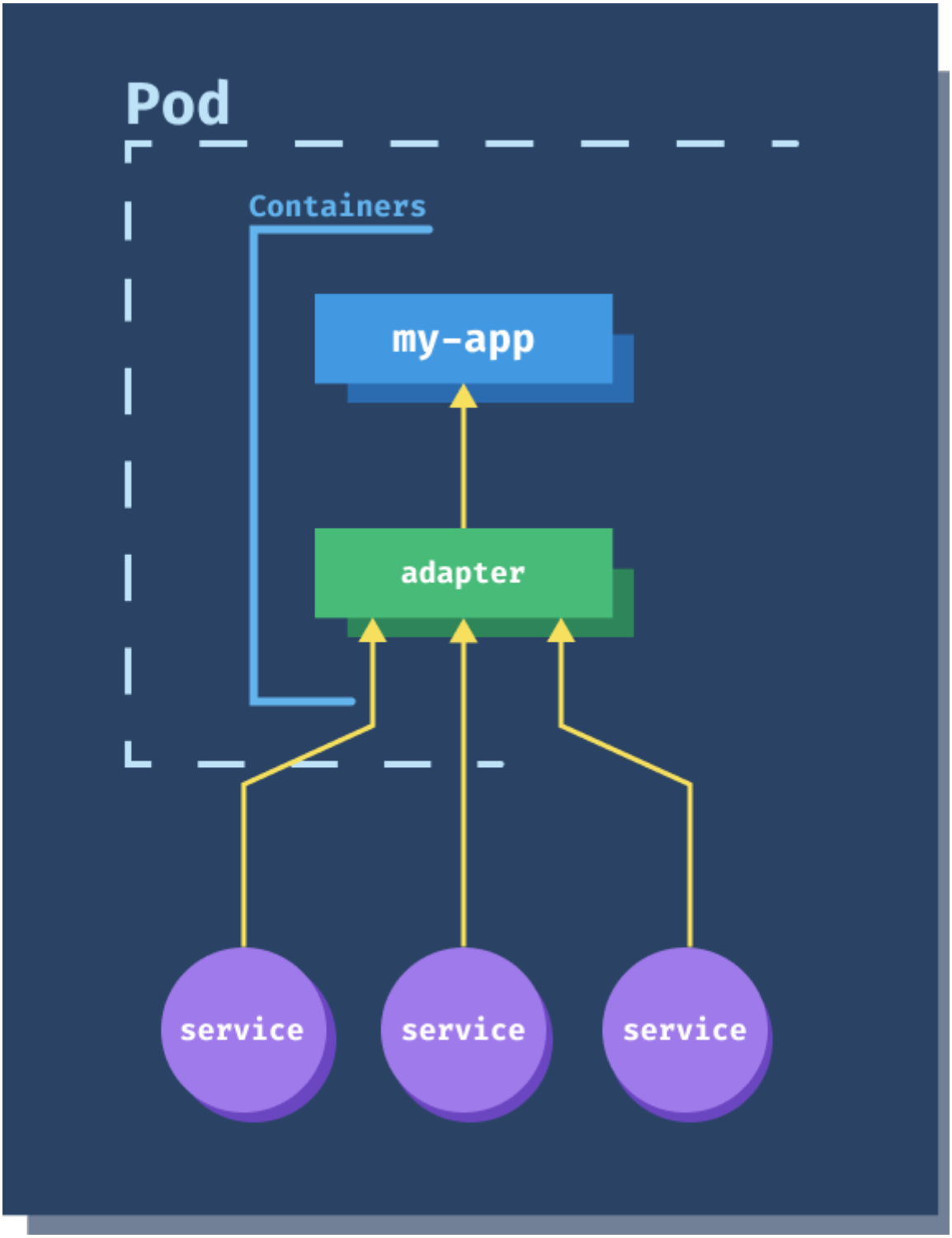


Figure 40. Adapter Pattern

Using the adapter pattern, you use common interfaces across multiple containers. An excellent example of this pattern are adapters that ensure all containers have the same monitoring interface. For example, adapter for exposing the application or container metrics on `/metrics` and port `9090`.

Your application might write the logs and metrics to a shared volume, and the adapter reads the data and serves it on a common endpoint and port. Using this approach, you can add the adapter container to each Pod to expose the metrics.

Another use of this pattern is for logging. The idea is similar as the metrics - your applications can use an internal logging format. Simultaneously, the adapter takes that format, cleans it up, adds additional information, and then serves it to the centralized log aggregator.

Application Health

The kubelet component in Kubernetes uses different health checking mechanisms to verify if the containers inside your Pods have started (startup probe), are ready to receive traffic (readiness probe), and are healthy (liveness probe).

To explain how health checks or probes in Kubernetes work, we will be using a sample application with endpoints defined in the table below.

Table 4. Endpoints

Endpoint	Description
`/healthy`	Returns an HTTP 200. If you set the `HEALTHY_SLEEP` environment variable, it will sleep for the period defined in the variable. For example, if you set `HEALTHY_SLEEP=5000`, the endpoint waits for 5 seconds before returning HTTP 200.
`/healthz`	The endpoint returns HTTP 200 for the first 10 seconds. After 10 seconds, it starts returning an HTTP 500.
`/ready`	Functionally equivalent to the `/healthy` endpoint (returns HTTP 200). Uses `READY_SLEEP` environment variable to sleep for the amount of millisecond defined in the variable before returning HTTP 200.
`/readyz`	The endpoint returns HTTP 500 for the first 10 seconds. After that, it starts returning an HTTP 200.
`/fail`	Returns an HTTP 500 error. Uses `FAIL_SLEEP` environment variable to sleep for the period (in milliseconds) defined in the variable before returning.

With the combination of the endpoints above and a couple of environment variables, we will simulate different failure scenarios and see how probes can help out.

There are three methods you can use to determine if containers are healthy or not:

- Invoke a command inside the container. The exit code determines if the liveness probe failed or not. A non-zero exit code indicates that the container is unhealthy
- Open a TCP socket. If the socket opens, the probe is successful, otherwise the probe fails
- Send an HTTP request to the provided URI. The HTTP response code is used to determine if the liveness probe succeeds or fails

With the combination of different methods and probes, you can ensure that your containers are

ready to receive traffic, healthy, and get restarted in case of a deadlock.

Application Liveness probe

The liveness probe indicates whether your application is still working as it's supposed to. In case the liveness probe fails, the **kubelet** will restart the failing container.

Without the liveness probe set, Kubernetes assumes the container healthy and keeps it running. Using this probe, you can handle scenarios where your code is deadlocked, or your application is not responding anymore. Without the probe, Kubernetes keeps the application running, however, if you provide and implement the probe, Kubernetes will restart the container in case the probe fails.

HTTP Probe

Let's use the App Health example we mentioned at the beginning of this chapter.

In the YAML, we define the liveness probe using an HTTP get request. Kubelet send the HTTP GET request to the `/healthz` endpoint on the container, on port `3000`. Kubernetes considers the container healthy, as long as the response is greater or equal than 200 and less than 400. In our example, we are returning HTTP 200 for the first 10 seconds, and then HTTP 500 afterward.

The `initialDelaySeconds` field tells Kubernetes to wait 3 seconds before it starts calling the probe. Similarly, the `periodSeconds` field specifies that the Kubernetes should perform the probe every second. "Performing the probe," in this case, means sending a GET request to the specified path and port on the container. The kubelet performs the first probe 4 seconds after the container starts (3 seconds is the initial delay and then 1 second for the first period).

ch7/liveness.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness
  labels:
    app.kubernetes.io/name: liveness
spec:
  containers:
    - name: web
      image: startkubernetes/app-health:0.1.0
      ports:
        - containerPort: 3000
      livenessProbe:
        httpGet:
          path: /healthz
          port: 3000
        initialDelaySeconds: 3
        periodSeconds: 1
```

Save the above YAML in `liveness.yaml` and create the Pod with `kubectl apply -f liveness.yaml`.

Let's use the `describe` command to look at the Pod events:

```
$ kubectl describe po liveness
...
Tolerations:     node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                 node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type     Reason     Age   From               Message
  ----     ------     ----  ----               -------
  Normal   Scheduled  12s   default-scheduler  Successfully assigned default/liveness
to minikube
  Normal   Pulled     11s   kubelet, minikube  Container image "startkubernetes/app-
health:0.1.0" already present on machine
  Normal   Created    11s   kubelet, minikube  Created container web
  Normal   Started    11s   kubelet, minikube  Started container web
  Warning  Unhealthy  0s    kubelet, minikube  Liveness probe failed: HTTP probe
failed with statuscode: 500
```

The Pod is healthy for the first 10 seconds or so. After that, looking at the last line in the output, the liveness probe fails, and Kubernetes restarts the container. Once the container restarts, the same story repeats - the liveness probe is healthy for 10 seconds, and then fails.

You can see the number of times Kubernetes restarted the container if you run the `kubectl get po` command and look at the `RESTARTS` column:

```
$ kubectl get po
NAME       READY   STATUS    RESTARTS   AGE
liveness   1/1     Running   5          2m31s
```

In addition to just specifying the path and port for the HTTP check, you can also use the `httpHeaders` field to specify the headers you want the probe to use. For example, if you want to add the `Host` header to the call, you could do it like this:

ch7/liveness-headers.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-headers
  labels:
    app.kubernetes.io/name: liveness-headers
spec:
  containers:
    - name: web
      image: startkubernetes/app-health:0.1.0
      imagePullPolicy: Always
      ports:
        - containerPort: 3000
      livenessProbe:
        httpGet:
          path: /healthz
          port: 3000
          httpHeaders:
            - name: Host
              value: liveness-host
        initialDelaySeconds: 3
        periodSeconds: 1
```

Save the above YAML in `liveness-headers.yaml` and create the Pod using `kubectl apply -f liveness-headers.yaml`. The App Health application automatically logs the headers from incoming requests, so after the Pod starts, you can look at the logs:

```
$ kubectl logs liveness-headers

> app-health@0.1.0 start /app
> node server.js

appHealth running on port 3000.
{"host":"liveness-host","user-agent":"kube-probe/1.18","accept-encoding":"gzip"
,"connection":"close"}
GET /healthz 200 7.083 ms - 2
```

Notice the `host` header value is set to `liveness-host`, just like we configured it in the probe configuration.

In addition to the fields mentioned above, you can set a couple of other fields to control the probes' behavior. I am mentioning the additional fields in the table below.

Table 5. Fields

Field name	Description
`timeoutSeconds`	The number of seconds after which the probe times out. The default value is set to 1 second. Consider increasing this value if your health check probe depends on other services (i.e., you're invoking other services to determine the health of your current service).
`successThreshold`	The number of consecutive successes for the probe to be considered successful again after having failed. For example, if set to 5, the health probe needs to succeed five times in a row after a failure to be considered successful. The default and minimum value are 1. For the liveness probe, this value must to be set to 1.
`failureThreshold`	The number of times probe is retried in case of a failure. If the probe is still failing after this threshold, the container is either restarted when using the liveness probe or marked as unhealthy if using the readiness probe. The default value is 3. The minimum value is 1.

Command Probe

The second mechanism is a command probe. With the comman probe, Kubernetes runs a command inside of your container to determine if the container is healthy or not. If the command returns with exit code 0, Kubernetes considers the container healthy. If exit code is different from 0, Kubernetes considers the container unhealthy.

You can use the command probe when you can't run an HTTP probe. Let's consider an example where your REST API running in the container requires an Authorization header. You can provide header to HTTP probe. However, you can't specify the authorization header value in the YAML. What you could do in this case is to mount the authorization token secret inside the container and then run `curl` as part of the command probe.

Here's how you can provide a command to execute to the liveness probe:

```
...
livenessProbe:
  exec:
    command:
    - curl
    - --fail
    - -u $(USER)
    - -p $(PASSWORD)
    - localhost:3000/healthz
initialDelaySeconds: 30
periodSeconds: 1
...
```

We are using `localhost` because the command will run inside the same container your application is running in.

TCP Probe

With the TCP probe, Kubernetes tries to establish a TCP connection on the specified port. The configuration is almost identical to the HTTP probe. Instead of using the `httpGet` field, you use `tcpSocket` field, like this:

```
...
livenessProbe:
  tcpSocket:
    port: 3000
  initialDelaySeconds: 10
  periodSeconds: 5
```

The above `tcpSocket` probe will try to open a socket on port `3000`, 15 seconds after the container starts. An excellent example of the TCP probe would be a gRPC service.

Application Startup probe

When you're dealing with applications that can take longer to start up, it might be tricky to set up a proper liveness check that's going to work both for the startup of the container and during the lifetime of the container. For that purpose, you can use the **startup probe** that uses the probe method (HTTP, TCP, or command), but with a higher failure threshold. The higher threshold is for cases where the application takes a long time to start.

```
livenessProbe:
  httpGet:
    path: /healthz
    port: 3000
  periodSeconds: 4
startupProbe:
  httpGet:
    path: /healthz
    port: 3000
  failureThreshold: 30
  periodSeconds: 4
```

Based on the startup probe's above settings, the container will have a maximum of 2 minutes (30 from the failure threshold setting multiplied by 4 seconds from the `periodSeconds` field) to finish the startup. The formula for calculating the maximum time is:

```
initialDelaySeconds + failureThreshold * periodSeconds
```

So if the probe is failing for the first 2 minutes, Kubernetes won't restart it yet. Once 2 minutes pass, the startup probe will be considered as failed.

If the application starts up within the 2 minutes, the liveness probe will take over and ensure the container stays alive.

Application Readiness probe

You can use the **readiness probe** to determine when your application is ready to start receiving traffic. In some cases, applications might take a while to startup. That could be because they depend on external services for the startup or loading a more considerable amount of data that takes time. During this time, you don't want to restart the container or send any requests to the container either.

If the readiness probe determines that the container is not ready yet, Kubernetes will mark the Pod as unhealthy and won't send any traffic. It will not restart the container; rather, it will invoke the probe based on the settings to determine when it's ready to receive traffic. Once the probe succeeds, the Pod is marked as healthy and can start receiving requests.

To demonstrate the readiness probe, we will use the `/readyz` endpoint. This endpoint will return HTTP 500 for the first 10 seconds (simulate the container doing some work). In those first 10 seconds, the container will be marked as unhealthy and won't receive any traffic.

The configuration for the readiness probe looks similar to the liveness probe. You only need to replace the `livenessProbe` key with `readinessProbe` key. Other configuration settings are also the same.

ch7/readiness.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: readiness
  labels:
    app.kubernetes.io/name: readiness
spec:
  containers:
    - name: web
      image: startkubernetes/app-health:0.1.0
      ports:
        - containerPort: 3000
      readinessProbe:
        httpGet:
          path: /readyz
          port: 3000
        initialDelaySeconds: 3
        periodSeconds: 1
```

Save the above YAML in `readiness.yaml` and create the Pod with `kubectl apply -f readiness.yaml`.

If you describe the Pod, you will notice that under the `Ready` and `ContainersReady` values under the `Conditions` section:

```
$ kubectl describe po readiness
...
Conditions:
  Type              Status
  Initialized       True
  Ready             False
  ContainersReady   False
  PodScheduled      True
...
```

If you re-run the describe command after 10 seconds, both values for `Ready` and `ContainersReady` will change to `True`, indicating the Pod and containers are ready:

```
$ kubectl describe po readiness
...
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
...
```

Similarly, if you look at the logs when the Pod starts, you will notice it is returning HTTP 500, and after 10 seconds, it starts returning HTTP 200:

```
$ kubectl logs readiness

> app-health@0.1.0 start /app
> node server.js

appHealth running on port 3000.
{"host":"172.17.0.2:3000","user-agent":"kube-probe/1.18","accept-encoding":"gzip"
,"connection":"close"}
GET /readyz 500 7.179 ms - 21
{"host":"172.17.0.2:3000","user-agent":"kube-probe/1.18","accept-encoding":"gzip"
,"connection":"close"}
...
GET /readyz 200 0.279 ms - 2
{"host":"172.17.0.2:3000","user-agent":"kube-probe/1.18","accept-encoding":"gzip"
,"connection":"close"}
GET /readyz 200 0.862 ms - 2
...
```

You can use the readiness probe together with the liveness probe to health and ready-check the same containers. Using both probes, you can ensure Kubernetes restarts the failed containers when they are unheathy and allows them to receive the requests only when they are ready to receive traffic.

Security in Kubernetes

What are service accounts?

To access the Kubernetes API server, you need an authentication token. The processes running inside your containers use a service account to authenticate with the Kubernetes API server. Just like a user account represents a human, a service account represents and provides an identity to your Pods.

Each Pod you create has a service account assigned to it. Even if you don't explicitly provide the service account name, Kubernetes sets the default service account to your Pods. This default service account (called `default`) is in every namespace in Kubernetes, which means that the account is bound to the namespace it lives in. You can try creating a new namespace and listing the service accounts (e.g. `kubectl get serviceaccount`), and you'll see there's a service account called `default`. A Pod can only use one service account, and they both need to be in the same namespace. However, multiple Pods can share the same service account.

I will be using Minikube in this section. Let's start by creating a Pod to see where the service account is specified:

```
$ kubectl run simple-pod --image=nginx
```

You can use `-o yaml` to get the YAML representation of the Pod, like this: `kubectl get po simple-pod -o yaml`. If you look through the output, you will notice the following line:

```
serviceAccountName: default
```

Even though we haven't explicitly set the service account name, Kubernetes assigned the `default` service account to the Pod.

Let's run `kubectl describe serviceaccount default` or `kubectl describe sa default` to see the details of the default service account:

```
$ kubectl describe sa default
Name:                default
Namespace:           default
Labels:              <none>
Annotations:         <none>
Image pull secrets:  <none>
Mountable secrets:   default-token-qjdzv
Tokens:              default-token-qjdzv
Events:              <none>
```

Like any other resource, the service account has the name, namespace, and labels and annotations. Additionally, it has the `Image pull secrets`, `Mountable secrets`, and `Tokens` sections. If you defined

image pull secrets (these are used by Pods to pull the images from private registries), Kubernetes will automatically added them to all Pods that are using this service account. The mountable secrets field is specifying the secrets that can be mounted by the Pods using this service account. Lastly, the tokens fields list all authentication tokens in the service account. Kubernetes automatically mounts the first token inside the container.

Here's how the YAML representation of the service account looks like, if you run `kubectl get sa default -o yaml`:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: "2020-09-03T22:09:47Z"
  name: default
  namespace: default
  resourceVersion: "320"
  selfLink: /api/v1/namespaces/default/serviceaccounts/default
  uid: 7395b383-6f90-4fef-946c-87bcc3211891
secrets:
- name: default-token-qjdzv
```

The mountable secret from the account (the `default-token-qjdzv` above) gets mounted automatically in each Pod under `/var/run/secrets/kubernetes.io/serviceaccount`. The Secret stores these three values:

- the authentication token (mounted as `token` file)
- namespace name (mounted as `namespace` file)
- public certificate authority of the API server (mounted as `ca.crt` file)

Kubernetes can use different authentication mechanisms or plugins - client certificates, bearer tokens, authenticating proxy, or HTTP basic auth - to authenticate API requests. Whenever the API server receives a request, the request goes through all configured plugins, and the plugins try to determine the requests' sender. The plugins try to extract the caller's identity from the request, and the first plugin that's able to extract that information will send it to the API server. At this point, the request will continue to the authorization phase.

The identity consists of the following attributes:

- Username (a string that identifies the user)
- User ID (UID) (a string that identifies the user - more unique than the username)
- Groups: a set of strings that indicats the groups user belongs (e.g., `developers`, `system:admins`, etc.)
- Other extra fields

The service account usernames use the following format:

```
system:serviceaccount:[namespace]:[service-account-name]
```

The API server uses this username to determine if the caller (the process inside the container, inside your Pod) can perform the desired action (for example, getting the Pods list from the API server).

Each service account can belong to one or more groups. These groups are used to grant permissions to multiple users at the same time. For example, a group called `administrators` grants administrative privileges to all accounts of that group. These groups are just simple, unique strings - `admins`, `developers`, etc.

Let's go back to our `simple-pod` and invoke the Kubernetes API using the service account token. First, we will get a shell inside the container:

```
$ kubectl exec -it simple-pod -- /bin/bash
root@simple-pod:/#
```

We will store the auth token in the `TOKEN` variable, so we can use it when invoking the API:

```
$ TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
```

If you're curious about the encoded information in the token, you can head to `https://jwt.io` and decode your token to look at the payload. Here's how the payload for my token looks like:

```
{
  "iss": "kubernetes/serviceaccount",
  "kubernetes.io/serviceaccount/namespace": "default",
  "kubernetes.io/serviceaccount/secret.name": "default-token-qjdzv",
  "kubernetes.io/serviceaccount/service-account.name": "default",
  "kubernetes.io/serviceaccount/service-account.uid": "7395b383-6f90-4fef-946c-
87bcc3211891",
  "sub": "system:serviceaccount:default:default"
}
```

We will use the `TOKEN` as the bearer token and invoke the Kubernetes API. The Kubernetes API is exposed through the Service called `kubernetes` in the default namespace.

Here's how we can try and invoke the API from within the container:

```
root@simple-pod:#/ curl -sSk -H "Authorization: Bearer $TOKEN"
https://kubernetes.default:443/api
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "192.168.64.9:8443"
    }
  ]
}
```

If we tried to access the namespaces or Pods, we would get a "403 Forbidden" response back. That's because the default service account doesn't have any permissions - Kubernetes treats the default service account as an unauthenticated user.

Here's the response we get back if we try to get the information about the `simple-pod` Pod:

```
root@simple-pod:/# curl -sSk -H "Authorization: Bearer $TOKEN"
https://kubernetes.default:443/api/v1/namespaces/default/pods/simple-pod
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {

  },
  "status": "Failure",
  "message": "pods \"simple-pod\" is forbidden: User
\"system:serviceaccount:default:default\" cannot get resource \"pods\" in API group
\"\" in the namespace \"default\"",
  "reason": "Forbidden",
  "details": {
    "name": "simple-pod",
    "kind": "pods"
  },
  "code": 403
}
```

The detailed message says that the user `system:serviceaccount:default:default` (note the first `default` is the namespace name, and the second one is the service account name) cannot get the Pods from the `default` namespace. What we could do is add the default service account to a group that has more permissions. However, that would be a horrible idea, because if you remember, the `default` service account gets automatically assigned to each Pod if the Pod doesn't specify its service account. A much better practice is to create a new service account and explicitly set it for the Pod.

You can exit the container by typing `exit` and then deleting the Pod by running `kubectl delete po simple-pod`.

Using Role-Based Access Control (RBAC)

Kubernetes manages the authorization (i.e., regulating access to resources based on roles) through the role based access control or RBAC for short.

Using RBAC, you can dynamically configure policies and control access to the Kubernetes resources. There are four Kubernetes resources related to RBAC - Role (and ClusterRole) and RoleBinding (and ClusterRoleBinding).

Roles

The Role resource contains the rules that represent a set of permissions. The Role is defined on the namespace level, and rules only apply within that namespace. The second resource is the ClusterRole resource, and this resource can be used to apply the permissions cluster-wide (across all namespaces).

The other couple of use cases for ClusterRole are if you are defining rules for cluster-scoped resources. An example of a cluster-scope resource is cluster nodes. You could apply rules for namespaced resources across all namespaces, such as all Services or all Pods in the cluster. Similarly, you would use a ClusterRole if you are defining permissions for non-resource endpoints (e.g., `/healthz`).

If we continue with the previous example where we tried to list the Pods from within the container, here's how we could define a Role called `pod-reader` that grants read access to Pods:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

The `apiGroups` indicates which core API group the role applies to. The empty value means that the rules apply to the core API group. You can get the list of all API groups by running `kubectl api-resources -o wide`. The `APIGROUP` column shows the name, and the `VERBS` column will show you the supported verbs for a particular API group. For example, the `deployments` fall under the `apps` API group, and the following verbs are supported: `[create delete deletecollection get list patch update watch]`. Since Pods are part of the core API, we don't have to specify an `apiGroup` for them.

The `resources` field holds the list of resources the rules apply to. In the above case, we are setting it to `pods`. If we wanted to apply the rules to any other resource from the core API group, we could add it to the list. For example, to apply the rules to Services and ConfigMaps, the `resources` field

value would be :`["pods", "services", "configmaps"]`.

The array in the `verbs` field maps to the HTTP verbs used when making the API's request. The table below shows how the HTTP verbs map to the verbs you can use in the Role (or ClusterRole) resource.

Table 6. HTTP Verbs

HTTP verb	Verb in Role
POST	create
GET, HEAD	get, list, watch
PUT	update
PATCH	patch
DELETE	delete, deletecollection

There's also the `resourceNames` field that you can set under the rules if you want to specify the exact resource the rules should apply to. The example below shows how you could create a rule that only applies to Deployment called `my-deployment`:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: deployment-role
rules:
- apiGroups: ["apps"]
  resources: ["deployments"]
  resourceName: ["my-deployment"]
  verbs: ["get"]
```

Once you have created the roles, you have to use one of the binding resources to grant the permissions (from the role) to the users.

Bindings

You use the RoleBinding and ClusterRoleBinding resources to bind the permissions from the Role or ClusterRole resource to the users. The users, in this case, could be groups or service account. Like with the Role and ClusterRole before, the RoleBinding grants the permissions within a specific namespace, whereas the ClusterRoleBinding grants the permissions cluster-wide.

Here's how we could create a RoleBinding that binds the `pod-reader` Role to a service account called `pod-reader-sa`:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: ServiceAccount
  name: pod-reader-sa
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Under `subjects`, you can define multiple subjects - these can either be service accounts, users, or groups. The binding assigns these subjects the role that's referenced under the `roleRef` field.

Now that we have a basic understanding of how RBAC and service accounts work together let's create a service account called `pod-reader-sa` service account, a Role, and a RoleBinding that grants the permission to read the Pods. We will then create the same Pod as we did at the beginning of this chapter and assign the `pod-reader-sa` service account to it.

First, let's create the service account:

ch8/pod-reader-sa.yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: pod-reader-sa
```

Save the above to `pod-reader.sa.yaml` and create the service account using `kubectl apply -f pod-reader-sa.yaml`.

ch8/pod-reader-role.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["list"]
```

NOTE Another way to create Roles is by using `kubectl` directly. For example `kubectl create role pod-reader --verb=list --resource=pods -n default`.

Next, we create the Role that only allows listing the Pods. This Role will prevent us from retrieving the Pod details, for example (that's the `get` verb). Save the above to `pod-reader-role.yaml` and create it with `kubectl apply -f pod-reader-role.yaml`.

To assign the Role to the service account we need to create the RoleBinding:

ch8/pod-reader-role-binding.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
  - kind: ServiceAccount
    name: pod-reader-sa
    namespace: default
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

NOTE To bind a Role to a subject with Kubernetes CLI, use `kubectl create rolebinding read-pods --role=pod-reader --serviceaccount=default:pod-reader-sa -n default`

You can describe the role binding, and it should look similar to this:

```
$ kubectl describe rolebinding read-pods
Name:         read-pods
Labels:       <none>
Annotations:  <none>
Role:
  Kind:  Role
  Name:  pod-reader
Subjects:
  Kind            Name           Namespace
  ----            ----           ---------
  ServiceAccount  pod-reader-sa  default
```

The figure below shows how the connection between the service account, Role, and the RoleBinding.

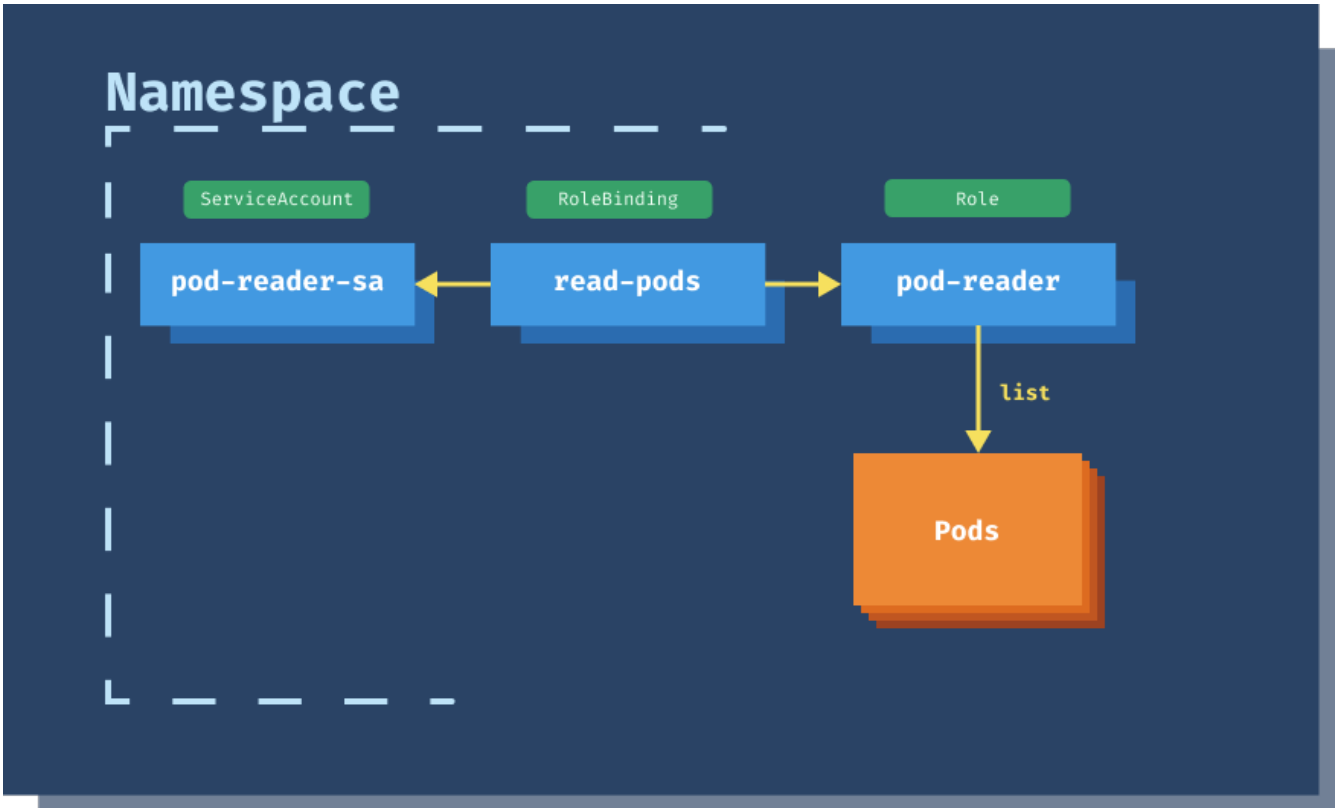


Figure 41. Service Account, Role, and RoleBinding

Finally, we can create the Pod with the `serviceAccountName` field set to `pod-reader-sa`.

ch8/pod-with-sa.yaml

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: simple-pod
  name: simple-pod
spec:
  serviceAccountName: pod-reader-sa
  containers:
    - image: nginx
      name: simple-pod
```

Save the above YAML to `pod-with-sa.yaml` and create the Pod with `kubectl apply -f pod-with-sa.yaml`.

Let's get the shell inside the container and then try send a request to the Kubernetes API:

```
$ kubectl exec -it simple-pod -- /bin/bash
root@simple-pod:/# TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
root@simple-pod:/# curl -sSk -H "Authorization: Bearer $TOKEN"
https://kubernetes.default:443/api/v1/namespaces/default/pods
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/namespaces/default/pods",
    "resourceVersion": "7768"
  },
  "items": [
    {
      "metadata": {
        "name": "simple-pod",
        "namespace": "default",
        "selfLink": "/api/v1/namespaces/default/pods/simple-pod",
...
```

This time the request worked! If you try to get the details about the `simple-pod` for example, the request fails as we don't have the `get` verb in the role. Similarly, if you try to list any other resources, the request will fail as well.

Security contexts

The way to apply security-related configuration is through the `securityContext` field. The objects that describe the security context are called PodSecurityContext and SecurityContext. You can set the security context at the Pod level or the container level. If you set the same values at both levels, the container `securityContext` value will take precedence.

Privileged containers

If you set the `privileged` setting to `true`, your container will run in a privileged mode, which nearly equals `root` on the container host. The default value is false. You would use this if your container needs to manipulate the network stack or access hardware devices on the host. In general, you shouldn't be running your container in the privileged mode at all.

```
...
securityContext:
  privileged: true
....
```

User (UID) and group ID (GID)

Using the `runAsUser` and `runAsGroup` fields, you can specify the UID or GID the containers' processes will use to execute. Let's consider the following snippet:

```
....
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
...
```

All processes in the container that uses the above security context will run as user ID 1000. The group ID for all processes in the containers is 3000. With `fsGroup` we are specifying that all processes will also be apart of the supplementary group with ID 2000.

Let's take a look at an example:

ch8/pod-ids.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-pod
  labels:
    app.kubernetes.io/name: hello
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  containers:
    - name: hello-container
      image: busybox
      command: ["sh", "-c", "sleep 3600"]
```

Save the above YAML to `pod-ids.yaml` and create the Pod with `kubectl apply -f pod-ids.yaml`. Let's get the terminal inside the Pod and run `ps` to look at the list of running processes:

```
$ kubectl exec -it hello-pod -- /bin/sh
/ $ ps
PID   USER     TIME  COMMAND
    1 1000      0:00 sleep 3600
   17 1000      0:00 /bin/sh
   27 1000      0:00 ps
/ $
```

Notice the ID in the `USER` column is 1000, just like we set it in the security context.

```
/ $ id
uid=1000 gid=3000 groups=2000
/ $
```

Similarly, if you run the `id` command, you will notice the `uid` set to `1000`, the `gid` set to `3000`, and supplemental group (`groups`) to `2000`.

Linux capabilities

Using the `capabilities` field, you can set a list of Linux capabilities you want to add or drop from the containers. Even though containers are running in isolated namespaces they don't have permissions to everything. Using these capabilities, you can fine-tune the permissions you want to grant to the containers.

For example, if you'd want your containers to use privileged ports (any port number smaller than 1024), you could use the `NET_BIND_SERVICE` capability. To add this capability to the container, you can specify it under the `capabilities` field under the container:

```
...
    securityContext:
      capabilities:
        add:
          - NET_BIND_SERVICE
```

Let's try setting this to a sample Pod:

ch8/add-cap-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-pod
  labels:
    app.kubernetes.io/name: hello
spec:
  containers:
    - name: hello-container
      image: debian
      command: ["sh", "-c", "sleep 3600"]
      securityContext:
        capabilities:
          add:
            - NET_BIND_SERVICE
```

NOTE We have updated the image name to `debian`, because it already has the `capsh` binary installed.

Once the container is up and running you can use the `capsh` command to look at the capabilities:

```
$ kubectl exec -it hello-pod -- capsh --print
Current: =
cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_se
tpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_se
tfcap+eip
Bounding set
=cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_s
etpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_s
etfcap
Securebits: 00/0x0/1'b0
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
uid=0(root)
gid=0(root)
groups=
```

You can find the `cap_net_bind_service` capability in the list. Also, notice all other capabilities that are available in the container by default. A good practice is to drop all capabilities first and then only add the capabilities your application need. To do that, you can use the `drop` field.

Delete the previous Pod with `kubectl delete po hello-pod` and let's create one that drops all capabilities:

ch8/drop-all-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-pod
  labels:
    app.kubernetes.io/name: hello
spec:
  containers:
    - name: hello-container
      image: debian
      command: ["sh", "-c", "sleep 3600"]
      securityContext:
        capabilities:
          drop:
            - all
```

If you run `capsh` this time, you will notice that all capabilities have been dropped:

```
$ kubectl exec -it hello-pod -- capsh --print
Current: =
Bounding set =
Securebits: 00/0x0/1'b0
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
uid=0(root)
gid=0(root)
groups=
```

The above capabilites are a good starting point for all your containers. The next step would be to dig deeper into individual capabilities your application might need and grant them. You can find the full reference and explanation of Linux capabilities here.

SELinux

Security-Enhanced Linux (SELinux) is a Linux kernel security module that allows you to have more control over who can access things in the system. SELinux is a labeling system, and every process, file, or a directory has a label. You can write policy rules that control access between labeled processes and labeled objects, and the OS kernel will enforce these rules. The detailed explanation of SELinux and how it works is out of scope for this book. Here are a couple of resources you can use to get more familiar with the SELinux:

- Your visual how-to guide for SELinux policy enforcement [https://opensource.com/business/13/11/selinux-policy-guide] (article by Daniel Walsh)
- Security-Enhanced Linux for mere mortals [https://www.youtube.com/watch?v=_WOKRaM-HI4] (talk by Thomas Cameron)

To assign SELinux labels to a container, you can use the `seLinuxOptions` field in the `securitySection`. Note that you can apply the `seLinuxOptions` at both Pod and container level.

```
...
securityContext:
  seLinuxOptions:
    level: "s0:c123,c456"
```

AppArmor

AppArmor is another Linux kernel security module. In allows you to confine applications to a limited set of resources and it's used together with the traditional permissions (users and groups). You can create AppArmor profiles that restrict actions such as reading, writing, or executing specific files or limiting access to networks. Default AppArmor profiles limit access to `/proc` and `/sys` locations, and this gives you a way to isolate the containers from the node they are running on.

Note that AppArmor is not available on all operating systems. Suppose you're running your Kubernetes cluster on Ubuntu or SUSE Linux (or running Minikube with the KVM driver on those

operating systems). In that case, the AppArmor is supported and enabled by default. To quickly check if the AppArmor is enabled, run the following command:

```
$ cat /sysmodule/apparmor/parameters/enabled
Y
```

The AppArmor profiles are defined per-container through an annotation with the following format:

```
container.apparmor.security.beta.kubernetes.io/<container_name>: <profile_name>
```

Here's an earlier example of a Pod but this time with the AppArmor profile selected for the container:

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-pod
  labels:
    app.kubernetes.io/name: hello
  annotations:
    container.apparmor.security.beta.kubernetes.io/hello-container: runtime/default
spec:
  containers:
    - name: hello-container
      image: busybox
      command: ["sh", "-c", "echo Hello from my container! && sleep 3600"]
```

The above annotation applies the `runtime/default` AppArmor profile to the `hello-container`. If you don't have AppArmor enabled on your cluster nodes, you can't run any Pods with the AppArmor annotation. The Pods will have a `Blocked` status, like this:

```
$ kubectl get po
NAME        READY   STATUS    RESTARTS   AGE
hello-pod   0/1     Blocked   0          65s
```

If you get more details from the Pod using the `describe` command, you will notice the following message in the output:

```
...
Annotations:  container.apparmor.security.beta.kubernetes.io/hello-container:
runtime/default
Status:       Pending
Reason:       AppArmor
Message:      Cannot enforce AppArmor: AppArmor is not enabled on the host
...
```

To check which AppArmor profiles operating systems loads on your cluster nodes, you can look in the `/sys/kernel/security/apparmor/profiles` file. To do that, you will have to SSH into your cluster nodes. Enabling SSH access to the nodes depends on the cloud provider your cluster is hosted in. If you're running Linux or Minikube with KVM driver, you're in luck because you can directly look at that `/sys/kernel/security/apparmor/profiles` folder.

Here's a sample output from one of the nodes on the cloud-managed cluster I used that shows all AppArmor profiles on the node:

```
$ sudo cat /sys/kernel/security/apparmor/profiles
docker-default (enforce)
/usr/lib/snapd/snap-confine (enforce)
/usr/lib/snapd/snap-confine//mount-namespace-capture-helper (enforce)
/usr/sbin/tcpdump (enforce)
/usr/lib/connman/scripts/dhclient-script (enforce)
/usr/lib/NetworkManager/nm-dhcp-helper (enforce)
/usr/lib/NetworkManager/nm-dhcp-client.action (enforce)
/sbin/dhclient (enforce)
/usr/lib/lxd/lxd-bridge-proxy (enforce)
/usr/bin/lxc-start (enforce)
lxc-container-default-with-nesting (enforce)
lxc-container-default-with-mounting (enforce)
lxc-container-default-cgns (enforce)
lxc-container-default (enforce)
```

Notice the word `enforce` in the parenthesis? AppArmor can run in two modes: `enforce` and `complain`. If the profile is loaded in enforcement mode, the policies in the profile will be enforced. However, if you load it in the `complain` mode, the policy will not be enforced. Instead policy violations will be reported (via syslog or auditd).

Let's create a sample policy and see how it looks like when it's enforced on a container.

deny-all.profile

```
#include <tunables/global>
profile apparmor-example-deny-all flags=(attach_disconnected) {
  #include <abstractions/base>

  file,
  # Deny all file reads/writes.
  deny /** rw,
}
```

The sample policy called `apparmor-example-deny-all` from the listing above denies all reads and writes to files. To add the profile, you can use `apparmor_parser` command:

```
$ sudo apparmor_parser deny-all.profile
```

NOTE	There won't be any output from the command if the AppArmor profile is loaded correctly.

If you look at all loaded profiles again, you'll notice our profile in the list:

```
$ sudo cat /sys/kernel/security/apparmor/profiles
apparmor-example-deny-all (enforce)
....
```

Let's see this AppArmor profile in action. We prefix the AppArmor profile name with `localhost` and set it the annotation:

ch8/pod-apparmor.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-pod
  labels:
    app.kubernetes.io/name: hello
  annotations:
    container.apparmor.security.beta.kubernetes.io/hello-container:
localhost/apparmor-example-deny-all
spec:
  containers:
    - name: hello-container
      image: busybox
      command: ["sh", "-c", "echo Hello from my container! && sleep 3600"]
```

The above annotation applies our AppArmor profile to the container called `hello-container`. Save the above YAML to `pod-apparmor.yaml` and create the Pod using `kubectl apply -f pod-armor.yaml`.

Once the Pod is running, let's try creating a file using the `touch` command:

```
$ kubectl exec -it hello-pod -- touch hello.txt
touch: hello.txt: Permission denied
command terminated with exit code 1
```

As expected, we get the "Permission denied" messsage because we applied the AppArmor profile to it. If we'd execute the same command against a Pod without the AppArmor profile, we'd be able to create the file.

One tool I'd like to mention is the profile generation utility called `aa-genprof`. You can install it by running `sudo apt install apparmor-utils`. Using this tool, you can automatically generate an AppArmor profile for your application.

Let's create a trivial Go application that writes a string to a file in the `/tmp` folder:

ch8/main.go

```
package main

import (
    "io/ioutil"
)

func main() {
    d1 := []byte("hello")
    err := ioutil.WriteFile("/tmp/file1.txt", d1, 0644)
    if err != nil {
        panic(err)
    }
}
```

To run the above app, run `go run main.go`. There won't be any output, but you can check that the app created the `/tmp/file1.txt` file. Let's also build the binary, and we will use the `aa-genprof` to generate the AppArmor profile for it:

```
$ go build main.go
$ sudo aa-genprof main
~/apparmor-test$ sudo aa-genprof main
Writing updated profile for /home/user/apparmor-test/main.
Setting /home/user/apparmor-test/main to complain mode.
....
Profiling: /home/user/apparmor-test/main

[(S)can system log for AppArmor events] / (F)inish
```

Now you have to run the binary from a different terminal window for the tool to generate the profile. Once you've done that, press the `S` key, so the tool reads the AppArmor events. This is what

you should see:

```
...
Reading log entries from /var/log/syslog.
Complain-mode changes:

Profile:  /home/user/apparmor-test/main
Path:     /tmp/file1.txt
Mode:     w
Severity: unknown

  [1 - /tmp/file1.txt]
[(A)llow] / (D)eny / (I)gnore / (G)lob / Glob with (E)xtension / (N)ew / Abo(r)t / (F
)inish / (M)ore
```

You can then decide if you want to Allow this action, Deny it, or pick any other available options. After you've decided what to do, you will be prompted to save the profile changes. You can now exit the tool (use the `F` key) and look at the generated profile. All profiles are stored under `/etc/apparmor.d` folder. Here's how the profile looks like if you've selected `Allow` action for writing to the `/tmp/file1.txt` file:

```
$ sudo cat /etc/apparmor.d/home.user.apparmor-test.main
#include <tunables/global>

/home/user/apparmor-test/main {
  #include <abstractions/base>

  /home/user/apparmor-test/main mr,
  /tmp/file1.txt w,
}
```

The profile would look similar if we'd deny the action:

```
  deny /tmp/file1.txt w,
```

Seccomp

Seccomp or secure computing mode allows a process to transition into a protected state where it can't make any system calls, except to `exit()`, `sigreturn()`, `read()` and `write()` to already-open file descriptors. In case the process tries to call any other system calls, the kernel will terminate it. You can provide the Secomp profile either at the Pod or container level. If you provide it at both levels, the options at the container level will take precedence.

You can define the seccomp profile name using the `seccompProfile` field, like this:

```
...
securityContext:
  seccompProfile:
    type: RuntimeDefault
```

There are three valid options for the type: `RuntimeDefault`, `Unconfined`, and `Localhost`. Additionally, you can also create your Seccomp profiles. You can reference them using the `Localhost` type and and a `localhostProfile` field pointing to the file in a kubelet root directory (e.g. `[kubelet]/seccomp/profiles/my-profile.json`). For example:

```
...
securityContext:
  seccompProfile:
    type: Localhost
    localhostProfile: profiles/my-profile.json
```

Pod security policies

With the PodSecurityPolicy API, you can define and manage all security-related fields in your Pods. For example, you can limit what can run on your cluster and with what level of privilege. For the PodSecurityPolicy to work, you need to enable corresponding admission controllers. The admission controller enforces the conditions defined in the PodSecurityPolicy.

In addition to the security settings we described previously, you can use the PodSecurityPolicy to control the following:

- Running privileged containers
- Using host namespaces
- Using the host network and ports
- Using volume types
- Using host filesystem
- Using user and group Ids of the container
- Restricting root privilege escalations
- Linux capabilities
- SELinux settings
- AppArmor profiles
- Seccomp profiles

If you want to try out the PodSecurityPolicy using Minikube, you have to make sure to start it with the `PodSecurityPolicy` admission controller and `pod-security-policy` addon enabled. Alternatively, you can use one of the cloud-managed clusters that have the PodSecurityPolicy enabled. Here's the command you can use to start the Minikube cluster with:

```
minikube start --extra-config=apiserver.enable-admission-plugins=PodSecurityPolicy
--addons=pod-security-policy
```

Once the cluster starts, you can run `kubectl get podsecuritypolicy` (or `psp`) to get the list of pod security policies defined in the cluster:

```
$ kubectl get podsecuritypolicy
NAME         PRIV    CAPS   SELINUX    RUNASUSER          FSGROUP     SUPGROUP
READONLYROOTFS   VOLUMES
privileged   true    *      RunAsAny   RunAsAny           RunAsAny    RunAsAny
false            *
restricted   false          RunAsAny   MustRunAsNonRoot   MustRunAs   MustRunAs
false            configMap,emptyDir,projected,secret,downwardAPI,persistentVolumeClaim
```

Installing the addon created a `privileged` and `restricted` PodSecurityPolicy. In addition to the policies, the addon created two ClusterRoles that grant the service account access to use these policies through the verb `use`. The two ClusterRoles are named `psp:privileged` and `psp:restricted`.

Here's how the privileged ClusterRole is defined:

```
$ kubectl describe clusterrole psp:privileged
Name:         psp:privileged
Labels:       addonmanager.kubernetes.io/mode=EnsureExists
Annotations:  <none>
PolicyRule:
  Resources                  Non-Resource URLs  Resource Names  Verbs
  ---------                  -----------------  --------------  -----
  podsecuritypolicies.policy  []                 [privileged]    [use]
```

The last resource that the addon created is the ClusterRoleBinding called `default:restricted`. This binding allows the `system:serviceaccounts` group access to the `psp-restricted` ClusterRole.

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: psp-restricted
subjects:
- kind: Group
  name: system:serviceaccounts
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: psp-restricted
  apiGroup: rbac.authorization.k8s.io
```

Let's create a Pod and see which policy gets applied to the Pod:

ch8/just-a-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: just-a-pod
spec:
  containers:
    - name: test
      image: busybox
      command: ["sh", "-c", "sleep 1h"]
```

Save the above YAML to `just-a-pod.yaml` and create the Pod using `kubectl apply -f just-a-pod.yaml`.

When the Pod starts, you can use the `-o yaml` to get the YAML and look for the `kubernetes.io/psp` annotation:

```
$ kubectl get po
NAME                                READY   STATUS                      RESTARTS
AGE
just-a-pod                          1/1     Running                     0
16s

$ kubectl get po just-a-pod -o yaml | grep kubernetes.io/psp
    kubernetes.io/psp: privileged
```

Kubernetes assigned the `privileged` pod security policy. Let's do another test, but this time we will create a Deployment instead of a Pod:

ch8/just-a-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: just-a-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test
  template:
    metadata:
      labels:
        app: test
    spec:
      containers:
        - name: test
          image: busybox
          command: ["sh", "-c", "sleep 1h"]
```

Save the above YAML to `just-a-deployment.yaml` and create the Deployment using `kubectl apply -f just-a-deployment.yaml`.

If you list the Pods, you will notice that the Pod is not created and the status is set to `CreateContainerConfigError`:

```
$ kubectl get po
NAME                                READY   STATUS                       RESTARTS
AGE
just-a-deployment-7fd6bd7964-pdhb8   0/1     CreateContainerConfigError   0
50s
just-a-pod                          1/1     Running                      0
5m21s
```

If you look at the errors in the events list, you will see a more detailed error:

```
$ kubectl get event | grep Error
9s          Warning   Failed              pod/just-a-deployment-7fd6bd7964-pdhb8
Error: container has runAsNonRoot and image will run as root
```

This error tells us that we need to provide the `runAsUser` setting in the security context. Why is that? The Pod template in the Deployment is identical to the Pod we created earlier.

When we created the Pod directly using `kubectl`, Kubernetes used our user credentials. Since you're the one who set up the cluster, you have cluster-admin privileges. When we created the Deployment, it was the Deployment and ReplicaSet controllers who created the Pods. In this case, Kubernetes used the ReplicaSet controllers' service account to create the Pods.

Let's check the policy that Kubernetes applied to the failing Pod:

```
$ kubectl get po just-a-deployment-7fd6bd7964-pdhb8 -o yaml | grep kubernetes.io/psp
    kubernetes.io/psp: restricted
```

Because we used a different service account, Kubernetes applied the `restricted` policy to the Pod. Per restricted policy we must define the `runAsNonRoot` setting and a couple of other settings:

```
 allowPrivilegeEscalation: false
  fsGroup:
    ranges:
    - max: 65535
      min: 1
    rule: MustRunAs
  requiredDropCapabilities:
  - ALL
  runAsUser:
    rule: MustRunAsNonRoot
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    ranges:
    - max: 65535
      min: 1
    rule: MustRunAs
```

The `MustRunAs` rule means that we **must** specify a value for those settings. Otherwise the Pod won't run. Let's update the Deployment and include these settings.

ch8/secure-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: just-a-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test
  template:
    metadata:
      labels:
        app: test
    spec:
      securityContext:
        runAsUser: 1000
        runAsGroup: 3000
        fsGroup: 2000
      containers:
        - name: test
          image: busybox
          command: ["sh", "-c", "sleep 1h"]
```

Save the above YAML to `secure-deployment.yaml` and create the Deployment using `kubectl apply -f secure-deployment.yaml`.

If you list the Pod this time, you will notice the Pod is running:

```
$ kubectl get po
NAME                                READY   STATUS    RESTARTS   AGE
just-a-deployment-7d996d5849-ph8hb   1/1     Running   0          3s
```

Network Policies

Using the NetworkPolicy resource, you can control the traffic flow for your applications in the cluster, at the IP address lever or port level (OSI layer 3 or 4).

NOTE Open Systems Interconnection model (OSI model) is a conceptual model that characterises and standardizes the communication functions, regardless of the underlying technology. For more information, see OCI model.

With the NetworkPolicy you can define how your Pod can communicate with various network entities over the cluster. There are three parts to defining the NetworkPolicy:

1. Select the Pods the policy applies to. You can do that using labels. For example, using `app=hello` applies the policy to all Pods with that label.

2. Decide if the policy applies for incoming (ingress) traffic, outgoing (egress) traffic, or both.
3. Define the ingress or egress rules by specifying IP blocks, ports, Pod selectors, or namespace selectors.

Here is a sample NetworkPolicy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: my-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: hello
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          owner: ricky
    - podSelector:
        matchLabels:
          version: v2
    ports:
    - protocol: TCP
      port: 8080
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 500
```

Let's break down the above YAML. The `podSelector` tells us that the policy applies to all Pods in the `default` namespace that have the `app: hello` label set. We are defining policy for both ingress and egress traffic.

The calls to the Pods policy applies to can be made from any IP within the CIDR block `172.17.0.0/16` (that's 65536 IP addresses, from 172.17.0.0 to 172.17.255.255), except for Pods with whose IP falls within the CIDR block 172.17.1.0/24 (256 IP addresses, from 172.17.1.0 to 172.17.1.255) to the port `8080`. Additionally, the calls to the Pods policy applies to can be coming from any Pod in the

namespace(s) with the label `owner: ricky` and any Pod from the `default` namespace, labeled `version: v2`.

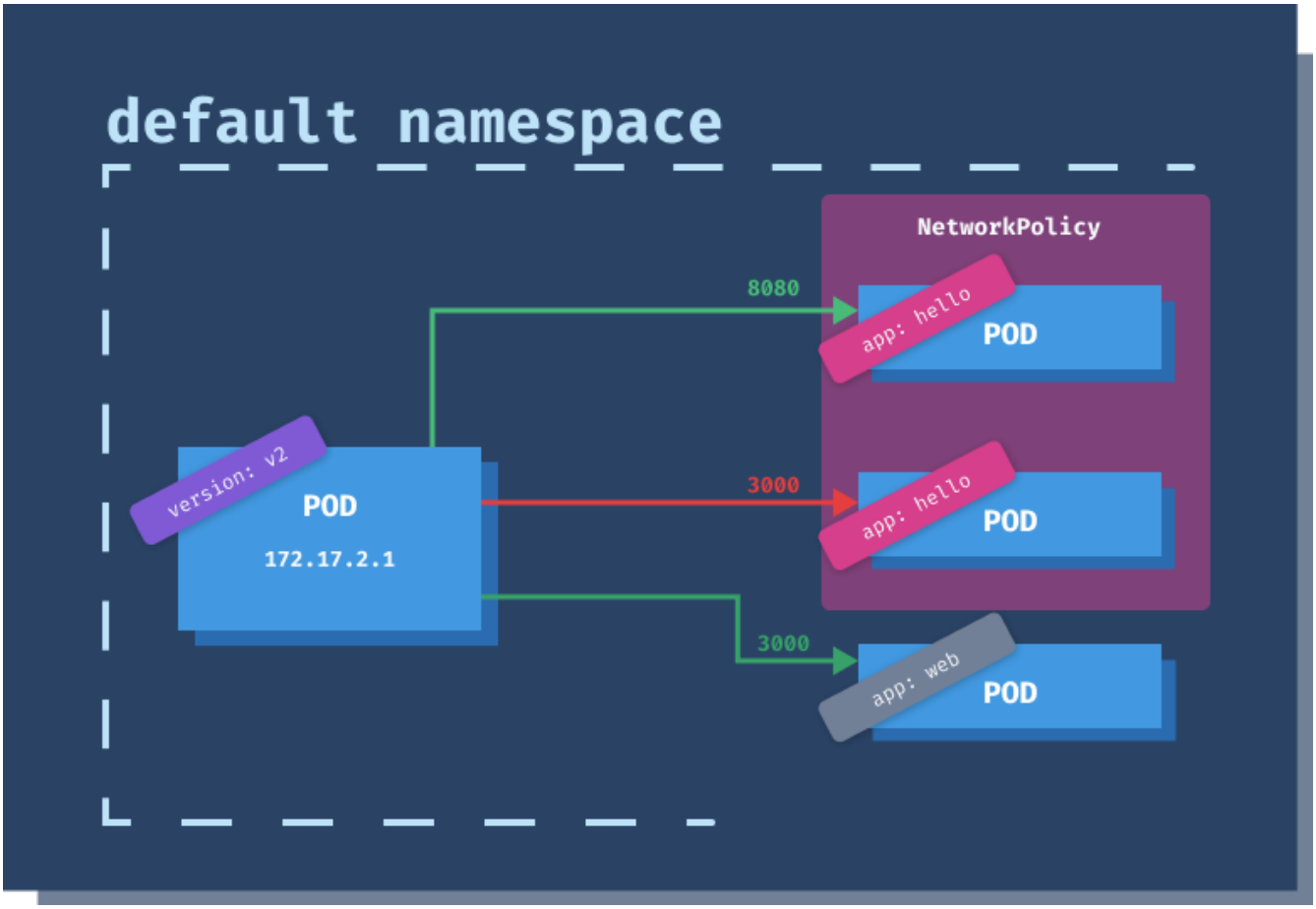


Figure 42. Ingress Network Policy

The egress policy specifies that Pods with the label `app: hello` in the `default` namespace can make calls to any IP within 10.0.0.0/24 (256 IP addresses, from 10.0.0.0, 10.0.0.255), but only to the port `5000`.

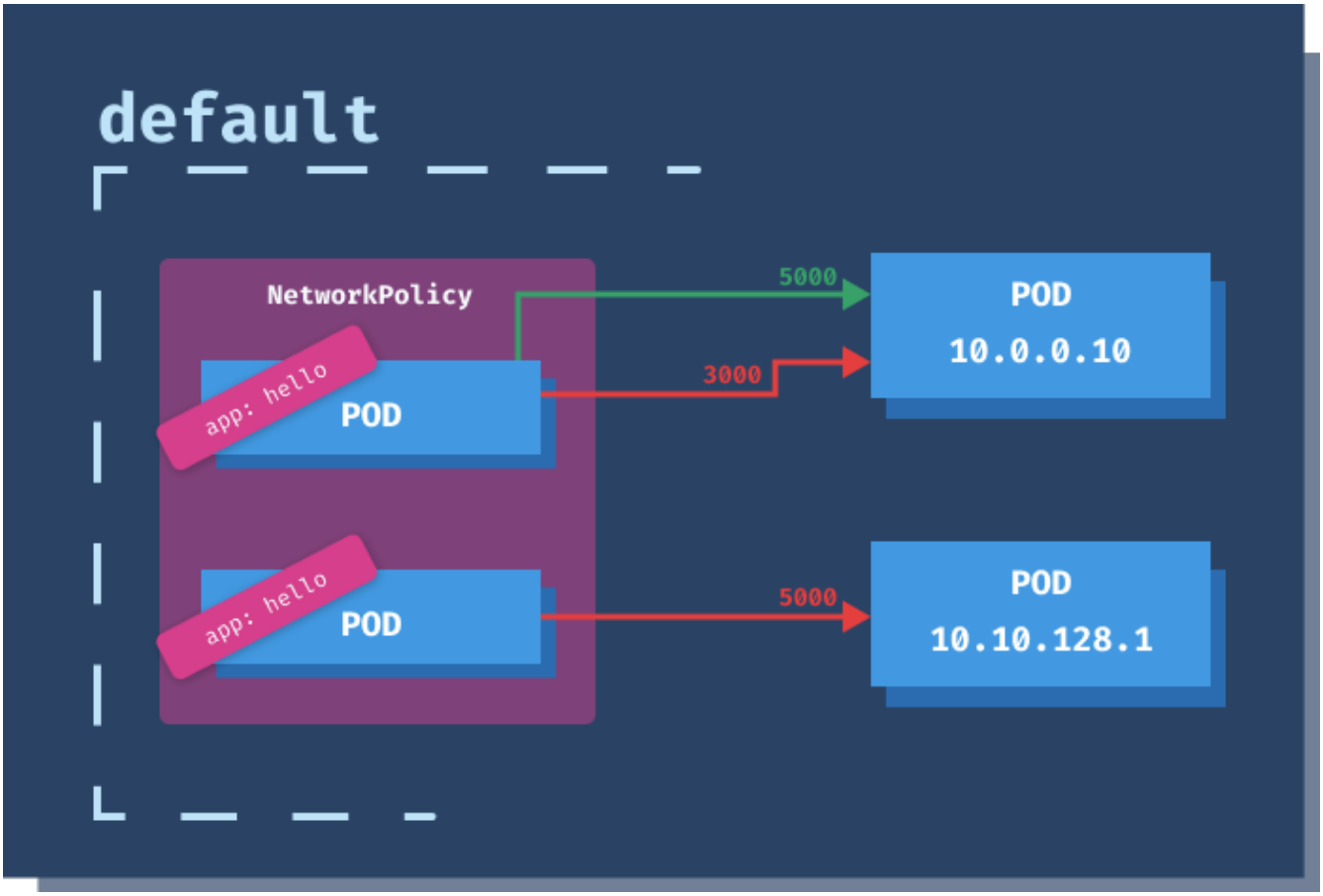


Figure 43. Egress Network Policy

The Pod and namespace selectors support **and** and **or** semantics. Let's consider the following snippet:

```
...
ingress:
- from:
  - namespaceSelector:
      matchLabels:
        user: ricky
    podSelector:
      matchLabels:
        app: website
...
```

The above snippet with a single element in the `from` array, includes all Pods with labels `app: website` from the namespace labeled `user: ricky`. This is the equivalent of **and** operator.

If you change the `podSelector` to be a separate element in the `from` array by adding `-`, you are using the **or** operator.

```
...
ingress:
- from:
  - namespaceSelector:
      matchLabels:
        user: ricky
  - podSelector:
      matchLabels:
        app: website
...
```

The above snippet includes all Pods labeled `app: website` or all Pods from the namespace with the label `user: ricky`.

Installing Cilium

Network policies are implemented (and rules enforce) through network plugins. If you don't install a network plugin, the policies won't have any effect.

I will use the Cilium plugin and install it on top of Minikube. You could also use a different plugin, such as Calico.

If you already have Minikube running, you will have to stop and delete the cluster (or create a separate one). You will have to start Minikube wit the `cni` flag for the Cilium to work correctly:

```
$ minikube start --network-plugin=cni
```

Once Minikube starts, you can install Cilium.

```
$ kubectl create -f
https://raw.githubusercontent.com/cilium/cilium/1.8.3/install/kubernetes/quick-
install.yaml
all/kubernetes/quick-install.yaml
serviceaccount/cilium created
serviceaccount/cilium-operator created
configmap/cilium-config created
clusterrole.rbac.authorization.k8s.io/cilium created
clusterrole.rbac.authorization.k8s.io/cilium-operator created
clusterrolebinding.rbac.authorization.k8s.io/cilium created
clusterrolebinding.rbac.authorization.k8s.io/cilium-operator created
daemonset.apps/cilium created
deployment.apps/cilium-operator created
```

Cilium is installed in `kube-system` namespace, so you can run `kubectl get po -n kube-system` and wait until the Cilium Pods are up and running.

Let's look at an example that demonstrates how to disable egress traffic from the Pods.

ch8/no-egress-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: no-egress-pod
  labels:
    app.kubernetes.io/name: hello
spec:
  containers:
    - name: container
      image: radial/busyboxplus:curl
      command: ["sh", "-c", "sleep 3600"]
```

Save the above YAML to `no-egress-pod.yaml` and create the Pod using `kubectl apply -f no-egress-pod.yaml`.

Once the Pod is running, let's try calling `google.com` using `curl`:

```
$ kubectl exec -it no-egress-pod -- curl -I -L google.com
HTTP/1.1 301 Moved Permanently
Location: http://www.google.com/
Content-Type: text/html; charset=UTF-8
Date: Thu, 24 Sep 2020 16:30:59 GMT
Expires: Sat, 24 Oct 2020 16:30:59 GMT
Cache-Control: public, max-age=2592000
Server: gws
Content-Length: 219
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN

HTTP/1.1 200 OK
...
```

The call completes successfully. Let's define a network policy that will prevent egress for Pods with `app.kubernetes.io/name: hello`:

ch8/deny-egress-hello.yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-egress
spec:
  podSelector:
    matchLabels:
      app.kubernetes.io/name: hello
  policyTypes:
    - Egress
```

If you run the same command this time, `curl` won't be able to resolve the host:

```
$ kubectl exec -it no-egress-pod -- curl -I -L google.com
curl: (6) Couldn't resolve host 'google.com'
```

Try running `kubectl edit pod no-egress-pod` and change the label value to `hello123`. Save the changes and then re-run the curl command. This time, the command works fine because we changed the Pod label, and the network policy does not apply to it anymore.

Common Network Policies

Let's look at a couple of scenarios and corresponding network policies.

Deny all egress traffic

Denies all egress traffic from the Pods in the namespace, and Pods cannot make any outgoing requests.

ch8/deny-all-egress.yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all-egress
spec:
  podSelector: {}
  policyTypes:
    - Egress
```

Deny all ingress traffic

Denies all ingress traffic, and Pods cannot receive any requests.

ch8/deny-all-ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all-ingress
spec:
  podSelector: {}
  policyTypes:
    - Ingress
```

Allow ingress traffic to specific Pods

Allow ingress to specific Pods, identified by a label.

ch8/pods-allow-all.yaml

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: pods-allow-all
spec:
  podSelector:
    matchLabels:
      app: my-app
  ingress:
    - {}
```

Deny ingress to specific Pods

Denies ingress to specific Pods, identified by a label.

ch8/pods-deny-all.yaml

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: pods-deny-all
spec:
  podSelector:
    matchLabels:
      app: my-app
  ingress: []
```

Restrict traffic to specific Pods

Allows traffic from certain Pods only. Allow traffic from `app: customers` to any frontend Pods (`role: frontend`) that are part of the same app (`app: customers`).

ch8/frontend-allow.yaml

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: frontend-allow
spec:
  podSelector:
    matchLabels:
      app: customers
      role: frontend
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: customers
```

Deny all traffic to and within a namespace

Denies all incoming traffic (no ingress rules defined) to all Pods (empty `podSelector`) in the `prod` namespace. Any calls from outside of the `default` namespace will be blocked and any calls between Pods in the same namespace.

ch8/prod-deny-all.yaml

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: prod-deny-all
  namespace: prod
spec:
  podSelector: {}
  ingress: []
```

Deny all traffic from other namespaces

Denies all traffic from other namespaces, coming to the Pods in the `prod` namespace. It matches all pods (empty `podSelector`) in the `prod` namespace and allows ingress from all Pods in the `prod` namespace, as the ingress `podSelector` is empty as well.

ch8/deny-other-namespaces.yaml

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-other-namespaces
  namespace: prod
spec:
  podSelector: {}
  ingress:
    - from:
        - podSelector: {}
```

Deny all egress traffic for specific Pods

Denies Pods labeled with `app: api` from making any external calls.

ch8/api-deny-egress.yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: api-deny-egress
spec:
  podSelector:
    matchLabels:
      app: api
  policyTypes:
    - Egress
  egress: []
```

Scaling and Resources

Scaling and autoscaling Pods

In Managing Pods with ReplicaSets, we discussed how to manually scale the Pods by increasing the value in the `replicas` field. Scaling where you're increasing the number of Pod instances is also called **horizontal scaling**.

The other type of scaling we will discuss in this chapter is **vertical scaling**, where you're increasing the number of resources (CPU and memory) containers can consume.

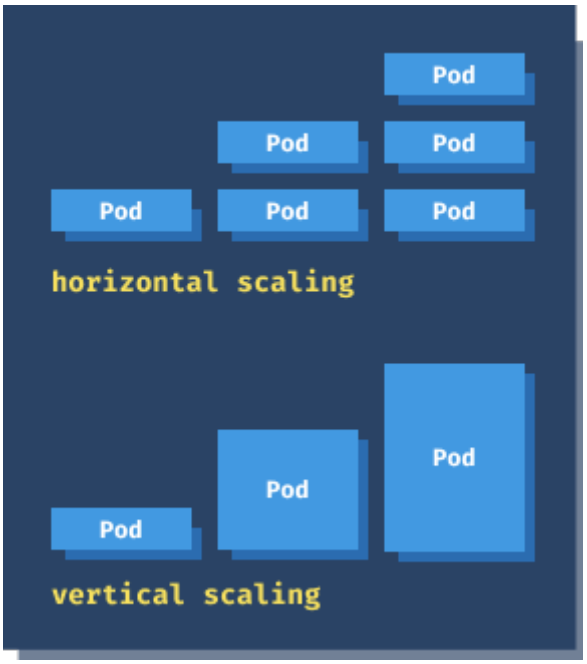


Figure 44. Scaling Pods

To gather the CPU and memory information, you need to install the **metrics server**. The metrics server aggregates the resource usage data across the cluster. Through the metrics API, you can get the number of resources used by nodes or Pods. There are two metrics reported: CPU and memory.

If you're using Minikube, you have to enable the `metrics-server` addon before using the HPA. To enable the `metrics-server` addon, run:

```
$ minikube addons enable metrics-server
  The 'metrics-server' addon is enabled
```

You can check metrics server was installed by looking at the `apiservice` and the Deployment called `metrics-server` that Minikube created in the `kube-system` namespace:

```
$ kubectl get apiservices | grep metrics
v1beta1.metrics.k8s.io                 kube-system/metrics-server   True        3m20s

$ kubectl get deployment metrics-server -n kube-system
NAME             READY   UP-TO-DATE   AVAILABLE   AGE
metrics-server   1/1     1            1           5m4s
```

With the metrics server installed, you can use the `kubectl top` command. The `top` command allows you to see the resource consumption for nodes and Pods.

```
$ kubectl top nodes
NAME       CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
minikube   437m         7%     594Mi           6%
```

Let's look at how you can specify the amount of resources your Pods need and how you can limit the resources they consume.

Resource requests and limits

For each container in a Pod, you can specify the resource **request**. Assuming your cluster has multiple nodes, Kubernetes uses this information to decide which node to place the Pod on. For example, if you set the memory request to 512 MiB and Kubernetes places the container on a node with 12 GiB, then the container can use up more memory (assuming no other Pods is running on that node).

To limit how much resources container consumes, you can specify the resource **limit**. The limit prevents the container from using more than you specified. If we continue with the previous example, if the request is set to 512 MiB and limit to 1024 MiB, regardless of how much memory is available on the node, the container will never use more than 1024 MiB. If the container's process tries to allocate more than the specified limit, the system kernel will terminate the process.

As mentioned earlier, there are two resources, CPU and memory. CPU represents the compute processing,and you can specify it in units of **Kubernetes CPUs**. The CPU resources are measured in **"cpu"** units. One "cpu" unit in Kubernetes is equivalent to 1 vCPU/Core for cloud providers and 1 hyperthread on bare-metal Intel processors. You can also request fractional units. For example, the value of `0.1` is equivalent to `100m` or 100 millicpu.

Memory is measured in bytes. The value can be expressed as an integer or a fixed-point number, using the following suffixes: E, P, T, G, M, K (exabyte, petabyte, terabyte, gigabyte, megabyte, and kilobyte). You can also use the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki (exbibyte, pebibyte, tebibyte, gibibyte, mebibyte, kibibyte). For example, 128 MiB is about 134 MB.

Both CPU and memory limits and requests can be specified under the `resources` field, like this:

ch9/memory-sample.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-sample
spec:
  containers:
    - name: stress
      image: polinux/stress
      resources:
        limits:
          memory: "200Mi"
        requests:
          memory: "100Mi"
      command: ["stress"]
      args: ["--vm", "1", "--vm-bytes", "150M", "--vm-hang", "1"]
```

Save the above YAML to `memory-sample.yaml` and create the Pod using `kubectl apply -f memory-sample.yaml`. If you look at the memory consumption with `top`, you will see the memory is at 150 Mi, which is above the requested (`100Mi`) and below the limit (`200Mi`):

```
$ kubectl top po memory-sample
NAME            CPU(cores)   MEMORY(bytes)
memory-sample   75m          150Mi
```

Let's see what happens if we try to exceed the memory limit. We will use the same limit and requests as before, but instead, we will pass in `250M` to the `stress` command.

NOTE `stress` is a Linux tool for load and stress testing. You can read more about it here.

ch9/exceed-limit.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-sample
spec:
  containers:
    - name: stress
      image: polinux/stress
      resources:
        limits:
          memory: "200Mi"
        requests:
          memory: "100Mi"
      command: ["stress"]
      args: ["--vm", "1", "--vm-bytes", "250M", "--vm-hang", "1"]
```

Save the above YAML to `exceed-limit.yaml` and create the Pod using `kubectl apply -f exceed-limit.yaml`. If you look at the Pods, you will notice Kubernetes killed the Pod:

```
$ kubectl get po
NAME                            READY   STATUS      RESTARTS   AGE
memory-sample                   0/1     OOMKilled   0          5s
```

The `OOMKilled` status means that Kubernetes killed the Pod because it exceeded the limit and is Out Of Memory. Let's kill the Pod with `kubectl delete pod memory-sample`.

Another scenario you might run into, especially if you're running a local cluster, is if you try to run too many workloads, and the nodes cannot accommodate them. Let's try to simulate this scenario by requesting more memory that's available in the cluster.

ch9/memory-hog.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-hog
spec:
  containers:
    - name: stress
      image: polinux/stress
      resources:
        limits:
          memory: "1000Gi"
        requests:
          memory: "1000Gi"
      command: ["stress"]
      args: ["--vm", "1", "--vm-bytes", "250M", "--vm-hang", "1"]
```

Save the above YAML to `memory-hog.yaml` and create the Pod using `kubectl apply -f memory-hog.yaml`.

If you look at the Pod, you will notice that it's `Pending`. Running the `describe` command shows the reason for the status:

```
$ kubectl describe po memory-hog
...
Events:
  Type     Reason            Age                From               Message
  ----     ------            ----               ----               -------
  Warning  FailedScheduling  43s (x2 over 43s)  default-scheduler  0/1 nodes are
available: 1 Insufficient memory.
```

Delete the Pod with `kubectl delete po memory-hog`.

You can set the CPU requests and limits similarly as you did the memory. If we set `cpu` field to a crazy number of 100 CPUs you will get a similar error as we did with the memory.

ch9/cpu-hog.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-hog
spec:
  containers:
    - name: stress
      image: polinux/stress
      resources:
        limits:
          cpu: "100"
        requests:
          cpu: "100"
      command: ["stress"]
      args: ["--cpu", "2"]
```

Save the above YAML to `cpu-hog.yaml` and create the Pod using `kubectl apply -f cpu-hog.yaml`. Then, look at the Pod details for the error:

```
$ kubectl describe po cpu-hog
...
  Type     Reason            Age                From               Message
  ----     ------            ----               ----               -------
  Warning  FailedScheduling  80s (x2 over 81s)  default-scheduler  0/1 nodes are
available: 1 Insufficient cpu.
```

Before continuing, delete the `cpu-hog` Pod using `kubectl delete po cpu-hog`.

When designing your Pods, make sure you configure the limits and requests. That way, you can make efficient use of the available resources. If you don't specify the limits, your containers don't have an upper bound. You could potentially run into a situation where a single container can use up all available memory.

Resource quotas

If you are working with a cluster shared between multiple team members and namespaces, it is important to enable resource quotas on the namespace. With a quoate you can limit the total resources that can be requested in a given namespaces.

In addition to the CPU and memory quotas, you can also enable **storage quotas** and **object count** quotas. The storage quota can limit the number of storage resources - the number of persistent volume claims and the sum of storage requests. With the object count quota, you can go a step further and limit the number of specific resources created in a namespace. For example, you can

limit the number of Secrets, Services, ConfigMaps, and other Kubernetes resources.

The resource for defining quotas is called a `ResourceQuota`. Here's a sample quota that limits the number of Pods to 5:

ch9/res-quota.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: res-quota
spec:
  hard:
    pods: "5"
```

Save the above YAML to `res-quota.yaml` and create the ResourceQuota using `kubectl apply -f res-quota.yaml`. Next, we will create Deployment and then try to scale it over the limit of 5 Pods.

ch9/quota-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello
  labels:
    app.kubernetes.io/name: hello
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: hello
  template:
    metadata:
      labels:
        app.kubernetes.io/name: hello
    spec:
      containers:
        - name: hello-container
          image: busybox
          command: ["sh", "-c", "echo Hello from my container! && sleep 3600"]
```

Save the above YAML to `quota-deployment.yaml` and create the Deployment using `kubectl apply -f quota-deployment.yaml`.

Now you can try and scale the Deployment to 10 replicas:

```
$ kubectl scale deploy hello --replicas=10
```

If you list the Pods, there will be five of them running. If you look at the events, you will see the

error Kubernetes reported:

```
$ kubectl get events | grep replicaset/hello
...
71s         Warning   FailedCreate        replicaset/hello-56f578b46f        Error
creating: po
ds "hello-56f578b46f-96t4j" is forbidden: exceeded quota: pods-limit, requested: pods
=1, used:
pods=5, limited: pods=5
...
```

Let's modify the quota and include some CPU and memory limits as well.

ch9/res-quota-mem-cpu.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: res-quota
spec:
  hard:
    pods: "5"
    memory: "200m"
    cpu: "0.5"
```

Save the above YAML to `res-quota-mem-cpu.yaml` and update the ResourceQuota using `kubectl apply -f res-quota-mem-cpu.yaml`.

You can look at the quota details using the `describe` command:

```
$ kubectl describe quota res-quota
Name:       res-quota
Namespace:  default
Resource    Used  Hard
--------    ----  ----
cpu         0     500m
memory      0     200m
pods        5     5
```

Let's scale down the Deployment back to 1 Pod with `kubectl scale deploy hello --replicas=1`. If you try to create a Pod that exceeds the memory or CPU, the Kubernetes CLI will fail right away. Here's a message you might get if that happens:

```
Error from server (Forbidden): pods "quota-pod" is forbidden: exceeded quota: res-
quota, requested: memory=100Mi, used: memory=0, limited: memory=200m
```

Similarly, if you try to create a Pod without the CPU and memory requests defined, you will get an

error. Here's what happens if you run `kubectl create deploy my-nginx --image=nginx` and then look at the ReplicaSet details:

```
$ kubectl describe replicaset my-nginx
...
Error creating: pods "my-nginx-6b74b79f57-dfljt" is forbidden: failed quota: res-
quota: must specify cpu,memory
...
```

Let's create a Pod that defines the memory and CPU requests:

ch9/quota-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: quota-pod
spec:
  containers:
    - name: quota-pod
      image: busybox
      command: ["sh", "-c", "sleep 3600"]
      resources:
        requests:
          memory: "100m"
          cpu: "0.1"
```

Save the above YAML to `quota-pod.yaml` and create the Pod using `kubectl apply -f quota-pod.yaml`.

If you describe the quota now, you will see how much memory and CPU is the Pod using:

```
$ kubectl describe quota
Name:       res-quota
Namespace:  default
Resource    Used  Hard
--------    ----  ----
cpu         100m  500m
memory      100m  200m
pods        2     5
```

Before we continue, make sure you delete the quota using `kubectl delete quota res-quota`, and the Deployment (`kubectl delete deploy hello`) and the Pod (`kubectl delete po quota-pod`).

Horizontal scaling

Previously, you've learned how to scale the Pods manually, and now you also know how to request and limit the resources. With the help of the metrics server, we can horizontally scale Pods. For

example, if the CPU utilization is getting high or close to the limit, you can add more replicas, and once the utilization drops, you can scale the Pods down.

The resource and controller you can use to scale Pods automatically is the **Horizontal Pod Autoscaler (HPA)**. HPA periodically checks the Pod resource utilization and calculates the number of replicas required, based on the HPA resource settings. Based on these values, it adjusts the replicas field.

Let's create a Deployment with an image that runs a computation that takes a while. Then, we will start sending request to it, to see the metrics we get from the Pods:

ch9/computations.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: computations
  labels:
    app.kubernetes.io/name: computations
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: computations
  template:
    metadata:
      labels:
        app.kubernetes.io/name: computations
    spec:
      containers:
        - name: computations
          image: startkubernetes/computations:0.1.0
          ports:
            - containerPort: 8080
          resources:
            limits:
              cpu: 100m
              memory: 10Mi
---
apiVersion: v1
kind: Service
metadata:
  name: computations
  labels:
    app.kubernetes.io/name: computations
spec:
  selector:
    app.kubernetes.io/name: computations
  ports:
    - port: 8080
```

Save the above YAML to `computations.yaml` and create the Service and Deployment with `kubectl apply -f computations.yaml`.

Next, we will create a Pod called `load` and run the shell inside. From inside the Pod, we will call the `computations` service in an endless loop:

```
$ kubectl run -it --rm load --image=radial/busyboxplus:curl -- /bin/sh
```

When you get the terminal inside the container, run the endless loop:

```
$ while true; do curl http://computations:8080; done
DoneDoneDoneDoneDoneDone
...
```

You will start getting back the response from the service (`Done`). Leave it running for a couple of minutes and then run the `kubectl top pod` command to see the load. You can also specify the `--containers` flag to show the container names:

```
$ kubectl top pod --containers
POD                            NAME           CPU(cores)   MEMORY(bytes)
computations-b6f8f97c4-d9rbg   computations   96m          5Mi
load                           load           95m          2Mi
```

Both the `computations` Pod and the `load` Pod report the memory and CPU usage. Notice the CPU usage for the computations Pod is getting close to `100m` - this is the limit we set in the Deployment.

Let's create a horizontal pod auto-scaler that will scale out the Pods when the average CPU utilization hits 50%. We also specify the upper bound, the max number of replicas, as we don't want to scale the Pod indefinitely. With the `scaleTargetRef` we are defining which resource to use (Deployment). Under `metrics` we are specifying the resource the HPA should use to determine if the Deployment should be scaled or not.

ch9/hpa.yaml

```
kind: HorizontalPodAutoscaler
apiVersion: autoscaling/v2beta2
metadata:
  name: computations
spec:
  maxReplicas: 10
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: computations
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

Save the above YAML to `hpa.yaml` and create the HorizontalPodAutoscaler using `kubectl apply -f hpa.yaml`. With the `curl` command from earlier still running, if you look at the details of the HPA, you should how many replicas were already created by the HPA. After a couple of minutes, the HPA will scale the Pod as shown in the `REPLICAS` column:

```
$ kubectl get hpa
NAME           REFERENCE                 TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
computations   Deployment/computations   59%/50%   1         10        3          3m8s
```

Looking at the `top` command, you will see that this time each Pod is consuming around 57m of CPU, which is under the limit of 100m and less than a single Pod was consuming:

```
$ kubectl top pod --containers
POD                            NAME           CPU(cores)   MEMORY(bytes)
computations-b6f8f97c4-4k5mx   computations   57m          3Mi
computations-b6f8f97c4-d9rbg   computations   57m          3Mi
computations-b6f8f97c4-mn2ld   computations   59m          4Mi
```

By scaling out the Pods, we distributed the load across multiple instances.

Let's stop the endless `curl` loop and observe what happens. You can pass the `-w` flag to the `kubectl get hpa` command to watch the changes:

```
$ kubectl get hpa -w
NAME           REFERENCE                 TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
computations   Deployment/computations   51%/50%   1         10        4
5m29s
computations   Deployment/computations   47%/50%   1         10        4
5m34s
```

The time it takes for HPA to scale down the replicas depends on the stabilization window setting in the field `stabilizationWindow`. The default stabilization window for scaling down is 300 seconds (5 minutes). Because of this, it will take at least 5 minutes for HPA to start scaling down the Pods.

In addition to CPU, you can also use other metrics, such memory, and even combine them:

```
...
  metrics:
  - type: Resource
    resource:
      name: memory
      targetAverageValue: 10M
...
```

You can use two other groups of metrics in the horizontal pod scaler: the custom and external metrics. The custom metric is any metric associated with a Kubernetes resource, while the external metric is any custom metric that's not related to a Kubernetes resource.

To use these metrics, you need to instrument your application (to emit the metrics) first, then install a metrics collector that's going to collect the desired metrics and pass them to the metrics server. A popular metrics collector is Prometheus[https://prometheus.io/]. Prometheus can collect the metrics from your containers. For the metrics to be available to the HPA, for example, you will also need to install a Prometheus Adapter[https://github.com/DirectXMan12/k8s-prometheus-adapter]. The Prometheus adapter is a metrics server, and it implements the same metrics API interface.

Once your application is emitting metrics, and Prometheus is collecting and exposing them through the metrics API, you can create an HPA that uses the custom metric. For example, let's say your application emits a metric called `invoke_count`, then you can write a HPA like this:

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: invoke-count-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-app
  minReplicas: 1
  maxReplicas: 10
  metrics:
    - type: Pods
      pods:
        metric:
          name: invoke_count
        target:
          type: AverageValue
          averageValue: 2
```

Using affinity, taints, and tolerations

Using node affinity, you can constrain which nodes your Pod is eligible to run on based on the node's labels. Node affinity can be set on Pods using the `affinity` and `nodeAffinity` fields. There are two types of node affinity - with the first one (`requiredDuringSchedulingIgnoredDuringExecution`) you can specify rules that **must** be met for a Pod to be scheduled onto a node, for example, "only run the Pod on node ABC". The second one (`preferredDuringSchedulingIgnoredDuringExecution`) specifies the preferences - for example, "try to run this Pod on node ABC, but if it's not possible, then run it somewhere else".

Let's look at the labels set on the Minikube node:

```
$ kubectl get node --show-labels
NAME       STATUS   ROLES    AGE     VERSION   LABELS
minikube   Ready    master   4h39m   v1.19.0   beta.kubernetes.io/arch
=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64,kubernetes.io/hostname=min
ikube,kubernetes.io/os=linux,minikube.k8s.io/commit=0c5e9de4ca6f9c55147ae7f90af97eff5b
efef5f,minikube.k8s.io/name=minikube,minikube.k8s.io/updated_at=2020_09_20T12_00_03_07
00,minikube.k8s.io/version=v1.13.0,node-role.kubernetes.io/master=
```

For example, if we wanted the Pods to end up on nodes with the following label `kubernetes.io/os=linux`, we could define the Pod like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/os
            operator: In
            values:
            - linux
  containers:
  - name: container
    image: busybox
    command: ["sh", "-c", "sleep 3600"]
```

If you deploy the above Pod to your Minikube cluster, it will all work. Let's say we wanted the Pod to end up on a node running Windows (e.g., `kubernetes.io/os=windows`).

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/os
            operator: In
            values:
            - windows
  containers:
  - name: container
    image: busybox
    command: ["sh", "-c", "sleep 3600"]
```

In this case, the Pod will stay pending because we don't have a node with that label. This is the error you will get if you describe the Pod:

```
$ kubectl describe po pod-affinity
...
  Type     Reason            Age              From               Message
  ----     ------            ----             ----               -------
  Warning  FailedScheduling  8s (x2 over 8s)  default-scheduler  0/1 nodes are
available: 1 node(s) didn't match node selector.
...
```

While the affinity 'attracts' Pods to specified nodes, **taints** do the opposite. You can use taints to keep Pods off specific nodes. You use taints in combination with **tolerations**. They work together to ensure Pods don't get scheduled onto appropriate nodes. You can apply taints onto nodes to mark them that they should not accept any Pods that do not tolerate them.

Each taint has three parts: a key, value, and effect, and it looks like this: `key=value:effect`. The key and value is something you can pick, but the effect can only be one of the following value: `NoSchedule`, `PreferNoSchedule`, or `NoExecute`.

One of the use cases for using taints and tolerations is to dedicate a set of nodes for specific users or running specialized hardware (GPUs). In that case, you only want to run workloads that need the specialized hardware on those nodes and keep the rest of the Pods off those nodes.

Here's an example of how you can use `kubectl taint` command to add a taint to the node:

```
$ kubectl taint nodes my-node special=true:NoSchedule
```

The above command adds a taint with a key `special` and value `true` to `my-node`. You could then define the tolerations in your Pods like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-taint
spec:
  tolerations:
  - key: "special"
    operator: "Equal"
    value: "true"
    effect: "NoSchedule"
  containers:
  - name: container
    image: busybox
    command: ["sh", "-c", "sleep 3600"]
```

The toleration in the above Pod YAML has the exact key, value, and the effect as the taint on the node. Since it matches the taint, you can schedule it on that node. Since the `tolerations` field is an array, you can specify multiple tolerations. In case of multiple tolerations, **all** of them need to match for the Pod to be scheduled.

If you already have a mix of Pods running on the node and want to evict the ones without the taint, you can use the `NoExecute` effect. When using `NoExecute` effect, you can also specify an optional `tolerationSeconds` field. The Pod that tolerates the taint remains running on the node for the duration specified in that field. Pods that don't specify the `tolerationSeconds` field will remain bound to the node forever, and the ones with no tolerations will be evicted immediately.

The difference between `NoSchedule` and `NoExecute` is that if the Pod is already running on the node before you apply the taint, the Pod won't get rescheduled if using `NoSchedule`. However, if you use `NoExecute` the Pod might get evicted.

The last effect is `PreferNoSchedule`. When using this effect, Kubernetes avoids scheduling Pods that don't tolerate the taint. However, the Pods might still get scheduled onto these nodes if other nodes are at capacity.

Extending Kubernetes

Using custom resource definitions (CRDs)

Throughout this book, we discussed various Kubernetes resources, their purpose, how to use them, and how they work. However, you might run into scenarios where the existing resources are too limiting or not precisely what you need. For these purposes, Kubernetes also support **custom resource definitions (CRDs)**.

Each resource in Kubernetes (Pod, Service, Deployment, etc.) is an endpoint in the Kubernetes API. This endpoint stores the collection of Kubernetes objects. You have seen an example of the API endpoint in the section called What are service accounts?. In that section, we were accessing the Pod information using an URL like this: `https://kubernetes.default:443/api/v1/namespaces/default/pods/simple-pod`. Other Kubernetes services have similar endpoints in the API, and you can use that API to get the objects or create, update, and delete it. You can check out the full Kubernetes API reference here.

Custom resources are a way of extending the Kubernetes API. You can develop your custom resource (for example, MyCoolPod) and install it on your cluster or any other Kubernetes clusters. Once the custom resource is installed, you can create its objects using the Kubernetes CLI, just like you would do it for Pods or Services. Let's say you registered a custom resource called `MyCoolPod`. You could use `kubectl get mycoolpod` or `kubectl describe mycoolpod pod-instance` to interact with the object of the MyCoolPod kind.

Custom resource on their own simply allows you to store and retrieve the data. To make them more powerful, you can combine them with a **custom controller**. Using a custom controller, you can get a proper declarative API that allows you to declare your resource's desired state and keep the object's current state in sync with the desired state. Just like the ReplicaSet controller does when it tries to maintain the Pod replicas.

Let's say we want to create a custom resource called PdfDocument that takes the text in the resource and saves it to an MD (Markdown) file and then converts it to a PDF file. We will use a Kubernetes Volume to store the MD and the PDF file. There are two init containers. The first one saves the text to .MD file, and the second one uses that .MD file to create a PDF file.

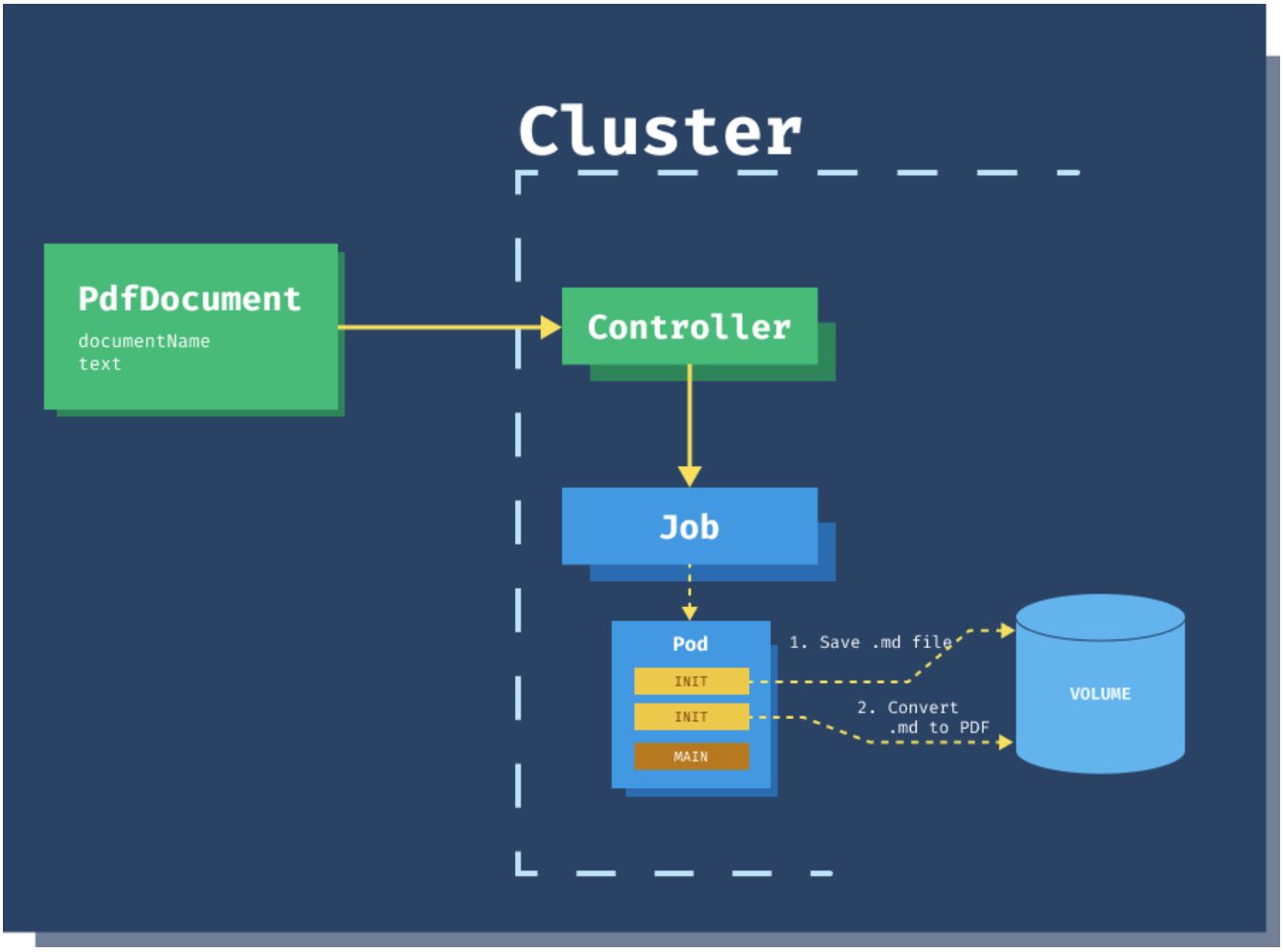


Figure 45. PdfDocument Flow

Here's how the resource could look like:

```
kind: PdfDocument
metadata:
    name: my-document
spec:
    documentName: my-text
    text: |
        ### This is a title
        Here is some **BOLD** text
```

If you try to create this resource using Kubernetes CLI, you'll get an error because you haven't provided the `apiVersion` field. We need a way to tell Kubernetes about the `PdfDocument` kind.

Create a CustomResourceDefinition

To be able to create custome resources, we need to tell Kubernetes about the resource. You can do that using a `CustomResourceDefinition` resource. Here's how we would create the CRD for the `PdfDocument` type:

ch10/pdf-crd.yaml

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: pdfdocuments.k8s.startkubernetes.com
spec:
  group: k8s.startkubernetes.com
  scope: Namespaced
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
            type: object
            properties:
                spec:
                  type: object
                  properties:
                    documentName:
                      type: string
                    text:
                      type: string
  names:
    kind: PdfDocument
    singular: pdfdocument
    plural: pdfdocuments
    shortNames:
    - pdf
    - pdfs
```

Save the YAML to `pdf-crd.yaml`. There's a lot of going on in the YAML. Refer to the table below to understand the different fields, sections, and their usage.

Table 7. CRD Fields

Field	Description	Example
`name`	Name of the CRD must match the following format: `plural.group`. The values are coming from the fields with the same names, from the `names` section	`pdfdocuments.k8s.startkubernetes.com`
`group`	Name of the group to use for the REST API. For example `/apis/[group]/[version]`	`k8s.startkubernetes.com`
`scope`	Resource can be Namespaced or Cluster scoped	`Namespaced` or `Cluster`

Field	Description	Example
`versions`	List of all supported versions of the API	See the YAML
`plural`	Plural name of the resource. Used in the REST API URL: `/apis/[group]/[version]/[plural]`	`pdfdocuments`
`singular`	Singular name of the resource. Use as an alias the CLI (e.g., `kubectl get [singular]`)	`pdfdocument`
`shortNames`	Short names used for the resource in the CLI	`pdf`, `pdfs`

The schema of the resource is specified using the OpenAPI v3.0 specification format. OpenAPI v3.0 spec also allows you to define a more structural schema, where you can define required fields or patterns field values need to conform to. For example, if we wanted to restrict the values for the `documentName` field to lowercase names with exactly ten characters, we could do it using the `pattern` field:

```
...
documentName:
  type: string
  pattern: '^[a-z]{10}$'
```

Let's create the CustomResourceDefinition.

```
$ kubectl apply -f pdf-crd.yaml
customresourcedefinition.apiextensions.k8s.io/pdfdocuments.k8s.startkubernetes.com
created
```

You can now use `pdf`, `pdfs` or `pdfdocument` to list the PdfDocument resouces in the cluster. The resource kind is also visible when you run the `api-resources command`:

```
$ kubectl get pdfdocument
No resources found in default namespace.

$ kubectl get pdf
No resources found in default namespace.

$ kubectl api-resources | grep pdf
NAME                              SHORTNAMES   APIGROUP
NAMESPACED   KIND
...
pdfdocuments                      pdf,pdfs     k8s.startkubernetes.com        true
PdfDocument
...
```

Kubernetes also creates a new namespaced REST API endpoint for the `pdfresources`. Let's look at how we can access the API. You have already talked to the Kubernetes API in the Bindings section, where we accessed the API through a Pod. We will use `kubectl proxy` command to set up a proxy to the API server this time.

Open a separate terminal window and start the proxy:

```
$ kubectl proxy --port=8080
Starting to serve on 127.0.0.1:8080
```

Leave the proxy running, and from a different terminal, you can now access the Kubernetes API. For example, to get the list of all supported APIs, run:

```
$ curl localhost:8080/apis
...
    {
      "name": "k8s.startkubernetes.com",
      "versions": [
        {
          "groupVersion": "k8s.startkubernetes.com/v1",
          "version": "v1"
        }
      ],
      "preferredVersion": {
        "groupVersion": "k8s.startkubernetes.com/v1",
        "version": "v1"
      }
    },
...
```

To access the PdfDocuments API, you have to use the API name and the version, like this:

```
$ curl localhost:8080/apis/k8s.startkubernetes.com/v1/namespaces/default/pdfdocuments
{"apiVersion":"k8s.startkubernetes.com/v1","items":[],"kind":"PdfDocumentList","metada
ta":{"continue":"","resourceVersion":"21553","selfLink":"/apis/k8s.startkubernetes.com
/v1/namespaces/default/pdfdocuments"}}
```

We get back an empty list of items,because we haven't created the PdfDocument resource yet. Now that the API is registered and we have the `apiVersion`, we can create and deploy the PdfDocument. The `apiVersion` consists of the group name and one of the support versions. In this case, the `apiVersion` is `k8s.startkubernetes.com/v1`.

ch10/my-document.yaml

```
apiVersion: k8s.startkubernetes.com/v1
kind: PdfDocument
metadata:
    name: my-document
spec:
    documentName: my-text
    text: |
        ### This is a title
        Here is some **BOLD** text
```

Save the above YAML to `my-document.yaml` and create the PdfDocument resource.

```
$ kubectl apply -f my-document.yaml
pdfdocument.k8s.startkubernetes.com/my-document created
```

If you list the `pdfs` you will see the resource we created. You can also look at the YAML representation of the resource and the fields Kubernetes added.

```
$ kubectl get pdfs
NAME          AGE
my-document   57s

$ kubectl describe pdf my-document
Name:         my-document
Namespace:    default
Labels:       <none>
Annotations:  <none>
API Version:  k8s.startkubernetes.com/v1
Kind:         PdfDocument
Metadata:
  Creation Timestamp:  2020-09-21T22:37:30Z
  Generation:          1
  Managed Fields:
    API Version:  k8s.startkubernetes.com/v1
    Fields Type:  FieldsV1
    fieldsV1:
      f:metadata:
        f:annotations:
          .:
          f:kubectl.kubernetes.io/last-applied-configuration:
      f:spec:
        .:
        f:documentName:
        f:text:
    Manager:         kubectl-client-side-apply
    Operation:       Update
    Time:            2020-09-21T22:37:30Z
  Resource Version:  21796
  Self Link:
/apis/k8s.startkubernetes.com/v1/namespaces/default/pdfdocuments/my-document
  UID:               62283c82-cbb3-4676-b71d-770004151c6d
Spec:
  Document Name:  my-text
  Text:           ### This is a title
Here is some **BOLD** text

Events:  <none>
```

The resource behaves just like any other Kubernetes resource. Without a controller, the resource is useless. Let's see how we can create a simple controller. The controller creates a Job whenever you create a new PdfDocument resource. The Job takes the text from the resource and uses init containers to create a PDF document from it.

Create a controller

The PdfDocument controller will watch the PdfDocument resources and act accordingly. Whenever you create or update a PdfDocument resource, the controller creates a Job that generates a PDF

document from the text inside the resource. I've used the Kubebuilder to build the CRD and the controller. You will also need Kustomize to build the controller locally.

Once you have installed Kubebuilder, you can initialize the project:

```
$ go mod init k8s.startkubernetes.com/v1
go: creating new go.mod: module k8s.startkubernetes.com/v1

$ kubebuilder init
Writing scaffold for you to edit...
Get controller runtime:
...
go build -o bin/manager main.go
Next: define a resource with:
$ kubebuilder create api
```

The kubebuilder creates the project structure and other files for the controller. The next step is to create the code, and struct for the custom resource using the `kubebuilder create api` command:

```
$ kubebuilder create api --group k8s.startkubernetes.com --version v1 --kind
PdfDocument
Create Resource [y/n]
y
Create Controller [y/n]
y
Writing scaffold for you to edit...
api/v1/pdfdocument_types.go
controllers/pdfdocument_controller.go
...
```

The command creates the `api/v1` folder with the emtpy resource type. I've added the two fields that we one in the `spec` section to the `pdfdocument_types.go` file:

```
type PdfDocumentSpec struct {
    // INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
    // Important: Run "make" to regenerate code after modifying this file

    DocumentName string `json:"documentName,omitempty"`
    Text         string `json:"text,omitempty"`
}
```

The next part is to implement a controller inside the `controllers/pdfdocument_controller.go` file. To implement the functionality, I am using two init containers (you can read more about in Init containers section). The first init container reads the `text` from the resource and stores it in a `.md` file on a Volume, shared between all containers.

Once the `.md` file is stored, the second init container runs `pandoc` to converts the `.md` file into the PDF

file. This file is stored on the shared volume as well.

Finally, the main container will just sleep, so that we can copy the result over to the local machine. To make this more realistic, you could use a persistent volume and store the converted PDF documents there. For the sake of simplicity and to demonstrate how controllers work, I am using a regular volume.

The logic in the `Reconcile` function in `pdfdocument_controller.go` looks like this:

```
    var pdfDoc k8sstartkubernetescomv1.PdfDocument
    if err := r.Get(ctx, req.NamespacedName, &pdfDoc); err != nil {
        log.Error(err, "unable to fetch PdfDocument")
        return ctrl.Result{}, client.IgnoreNotFound(err)
    }

    jobSpec, err := r.createJob(pdfDoc)
    if err != nil {
        log.Error(err, "failed to create Job spec")
        return ctrl.Result{}, client.IgnoreNotFound(err)
    }

    if err := r.Create(ctx, &jobSpec); err != nil {
        log.Error(err, "unable to create Job")
    }
```

The main function of every controller is the `Reconcile` function. During the reconciliation process, the controller needs to ensure that the actual state in the cluster matches the desired state in the object. Each controller focuses on a single kind of resource, but that doesn't prevent you from interacting with other kinds and resources. In our case, we will be creating a Job in addition to reading the PdfDocument kind.

If we encounter any errors during the reconciliation, we return an error from the function. Otherwise, we return an empty result that indicates we successfully reconciled the object.

The full source code for the controller is available `customcontroller` folder with other source code.

To test the controller against the current cluster, you can use the `make run` command:

```
$ make run
/Users/peterj/projects/go/bin/controller-gen object:headerFile="hack/boilerpla
te.go.txt" paths="./..."
go fmt ./...
go vet ./...
/Users/peterj/projects/go/bin/controller-gen "crd:trivialVersions=true" rbac:r
oleName=manager-role webhook paths="./..." output:crd:artifacts:config=config/
crd/bases
go run ./main.go
2020-09-21T18:26:49.349-0700    INFO    controller-runtime.metrics      metric
s server
....
```

Once the controller is running, you can create the PdfDocument resource, just like you did before.

ch10/my-document.yaml

```
apiVersion: k8s.startkubernetes.com/v1
kind: PdfDocument
metadata:
    name: my-document
spec:
    documentName: my-text
    text: |
        ### This is a title
        Here is some **BOLD** text
```

Save the above YAML to `my-document.yaml` and create the PdfDocument resource using `kubectl apply -f my-document.yaml`. As Kubernetes creates the resource you will notice the following output from the controller:

```
2020-09-21T18:41:26.852-0700    DEBUG   controller-runtime.controller   Successfully
Reconciled      {"controller": "pdfdocument", "request": "default/my-document"}
```

This output message means that the PdfDocument resource was created and that the controller also created a Job:

```
$ kubectl get pdfdocument
NAME          AGE
my-document   37s

$ kubectl get job
NAME              COMPLETIONS   DURATION   AGE
my-document-job   0/1           66s        66s

$ kubectl get po
NAME                    READY   STATUS    RESTARTS   AGE
my-document-job-rrvzz   1/1     Running   0          70s
```

Note that the Job won't complete until the Pod finishes executing - we are running a sleep command, so that it gives us enough time to grab the converted PDF document from the Volume.

To copy the PDF file from the container to the local machine, you can use the `kubectl cp` command:

```
$ kubectl cp my-document-job-rrvzz:/data/my-text.pdf ${PWD}/my-text.pdf
```

The above command copies a file from `/data/my-text.pdf` location inside the container (which is on the shared volume), and copies it to the current folder you're running `kubectl` in.

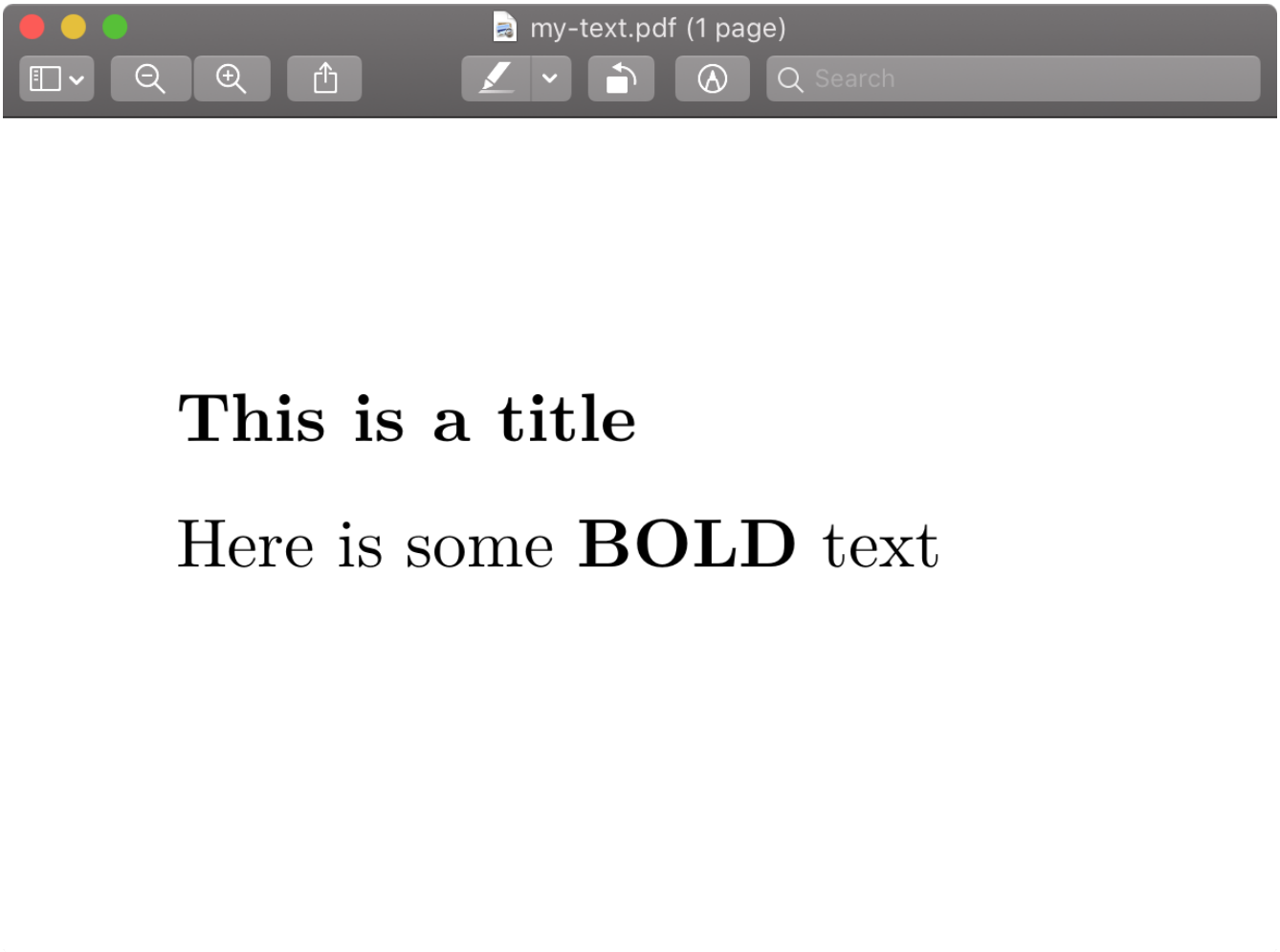


Figure 46. Generated PDF Document

Ensure you delete the CRD and the resource as the `make run` is not going to remove the deployed resources.

Kubernetes Operators

Operators use custom resources and controllers to manage applications and their components. One of the Kubernetes operators' purposes is to automate tasks that a person operating a Kubernetes application would do. That way, you can simplify the installation and administration of your applications.

Let's look at a concrete example with the Istio service mesh. Installing and managing Istio service mesh was (and still is, to a certain extent) a complicated task. There are multiple components and various resources involved in operating a service mesh on Kubernetes. The way you would install Istio is to either render a full YAML template based on a configuration profile and deploy that, or you would use Helm. The initial installation usually worked fine. However, operators would run into issues later when they tried to re-configure the installation or upgrade components as it was challenging to administer.

For that reason, Istio decided to implement an operator. Using the operator Istio significantly simplified the installation and administration of the mesh. Previously, you would have to run Helm and depend on a third-party tool for the installation. With the operator, you create a custom resource (`IstioControlPlane`) that 'describes' how you want your service mesh to look like. You can select a profile or fine-tune any of the configuration settings of the mesh.

After you install the operator in your cluster, you can install the mesh using the custom resource, like the one below:

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  namespace: istio-system
  name: my-istiocontrolplane
spec:
  profile: demo
```

The controller responds to the resource being created, and starts installing Istio. Later, suppose you decide to change the configuration setting, you can use a different installation profile or even upgrade the installation, by updating the IstioOperator resource. The operator takes the update resources and reconfigures the installation. The knowledge that one needs to operate an application is now coded inside an operator, for everyone to use.

In the Create a controller section, we explained how to use Kubebuilder to build a custom resource controller. Just to be exact - the custom resource controller and operators are using the same things behind the scenes. Other tools are available for creating controllers or operators, such as KUDO and Operator Operator Framework. KUDO provides you with a declarative approach to building Kubernetes operators, while the Operator Framework features an Operator SDK that makes it easier to build, test, and package Operators.

Instead of writing custom code, as we did in Create a controller, some of these tools allow you to create Helm-based operators, for example.

For example, let's say you are creating an operator for your Cool App, called `coolapp-operator`. Using the Operator SDK, you can create a custom resource called `CoolApp` and handle the reconciliation logic.

The first step is to initialize the operator using the Helm plugin, by running `operator-sdk init --plugins=helm`. This command will create initial the project structure, including the configuration, Dockerfile, and Makefile.

Then, just like before, you need to create an API - this includes the group name, version, and the resource kind. For example:

```
$  operator-sdk create api --group k8s.startkubernetes.com --version v1 --kind CoolApp
Created helm-charts/coolapp
Generating RBAC rules
WARN[0000] The RBAC rules generated in config/rbac/role.yaml are based on the chart's
default manifest. Some rules may be missing for resources that are only enabled with
custom values, and some existing rules may be overly broad. Double check the rules
generated in config/rbac/role.yaml to ensure they meet the operator's permission
requirements.
```

The above command will create the Helm chart, include the Deployment, Ingress, Service, ServiceAccount, and HorizontalPodAutoscaler resources for your application. Alternatively, if you already have a Helm chart you want to use, you can provide it using the `--help-chart-repo` flag to the `create api` command.

The interesting part of this operator is in the `watches.yaml` file. In that file, you specify which resource you want the operator to watch for and create new Helm releases for:

```
# Use the 'create api' subcommand to add watches to this file.
- group: k8s.startkubernetes.com.my.domain
  version: v1
  kind: CoolApp
  chart: helm-charts/coolapp
# +kubebuilder:scaffold:watch
```

Helm 101

Helm is the package manager for Kubernetes. As you've seen throughout this book, several Kubernetes resources together for your application. If you're deploying a single resource, you can probably manage the YAML file and deployments manually. However, that's usually not the case. If you try to manually manage, track, and update YAML files for your application, you will quickly notice that it can get relatively complicated, and it doesn't make sense.

Helm can help with this complexity. Helm allows you to templatize your YAML files. Let's take a simple Deployment, for example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: simple-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test
  template:
    metadata:
      labels:
        app: test
    spec:
      containers:
      - name: test
        image: busybox
        command: [ "sh", "-c", "sleep 1h"]
```

One of the things you will probably be updating in the YAML is the image name. Helm allows to 'extract' these values into a separate values file (usually called `values.yaml`, but you can name it whatever you like). If we take the above Deployment and use the variable, here's how it would look:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: simple-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test
  template:
    metadata:
      labels:
        app: test
    spec:
      containers:
      - name: test
        image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
        command: [ "sh", "-c", "sleep 1h"]
```

Notice the image value consists of two values: the repository and tag. You can then store the actual values that will be replaced by Helm, in a `values.yaml` file, like this:

```
image:
  repository: busybox
  tag: "1.32"
```

The collection of all templatized files is called a **Helm chart**.

If you try to deploy the above Deployment YAML, you will get an error. However, you can use Helm to take the values from the `values.yaml` file and replace them in Deployment file.

```
$ helm install -f values.yaml ./mychart
```

Once you install a Helm chart, you can upgrade it or create new releases by modifying the values file or passing in values you want to replace from the command line.

Back to the `watches.yaml` file and the Operator SDK. Let's look at the `config/samples/k8s.startkubernetes.com_v1_coolapp.yaml` file. The file shows an example of a CoolApp resource that define how you should configure the CoolApp.

```
apiVersion: k8s.startkubernetes.com.my.domain/v1
kind: CoolApp
metadata:
  name: coolapp-sample
spec:
  # Default values copied from <project_dir>/helm-charts/coolapp/values.yaml
  affinity: {}
  autoscaling:
    enabled: false
    maxReplicas: 100
    minReplicas: 1
    targetCPUUtilizationPercentage: 80
  fullnameOverride: ""
  ...
```

The values in this resource are coming from the Helm `values.yaml` file. So if you deploy the CoolApp resource, the Operator SDK will invoke Helm behind the scenes and deploy the chart. Later, when you decide to change your application's configuration, the Operator SDK will do the reconciliation and make sure your application is updated accordingly.

To deploy the CoolApp operator, you can run `make install` from the operator folder. This will deploy the CRD:

```
$ kubectl api-resources | grep cool
coolapps                                      k8s.startkubernetes.com.my.domain
true         CoolApp
```

The next step is to build the operator image, push it to a Docker registry, and deploy it to the cluster.

NOTE	If you're building your image, make sure to replace the image name in the command below.

```
export IMG=startkubernetes/coolapp-operator:0.1.0
make docker-build docker-push IMG=$IMG
```

After you pushed the image, you can deploy the operator:

```
$ make deploy IMG=startkubernetes/coolapp-operator:0.1.0
....
rolebinding.rbac.authorization.k8s.io/coolapp-operator-leader-election-rolebinding
created
clusterrolebinding.rbac.authorization.k8s.io/coolapp-operator-manager-rolebinding
created
clusterrolebinding.rbac.authorization.k8s.io/coolapp-operator-proxy-rolebinding
created
service/coolapp-operator-controller-manager-metrics-service created
deployment.apps/coolapp-operator-controller-manager created
```

Operator SDK creates the necessary Kubernetes resources and deploys the operator to `coolapp-operator-system` namespace:

```
$ kubectl get po -n coolapp-operator-system
NAME                                                READY   STATUS    RESTARTS
AGE
coolapp-operator-controller-manager-54bf54c4f9-kjvkb   2/2     Running   0
107s
```

Let's tail the logs from the `manager` container in the above Pod. Open a separate terminal window, and run: `kubectl logs coolapp-operator-controller-manager-54bf54c4f9-kjvkb -n coolapp-operator-system -c manager -f`.

Now we can try deploying the sample CoolApp resource from `/config/samples/k8s.startkubernetes.com_v1_coolapp.yaml` by running `kubectl apply -f /config/samples/k8s.startkubernetes.com_v1_coolapp.yaml`.

As you deploy the resource you will notice the logs from the `manager` containers saying that the release was installed:

```
...
{"level":"info","ts":1600803466.9313226,"logger":"helm.controller","msg":"Installed
release","namespace":"default","name":
"coolapp-sample","apiVersion":"k8s.startkubernetes.com.my.domain/v1","kind":"CoolApp"
,"release":"coolapp-sample"}
...
{"level":"info","ts":1600803470.5439534,"logger":"helm.controller","msg":"Reconciled
release","namespace":"default","name"
:"coolapp-sample","apiVersion":"k8s.startkubernetes.com.my.domain/v1","kind":
"CoolApp","release":"coolapp-sample"}
...
```

The operator installed the CoolApp using Helm. Under the covers, Helm created a release:

```
$ helm ls
NAME            NAMESPACE       REVISION        UPDATED
STATUS          CHART           APP VERSION
coolapp-sample  default         1               2020-09-22 19:37:43.6701905 +0000 UTC
deployed        coolapp-0.1.0   1.16.0
```

If you would edit the original resource, you will notice that the operator will pick it up and update the Helm release. To remove the deployed operator and other resources, run `make undeploy` from the root controller folder.

Practical Kubernetes

Using an Ingress controller for SSL termination

SSL stands for secure socket layer protocol. The SSL termination, or also called SSL offloading, is the process of decrypting encrypted traffic. When encrypted traffic hits the ingress controller, it gets decrypted there and then passed to the backend applications. Doing SSL termination at the ingress controller level also lessens the burden on your server. You are only doing it once at the ingress controller level and not in each application.

I will be using a cloud-managed cluster and an actual domain name to demonstrate how to set up SSL termination. I'll be using the Ambassador controller, cert-manager for managing and issuing TLS certificates, Let's Encrypt as the certificate authority (CA), and Helm to install some of the components.

Before continuing, making sure you have installed Helm by following the instructions here. You can run `helm version` to make sure Helm is installed:

```
$ helm version
version.BuildInfo{Version:"v3.2.4",
GitCommit:"0ad800ef43d3b826f31a5ad8dfbb4fe05d143688", GitTreeState:"dirty",
GoVersion:"go1.14.3"}
```

NOTE Helm is a package manager for Kubernetes. Instead of dealing with individual deployments, services, configuration maps, secrets, and other Kubernetes resources, Helm packages them into "charts". A chart is a collection of different Kubernetes resource files. You can then take the charts and version, deploy, upgrade, and manage them as a single unit.

Deploying the sample application

As a sample application, I will be using the Dog Pic website.

practical/dogpic-app.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dogpic-web
  labels:
    app.kubernetes.io/name: dogpic-web
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: dogpic-web
  template:
    metadata:
      labels:
        app.kubernetes.io/name: dogpic-web
    spec:
      containers:
        - name: dogpic-container
          image: learncloudnative/dogpic-service:0.1.0
          ports:
            - containerPort: 3000
---
kind: Service
apiVersion: v1
metadata:
  name: dogpic-service
  labels:
    app.kubernetes.io/name: dogpic-web
spec:
  selector:
    app.kubernetes.io/name: dogpic-web
  ports:
    - port: 3000
      name: http
```

Save the above YAML in `dogpic-app.yaml` file and use `kubectl apply -f dogpic-app.yaml` to create the deployment and service.

Deploying cert-manager

We will deploy the cert-manager inside the cluster. As the name suggests, the cert-manager will deal with certificates. So, whenever we need a new certificate or renew an existing certificate, the cert-manager will do that for us.

The first step is to create a namespace to deploy the cert-manager in:

```
$ kubectl create ns cert-manager
namespace/cert-manager created
```

Next, we will add the `jetstack` Helm repository and refresh the local repository cache:

```
$ helm repo add jetstack https://charts.jetstack.io
"jetstack" has been added to your repositories

$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "jetstack" chart repository
Update Complete. ⎈ Happy Helming!⎈
```

Now we are ready to install the cert-manager. Run the following Helm command to install the cert-manager:

```
helm install \
  cert-manager jetstack/cert-manager \
  --namespace cert-manager \
  --version v0.15.1 \
  --set installCRDs=true
```

In the output, you will notice the message saying that Helm deployed the cert-manager successfully.

Before we can use it, we need to set up either a `ClusterIssuer` or an `Issuer` resource and configure it. This resource represents a certificate signing authority (CA) and allows cert-manager to issue certificates.

The difference between a `ClusterIssuer` and an `Issuer` is that the `ClusterIssuer` operates at the cluster level and the `Issuer` resource works on a namespace. For example, you could configure different `Issuer` resources for each namespace. Alternatively, you could create a `ClusterIssuer` to issue certificates in any namespace.

Cert-manager supports multiple issuer types. Let's Encrypt uses the ACME protocol, and therefore we will configure an ACME issuer type. These protocols support different challenge mechanisms to determine and verify domain ownership.

Challenges

In the ACME protocol, cert-manager supports two challenges to verify the domain ownership: the `HTTP-01` and `DNS-01` challenge. You can read more details about each one of these on Let's Encrypt website.

In short, the HTTP-01 challenge is the most common challenge type. The challenge involves a file with a token that you put in a certain location on your server. For example: `http://[my-cool-domain]/.well-known/acme-challenge/[token-file]`.

The DNS-01 challenge involves modifying a DNS record for your domain. To pass this challenge, you need to create a TXT DNS record with a specific value under the domain you want to claim. Using the DNS-01 challenge only makes sense if your domain registrar has an API that automatically updates the DNS records. See the full list of providers that integrate with the Let's Encrypt DNS validation.

I will be using the HTTP-01 challenge as it is more generic than the DNS-01, which depends on your domain registrar.

Let's deploy a `ClusterIssuer` we will be using. Make sure you replace the email with your email address:

practical/cluster-issuer.yaml

```
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    email: hello@example.com
    server: https://acme-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: letsencrypt-prod
    solvers:
      - http01:
          ingress:
            class: nginx
        selector: {}
```

Let's make sure ClusterIssuer gets created, and it's ready by running `kubectl describe clusterissuer` and confirming that the ACME account was registered (i.e., the email address you provided):

```
...
Status:
  Acme:
    Last Registered Email:  hello@example.com
    Uri:                    https://acme-v02.api.letsencrypt.org/acme/acct/89498526
  Conditions:
    Last Transition Time:  2020-06-22T20:36:04Z
    Message:               The ACME account was registered with the ACME server
    Reason:                ACMEAccountRegistered
    Status:                True
    Type:                  Ready
Events:                    <none>
```

Similarly, if you run `kubectl get clusterissuer` you should see the indication that the `ClusterIssuer` is ready:

```
$ kubectl get clusterissuer
NAME               READY   AGE
letsencrypt-prod   True    2m30s
```

Later on, once we deployed the Ingress controller and set up the DNS record on the domain, we will also create a `Certificate` resource.

Ambassador

To install Ambassador gateway, run the two commands below. The first one will take care of installing all CRD (custom resource definitions), and the second one installs the RBAC (Role-Based Access Control) resources and creates the Ambassador deployment.

```
$ kubectl apply -f https://www.getambassador.io/yaml/ambassador/ambassador-crds.yaml
...
$ kubectl apply -f https://www.getambassador.io/yaml/ambassador/ambassador-rbac.yaml
```

Finally, we need to create a LoadBalancer service that exposes two ports: 80 for HTTP traffic and 443 for HTTPS.

practical/ambassador-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: ambassador
spec:
  type: LoadBalancer
  ports:
    - name: http
      port: 80
      targetPort: 8080
    - name: https
      port: 443
      targetPort: 8443
  selector:
    service: ambassador
```

Save the above YAML in `ambassador-svc.yaml` file and run `kubectl apply -f ambassador-svc.yaml`.

NOTE	Deploying the above service will create a load balancer in your cloud providers account.

If you list the services, you will notice an External IP assigned to the `ambassador` service:

```
$ kubectl get svc
NAME               TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)          AGE
ambassador         LoadBalancer   10.0.78.66    51.143.120.54   80:31365/TCP     97s
ambassador-admin   NodePort       10.0.65.191   <none>          8877:30189/TCP   4m20s
kubernetes         ClusterIP      10.0.0.1      <none>          443/TCP          30d
```

Now that we have the External IP address, you can go to the website where you registered your domain and create an A DNS record that will point the domain to the external IP. Pointing the domain to an external IP will allow you to enter `http://[my-domain.com]` in your browser, and it will resolve to the above IP address (the ingress controller inside the cluster).

I will be using my domain called `startkubernetes.com`. I will set up a subdomain `dogs.startkubernetes.com` to point to my load balancer (e.g. `51.143.120.54`) using an A record. Regardless of where you registered your domain, you should be able to update the DNS records. Check the documentation on your domain registrars website on how to do that.

Let's set up an Ingress resource, so we can reach the Dog Pic website we deployed on the subdomain (make sure you replace the `dogs.startkubernetes.com` with your domain or subdomain name):

practical/dogs-ingress.yaml

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: my-ingress
  annotations:
    kubernetes.io/ingress.class: ambassador
spec:
  rules:
    - host: dogs.startkubernetes.com
      http:
        paths:
          - backend:
              serviceName: dogpic-service
              servicePort: 3000
```

With ingress deployed, you can open `http://dogs.startkubernetes.com`. You should see the Dog Pic website below.

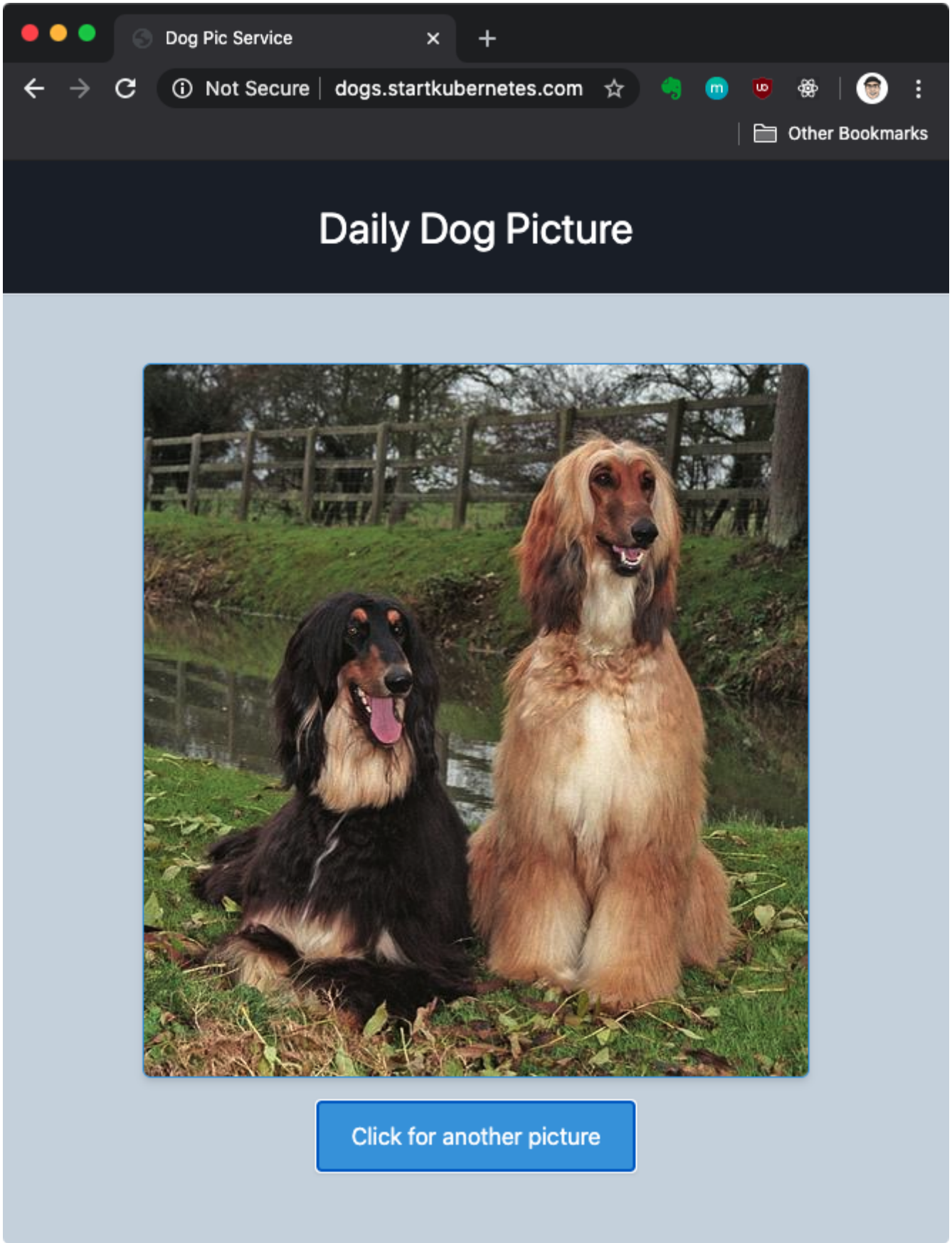


Figure 47. Dog Pic Website

Requesting a certificate

To request a new certificate, you need to create a `Certificate` resource. This resource includes the issuer reference (`ClusterIssuer` we created earlier), DNS names we want to request certificates for

(dogs.startkubernetes.com), and the Secret name the certificate will be stored in.

practical/certificate.yaml

```
apiVersion: cert-manager.io/v1alpha2
kind: Certificate
metadata:
  name: ambassador-certs
  namespace: default
spec:
  secretName: ambassador-certs
  issuerRef:
    name: letsencrypt-prod
    kind: ClusterIssuer
  dnsNames:
    - dogs.startkubernetes.com
```

Make sure you replace the dogs.startkubernetes.com with your domain name. Once you've done that, save the YAML in cert.yaml and create the certificate using kubectl apply -f -cert.yaml.

If you list the pods, you will notice a new pod called cm-acme-http-solver:

```
$ kubectl get po
NAME                           READY   STATUS    RESTARTS   AGE
ambassador-9db7b5d76-jlcdg     1/1     Running   0          22h
ambassador-9db7b5d76-qcwgk     1/1     Running   0          22h
ambassador-9db7b5d76-xsfw4     1/1     Running   0          22h
cm-acme-http-solver-qzh6l      1/1     Running   0          25m
dogpic-web-7bf547bd54-f2pff    1/1     Running   0          22h
```

Cert-manager created this pod to serve the token file as explained in the Challenges section and verify the domain name.

You can also look at the logs from the pod to see the values pod expects for the challenge:

```
$ kubectl logs cm-acme-http-solver-qzh6l
I0622 20:39:26.712391       1 solver.go:39] cert-manager/acmesolver "msg"="starting
listener"  "expected_domain"="dogs.startkubernetes.com" "expected_key"
="iqUZlG9v1K8czpAKaTpLfL278piwf-
mN4VZNvuwD0Ks.xonKHFvEQg2Ox_mI0cPM7UpCUHfu6H4aKtRcdrpiLik" "expected_token"
="iqUZlG9v1K8czpAKaTpLfL278piwf-mN4VZNvuwD0Ks" "listen_port"=8089
```

However, this pod is not exposed, so there's no way for Let's Encrypt to access it and do the challenge. So we need to expose this pod through an ingress. This involves creating a Kubernetes Service that points to the pod and updating the ingress. To update the ingress, we will use the Mapping resource from Ambassador. This resource defines a mapping to redirect requests with the prefix ./well-known/acme-challenge to the Kubernetes service that goes to the pod.

practical/challenge.yaml

```
apiVersion: getambassador.io/v2
kind: Mapping
metadata:
  name: challenge-mapping
spec:
  prefix: /.well-known/acme-challenge/
  rewrite: ""
  service: challenge-service
---
apiVersion: v1
kind: Service
metadata:
  name: challenge-service
spec:
  ports:
    - port: 80
      targetPort: 8089
  selector:
    acme.cert-manager.io/http01-solver: "true"
```

Store the above in `challenge.yaml` and deploy it using `kubectl apply -f challenge.yaml`. The cert-manager will retry the challenge and issue the certificate.

You can run `kubectl get cert` and confirm the `READY` column shows `True`, like this:

```
$ kubectl get cert
NAME               READY   SECRET             AGE
ambassador-certs   True    ambassador-certs   35m
```

Here are the steps we followed to request a certificate and a figure to visualize the process.

1. Request the certificate by creating the `Certificate` resource.
2. Cert-manager creates the `http-solver` pod (exposed through the `challenge-service` we created)
3. Cert-manager uses the issuer referenced in the `Certificate` and requests the certificates for the `dnsNames` from the authority (Let's Encrypt)
4. The authority sends the challenge for the `http-solver` to prove that we own the domains and checks that the challenges are solved (i.e. downloads the file from `/.well-known/acme-challenge/`)
5. Issued certificate and key are stored in Secret, referenced by the Issuer resource

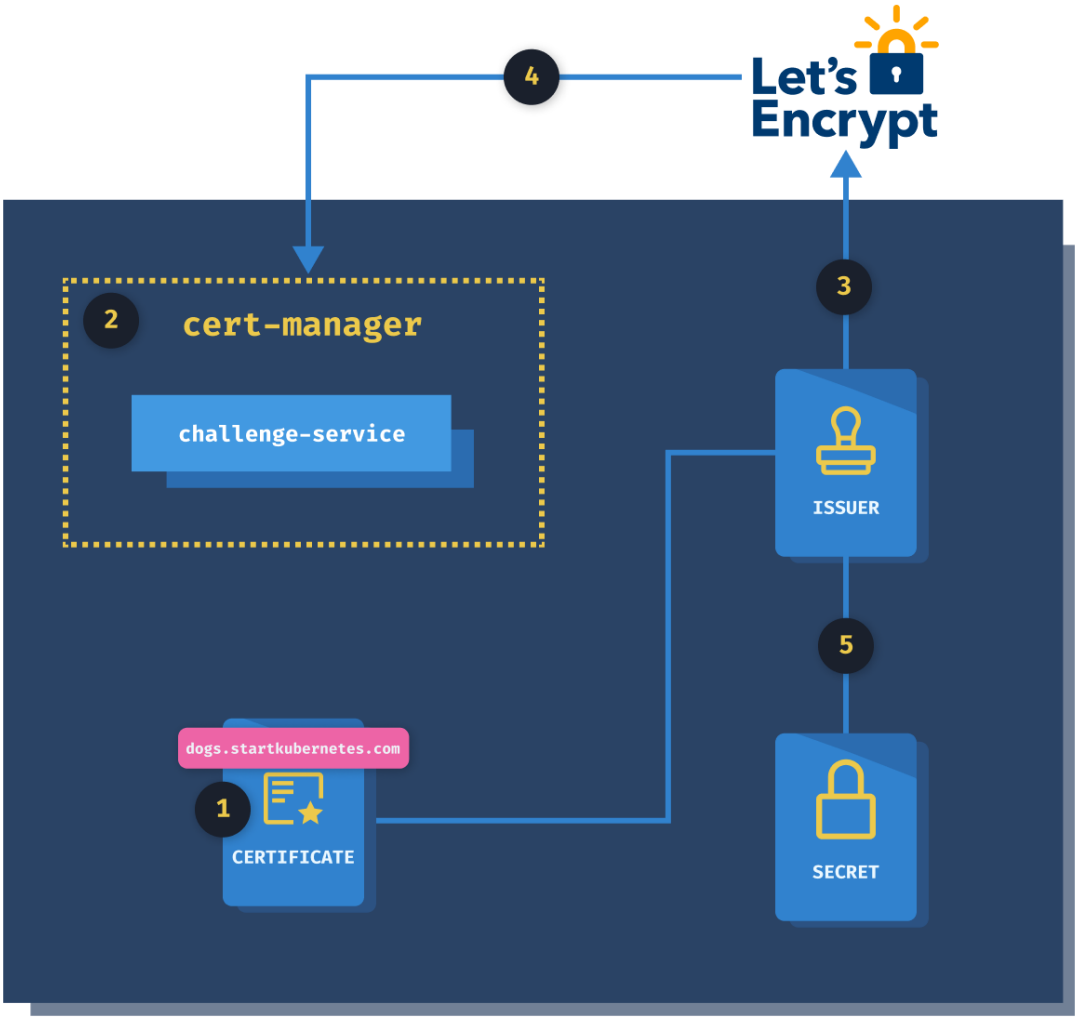


Figure 48. Requesting Certificates

Configuring TLS in Ingress

To secure an Ingress, we have to specify a Secret that contains the certificate and the private key. We defined the `ambassador-certs` secret name in the `Certificate` resource we created earlier.

practical/ingres-tls.yaml

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: my-ingress
  annotations:
    kubernetes.io/ingress.class: ambassador
spec:
  tls:
    - hosts:
        - dogs.startkubernetes.com
      secretName: ambassador-certs
  rules:
    - host: dogs.startkubernetes.com
      http:
        paths:
          - path: /
            backend:
              serviceName: dogpic-service
              servicePort: 3000
```

Under resource specification (`spec`), we use the `tls` key to specify the hosts and the secret name where the certificate and private key are stored.

Save the above YAML in `ingress-tls.yaml` and apply it with `kubectl apply -f ingress-tls.yaml`.

If you navigate to your domain using https (e.g. `https://dogs.startkubernetes.com`) you will see that the connection is secure, and it is using a valid certificate from Let's Encrypt.

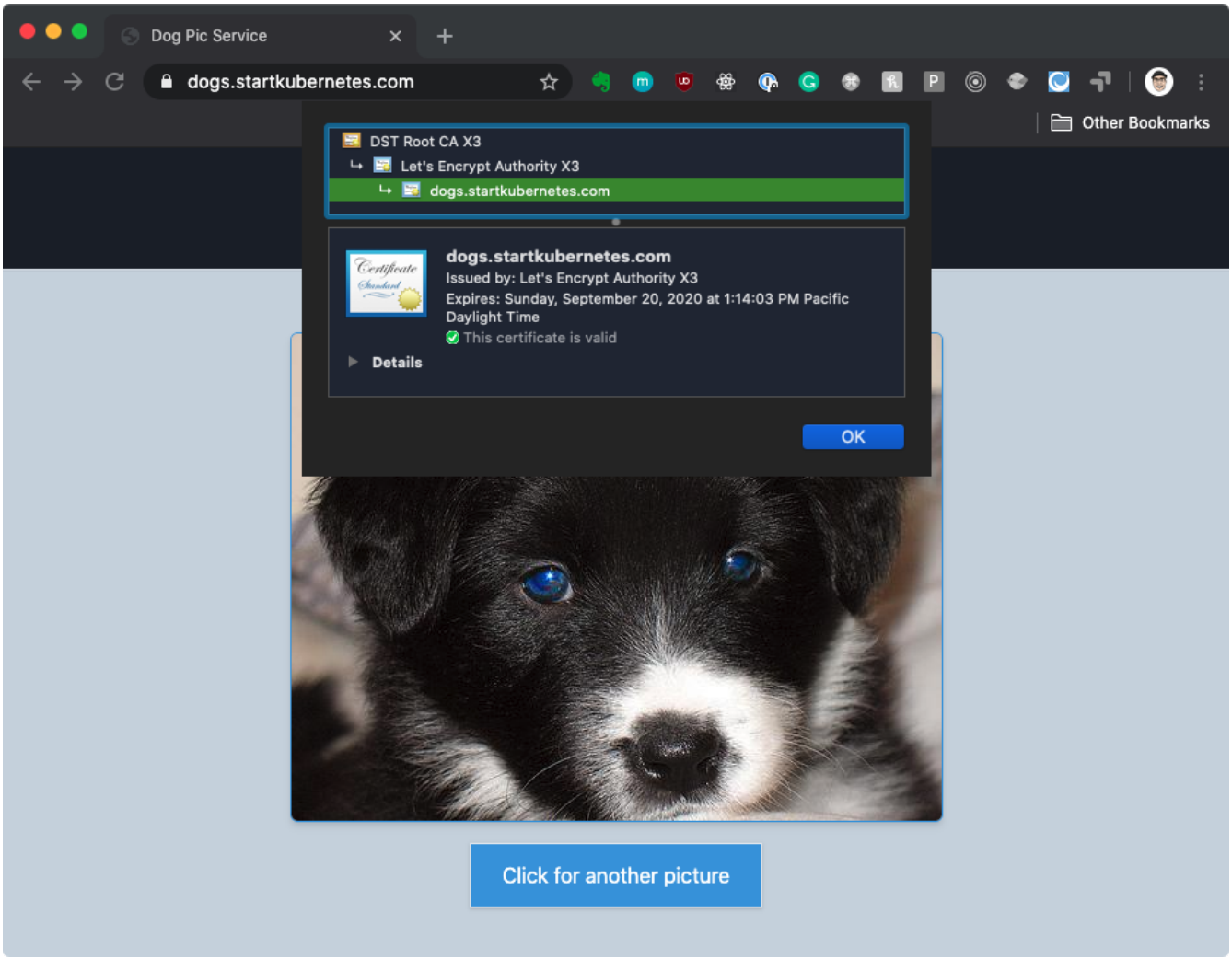


Figure 49. Dog Pic Website via HTTPS

Cleanup

Use the commands below to remove the everything you installed in this section:

```
kubectl delete cert ambassador-certs
kubectl delete secret ambassador-certs
kubectl delete -f https://www.getambassador.io/yaml/ambassador/ambassador-crds.yaml
kubectl delete -f https://www.getambassador.io/yaml/ambassador/ambassador-rbac.yaml
kubectl delete svc ambassador
helm uninstall cert-manager -n cert-manager
kubectl delete svc dogpic-service challenge-service
kubectl delete deploy dogpic-web
kubectl delete ing my-ingress
```