

[Donate](#)

Stay safe, friends. Learn to code from home. Use our free 2,000 hour curriculum.

20 AUGUST 2020 / [#KUBERNETES](#)

The Kubernetes Handbook



Farhan Hasin Chowdhury



Kubernetes is an open-source container orchestration platform that automates the deployment, management, scaling, and networking of containers.

It was developed by Google using the Go Programming Language, and this amazing technology has been open-source since 2014.

According to the Stack Overflow Developer Survey - 2020,

[Donate](#)

platform.

Apart from being very powerful, Kubernetes is known as quite hard to get started with. I won't say it's easy, but if you are equipped with the prerequisites and go through this guide attentively and with patience, you should be able to:

- Get a solid understanding of the fundamentals.
- Create and manage Kubernetes clusters.
- Deploy (almost) any application to a Kubernetes cluster.

Prerequisites

- Familiarity with JavaScript
- Familiarity with the Linux Terminal
- Familiarity with Docker (suggested read: [The Docker Handbook](#))

Project Code

Code for the example projects can be found in the following repository:

[fhsinhy/kubernetes-handbook-projects](#)

Project codes used in "The Kubernetes Handbook" :notebook: - fhsinhy/kubernetes-...

 fhsinhy • GitHub



Spare a  to keep me motivated

Table of Contents

- [Introduction to Container Orchestration and Kubernetes](#)
- [Installing Kubernetes](#)
- [Hello World in Kubernetes](#)
 - [Kubernetes Architecture](#)
 - [Control Plane Components](#)
 - [Node Components](#)
 - [Kubernetes Objects](#)
 - [Pods](#)
 - [Services](#)
 - [The Full Picture](#)
 - [Getting Rid of Kubernetes Resources](#)
- [Declarative Deployment Approach](#)
 - [Writing Your First Set of Configurations](#)
 - [The Kubernetes Dashboard](#)
- [Working with Multi-Container Applications](#)
 - [Deployment Plan](#)
 - [Replication Controllers, Replica Sets, and Deployments](#)
 - [Creating Your First Deployment](#)
 - [Inspecting Kubernetes Resources](#)
 - [Getting Container Logs from Pods](#)
 - [Environment Variables](#)
 - [Creating The Database Deployment](#)

[Donate](#)

- [Dynamic Provisioning of Persistent Volumes](#)
- [Connecting Volumes with Pods](#)
- [Wiring Everything Up](#)
- [Working with Ingress Controllers](#)
 - [Setting-up NGINX Ingress Controller](#)
 - [Secrets and Config Maps in Kubernetes](#)
 - [Performing Update Rollouts in Kubernetes](#)
 - [Combining Configurations](#)
- [Troubleshooting](#)
- [Conclusion](#)

Introduction to Container Orchestration and Kubernetes

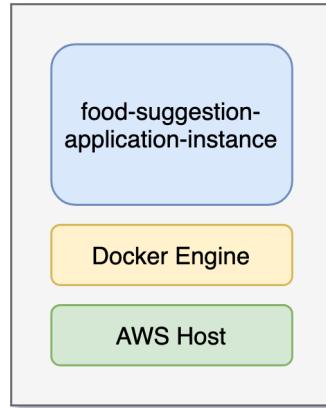
According to [Red Hat](#) –

"Container orchestration is the process of automating the deployment, management, scaling, and networking tasks of containers.

It can be used in any environment where you use containers and can help you deploy the same application across different environments without requiring any redesigning".

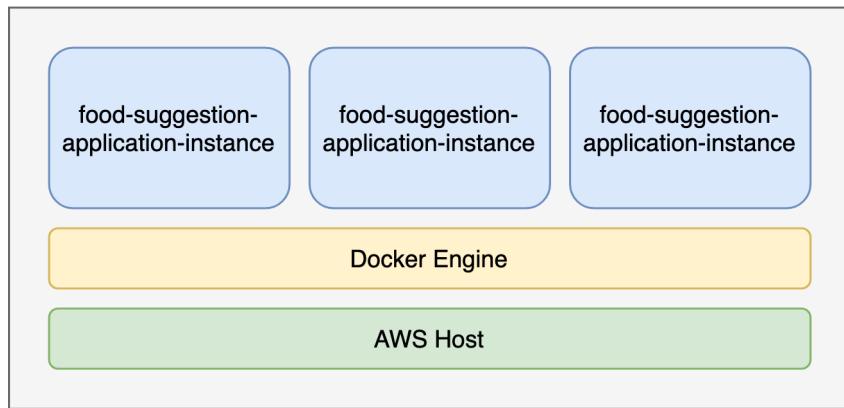
Let me show you an example. Assume that you have developed an amazing application that suggests to people what they should eat depending on the time of day.

Now assume that you've containerized the application using Docker and deployed it on AWS.

[Donate](#)

If the application goes down for any reason, the users lose access to your service immediately.

To solve this issue, you can make multiple copies or replicas of the same application and make it highly available.



Even if one of the instances goes down, the other two will be available to the users.

Now assume that your application has become wildly popular among the night owls and your servers are being flooded with requests at night, while you're sleeping.

What if all the instances go down due to overload? Who's going to

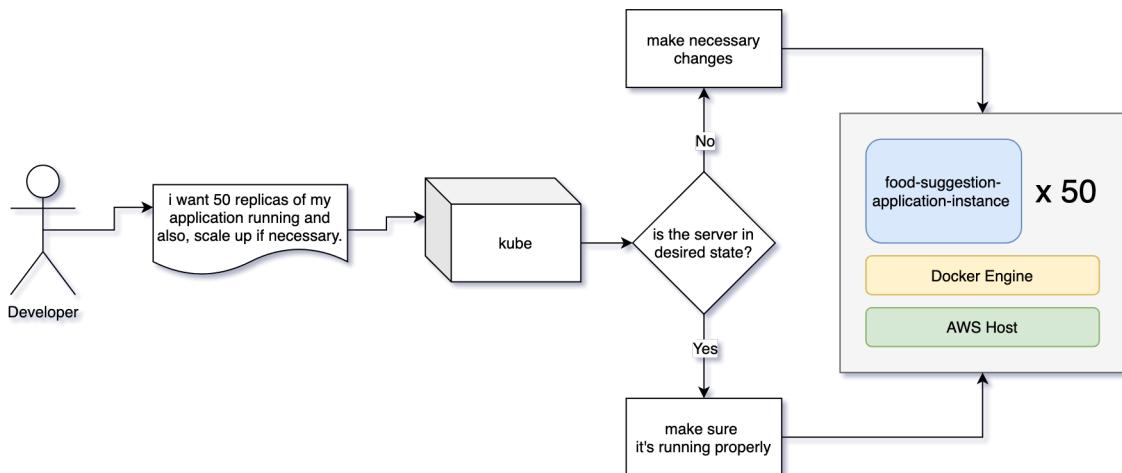
[Donate](#)

application, who's going to check on their health? How are going to set-up the networking so that requests hit the right endpoint? Load balancing is going to be a big concern as well, isn't it?

Kubernetes can make things much easier for these kinds of situations. It's a container orchestration platform that consists of several components and it works tirelessly to keep your servers in the state that you desire.

Assume that you want to have 50 replicas of your application running continuously. Even if there is a sudden rise in the user count, the server needs to be scaled up automatically.

You just tell your needs to Kubernetes and it will do the rest of the heavy lifting for you.



Kubernetes will not only implement the state, it'll also maintain it. It will make additional replicas if any of the old ones dies, manage the networking and storage, rollout or rollback updates, or even upscale the server if ever necessary.

Installing Kubernetes

Running Kubernetes in your local machine is actually a lot different

[Donate](#)

running, you need two programs.

- [minikube](#) - it runs a single-node Kubernetes cluster inside a Virtual Machine (VM) on your local computer.
- [kubectl](#) - The Kubernetes command-line tool, which allows you to run commands against Kubernetes clusters.

Apart from these two programs, you'll also need a hypervisor and a containerization platform. [Docker](#) is the obvious choice for the containerization platform. Recommended hypervisors are as follows:

- [Hyper-V](#) for Windows
- [HyperKit](#) for Mac
- [Docker](#) for Linux

Hyper-V comes built into Windows 10 (Pro, Enterprise, and Education) as an optional feature and can be turned on from the control panel.

HyperKit comes bundled with Docker Desktop for Mac as a core component.

And on Linux, you can bypass the entire hypervisor layer by using Docker directly. It's much faster than using any hypervisor and is the recommended way to run Kubernetes on Linux.

You may go ahead and install any of the above mentioned hypervisors. Or if you want to keep things simple, just get [VirtualBox](#).

For the rest of the article, I'll assume that you're using VirtualBox.

[Donate](#)

shouldn't be that much of a difference.

I'll be using `minikube` with the Docker driver on a [Ubuntu](#) machine throughout the entire article.

Once you have installed the hypervisor and the containerization platform, it's time to install the `minikube` and `kubectl` programs.

`kubectl` usually comes bundled with Docker Desktop on Mac and Windows. Installation instructions for Linux can be found [here](#).

`minikube`, on the other hand, has to be installed on all three of the systems. You can use [Homebrew](#) on Mac, and [Chocolatey](#) on Windows to install `minikube`. Installation instructions for Linux can be found [here](#).

Once you've installed them, you can test out both programs by executing the following commands:

```
minikube version
```

```
# minikube version: v1.12.1
# commit: 5664228288552de9f3a446ea4f51c6f29bbdd0e0
```

```
kubectl version
```

```
# Client Version: version.Info{Major:"1", Minor:"18", GitVersion:
# Server Version: version.Info{Major:"1", Minor:"18", GitVersion:
```

If you've downloaded the right versions for your operating system and have set up the paths properly, you should be ready to go.

As I've already mentioned, `minikube` runs a single-node Kubernetes cluster inside a Virtual Machine (VM) on your local computer. I'll explain clusters and nodes in greater details in an upcoming section.

[Donate](#)

hypervisor of choice and treats that as a Kubernetes cluster.

If you face any problems in this section please have a look at the [Troubleshooting](#) section at the end of this article.

Before you start `minikube`, you have to set the correct hypervisor driver for it to use. To set VirtualBox as the default driver, execute the following command:

```
minikube config set driver virtualbox  
# ! These changes will take effect upon a minikube delete and th
```

You can replace `virtualbox` with `hyperv`, `hyperkit`, or `docker` as per your preference. This command is necessary for the first time only.

To start `minikube`, execute the following command:

```
minikube start  
# 😊 minikube v1.12.1 on Ubuntu 20.04  
# ✨ Using the virtualbox driver based on existing profile  
# 👍 Starting control plane node minikube in cluster minikube  
# 🚀 Updating the running virtualbox "minikube" VM ...  
# 🐳 Preparing Kubernetes v1.18.3 on Docker 19.03.12 ...  
# 🔎 Verifying Kubernetes components...  
# 🌟 Enabled addons: default-storageclass, storage-provisioner  
# 💪 Done! kubectl is now configured to use "minikube"
```

You can stop `minikube` by executing the `minikube stop` command.

[Donate](#)

Now that you have Kubernetes on your local system, it's time to get your hands dirty. In this example you'll be deploying a very simple application to your local cluster and getting familiar with the fundamentals.

There will be terminologies like **pod**, **service**, **load balancer**, and so on in this section. Don't stress if you don't understand them right away. I'll go into great details explaining each of them in [The Full Picture](#) sub-section.

If you've started `minikube` in the previous section then you're ready to go. Otherwise you'll have to start it now. Once `minikube` has started, execute the following command in your terminal:

```
kubectl run hello-kube --image=fhsinchy/hello-kube --port=80  
# pod/hello-kube created
```

You'll see the `pod/hello-kube created` message almost immediately. The [run](#) command runs the given container image inside a pod.

Pods are like a box that encapsulates a container. To make sure the pod has been created and is running, execute the following command:

```
kubectl get pod  
  
# NAME      READY  STATUS    RESTARTS   AGE  
# hello-kube  1/1    Running   0          3m3s
```

[Donate](#)

like `ContainerCreating`, wait for a minute or two and check again.

Pods by default are inaccessible from outside the cluster. To make them accessible, you have to expose them using a service. So, once the pod is up and running, execute the following command to expose the pod:

```
kubectl expose pod hello-kube --type=LoadBalancer --port=80
# service/hello-kube exposed
```

To make sure the load balancer service has been created successfully, execute the following command:

```
kubectl get service
```

#	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
#	hello-kube	LoadBalancer	10.109.60.75	<pending>	80:308
#	kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP

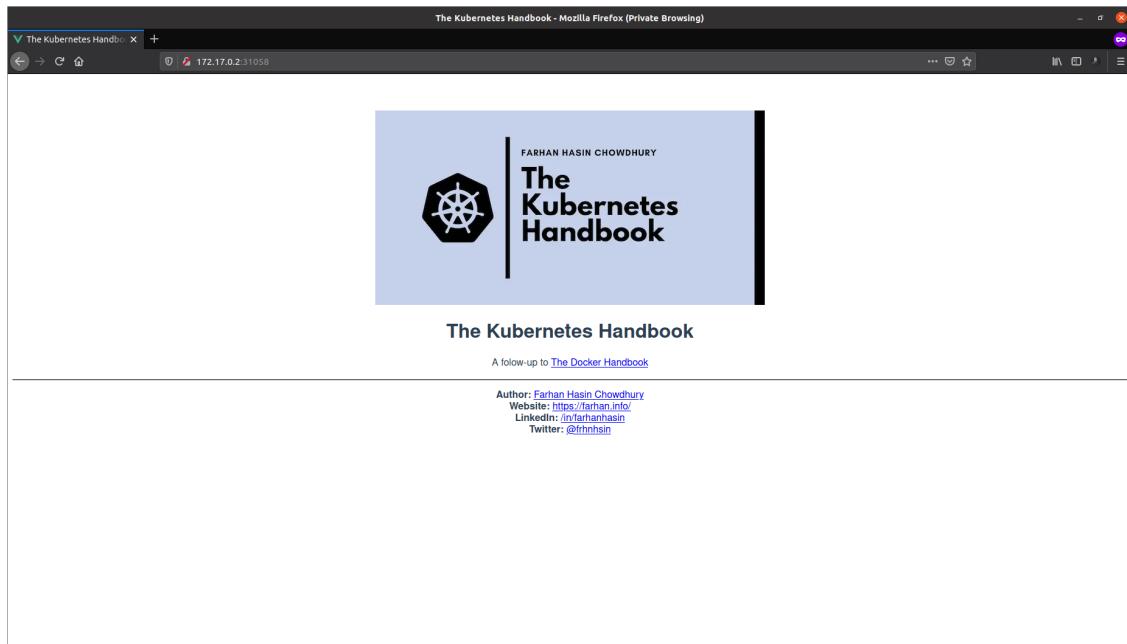
Make sure you see the `hello-kube` service in the list. Now that you have a pod running that is exposed, you can go ahead and access that. Execute the following command to do so:

```
minikube service hello-kube
```

```
# |-----|-----|-----|-----|
# | NAMESPACE | NAME | TARGET PORT | URL
# |-----|-----|-----|-----|
# | default | hello-kube | 80 | http://192.168.99.101:80 |
# |-----|-----|-----|-----|
# 🎉 Opening service default/hello-kube in default browser...
```

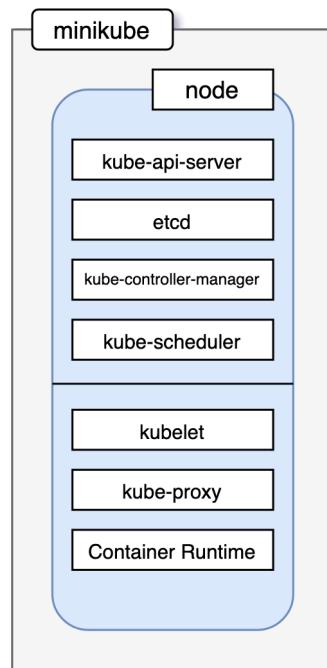
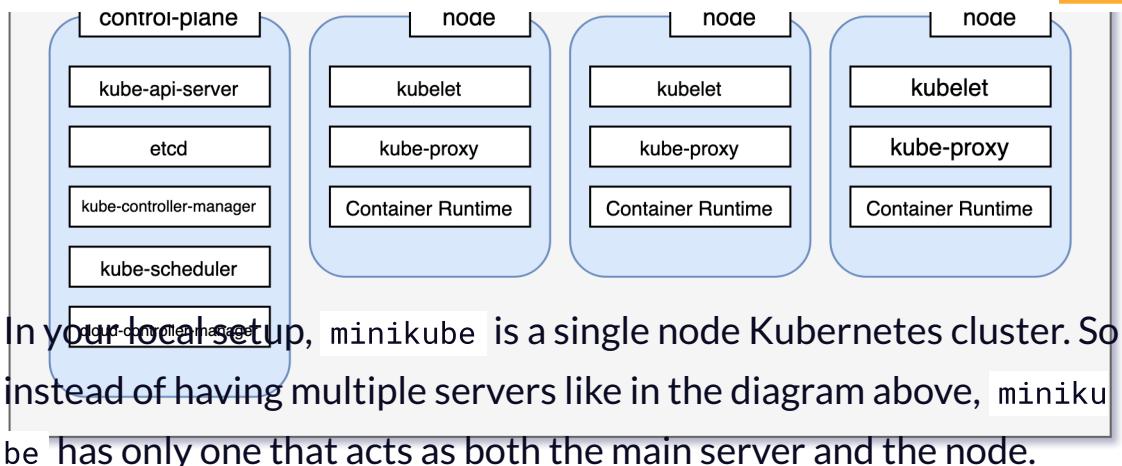
[Donate](#)

Your default web browser should open automatically and you should see something like this:



Kubernetes Architecture

In the world of Kubernetes, a **node** can be either a physical or a virtual machine with a given role. A collection of such machines or servers using a shared network to communicate between each other is called a **cluster**.

[Donate](#)

Each server in a Kubernetes cluster gets a role. There are two possible roles:

- **control-plane** – Makes most of the necessary decisions and acts as sort of the brains of the entire cluster. This can be a single server or a group of server in larger projects.
- **node** – Responsible for running workloads. These servers are usually micro managed by the control plane and carries out various tasks following supplied instructions.

[Donate](#)

The number and type of those components can vary depending on the role a server has in your cluster. That means the nodes do not have all the components that the control plane has.

In the upcoming subsections, you'll have a more detailed look into the individual components that make up a Kubernetes cluster.

Control Plane Components

The control plane in a Kubernetes cluster consists of **five** components. These are as follows:

1. **kube-api-server**: This acts as the entrance to the Kubernetes control plane, responsible for validating and processing requests delivered using client libraries like the `kubectl` program.
2. **etcd**: This is a distributed key-value store which acts as the single source of truth about your cluster. It holds configuration data and information about the state of the cluster. etcd is an open-source project and is developed by the folks behind Red Hat. The source code of the project is hosted on the etcd-io/etcd GitHub repo.
3. **kube-controller-manager**: The controllers in Kubernetes are responsible for controlling the state of the cluster. When you let Kubernetes know what you want in your cluster, the controllers make sure that your request is fulfilled. The `kube-controller-manager` is all the controller processes grouped into a single process.
4. **kube-scheduler**: Assigning task to a certain node considering its available resources and the requirements of the task is known as scheduling. The `kube-scheduler` component does the task of scheduling in Kubernetes making sure none of the servers in the cluster is overloaded.

[Donate](#)

environment, this component lets you wire-up your cluster with your cloud provider's ([GKE/EKS](#)) API. This way, the components that interact with that cloud platform stays isolated from components that just interact with your cluster. In a local cluster like `minikube`, this component doesn't exist.

Node Components

Compared to the control plane, nodes have a very small number of components. These components are as follows:

1. **kubelet**: This service acts as the gateway between the control plain and each of the nodes in a cluster. Every instructions from the control plain towards the nodes, passes through this service. It also interacts with the `etcd` store to keep the state information updated.
2. **kube-proxy**: This small service runs on each node server and maintains network rules on them. Any network request that reaches a service inside your cluster, passes through this service.
3. **Container Runtime**: Kubernetes is a container orchestration tool hence it runs applications in containers. This means that every node needs to have a container runtime like [Docker](#) or [rkt](#) or [cri-o](#).

Kubernetes Objects

According to the Kubernetes [documentation](#) –

"Objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your

[Donate](#)

applications are running, the resources available to them, and the policies around their behaviour."

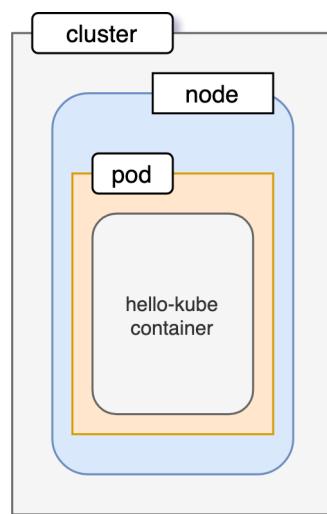
When you create a Kubernetes object, you're effectively telling the Kubernetes system that you want this object to exist no matter what and the Kubernetes system will constantly work to keep the object running.

Pods

According to the Kubernetes [documentation](#) –

"Pods are the smallest deployable units of computing that you can create and manage in Kubernetes".

A pod usually encapsulates one or more containers that are closely related sharing a life cycle and consumable resources.



Although a pod can house more than one container, you shouldn't just put containers in a pod willy nilly. Containers in a pod must be so closely related, that they can be treated as a single application.

As an example, your back-end API may depend on the database but

[Donate](#)

Throughout this entire article, you won't see any pod that has more than one container running.

Usually, you should not manage a pod directly. Instead, you should work with higher level objects that can provide you much better manageability. You'll learn about these higher level objects in later sections.

Services

According to the Kubernetes [documentation](#) –

"A service in Kubernetes is an abstract way to expose an application running on a set of pods as a network service".

Kubernetes pods are ephemeral in nature. They get created and after some time when they get destroyed, they do not get recycled.

Instead new identical pods take the places of the old ones. Some higher level Kubernetes objects are even capable of creating and destroying pods dynamically.

A new IP address is assigned to each pod at the time of their creation. But in case of a high level object that can create, destroy, and group together a number of pods, the set of pods running in one moment in time could be different from the set of pods running that application a moment later.

This leads to a problem: if some set of pods in your cluster depends on another set of pods within your cluster, how do they find out and keep track of each other's IP addresses?

The Kubernetes [documentation](#) says –

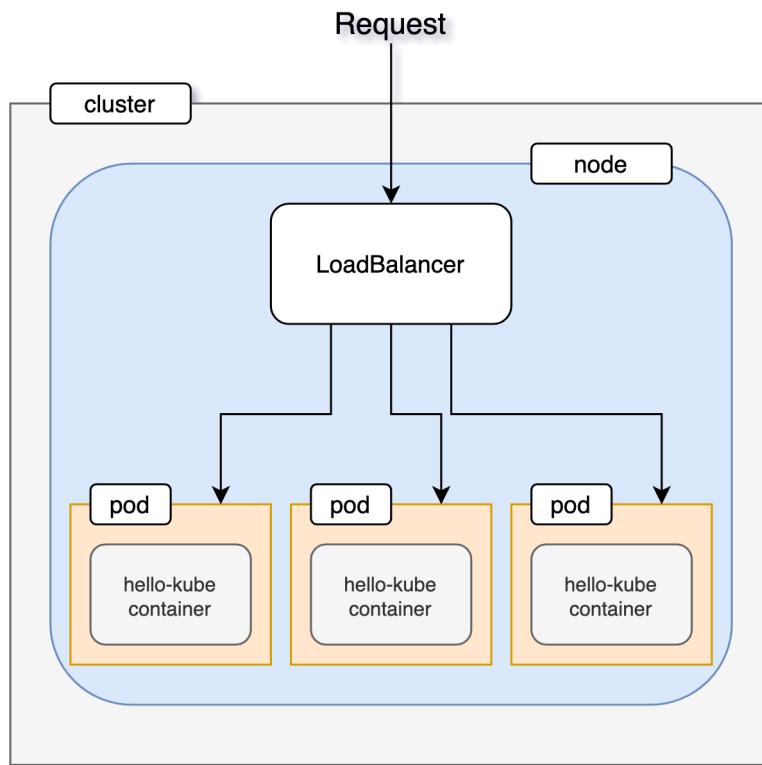
"a Service is an abstraction which defines a logical set of Pods

[Donate](#)

Which essentially means that a Service groups together a number of pods that perform the same function and presents them as a single entity.

This way, the confusion of keeping track of multiple pods goes out of the window as that single Service now acts as a sort of communicator for all of them.

In the `hello-kube` example, you created a `LoadBalancer` type of service which allows requests from outside the cluster connect to pods running inside the cluster.



Any time you need to give access to one or more pods to another application or to something outside of the cluster, you should create a service.

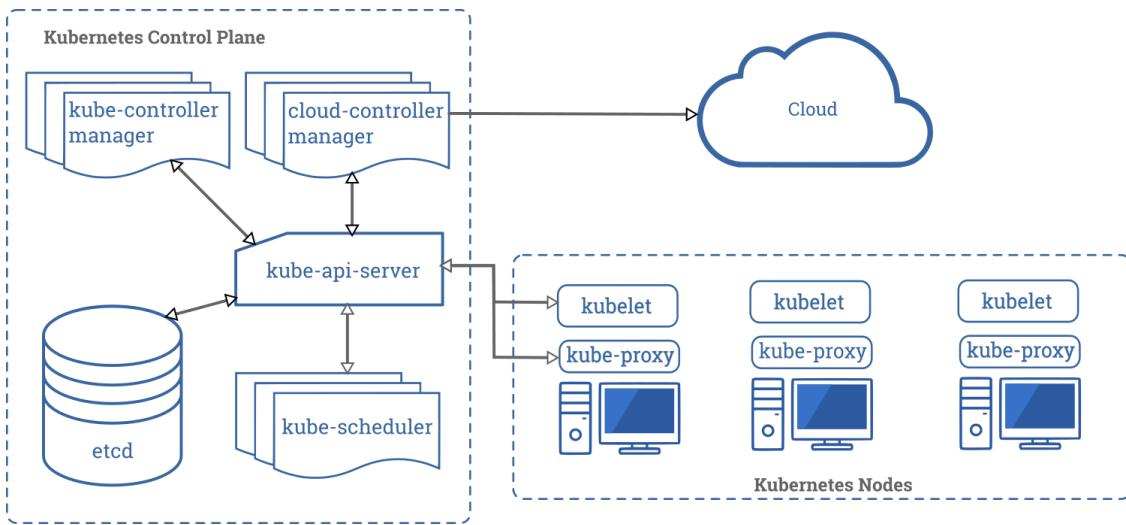
For instance, if you have a set of pods running web servers that

[Donate](#)

necessary abstraction.

The Full Picture

Now that you have a proper understanding of the individual Kubernetes components, here is a visual representation of how they work together behind the scenes:



<https://kubernetes.io/docs/concepts/overview/components/>

Before I get into explaining the individual details, have a look at what the [Kubernetes documentation](#) has to say –

"To work with Kubernetes objects – whether to create, modify, or delete them – you'll need to use the [Kubernetes API](#). When you use the `kubectl` command-line interface, the CLI makes the necessary Kubernetes API calls for you."

The first command that you ran was the `run` command. It was as follows:

```
kubectl run hello-kube --image=fhsinchy/hello-kube --port=80
```

[Donate](#)~~THE RUN COMMAND IS RESPONSIBLE FOR CREATING A NEW POD CONTAINING~~

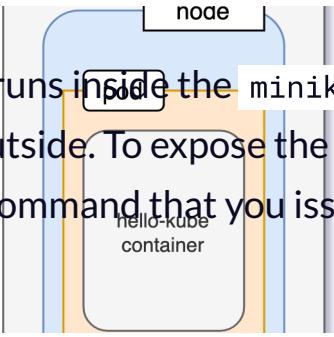
the given image. Once you've issued this command, following sets of events occur inside the Kubernetes cluster:

- The `kube-api-server` component receives the request, validates it and processes it.
- The `kube-api-server` then communicates with the `kubelet` component on the node and provides the instructions necessary for creating the pod.
- The `kubelet` component then starts working on making the pod up and running and also keeps the state information updated in the `etcd` store.

Generic syntax for the `run` command is as follows:

```
kubectl run <pod name> --image=<image name> --port=<port to expos
```

You can run any valid container image inside a pod. The [fhsinchy/hello-kube](#) Docker image contains a very simple JavaScript application that runs on port 80 inside the container. The `--port=80` option allows the pod to expose port 80 from inside the container.

[Donate](#)

The newly created pod runs **inside the minikube cluster** and is inaccessible from the outside. To expose the pod and make it accessible, the second command that you issued was as follows:

```
kubectl expose pod hello-kube --type=LoadBalancer --port=80
```

The `expose` command is responsible for creating a Kubernetes service of type `LoadBalancer` that allows users to access the application running inside the pod.

Just like the `run` command, the `expose` command execution goes through same sort of steps inside the cluster. But instead of a pod, the `kube-api-server` provides instructions necessary for creating a service in this case to the `kubelet` component.

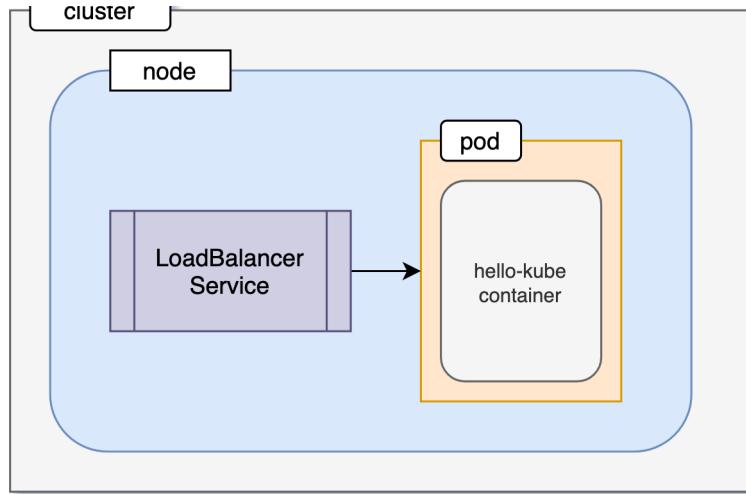
Generic syntax for the `expose` command is as follows:

```
kubectl expose <resource kind to expose> <resource name> --type=<
```

The object type can be any valid Kubernetes object type. The name has to match up with the object name you're trying to expose.

`--type` indicates the type of service you want. There are four different types of services available for internal or external networking.

Lastly, the `--port` is the port number you want to expose from the running container.

[Donate](#)

Once the service has been created, the last piece of the puzzle was to access the application running inside the pod. To do that, the command you executed was as follows:

```
minikube service hello-kube
```

Unlike the previous ones, this last command doesn't go to the `kube-api-server`. Rather it communicates with the local cluster using the `minikube` program. The `service` command for `minikube` returns a full URL for a given service.

When you created the `hello-kube` pod with the `--port=80` option, you instructed Kubernetes to let the pod expose port 80 from inside the container but it wasn't accessible from outside the cluster.

Then when you created the `LoadBalancer` service with the `--port=80` option, it mapped port 80 from that container to an arbitrary port in the local system making it accessible from outside the cluster.

On my system, the `service` command returns `192.168.99.101:308`

[Donate](#)

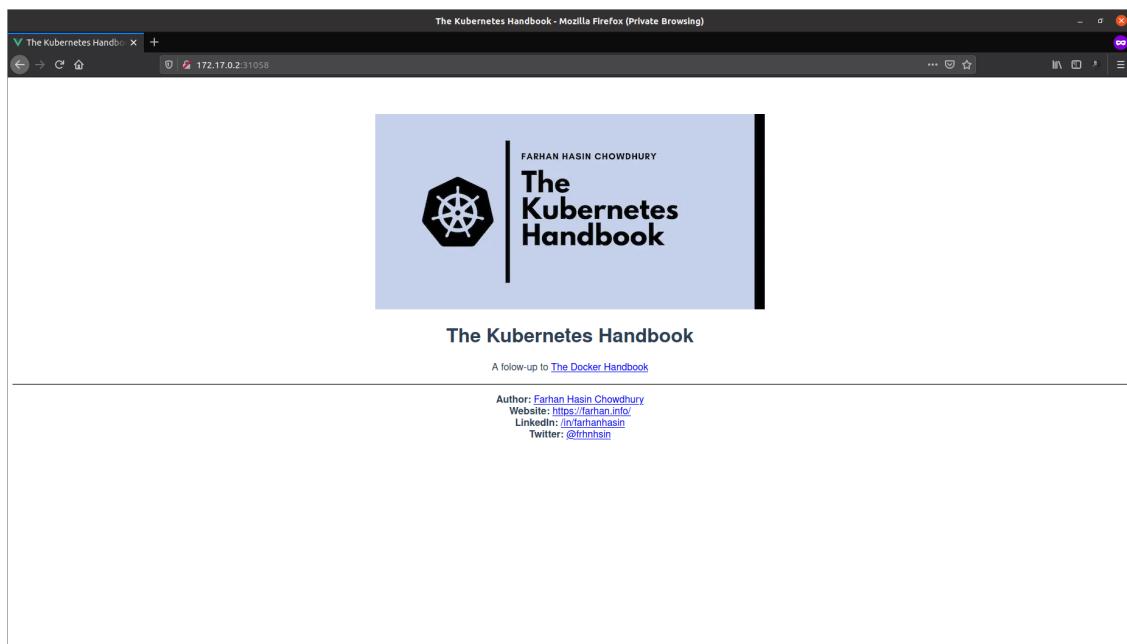
ube virtual machine. You can verify this by executing the following command:

```
minikube ip  
# 192.168.99.101
```

To verify that the 30484 port points to port 80 inside the pod, you can execute the following command:

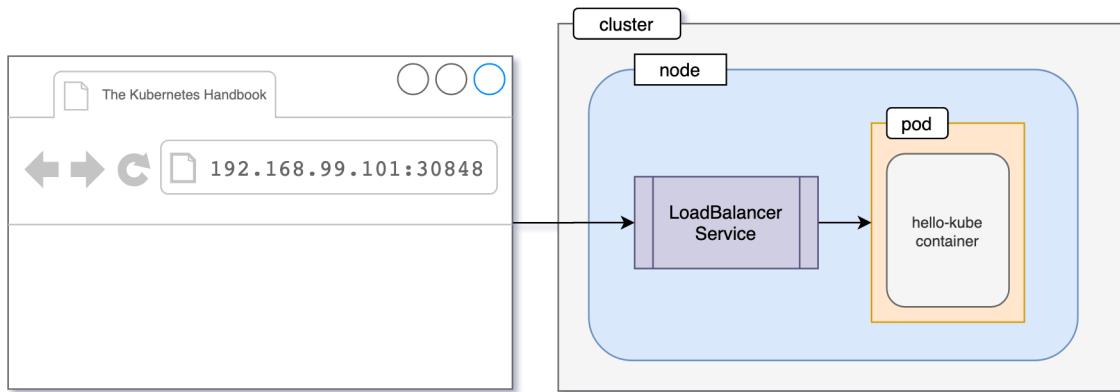
```
kubectl get service hello-kube  
# NAME          TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)  
# hello-kube    LoadBalancer   10.109.60.75   <pending>       80:30484/TCP
```

On the PORT(S) column, you can see that port 80 indeed maps to port 30484 on the local system. So instead of running the service command you can just inspect the IP and port and then put it into your browser manually to access the hello-kube application.



[Donate](#)

Now, the final state of the cluster can be visualized as follows:



If you're coming from Docker, then the significance of using a service in order to expose a pod may seem a bit too verbose to you at the moment.

But as you go into the examples that deal with more than one pod, you'll start to appreciate everything that Kubernetes has to offer.

Getting Rid of Kubernetes Resources

Now that you know how to create Kubernetes resources like pods and Services, you need to know how to get rid of them. The only way to get rid of a Kubernetes resource is to delete it.

You can do that by using the `delete` command for `kubectl`. Generic syntax of the command is as follows:

```
kubectl delete <resource type> <resource name>
```

[Donate](#)

```
kubectl delete pod hello-kube  
# pod "hello-kube" deleted
```

And to delete a service named `hello-kube` the command will be as follows:

```
kubectl delete service hello-kube  
# service "hello-kube" deleted
```

Or if you're in a destructive mood, you can delete all objects of a kind in one go using the `--all` option for the `delete` command. Generic syntax for the option is as follows:

```
kubectl delete <object type> --all
```

So to delete all pods and services you have to execute `kubectl delete pod --all` and `kubectl delete service --all` respectively.

Declarative Deployment Approach

To be honest, the `hello-kube` example you just saw in the previous section is not an ideal way of performing deployment with Kubernetes.

[Donate](#)

approach which means you had to execute every command one after the another manually. Taking an imperative approach defies the entire point of Kubernetes.

An ideal approach to deployment with Kubernetes is the **declarative approach**. In it you, as a developer, let Kubernetes know the state you desire your servers to be in and Kubernetes figures out a way to implement that.

In this section you'll be deploying the same `hello-kube` application in a declarative approach.

If you haven't already cloned the code repository linked above, then go ahead and grab that now.

Once you have that, go inside the `hello-kube` directory. This directory contains the code for the `hello-kube` application as well as the `Dockerfile` for building the image.

```
└── Dockerfile
└── index.html
└── package.json
└── public
└── src
```

2 directories, 3 files

The JavaScript code lives inside the `src` folder but that's not of interest to you. The file you should be looking at is the `Dockerfile` because it can give you insight into how you should plan your deployment. The contents of the `Dockerfile` are as follows:

```
FROM node as builder
```

[Donate](#)

```
COPY ./package.json ./
RUN npm install
COPY ..
RUN npm run build

EXPOSE 80

FROM nginx
COPY --from=builder /usr/app/dist /usr/share/nginx/html
```

As you can see, this is a multi-staged build process.

- The first stage uses `node` as the base image and compiles the JavaScript application into a bunch of production ready files.
- The second stage copies the files built during the first stage, and pastes them inside the default NGINX document root.
Given that the base image for the second phase is `nginx`, the resulting image will be an `nginx` image serving the files built during the first phase on port 80 (default port for `nginx`).

Now to deploy this application on Kubernetes, you'll have to find a way to run the image as a container and make port 80 accessible from the outside world.

Writing Your First Set of Configurations

In the declarative approach, instead of issuing individual commands in the terminal, you instead write down the necessary configuration in a YAML file and feed that to Kubernetes.

In the `hello-kube` project directory, create another directory named `k8s`. `k8s` is short for k(ubernetes = 8 character)s.

You don't need to name the folder this way, you can name it

It's not even necessary to keep it within the project directory. These configuration files can live anywhere in your computer, as they have no relation to the project source code.

Now inside that `k8s` directory, create a new file named `hello-kube-pod.yaml`. I will go ahead and write the code for the file first and then I'll go line by line and explain it to you. The content for this file is as follows:

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-kube-pod
  labels:
    component: web
spec:
  containers:
    - name: hello-kube
      image: fhsinchy/hello-kube
      ports:
        - containerPort: 80
```

Every valid Kubernetes configuration file has four required fields.

They are as follows:

- `apiVersion` : Which version of the Kubernetes API you're using to create this object. This value may change depending on the kind of object you are creating. For creating a `Pod` the required version is `v1` .
- `kind` : What kind of object you want to create. Objects in Kubernetes can be of many kinds. As you go through the article, you'll learn about a lot of them, but for now, just understand that you're creating a `Pod` object.

[Donate](#)

Under this field you can have information like `name`, `labels`, `annotation` etc. The `metadata.name` string will show up on the terminal and will be used in `kubectl` commands. The key value pair under the `metadata.labels` field doesn't have to be components: `web`. You can give it any label like `app: hello-kube`. This value will be used as the selector when creating the `LoadBalancer` service very soon.

- `spec` : contains the state you desire for the object. The `spec.containers` sub-field contains information about the containers that will run inside this `Pod`. The `spec.containers.name` value is what the container runtime inside the node will assign to the newly created container. The `spec.containers.image` is the container image to be used for creating this container. And the `spec.containers.ports` field holds configuration regarding various ports configuration. `containerPort: 80` indicates that you want to expose port 80 from the container.

Now to feed this configuration file to Kubernetes, you'll use the `apply` command. Generic syntax for the command is as follows:

```
kubectl apply -f <configuration file>
```

To feed a configuration file named `hello-kube-pod.yaml`, the command will be as follows:

```
kubectl apply -f hello-kube-pod.yaml  
# pod/hello-kube-pod created
```

[Donate](#)

command:

```
kubectl get pod
```

```
# NAME      READY   STATUS    RESTARTS   AGE
# hello-kube 1/1     Running   0          3m3s
```

You should see `Running` in the `STATUS` column. If you see something like `ContainerCreating` wait for a minute or two and check again.

Once the `Pod` is up and running, it's time for you to write the configuration file for the `LoadBalancer` service.

Create another file inside the `k8s` directory called `hello-kube-load-balancer-service.yaml` and put following code in it:

```
apiVersion: v1
kind: Service
metadata:
  name: hello-kube-load-balancer-service
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 80
  selector:
    component: web
```

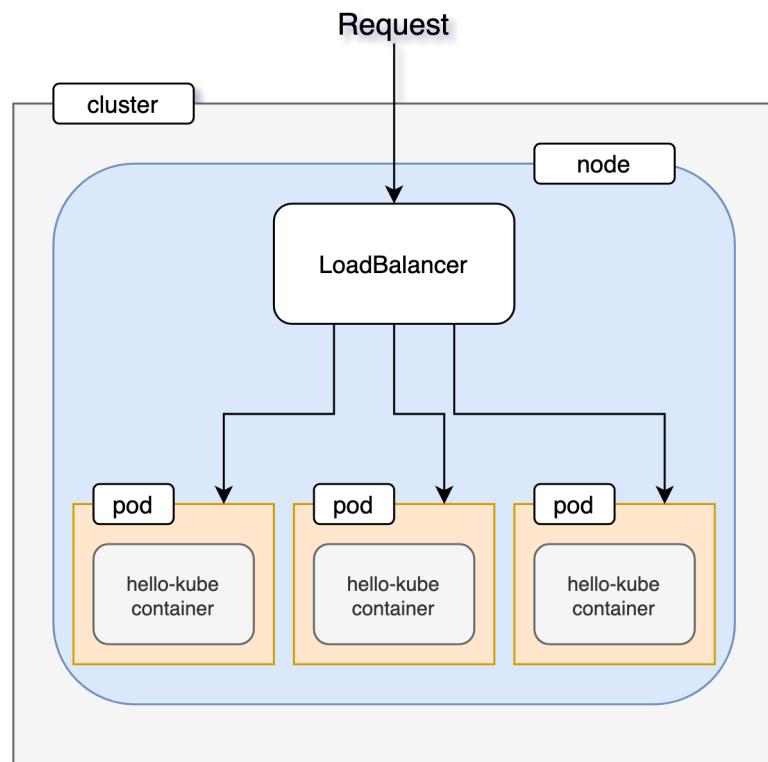
Like the previous configuration file, `apiVersion`, `kind`, and `meta`
`da` fields serve the same purpose here. As you can see there are no
`labels` field inside `metadata` here. That's because a service selects
other objects using `labels`, other objects don't select a service.

[Donate](#)

objects, other objects don't set-up an access policy for a service.

Inside the `spec` field you can see a new set of values. Unlike a `Pod`, services have four types. These are `ClusterIP`, `NodePort`, `LoadBalancer`, and `ExternalName`.

In this example, you're using the type `LoadBalancer`, which is the standard way for exposing a service outside the cluster. This service will give you an IP address that you can then use to connect to the applications running inside your cluster.



The `LoadBalancer` type requires two port values to work properly. Under the `ports` field, the `port` value is for accessing the pod itself and its value can be anything you want.

The `targetPort` value is the one from inside the container and has to match up with the port that you want to expose from inside the

[Donate](#)

I've already said that the `hello-kube` application runs on port 80 inside the container. You've even exposed this port in the `Pod` configuration file, so the `targetPort` will be `80`.

The `selector` field is used to identify the objects that will be connected to this service. The `component: web` key-value pair has to match up with the key-value pair under the `labels` field in the `Pod` configuration file. If you've used some other key value pair like `app: hello-kube` in that configuration file, use that instead.

To feed this file to Kubernetes you will again use the `apply` command. The command for feeding a file named `hello-kube-load-balancer-service.yaml` will be as follows:

```
kubectl apply -f hello-kube-load-balancer-service.yaml  
# service/hello-kube-load-balancer-service created
```

To make sure the load balancer has been created successfully execute the following command:

```
kubectl get service  
  
# NAME                      TYPE        CLUSTER-IP  
# hello-kube-load-balancer-service   LoadBalancer  10.107.231.12  
# kubernetes                   ClusterIP  10.96.0.1
```

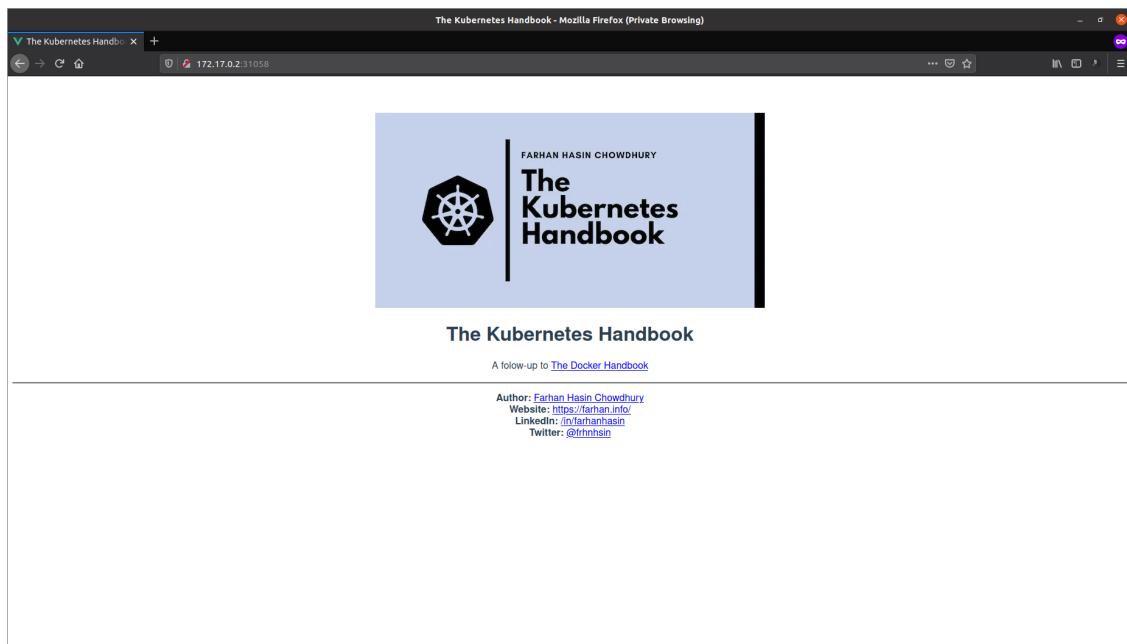
Make sure you see the `hello-kube-load-balancer-service` name in the list. Now that you have a pod running that is exposed, you can go ahead and access that. Execute the following command to do so:

[Donate](#)

minikube service hello-kube

```
# |-----|-----|-----|-----|-----|
# | NAMESPACE |           NAME           | TARGET PORT |
# |-----|-----|-----|-----|-----|
# | default   | hello-kube-load-balancer-service |     80    |
# |-----|-----|-----|-----|-----|
# 🎉  Opening service default/hello-kube-load-balancer in default
```

Your default web browser should open automatically and you should see something like this:



You can also feed both files together instead of feeding them individually. To do that you can replace the file name with the directory name as follows:

```
kubectl apply -f k8s
```

```
# service/hello-kube-load-balancer-service created
# pod/hello-kube-pod created
```

[Donate](#)

k8s directory.

If you're inside the k8s directory, you can use a dot (.) to refer to the current directory. When mass applying configurations, it can be a good idea to get rid of resources created previously. That way the possibility of conflicts becomes much lower.

The declarative approach is the ideal one when working with Kubernetes. Except for some special cases, that you'll see near the end of the article.

The Kubernetes Dashboard

In a previous section, you used the `delete` command to get rid of a Kubernetes object.

In this section, however, I thought introducing the dashboard would be great idea. The Kubernetes Dashboard is a graphical UI that you can use to manage your workloads, services, and more.

To launch the Kubernetes Dashboard, execute the following command in your terminal:

```
minikube dashboard

# 🤔 Verifying dashboard health ...
# 🚀 Launching proxy ...
# 🤔 Verifying proxy health ...
# 🎉 Opening http://127.0.0.1:52393/api/v1/namespaces/kubernetes-
```

The dashboard should open automatically in your default browser:

[Donate](#)

Name	Namespace	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created
hello-kube-pod	default	component: web	minikube	Running	0	-	-	34 seconds ago
hello-kube	default	run: hello-kube	minikube	Running	0	-	-	7 minutes ago

The UI is pretty user-friendly and you are free to roam around here. Although it's completely possible to create, manage, and delete objects from this UI, I'll be using the CLI for the rest of this article.

Here in the *Pods* list, you can use the three dots menu on the right side to *Delete* the Pod. You can do the same with the *LoadBalancer* service as well. In fact the *Services* list is conveniently placed right after the *Pods* list.

You can close the dashboard by hitting the `Ctrl + c` key combination or closing the terminal window.

Working with Multi-Container Applications

So far you've worked with applications that run within a single container.

In this section, you'll be working with an application consisting of two containers. You'll also get familiar with `Deployment`, `ClusterIP`, `PersistentVolume`, `PersistentVolumeClaim` and some debugging techniques.

[Donate](#)

with full CRUD functionality. The application uses PostgreSQL as its database system. So you're not only going to deploy the application but also set-up internal networking between the application and the database.

The code for the application is inside the `notes-api` directory inside the project repo.

```
.  
└── api  
└── docker-compose.yaml  
└── postgres
```

2 directories, 1 file

The application source code resides inside the `api` directory and the `postgres` directory contains a `Dockerfile` for creating the custom `postgres` image. The `docker-compose.yaml` file contains the necessary configuration for running the application using `docker-compose`.

Just like with the previous project, you can look into the individual `Dockerfile` for each service to get a sense of how the application runs inside the container.

Or you can just inspect the `docker-compose.yaml` and plan your Kubernetes deployment using that.

```
version: "3.8"
```

```
services:  
  db:  
    build:
```

[Donate](#)

```

environment:
  POSTGRES_PASSWORD: 63eaQB9wtLqmNBpg
  POSTGRES_DB: notesdb

api:
  build:
    context: ./api
    dockerfile: Dockerfile.dev
  ports:
    - 3000:3000
  volumes:
    - /usr/app/node_modules
    - ./api:/usr/app
  environment:
    DB_CONNECTION: pg
    DB_HOST: db
    DB_PORT: 5432
    DB_USER: postgres
    DB_DATABASE: notesdb
    DB_PASSWORD: 63eaQB9wtLqmNBpg

```

Looking at the `api` service definition, you can see that the application runs on port 3000 inside the container. It also requires a bunch of environment variables to function properly.

The volumes can be ignored as they were necessary for development purposes only and the build configuration is Docker-specific. So the two sets of information that you can carry over to your Kubernetes configuration files almost unchanged are as follows:

- Port mappings – because you'll have to expose the same port from the container.
- Environment variables – because these variables are going to be the same across all environments (the values are going to change, though).

The `db` service is even simpler. All it has is a bunch of environment

[Donate](#)

custom one.

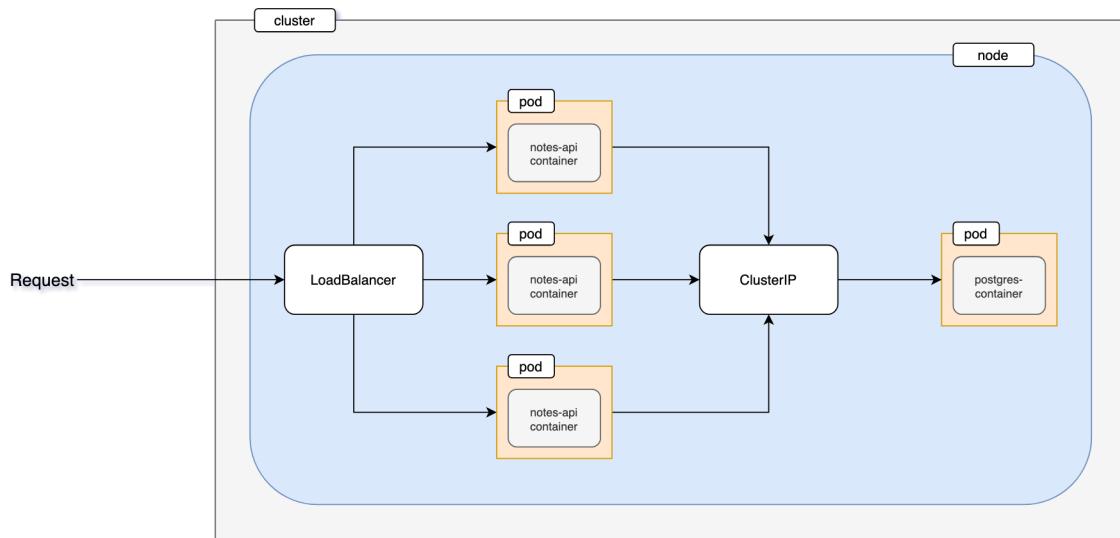
But the only reason for a custom image is if you want the database instance to come with the `notes` table pre-created.

This table is necessary for the application. If you look inside the `postgres/docker-entrypoint-initdb.d` directory, you'll see a file named `notes.sql` which is used for creating the database during initialization.

Deployment Plan

Unlike the previous project you deployed, this project is going to be a bit more complicated.

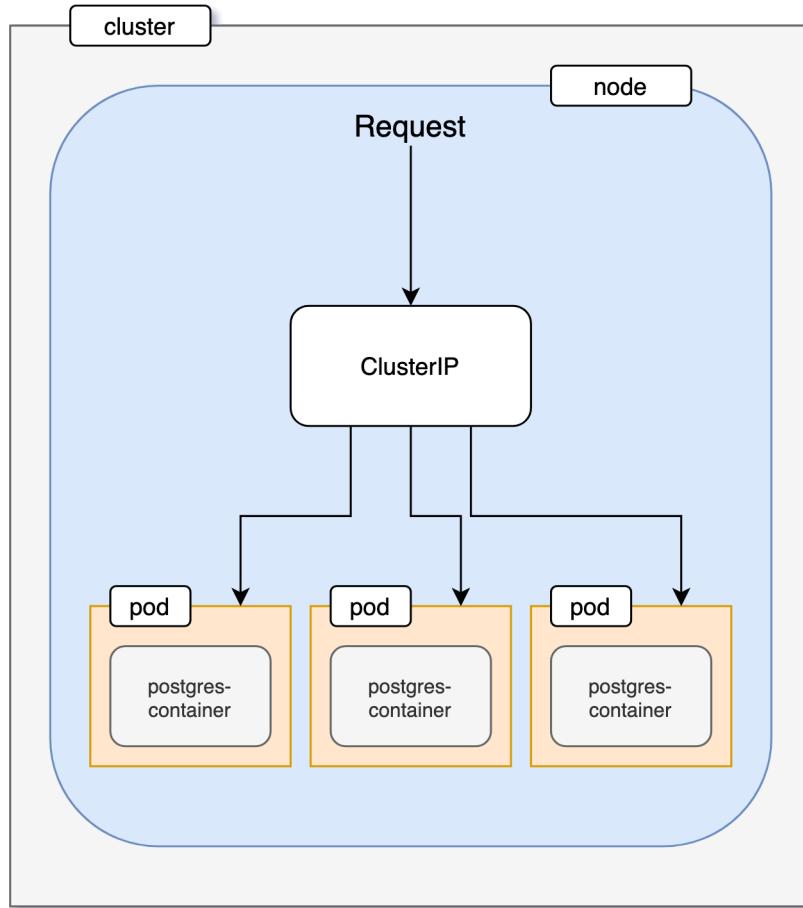
In this project, you'll create not one but three instances of the notes API. These three instances will be exposed outside of the cluster using a `LoadBalancer` service.



Apart from these three instances, there will be another instance of the PostgreSQL database system. All three instances of the notes

[Donate](#)

ClusterIP ~~Service~~ is another type of Kubernetes service that exposes an application within your cluster. That means no outside traffic can reach the application using a ClusterIP service.



In this project, the database has to be accessed by the notes API only, so exposing the database service within the cluster is an ideal choice.

I've already mentioned in a previous section that you shouldn't create pods directly. So in this project, you'll be using a Deployment instead of a Pod .

Replication Controllers, Replica Sets, and Deployments

According to the [Kubernetes documentation](#) -

[Donate](#)

In Kubernetes, controllers are control loops that watch the

state of your cluster, then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state. A control loop is a non-terminating loop that regulates the state of a system."

A `ReplicationController`, as the name suggests allows you to easily create multiple replicas very easily. Once the desired number of replicas are created, the controller will make sure that the state stays that way.

If after some time you decide to lower the number of replicas, then the `ReplicationController` will take actions immediately and get rid of the extra pods.

Otherwise if the number of replicas becomes lower than what you wanted (maybe some of the pods have crashed) the `ReplicationController` will create new ones to match the desired state.

As useful as they may sound to you, the `ReplicationController` is not the recommended way of creating replicas nowadays. A newer API called a `ReplicaSet` has taken the place.

Apart from the fact that a `ReplicaSet` can provide you with a wider range of selection option, both `ReplicationController` and `ReplicaSet` are more or less the same thing.

Having a wider range of selector options is good but what's even better is having more flexibility in terms of rolling out and rolling back updates. This is where another Kubernetes API called a `Deployment` comes in.

A `Deployment` is like an extension to the already nice `ReplicaSet` API. `Deployment` not only allows you to create replicas in no time,

[Donate](#)

function with just one or two `kubectl` commands.

REPLICATIONCONTROLLER	REPLICASET
Allows the creation of multiple pods easily	Allows the creation of mi
The original method of replication in Kubernetes	Has more flexible selecto

In this project, you'll be using a `Deployment` to maintain the application instances.

Creating Your First Deployment

Let's begin by writing the configuration file for the notes API deployment. Create a `k8s` directory inside the `notes-api` project directory.

Inside that directory, create a file named `api-deployment.yaml` and put following content in it:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      component: api
  template:
    metadata:
      labels:
        component: api
  spec:
    containers:
      - name: api
        image: fhsinchy/notes-api
        ports:
          - containerPort: 3000
```

[Donate](#)

In this file, the `apiVersion`, `kind`, `metadata` and `spec` fields serve the same purpose as the previous project. Notable changes in this file from the last one are as follows:

- For creating a Pod, the required `apiVersion` was `v1`. But for creating a Deployment, the required version is `apps/v1`. Kubernetes API versions can be a bit confusing at times, but as you keep working with Kubernetes you'll get the hang of them. Also you can consult the official [docs](#) for example YAML files to refer to. The kind is `Deployment` which is pretty self-explanatory.
- `spec.replicas` defines the number of running replicas. Setting this value to `3` means you let Kubernetes know that you want three instances of your application running at all times.
- `spec.selector` is where you let the `Deployment` know which pods to control. I've already mentioned that a `Deployment` is an extension to `ReplicaSet` and can control a set of Kubernetes objects. Setting `selector.matchLabels` to `component: api` means this `Deployment` will control the pods that have a label of `component: api`. This line is letting Kubernetes know that you want this `Deployment` to control all the pods having the `component: api` label.
- `spec.template` is the template for configuring the pods. It's almost the same as the previous configuration file.

Now to see this configuration in action, apply the file just like in the previous project:

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

[Donate](#)

To make sure the `Deployment` has been created, execute the following command:

```
kubectl get deployment
```

```
# NAME          READY   UP-TO-DATE   AVAILABLE   AGE
# api-deployment  0/3      3           0           2m7s
```

If you look at the `READY` column, you'll see `0/3`. This means the pods have not been created yet. Wait a few minutes and try once again.

```
kubectl get deployment
```

```
# NAME          READY   UP-TO-DATE   AVAILABLE   AGE
# api-deployment  0/3      3           0           28m
```

As you can see, I have waited nearly half an hour and still none of the pods are ready. The API itself is only a few hundred kilobytes. A deployment of this size shouldn't have taken this long. Which means there is a problem and we have to fix that.

Inspecting Kubernetes Resources

Before you can solve a problem, you have to first find out the origin. A good starting point is the `get` command.

You already know the `get` command that prints a table containing important information about one or more Kubernetes resources.

Generic syntax of the command is as follows:

[Donate](#)

```
kubectl get <resource type> <resource name>
```

To run the `get` command on your `api-deployment`, execute the following line of code in your terminal:

```
kubectl get deployment api-deployment
```

```
# NAME          READY   UP-TO-DATE   AVAILABLE   AGE
# api-deployment  0/3       3           0           15m
```

You can omit the `api-deployment` name to get a list of all available deployments. You can also run the `get` command on a configuration file.

If you would like to get information about the deployments described in the `api-deployment.yaml` file, the command should be as follows:

```
kubectl get -f api-deployment
```

```
# NAME          READY   UP-TO-DATE   AVAILABLE   AGE
# api-deployment  0/3       3           0           18m
```

By default, the `get` command shows a very small amount of information. You can get more out of it by using the `-o` option.

The `-o` option sets the output format for the `get` command. You can use the `wide` output format to see more details.

[Donate](#)

#	NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAIN
#	api-deployment	0/3	3	0	19m	api

As you can see, now the list contains more information than before. You can learn about the options for the `get` command from the [official docs](#).

Running `get` on the `Deployment` doesn't spit out anything interesting, to be honest. In such cases, you have to get down to the lower level resources.

Have a look at the pods list and see if you can find something interesting there:

```
kubectl get pod
```

#	NAME	READY	STATUS	RE
#	api-deployment-d59f9c884-88j45	0/1	CrashLoopBackOff	10
#	api-deployment-d59f9c884-96hfr	0/1	CrashLoopBackOff	10
#	api-deployment-d59f9c884-pzdxg	0/1	CrashLoopBackOff	10

Now this is interesting. All the pods have a `STATUS` of `CrashLoopBackOff` which is new. Previously you've only seen `ContainerCreating` and `Running` statuses. You may see `Error` in place of `CrashLoopBackOff` as well.

Looking at the `RESTARTS` column you can see that the pods have been restarted 10 times already. This means for some reason the pods are failing to startup.

Now to get a more detailed look at one of the pods, you can use another command called `describe`. It's a lot like the `get` command.

[Donate](#)

```
kubectl get <resource type> <resource name>
```

To get details of the `api-deployment-d59f9c884-88j45` pod, you can execute the following command:

```
kubectl describe pod api-deployment-d59f9c884-88j45

# Name:          api-deployment-d59f9c884-88j45
# Namespace:     default
# Priority:      0
# Node:          minikube/172.28.80.217
# Start Time:    Sun, 09 Aug 2020 16:01:28 +0600
# Labels:         component=api
#                 pod-template-hash=d59f9c884
# Annotations:   <none>
# Status:        Running
# IP:            172.17.0.4
# IPs:
#   IP:          172.17.0.4
# Controlled By: ReplicaSet/api-deployment-d59f9c884
# Containers:
#   api:
#     Container ID:  docker://d2bc15bda9bf4e6d08f7ca8ff5d3c85936
#     Image:         fhsinchy/notes-api
#     Image ID:     docker-pullable://fhsinchy/notes-api@sha256:...
#     Port:          3000/TCP
#     Host Port:    0/TCP
#     State:        Waiting
#       Reason:     CrashLoopBackOff
#     Last State:   Terminated
#       Reason:     Error
#       Exit Code:  1
#     Started:     Sun, 09 Aug 2020 16:13:12 +0600
#     Finished:    Sun, 09 Aug 2020 16:13:12 +0600
#     Ready:        False
#     Restart Count: 10
#     Environment:  <none>
#     Mounts:
#       /var/run/secrets/kubernetes.io/serviceaccount from default-
#     # Conditions:
```

[Donate](#)

```

#   Ready          False
#   ContainersReady  False
#   PodScheduled    True
# Volumes:
#   default-token-gqfr4:
#       Type:        Secret (a volume populated by a Secret)
#       SecretName: default-token-gqfr4
#       Optional:   false
# QoS Class:      BestEffort
# Node-Selectors: <none>
# Tolerations:   node.kubernetes.io/not-ready:NoExecute for 300s
#                  node.kubernetes.io/unreachable:NoExecute for 300s
# Events:
#   Type  Reason  Age           From
#   ----  -----  --  -----
#   Normal Scheduled <unknown>    default-sched
#   Normal Pulled   2m40s (x4 over 3m47s)  kubelet, mini
#   Normal Created   2m40s (x4 over 3m47s)  kubelet, mini
#   Normal Started   2m40s (x4 over 3m47s)  kubelet, mini
#   Normal Pulling   107s (x5 over 3m56s)   kubelet, mini
#   Warning BackOff  <invalid> (x44 over 3m32s) kubelet, mini

```

The most interesting part in this entire wall of text is the `Events` section. Have a closer look:

Events:			
Type	Reason	Age	From
---	-----	----	-----
Normal	Scheduled	<unknown>	default-sched
Normal	Pulled	2m40s (x4 over 3m47s)	kubelet, mini
Normal	Created	2m40s (x4 over 3m47s)	kubelet, mini
Normal	Started	2m40s (x4 over 3m47s)	kubelet, mini
Normal	Pulling	107s (x5 over 3m56s)	kubelet, mini
Warning	BackOff	<invalid> (x44 over 3m32s)	kubelet, mini

From these events, you can see that the container image was pulled successfully. The container was created as well, but it's evident from the Back-off restarting failed container that the container failed to startup.

[Donate](#)

The `describe` command is very similar to the `get` command and has the same sort of options.

You can omit the `api-deployment-d59f9c884-88j45` name to get information about all available pods. Or you can also use the `-f` option to pass a configuration file to the command. Visit the official [docs](#) to learn more.

Now that you know that there is something wrong with the container, you have to go down to the container level and see what's going on there.

Getting Container Logs from Pods

There is another `kubectl` command called `logs` that can help you to get the container logs from inside a pod. Generic syntax for the command is as follows:

```
kubectl logs <pod>
```

To view the logs inside the `api-deployment-d59f9c884-88j45` pod, the command should be as follows:

```
kubectl logs api-deployment-d59f9c884-88j45

# > api@1.0.0 start /usr/app
# > cross-env NODE_ENV=production node bin/www

# /usr/app/node_modules/knex/lib/client.js:55
#     throw new Error(`knex: Required configuration option 'clie
^

# Error: knex: Required configuration option 'client' is missing.
#       at new Client (/usr/app/node_modules/knex/lib/client.js:55:
```

[Donate](#)

```

#      at Module._compile (internal/modules/cjs/loader.js:1138:30)
#      at Object.Module._extensions..js (internal/modules/cjs/loader.js:1091:12)
#      at Module.load (internal/modules/cjs/loader.js:986:32)
#      at Function.Module._load (internal/modules/cjs/loader.js:878:12)
#      at Module.require (internal/modules/cjs/loader.js:1026:19)
#      at require (internal/modules/cjs/helpers.js:72:18)
#      at Object.<anonymous> (/usr/app/services/index.js:1:14)
# npm ERR! code ELIFECYCLE
# npm ERR! errno 1
# npm ERR! api@1.0.0 start: `cross-env NODE_ENV=production node $(
# npm ERR! Exit status 1
# npm ERR!
# npm ERR! Failed at the api@1.0.0 start script.
# npm ERR! This is probably not a problem with npm. There is likely
# npm ERR! A complete log of this run can be found in:
# npm ERR!     /root/.npm/_logs/2020-08-09T10_28_52_779Z-debug.log

```

Now this is what you need to debug the problem. Looks like the [knex.js](#) library is missing a required value, which is preventing the application from starting. You can learn more about the `logs` command from the official [docs](#).

This is happening because you're missing some required environment variables in the deployment definition.

If you take another look at the `api` service definition inside the `docker-compose.yaml` file, you should see something like this:

```

api:
  build:
    context: ./api
    dockerfile: Dockerfile.dev
  ports:
    - 3000:3000
  volumes:
    - /usr/app/node_modules
    - ./api:/usr/app
  environment:

```

[Donate](#)

```
DB_PORT: 5432
DB_USER: postgres
DB_DATABASE: notesdb
DB_PASSWORD: 63eaQB9wtLqmNBpg
```

These environment variables are required for the application to communicate with the database. So adding these to the deployment configuration should fix the issue.

Environment Variables

Adding environment variables to a Kubernetes configuration file is very straightforward. Open up the `api-deployment.yaml` file and update its content to look like this:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      component: api
  template:
    metadata:
      labels:
        component: api
  spec:
    containers:
      - name: api
        image: fhsinchy/notes-api
        ports:
          - containerPort: 3000

        # these are the environment variables
        env:
          - name: DB_CONNECTION
            value: pg
```

[Donate](#)

The `containers.env` field contains all the environment variables. If you look closely, you'll see that I haven't added all the environment variables from the `docker-compose.yaml` file. I have added only one.

The `DB_CONNECTION` indicates that the application is using a PostgreSQL database. Adding this single variable should fix the problem.

Now apply the configuration file again by executing the following command:

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

Unlike the other times, the output here says that a resource has been `configured`. This is the beauty of Kubernetes. You can just fix issues and re-apply the same configuration file immediately.

Now use the `get` command once more to make sure everything is running properly.

```
kubectl get deployment  
# NAME          READY   UP-TO-DATE   AVAILABLE   AGE  
# api-deployment 3/3     3           3           68m  
  
kubectl get pod  
# NAME                           READY   STATUS    RESTARTS  
# api-deployment-66cdd98546-19x8q  1/1     Running   0  
# api-deployment-66cdd98546-mbfw9  1/1     Running   0  
# api-deployment-66cdd98546-pntxv  1/1     Running   0
```

[Donate](#)

All three pods are running and the `Deployment` is running fine as well.

Creating the Database Deployment

Now that the API is up and running, it's time to write the configuration for the database instance.

Create another file called `postgres-deployment.yaml` inside the `k8s` directory and put the following content in it:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      component: postgres
  template:
    metadata:
      labels:
        component: postgres
  spec:
    containers:
      - name: postgres
        image: fhsinchy/notes-postgres
        ports:
          - containerPort: 5432
        env:
          - name: POSTGRES_PASSWORD
            value: 63eaQB9wtLqmNBpg
          - name: POSTGRES_DB
            value: notesdb
```

The configuration itself is very similar to the previous one. I am not going to explain everything in this file – hopefully you understand it

[Donate](#)

far.

PostgreSQL runs on port 5432 by default, and the `POSTGRES_PASSWORD` variable is required for running the `postgres` container. This password will also be used for connecting to this database by the API.

The `POSTGRES_DB` variable is optional. But because of the way this project has been structured, it's necessary here – otherwise the initialization will fail.

You can learn more about the official [postgres](#) Docker image from their Docker Hub page. For the sake of simplicity, I'm keeping the replica count to 1 in this project.

To apply this file, execute the following command:

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

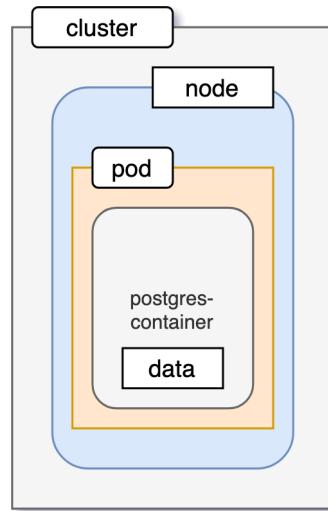
Use the `get` command to ensure that the deployment and the pods are running properly:

```
kubectl get deployment  
  
# NAME           READY   UP-TO-DATE   AVAILABLE   AGE  
# postgres-deployment   1/1     1           1           13m  
  
kubectl get pod  
  
# NAME                           READY   STATUS    RESTARTS  
# postgres-deployment-76fcc75998-mwnb7   1/1     Running   0
```

[Donate](#)

is a big issue with the database deployment.

If you've worked with any database system before, you may already know that databases store data in the filesystem. Right now the database deployment looks like this:

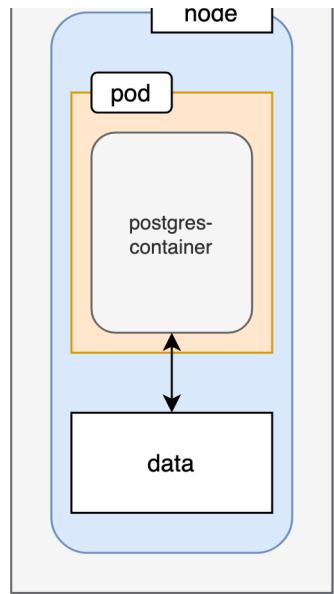


The `postgres` container is encapsulated by a pod. Whatever data is saved stays within the internal filesystem of the container.

Now, if for some reason, the container crashes or the pod encapsulating the container goes down, all data persisted inside the filesystem will be lost.

Upon crashing, Kubernetes will create a new pod to maintain the desired state, but there is no data carry over mechanism between the two pods whatsoever.

To solve this issue, you can store the data in a separate space outside the pod within the cluster.

[Donate](#)

Managing such storage is a distinct problem from managing compute instances. The `PersistentVolume` subsystem in Kubernetes provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed.

Persistent Volumes and Persistent Volume Claims

According to the [Kubernetes documentation](#) —

"A `PersistentVolume` (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using a `StorageClass`. It is a resource in the cluster just like a node is a cluster resource."

Which essentially means that a `PersistentVolume` is a way to take a slice from your storage space and reserve that for a certain pod. Volumes are always consumed by pods and not some high level object like a deployment.

If you want to use a volume with a deployment that has multiple pods, you'll have to go through some additional steps.

[Donate](#)

the `k8s` directory and put following content in that file:

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

The `apiVersion`, `kind`, and `metadata` serve the same purpose as any other configuration file. The `spec` field, however, contains some new fields.

- `spec.storageClassName` indicates the class name for this volume. Assume that a cloud provider has three kinds of storage available. These can be *slow*, *fast*, and *very fast*. The kind of storage you get from the provider will depend on the amount of money you're paying. If you ask for a very fast storage, you'll have to pay more. These different types of storage are the classes. I am using `manual` as an example here. You can use whatever you like in your local cluster.
- `spec.capacity.storage` is the amount of storage this volume will have. I am giving it 5 gigabytes of storage in this project.
- `spec.accessModes` sets the access mode for the volume. There are three possible access modes. `ReadWriteOnce` means the volume can be mounted as read-write by a single node. `ReadWriteMany` on the other hand means the volume

[Donate](#)

means the volume can be mounted read-only by many nodes.

- `spec.hostPath` is something development specific. It indicates the directory in your local single node cluster that'll be treated as persistent volume. `/mnt/data` means that the data saved in this persistent volume will live inside the `/mnt/data` directory in the cluster.

To apply this file, execute the following command:

```
kubectl apply -f database-persistent-storage.yaml  
# persistentvolume/database-persistent-volume created
```

Now use the `get` command to verify that the volume has been created:

```
kubectl get persistentvolume  
# NAME                      CAPACITY   ACCESS MODES  RECLAIM  
# database-persistent-volume  5Gi        RWO          Retain
```

Now that the persistent volume has been created, you need a way to let the postgres pod access it. This is where a `PersistentVolumeClaim` (PVC) comes in.

A persistent volume claim is a request for storage by a pod. Assume that in a cluster, you have quite a lot of volumes. This claim will define the characteristics that a volume must meet to be able to satisfy a pods' necessities.

[Donate](#)

the store and the salesperson shows you the following models:

MODEL 1	MODEL 2	MODEL 3
128GB	256GB	512GB
SATA	NVME	SATA

Now, you claim for a model that has at least 200GB of storage capacity and is an NVME drive.

The first one has less than 200GB and is SATA, so it doesn't match your claim. The third one has more than 200GB, but is not NVME. The second one however has more than 200GB and is also an NVME. So that's the one you get.

The SSD models that the salesperson showed you are equivalent to persistent volumes and your requirements are equivalent to persistent volume claims.

Create another new file called `database-persistent-volume-claim.yaml` inside the `k8s` directory and put the following content in that file:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: database-persistent-volume-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

[Donate](#)

purpose as any other configuration file.

- `spec.storageClass` in a claim configuration file indicates the type of storage this claim wants. That means any `PersistentVolume` that has `spec.storageClass` set to `manual` is suitable to be consumed by this claim. If you have multiple volumes with the `manual` class, the claim will get any one of them and if you have no volume with `manual` class – a volume will be provisioned dynamically.
- `spec.accessModes` again sets the access mode here. This indicates that this claim wants a storage that has an `accessMode` of `ReadWriteOnce`. Assume that you have two volumes with class set to `manual`. One of them has its `accessModes` set to `ReadWriteOnce` and the other one to `ReadWriteMany`. This claim will get the one with `ReadWriteOnce`.
- `resources.requests.storage` is the amount of storage this claim wants. `2Gi` doesn't mean the given volume must have exactly 2 gigabytes of storage capacity. It means that it must have at least 2 gigabytes. I hope you remember that you set the capacity of the persistent volume to be 5 gigabytes, which is more than 2 gigabytes.

To apply this file, execute the following command:

```
kubectl apply -f database-persistent-storage-claim.yaml  
# persistentvolumeclaim/database-persistent-volume-claim created
```

[Donate](#)

created:

```
kubectl get persistentvolumeclaim  
  
# NAME                      STATUS        VOLUME  
# database-persistent-volume-claim   Bound        database-persistent
```

Look at the `VOLUME` column. This claim is bound to the `database-persistent-volume` persistent volume that you created earlier. Also look at the `CAPACITY`. It's `5Gi`, because the claim requested a volume with at least 2 gigabytes of storage capacity.

Dynamic Provisioning of Persistent Volumes

In the previous sub-section, you've made a persistent volume and then created a claim. But, what if there isn't any persistent volume previously provisioned?

In such cases, a persistent volume compatible with the claim will be provisioned automatically.

To begin this demonstration, remove the previously created persistent volume and the persistent volume claim with the following commands:

```
kubectl delete persistentvolumeclaim --all  
  
# persistentvolumeclaim "database-persistent-volume-claim" deleted  
  
kubectl delete persistentvolumeclaim --all  
  
# persistentvolume "database-persistent-volume" deleted
```

[Donate](#)

update its content to be as follows:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: database-persistent-volume-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

I've removed the `spec.storageClass` field from the file. Now re-apply the `database-persistent-volume-claim.yaml` file without applying the `database-persistent-volume.yaml` file:

```
kubectl apply -f database-persistent-volume-claim.yaml
# persistentvolumeclaim/database-persistent-volume-claim created
```

Now use the `get` command to look at the claim information:

```
kubectl get persistentvolumeclaim
# NAME                      STATUS        VOLUME
# database-persistent-volume-claim   Bound       pvc-525ae8af-00d3-4
```

As you can see, a volume with `pvc-525ae8af-00d3-4cc7-ae47-866aa13dfffd5` name and storage capacity of `2Gi` has been provisioned and bound to the claim dynamically.

[Donate](#)

volume for the rest of this project. I'll be using a dynamically provisioned one.

Connecting Volumes with Pods

Now that you have created a persistent volume and a claim, it's time to let the database pod use this volume.

You do this by connecting the pod to the persistent volume claim you made in the previous sub-section. Open up the `postgres-deployment.yaml` file and update its content to be as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      component: postgres
  template:
    metadata:
      labels:
        component: postgres
    spec:
      # volume configuration for the pod
      volumes:
        - name: postgres-storage
          persistentVolumeClaim:
            claimName: database-persistent-volume-claim
      containers:
        - name: postgres
          image: fhsinchy/notes-postgres
          ports:
            - containerPort: 5432
      # volume mounting configuration for the container
      volumeMounts:
        - name: postgres-storage
          mountPath: /var/lib/postgresql/data
          subPath: postgres
```

[Donate](#)

```
  value: 63eaQB9wtLqmNBpg
- name: POSTGRES_DB
  value: notesdb
```

I've added two new fields in this configuration file.

- `spec.volumes` field contains the necessary information for the pod to find the persistent volume claim.
`spec.volumes.name` can be anything you want.
`spec.volumes.persistentVolumeClaim.claimName` has to match the `metadata.name` value from the `database-persistent-volume-claim.yaml` file.
- `containers.volumeMounts` contains information necessary for mounting the volume inside the container.
`containers.volumeMounts.name` has to match the value from `spec.volumes.name`. `containers.volumeMounts.mountPath` indicates the directory where this volume will be mounted.
`/var/lib/postgresql/data` is the default data directory for PostgreSQL. `containers.volumeMounts.subPath` indicates a directory that will be created inside the volume. Assume that you're using the same volume with other pods as well. In that case you can put pod-specific data inside another directory inside that volume. All data saved inside the `/var/lib/postgresql/data` directory will go inside a `postgres` directory within the volume.

Now re-apply the `postgres-deployment.yaml` file by executing the following command:

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

Now you have a proper database deployment with a much smaller risk of data loss.

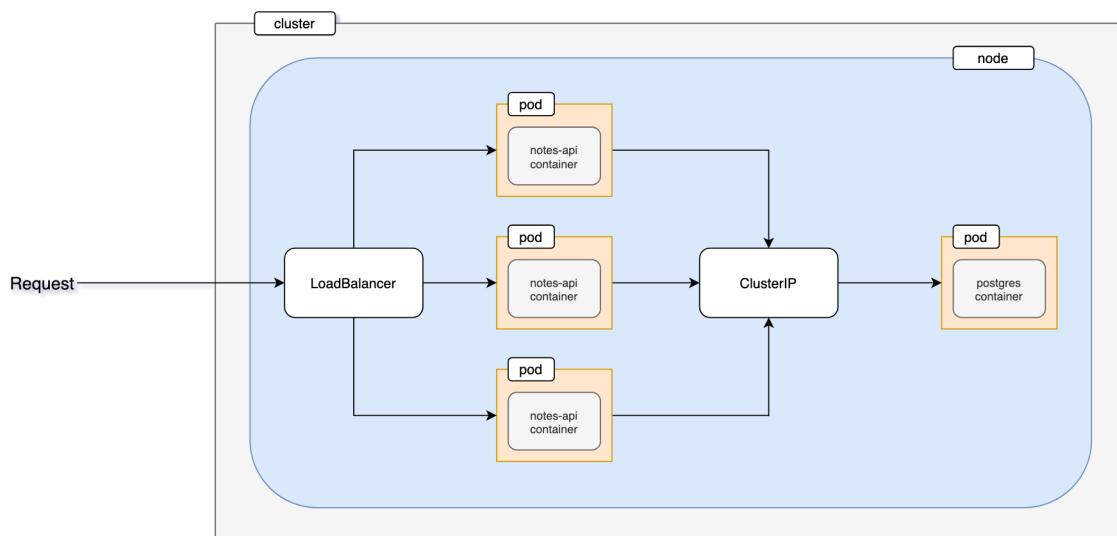
One thing that I would like to mention here is that the database deployment in this project has only one replica. If there were more than one replica, things would have been different.

Multiple pods accessing the same volume without them knowing about each others' existence can bring catastrophic results. In such cases creating sub directories for the pods inside that volume can be a good idea.

Wiring Everything Up

Now that you have both the API and database running, it's time to finish some unfinished business and set-up the networking.

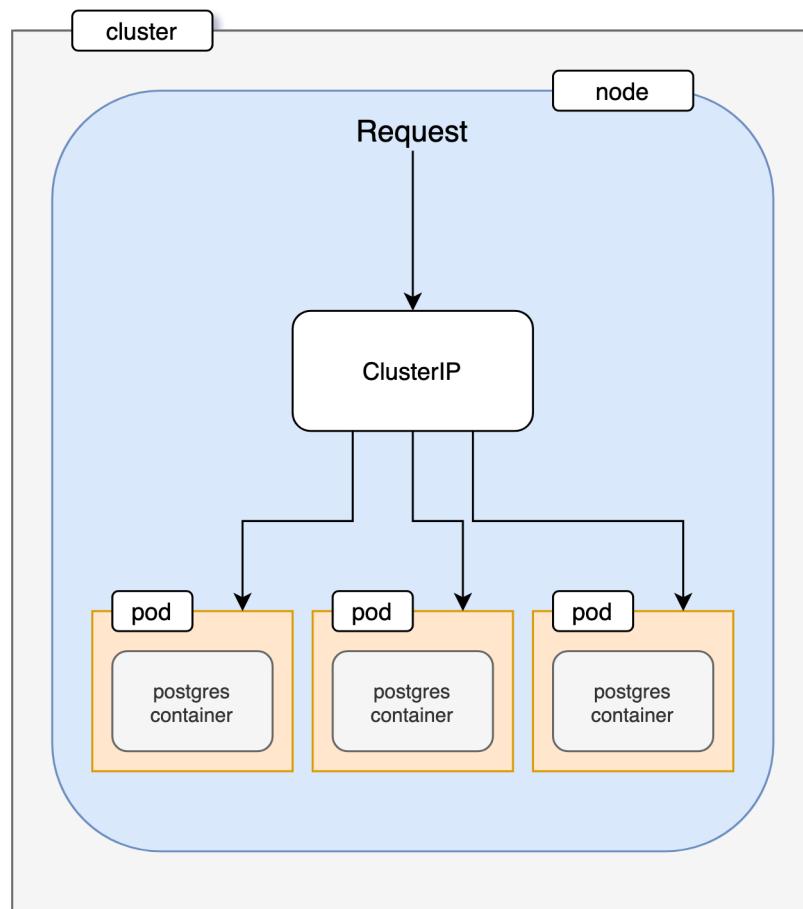
You've already learned in previous sections that to set up networking in Kubernetes, you use services. Before you start writing the services, have a look at the networking plan that I have for this project.



[Donate](#)

- The database will only be exposed within the cluster using a ClusterIP service. No external traffic will be allowed.
- The API deployment, however, will be exposed to the outside world. Users will communicate with the API and the API will communicate with the database.

You've previously worked with a `LoadBalancer` service that exposes an application to the outside world. The `ClusterIP` on the other hand exposes an application within the cluster and allows no outside traffic.



Given that the database service should be available only within the cluster, a `ClusterIP` service is the perfect fit for this scenario.

Create a new file called `postgres-cluster-ip-service.yaml` inside

[Donate](#)

```
apiVersion: v1
kind: Service
metadata:
  name: postgres-cluster-ip-service
spec:
  type: ClusterIP
  selector:
    component: postgres
  ports:
    - port: 5432
      targetPort: 5432
```

As you can see, the configuration file for a `ClusterIP` is identical to one for a `LoadBalancer`. The only thing that differs is the `spec.type` value.

You should be able to interpret this file without any trouble by now. 5432 is the default port that PostgreSQL runs on. That's why that port has to be exposed.

The next configuration file is for the `LoadBalancer` service, responsible for exposing the API to the outside world. Create another file called `api-load-balancer-service.yaml` and put the following content in it:

```
apiVersion: v1
kind: Service
metadata:
  name: api-load-balancer-service
spec:
  type: LoadBalancer
  ports:
    - port: 3000
      targetPort: 3000
  selector:
    component: api
```

[Donate](#)

This configuration is identical to the one you've written in a previous section. The API runs in port 3000 inside the container and that's why that port has to be exposed.

The last thing to do is to add the rest of the environment variables to the API deployment. So open up the `api-deployment.yaml` file and update its content like this:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      component: api
  template:
    metadata:
      labels:
        component: api
  spec:
    containers:
      - name: api
        image: fhsinchy/notes-api
        ports:
          - containerPort: 3000
    env:
      - name: DB_CONNECTION
        value: pg
      - name: DB_HOST
        value: postgres-cluster-ip-service
      - name: DB_PORT
        value: '5432'
      - name: DB_USER
        value: postgres
      - name: DB_DATABASE
        value: notesdb
      - name: DB_PASSWORD
        value: 63eaQB9wtLqmNBpg
```

[Donate](#)

Previously there was just the `DB_CONNECTION` variable under `spec.containers.env`. The new variables are as follows:

- `DB_HOST` indicates the host address for the database service.
In a non-containerized environment the value is usually `127.0.0.1`. But in a Kubernetes environment, you don't know the IP address of the database pod. Hence you just use the service name that exposes the database instead.
- `DB_PORT` is the port exposed from the database service, which is `5432`.
- `DB_USER` is the user for connecting to the database. `postgres` is the default username.
- `DB_DATABASE` is the database that the API will connect to. This has to match with the `spec.containers.env.DB_DATABASE` value from the `postgres-deployment.yaml` file.
- `DB_PASSWORD` is the password for connecting to the database. This has to match with the `spec.containers.env.DB_PASSWORD` value from the `postgres-deployment.yaml` file.

With that done, now you're ready to test out the API. Before you do that, I'll suggest applying all the configuration files once again by executing the following command:

```
kubectl apply -f k8s

# deployment.apps/api-deployment created
# service/api-load-balancer-service created
# persistentvolumeclaim/database-persistent-volume-claim created
```

[Donate](#)

If you face any errors, just delete all resources and re-apply the files. The services, the persistent volumes, and the persistent volume claims should be created instantly.

Use the `get` command to make sure the deployments are all up and running:

```
kubectl get deployment
```

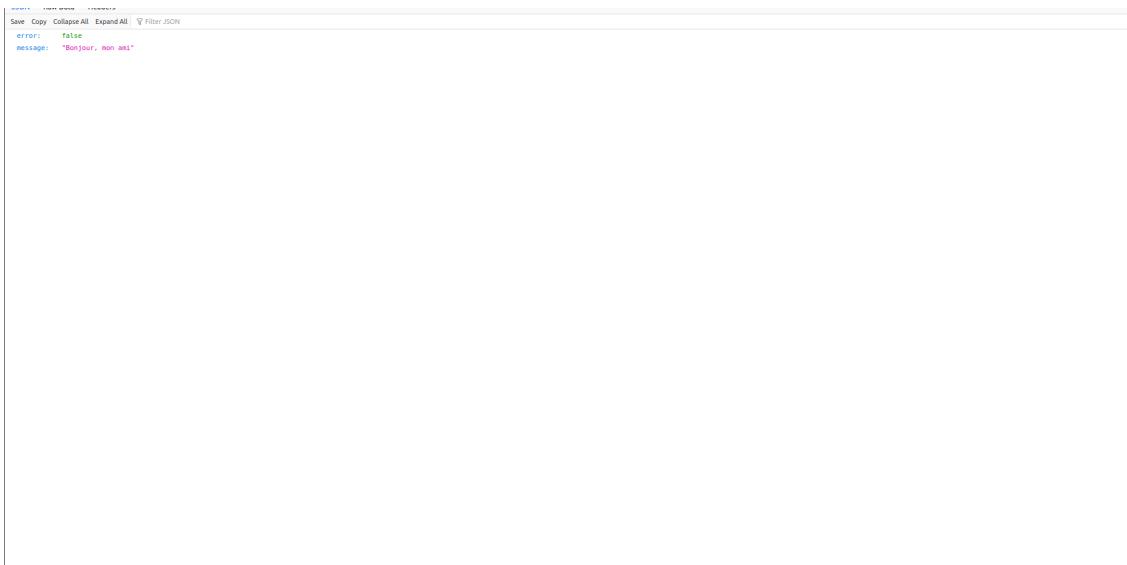
```
# NAME           READY   UP-TO-DATE   AVAILABLE   AGE
# api-deployment 3/3     3            3           106s
# postgres-deployment 1/1     1            1           106s
```

As you can see from the `READY` column, all the pods are up and running. To access the API, use the `service` command for `minikube`.

```
minikube service api-load-balancer-service
```

```
# |-----|-----|-----|-----|-----|
# | NAMESPACE |          NAME          | TARGET PORT |
# |-----|-----|-----|-----|-----|
# | default    | api-load-balancer-service |      3000 | http://
# |-----|-----|-----|-----|-----|
# * Opening service default/api-load-balancer-service in default
```

The API should open automatically in your default browser:

[Donate](#)

```
Save Copy Collapse All Expand All Filter JSON
{
  "error": false,
  "message": "Bonjour, mon ami"
}
```

This is the default response for the API. You can also use <http://12.19.186.112:31546/> with some API testing tool like [Insomnia](#) or [Postman](#) to test out the API. The API has full CRUD functionality.

You can see the tests that come with the API source code as documentation. Just open up the `api/tests/e2e/api/routes/notes.test.js` file. You should be able to understand the file without much hassle if you have experience with JavaScript and [express](#).

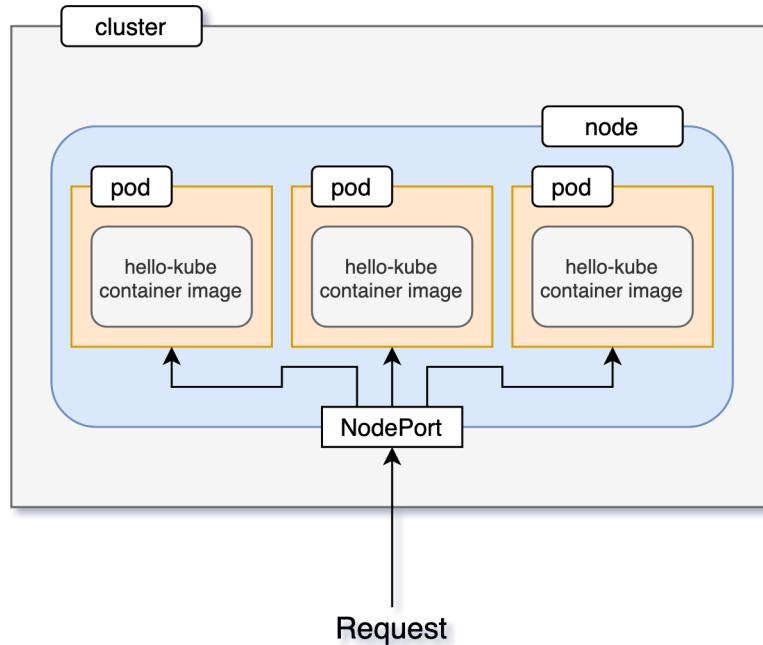
Working with Ingress Controllers

So far in this article, you've used `ClusterIP` to expose an application within the cluster and `LoadBalancer` to expose an application outside the cluster.

Although I've cited `LoadBalancer` as the standard service kind for exposing an application outside the cluster, it has some cons.

When using `LoadBalancer` services to expose applications in cloud environment, you'll have to pay for each exposed services individually which can be expensive in case of huge projects.

There is another kind of service called `NodePort` that can be used as an alternative to the `LoadBalancer` kind of services.

[Donate](#)

`NodePort` opens a specific port on all the nodes in your cluster, and handles any traffic that comes through that open port.

As you already know, services group together a number of pods, and control the way they can be accessed. So any request that reaches the service through the exposed port will end up in the correct pod.

An example configuration file for creating a `NodePort` can be as follows:

```

apiVersion: v1
kind: Service
metadata:
  name: hello-kube-node-port
spec:
  type: NodePort
  ports:
    - port: 8080
      targetPort: 8080
      nodePort: 31515
  selector:
    component: web

```

[Donate](#)

The `spec.ports.nodePort` field here must have a value between 30000 and 32767. This range is out of the well-known ports usually used by various services but is also unusual. I mean how many times do you see a port with so many digits?

You can try to replace the `LoadBalancer` services you created in the previous sections with a `NodePort` service. This shouldn't be tough and can be treated as a test for what you've learned so far.

To solve the issues I've mentioned the `Ingress` API was created. To be very clear, `Ingress` is actually not a type of service. Instead, it sits in front of multiple services and acts as a router of sorts.

An `IngressController` is required to work with `Ingress` resources in your cluster. A list of available ingress controllers can be found in the [Kubernetes documentation](#).

Setting up NGINX Ingress Controller

In this example, you'll extend the notes API by adding a front end to it. And instead of using a service like `LoadBalancer` or `NodePort`, you'll use `Ingress` to expose the application.

The controller you'll be using is the [NGINX Ingress Controller](#) because [NGINX](#) will be used for routing requests to different services here. The NGINX Ingress Controller makes it very easy to work with NGINX configurations in a Kubernetes cluster.

The code for the project lives inside the `fullstack-notes-application` directory.

.

[Donate](#)

```

└── docker-compose.yaml
└── k8s
    ├── api-deployment.yaml
    ├── database-persistent-volume-claim.yaml
    ├── postgres-cluster-ip-service.yaml
    └── postgres-deployment.yaml
└── nginx
└── postgres

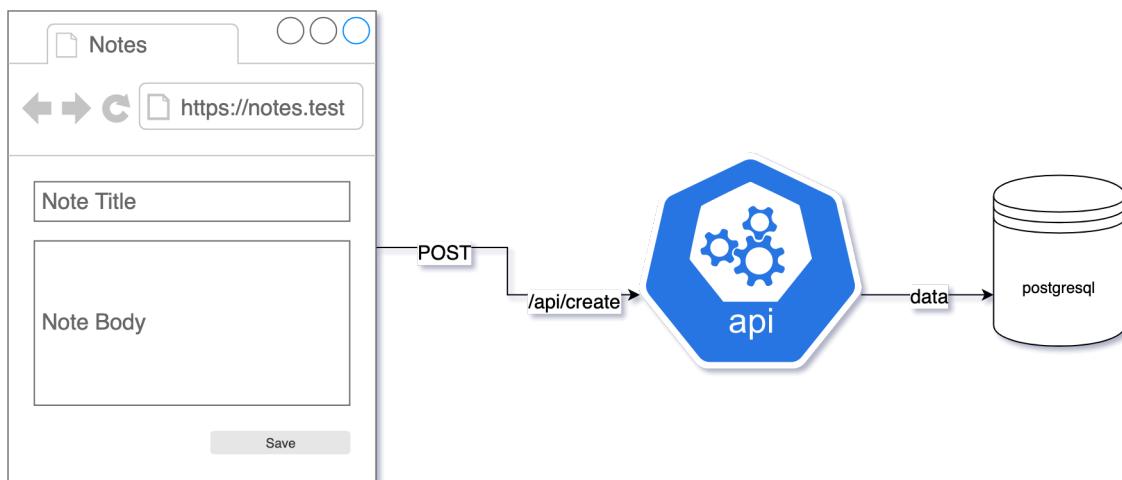
```

5 directories, 1 file

You'll see a `k8s` directory in there. It contains all the configuration files you wrote in the last sub-section, except the `api-load-balancer-service.yaml` file.

The reason for that is, in this project, the old `LoadBalancer` service will be replaced with an `Ingress`. Also, instead of exposing the API, you'll expose the front-end application to the world.

Before you start writing the new configuration files, have a look at how things are going to work behind the scenes.



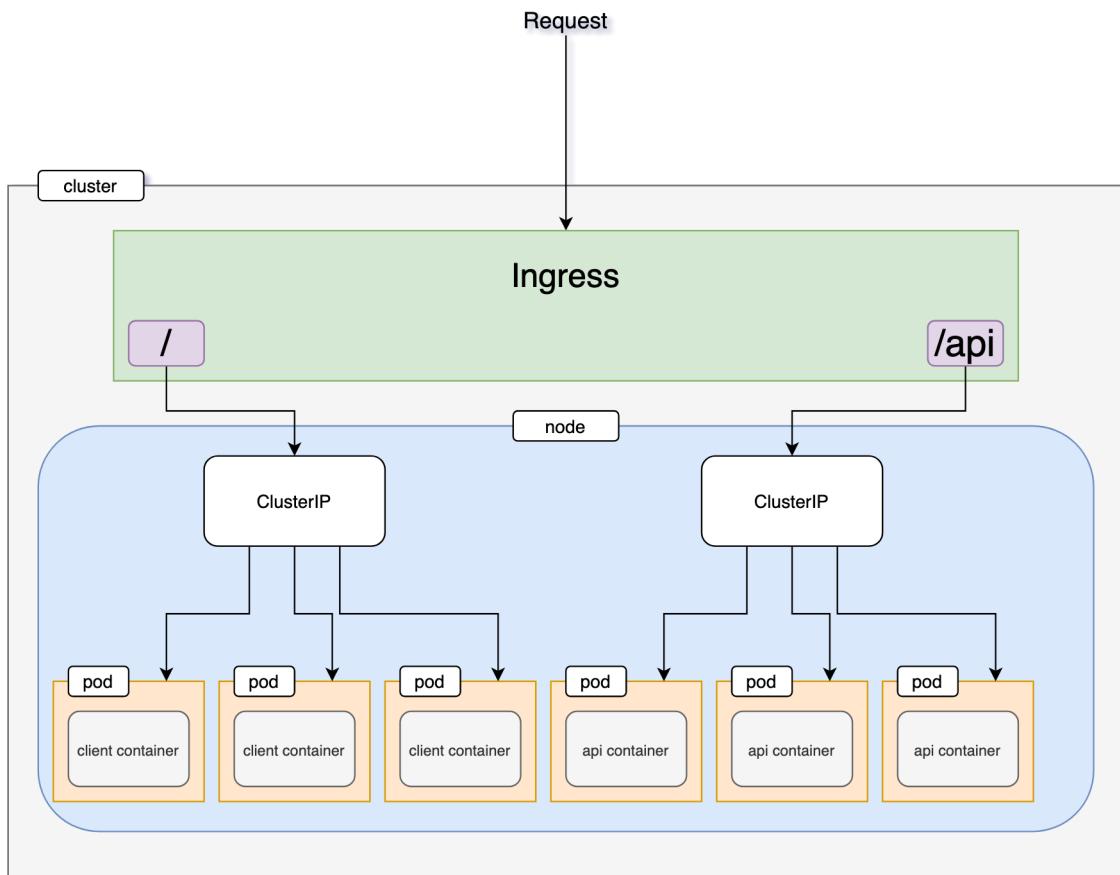
A user visits the front-end application and submits the necessary data. The front-end application then forwards the submitted data to

[Donate](#)

The API then persists the data in the database and also sends it back to the front-end application. Then routing of the requests is achieved using NGINX.

You can have a look at the `nginx/production.conf` file to understand how this routing has been set-up.

Now the necessary networking required to make this happen is as follows:



This diagram can be explained as follows:

- The Ingress will act as the entry-point and router for this application. This is an NGINX type Ingress so the port will be the default nginx port which is 80.

[Donate](#)

front-end application (the service on the left). So if the URL for this application is `https://kube-notes.test`, then any request coming to `https://kube-notes.test/foo` or `https://kube-notes.test/bar` will be handled by the front-end application.

- Every request that comes to `/api` will be routed towards the back-end API (the service on the right). So if the URL again is `https://kube-notes.test`, then any request coming to `https://kube-notes.test/api/foo` or `https://kube-notes.test/api/bar` will be handled by the back-end API.

It was totally possible to configure the `Ingress` service to work with sub-domains instead of paths like this, but I chose the path-based approach because that's how my application is designed.

In this sub-section, you'll have to write four new configuration files.

- `ClusterIP` configuration for the API deployment.
- `Deployment` configuration for the front-end application.
- `ClusterIP` configuration for the front-end application.
- `Ingress` configuration for the routing.

I'll go through the first three files very quickly without spending much time explaining them.

The first one is the `api-cluster-ip-service.yaml` configuration and the contents of the file are as follows:

```
apiVersion: v1
kind: Service
metadata:
  name: api-cluster-ip-service
spec:
  type: ClusterIP
```

[Donate](#)

```
ports:  
  - port: 3000  
    targetPort: 3000
```

Although in the previous sub-section you exposed the API directly to the outside world, in this one, you'll let the `Ingress` do the heavy lifting while exposing the API internally using a good old `ClusterIP` service.

The configuration itself should be pretty self-explanatory at this point, so I won't be spending any time explaining it.

Next, create a file named `client-deployment.yaml` responsible for running the front-end application. Contents of the file are as follows:

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: client-deployment  
spec:  
  replicas: 3  
  selector:  
    matchLabels:  
      component: client  
  template:  
    metadata:  
      labels:  
        component: client  
  spec:  
    containers:  
      - name: client  
        image: fhsinchy/notes-client  
        ports:  
          - containerPort: 8080  
        env:  
          - name: VUE_APP_API_URL  
            value: /api
```

It's almost identical to the the `api-deployment.yaml` file and I'm assuming that you'll be able to interpret this configuration file by yourself.

The `VUE_APP_API_URL` environment variable here indicates the path to which the API requests should be forwarded. These forwarded requests will be in turn handled by the `Ingress`.

To expose this client application internally another `ClusterIP` service is necessary. Create a new file called `client-cluster-ip-service.yaml` and put the following content in it:

```
apiVersion: v1
kind: Service
metadata:
  name: client-cluster-ip-service
spec:
  type: ClusterIP
  selector:
    component: client
  ports:
    - port: 8080
      targetPort: 8080
```

All this does is expose port 8080 within the cluster on which the front end application runs by default.

Now that the boring old configurations are done, the next configuration is the `ingress-service.yaml` file and the content of the file is as follows:

```
apiVersion: extensions/v1beta1
```

[Donate](#)

```

name: ingress-service
annotations:
  kubernetes.io/ingress.class: nginx
  nginx.ingress.kubernetes.io/rewrite-target: /$1
spec:
  rules:
    - http:
        paths:
          - path: /?(.*)
            backend:
              serviceName: client-cluster-ip-service
              servicePort: 8080
          - path: /api/?(.*)
            backend:
              serviceName: api-cluster-ip-service
              servicePort: 3000

```

This configuration file may look quite a bit unfamiliar to you but it's actually pretty straightforward.

- The `Ingress API` is still in beta phase thus the `apiVersion` is `extensions/v1beta1`. Although in beta, the API is very stable and usable in production environments.
- The `kind` and `metadata.name` fields serve the same purpose as any of the configurations you wrote earlier.
- `metadata.annotations` can contain information regarding the Ingress configuration. The `kubernetes.io/ingress.class: nginx` indicates that the Ingress object should be controlled by the `ingress-nginx controller`. `nginx.ingress.kubernetes.io/rewrite-target` indicates that you want to rewrite the URL target in places.
- `spec.rules.http.paths` contains configuration regarding the individual path routings you previously saw inside the `nginx/production.conf` file. The `paths.path` field indicates the path that should be routed. `backend.serviceName` is the service that the aforementioned path should be routed

[Donate](#)

that service.

- `/?(.*)` and `/api/?(.*)` are simple regex which means that `?(.*)` part will be routed towards the designated services.

The way you configure rewrites can change from time to time, so checking out the official [docs](#) would be good idea.

Before you apply the new configurations, you'll have to activate the `ingress` addon for `minikube` using the `addons` command. The generic syntax is as follows:

```
minikube addons <option> <addon name>
```

To activate the `ingress` addon, execute the following command:

```
minikube addons enable ingress

# 🔎 Verifying ingress addon...
# 🌟 The 'ingress' addon is enabled
```

You can use the `disable` option for the `addon` command to disable any addon. You can learn more about the `addon` command in the official [docs](#).

Once the addon has been activated, you may apply the configuration files. I would suggest deleting all resources (services, deployments, and persistent volume claims) before applying the new ones.

```
kubectl delete ingress --all
```

[Donate](#)

```
kubectl delete service --all

# service "api-cluster-ip-service" deleted
# service "client-cluster-ip-service" deleted
# service "kubernetes" deleted
# service "postgres-cluster-ip-service" deleted

kubectl delete deployment --all

# deployment.apps "api-deployment" deleted
# deployment.apps "client-deployment" deleted
# deployment.apps "postgres-deployment" deleted

kubectl delete persistentvolumeclaim --all

# persistentvolumeclaim "database-persistent-volume-claim" deleted

kubectl apply -f k8s

# service/api-cluster-ip-service created
# deployment.apps/api-deployment created
# service/client-cluster-ip-service created
# deployment.apps/client-deployment created
# persistentvolumeclaim/database-persistent-volume-claim created
# ingress.extensions/ingress-service created
# service/postgres-cluster-ip-service created
# deployment.apps/postgres-deployment created
```

Wait until all the resources have been created. You can utilize the `get` command to ensure that. Once all of them are running, you can access the application at the IP address of the `minikube` cluster. To get the IP, you can execute the following command:

```
minikube ip
```

```
# 172.17.0.2
```

You can also get this IP address by running inspecting the Ingress :

[Donate](#)

```
kubectl get ingress
```

#	NAME	CLASS	HOSTS	ADDRESS	PORTS	AGE
#	ingress-service	<none>	*	172.17.0.2	80	2m33s

As you can see, the IP and port is visible under the ADDRESS and PORTS columns. By accessing 127.17.0.2:80 , you should land directly on the notes application.

You can perform simple CRUD operations in this application. Port 80 is the default port for NGINX, so you don't need to write the port number in the URL.

You can do a lot with this ingress controller if you know how to configure NGINX. After all, that's what this controller is used for – storing NGINX configurations on a Kubernetes ConfigMap , which you'll be learning about in the next sub-section.

Secrets and Config Maps in Kubernetes

So far in your deployments, you've stored sensitive information such as POSTGRES_PASSWORD in plain text, which is not a very good idea.

To store such values in your cluster you can use a Secret which is a much more secure way of storing passwords, tokens, and so on.

The next step may not work the same in the Windows command line. You can use git bash or cmder for the task.

To store information in a Secret you have to first pass your data through base64. If the plain text password is 63eaQB9wtLqmNBpg

```
echo -n "63eaQB9wtLqmNBpg" | base64  
# NjNlYVFC0Xd0THFtTkJwZw==
```

This step is not optional, you have to run the plain text string through base64. Now create a file named `postgres-secret.yaml` inside the `k8s` directory and put following content in there:

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: postgres-secret  
data:  
  password: NjNlYVFC0Xd0THFtTkJwZw==
```

The `apiVersion`, `kind`, and `metadata` fields are pretty self-explanatory. The `data` field holds the actual secret.

As you can see, I've created a key-value pair where the key is `password` and the value is `NjNlYVFC0Xd0THFtTkJwZw==`. You'll be using the `metadata.name` value to identify this `Secret` in other configuration files and the key to access the password value.

Now to use this secret inside your database configuration, update the `postgres-deployment.yaml` file as follows:

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: postgres-deployment  
spec:
```

[Donate](#)

```

matchLabels:
  component: postgres
template:
  metadata:
    labels:
      component: postgres
spec:
  volumes:
    - name: postgres-storage
      persistentVolumeClaim:
        claimName: database-persistent-volume-claim
  containers:
    - name: postgres
      image: fhsinchy/notes-postgres
      ports:
        - containerPort: 5432
      volumeMounts:
        - name: postgres-storage
          mountPath: /var/lib/postgresql/data
          subPath: postgres
      env:
        # not putting the password directly anymore
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef:
              name: postgres-secret
              key: password
        - name: POSTGRES_DB
          value: notesdb

```

As you can see, the entire file is the same except the `spec.template.spec.containers.env` field.

The `name` environment variable used to store the password value was in plain text before. But now there is a new `valueFrom.secretKeyRef` field.

The `name` field here refers to the name of the `Secret` you created moments ago, and the `key` value refers to the key from the key-

[Donate](#)

Apart from the database configuration, Kubernetes have to update the `api-deployment.yaml` file as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      component: api
  template:
    metadata:
      labels:
        component: api
  spec:
    containers:
      - name: api
        image: fhsinchy/notes-api
        ports:
          - containerPort: 3000
    env:
      - name: DB_CONNECTION
        value: pg
      - name: DB_HOST
        value: postgres-cluster-ip-service
      - name: DB_PORT
        value: '5432'
      - name: DB_USER
        value: postgres
      - name: DB_DATABASE
        value: notesdb
        # not putting the password directly anymore
      - name: DB_PASSWORD
        valueFrom:
          secretKeyRef:
            name: postgres-secret
            key: password
```

Now apply all these new configurations by executing the following

[Donate](#)

```
kubectl apply -f k8s

# service/api-cluster-ip-service created
# deployment.apps/api-deployment created
# service/client-cluster-ip-service created
# deployment.apps/client-deployment created
# persistentvolumeclaim/database-persistent-volume-claim created
# secret/postgres-secret created
# ingress.extensions/ingress-service created
# service/postgres-cluster-ip-service created
# deployment.apps/postgres-deployment created
```

Depending on the state of your cluster, you may see a different set of output.

In case you're having any issue, delete all Kubernetes resources and create them again by applying the configs.

Use the `get` command to inspect and make sure all the pods are up and running.

Now to test out the new configuration, access the notes application using the `minikube` IP and try creating new notes. To get the IP, you can execute the following command:

```
minikube ip
```

```
# 172.17.0.2
```

By accessing `127.17.0.2:80`, you should land directly on the notes application.

[Donate](#)

The screenshot shows a web-based note-taking application. At the top, there are navigation links: 'Notes', 'HOME', and 'CREATE'. Below the header, a note is displayed with the title 'The standard Lorem Ipsum passage, used since the 1500s'. The note's content is a placeholder text from the 'Lorem Ipsum' generator, discussing the standard text used in printing and typesetting since the 1500s. A red 'Delete' button is located at the bottom right of the note's content area.

There is another way to create secrets without any configuration file. To create the same `Secret` using `kubectl`, execute the following command:

```
kubectl create secret generic postgres-secret --from-literal=pass  
# secret/postgres-secret created
```

This is a more convenient approach as you can skip the whole base64 encoding step. The secret in this case will be encoded automatically.

A `ConfigMap` is similar to a `Secret` but is meant to be used with non sensitive information.

To put all the other environment variables in the API deployment inside a `ConfigMap`, create a new file called `api-config-map.yaml` inside the `k8s` directory and put following content in it:

```
apiVersion: v1
```

[Donate](#)

```

name: api-config-map
data:
  DB_CONNECTION: pg
  DB_HOST: postgres-cluster-ip-service
  DB_PORT: '5432'
  DB_USER: postgres
  DB_DATABASE: notesdb

```

`apiVersion`, `kind` and `metadata` are again self-explanatory. The `data` field can hold the environment variables as key-value pairs.

Unlike the `Secret`, the keys here have to match the exact key required by the API. Thus, I have sort of copied the variables from a `api-deployment.yaml` file and pasted them here with a slight modification in the syntax.

To make use of this secret in the API deployment, open up the `api-deployment.yaml` file and update its content as follows:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      component: api
  template:
    metadata:
      labels:
        component: api
  spec:
    containers:
      - name: api
        image: fhsinchy/notes-api
        ports:
          - containerPort: 3000
# not putting environment variables directly

```

[Donate](#)

```
    name: api-config-map
  env:
    - name: DB_PASSWORD
      valueFrom:
        secretKeyRef:
          name: postgres-secret
          key: password
```

The entire file is almost unchanged except the `spec.template.spec.containers.env` field.

I have moved the environment variables to the `ConfigMap`. `spec.template.spec.containers.envFrom` is used to get data from a `ConfigMap`. `configMapRef.name` here indicates the `ConfigMap` from where the environment variables will be pulled.

Now apply all these new configurations by executing the following command:

```
kubectl apply -f k8s

# service/api-cluster-ip-service created
# configmap/api-config-map created
# deployment.apps/api-deployment created
# service/client-cluster-ip-service created
# deployment.apps/client-deployment created
# persistentvolumeclaim/database-persistent-volume-claim created
# ingress.extensions/ingress-service configured
# service/postgres-cluster-ip-service created
# deployment.apps/postgres-deployment created
# secret/postgres-secret created
```

Depending on the state of your cluster, you may see a different set of output.

[Donate](#)

- resources and create them again by applying the configs.

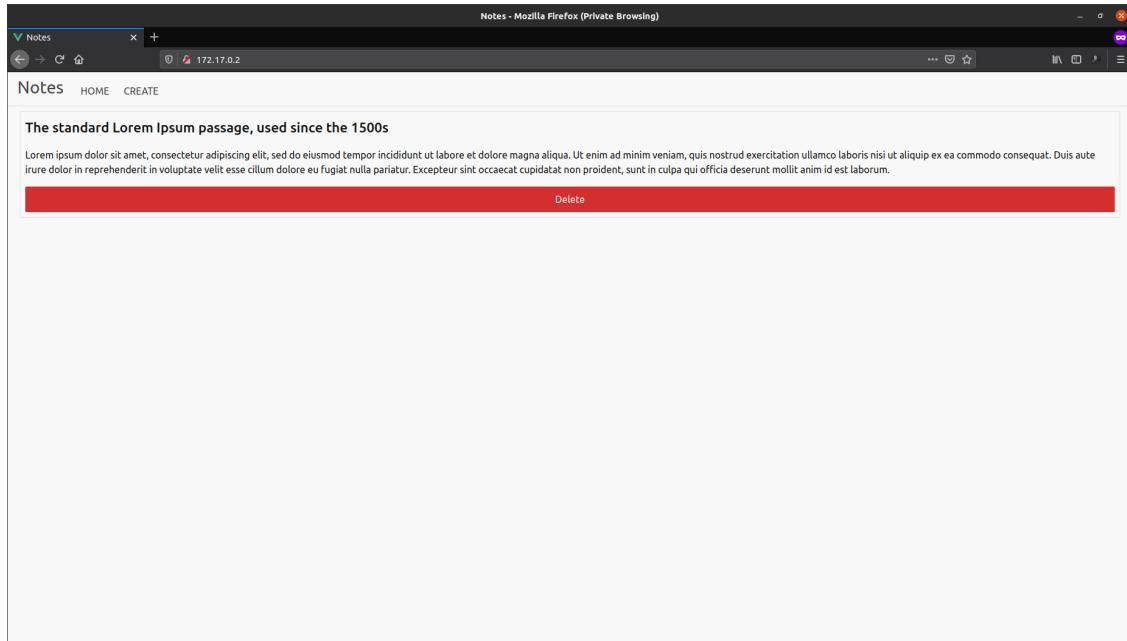
Upon making sure that the pods are up and running using the `get` command, access the notes application using the `minikube` IP and try creating new notes.

To get the IP, you can execute the following command:

```
minikube ip
```

```
# 172.17.0.2
```

By accessing `127.17.0.2:80`, you should land directly on the notes application.



Secret and ConfigMap have a few more tricks up their sleeves that I'm not going to get into right now. But if you're curious, you may check out the official [docs](#).

[Donate](#)

Now that you've successfully deployed an application consisting of multiple containers on Kubernetes, it's time to learn about performing updates.

As magical as Kubernetes may seem to you, updating a container to a newer image version is a bit of a pain. There are multiple approaches that people often take to update a container, but I am not going to go through all of them.

Instead, I'll jump right into the approach that I mostly take in updating my containers. If you open up the `client-deployment.yaml` file and look into the `spec.template.spec.containers` field, you'll find something that looks like this:

```
containers:
  - name: client
    image: fhsinchy/notes-client
```

As you can see, in the `image` field I haven't used any image tag. Now if you think that adding `:latest` at the end of the image name will ensure that the deployment always pulls the latest available image, you'd be dead wrong.

The approach that I usually take is an imperative one. I've already mentioned in a previous section that, in a few cases, using an imperative approach instead of a declarative one is a good idea. Creating a `Secret` or updating a container is such a case.

The command you can use to perform the update is the `set` command, and the generic syntax is as follows:

[Donate](#)

The resource type is `deployment` and resource name is `client-deployment`. The container name can be found under the `containers` field inside the `client-deployment.yaml` file, which is `client` in this case.

I have already build a version of the `fhsinchy/notes-client` image with a tag of `edge` that I'll be using to update this deployment.

So the final command should be as follows:

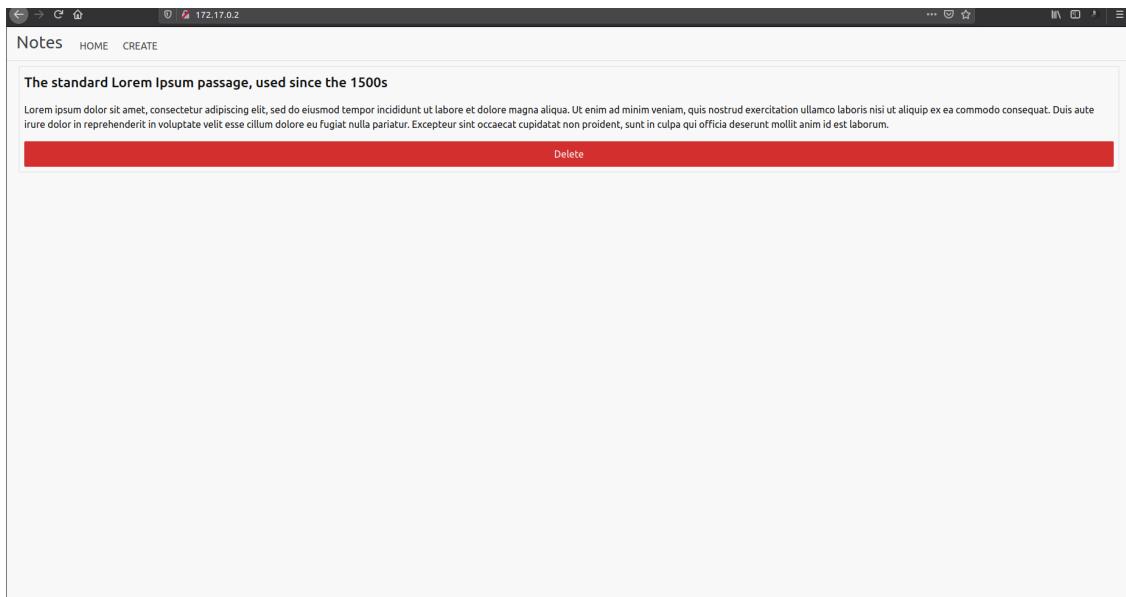
```
kubectl set image deployment/client-deployment client=fhsinchy/nc  
# deployment.apps/client-deployment image updated
```

The update process may take a while, as Kubernetes will recreate all the pods. You can run the `get` command to know if all the pods are up and running again.

Once they've all been recreated, access the notes application using the `minikube` IP and try creating new notes. To get the IP, you can execute the following command:

```
minikube ip  
# 172.17.0.2
```

By accessing `127.17.0.2:80`, you should land directly on the notes application.

[Donate](#)

Given that I haven't made any actual changes to the application code, everything will remain the same. You can ensure that the pods are using the new image using the `describe` command.

```
kubectl describe pod client-deployment-849bc58bcc-gz26b | grep ']'  
  
# Image:          fhsinchy/notes-client:edge  
# Image ID:       docker-pullable://fhsinchy/notes-client@sha256:
```

The `grep` command is available on Mac and Linux. If you're on Windows, use git bash instead of the windows command line.

Although the imperative update process is a bit tedious, but it can be made much easier by using a good CI/CD workflow.

Combining Configurations

As you've already seen, the number of configuration files in this project is pretty huge despite only having three containers in it.

You can actually combine configuration files as follows:

[Donate](#)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: client-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      component: client
  template:
    metadata:
      labels:
        component: client
  spec:
    containers:
      - name: client
        image: fhsinchy/notes-client
        ports:
          - containerPort: 8080
        env:
          - name: VUE_APP_API_URL
            value: /api
```

```
apiVersion: v1
kind: Service
metadata:
  name: client-cluster-ip-service
spec:
  type: ClusterIP
  selector:
    component: client
  ports:
    - port: 8080
      targetPort: 8080
```

As you can see, I've combined the contents of the `client-deployment.yaml` and `client-cluster-ip-service.yaml` file using a delimiter (`---`). Although it's possible and can help in projects where the

[Donate](#)

separate, clean, and concise.

Troubleshooting

In this section, I'll be listing some common issues that you may face during your time with Kubernetes.

- If you're on Windows or Mac and using the Docker driver for `minikube`, the `Ingress` plugin will not work.
- If you have [Laravel Valet](#) running on Mac and are using the HyperKit driver for `minikube`, it'll fail to connect to the internet. Turning off the `dnsmasq` service will resolve the issue.
- If you have a Ryzen (mine is R5 1600) PC and are running Windows 10, the VirtualBox driver may fail to start due to the lack of support for nested virtualization. You'll have to use the Hyper-V driver on Windows 10 (Pro, Enterprise, and Education). For the Home edition users, sadly there is no safe option on that hardware.
- If you're running Windows 10 (Pro, Enterprise, and Education) with the Hyper-V driver for `minikube`, the VM may fail to start with a message regarding insufficient memory. Don't panic, and execute the `minikube start` command once again to start the VM properly.
- If you see some of the commands executed in this article missing or misbehaving in the Windows command line, use [git bash](#) or [cmder](#) instead.

I would suggest installing a good Linux distribution on your system and using the Docker driver for `minikube`. This is by far the fastest and most reliable set-up.

Conclusion

[Donate](#)

you've spent reading this article. I hope you've enjoyed your time and have learned all the essentials of Kubernetes.

You can follow me on Twitter [@frhnhsin](#) or connect with me on LinkedIn at [/in/farhanhasin](#) if you prefer that.



Farhan Hasin Chowdhury

Programmer, Voracious Reader and Video Game Enthusiast

If you read this far, tweet to the author to show them you care.

[Tweet a thanks](#)

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

[Get started](#)

freeCodeCamp is a donor-supported tax-exempt 501(c)(3) nonprofit organization (United States Federal Tax Identification Number: 82-0779546)

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public. We also have thousands of freeCodeCamp study groups around the world.

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers, services, and staff.

You can [make a tax-deductible donation here](#).

[Donate](#)[JavaScript Closure](#)[JavaScript Promise](#)[CSS Box Shadow](#)[What is GitHub?](#)[Python List Append](#)[Python Sort List](#)[JavaScript Array Sort](#)[Comments in JSON](#)[Symlink in Linux](#)[What is Kanban?](#)[Linux Grep Command](#)[Python Write to File](#)[What is DNS?](#)[CSS Media Queries](#)[Primary Key SQL](#)[HTML Entities](#)[SQL Update Statement](#)[Excel VBA](#)[Screenshot on PC](#)[LOOKUP in Excel](#)[What is a Proxy Server?](#)[Arrow Function JavaScript](#)[Cat Command in Linux](#)[Remove Duplicates in Excel](#)[CSS Background Image](#)[dllhost.exe COM Surrogate](#)[HTML Background Color](#)[Boolean Algebra Truth Table](#)[CSS Comment Example](#)[Video Chat for Android](#)

Our Nonprofit

[About](#) [Alumni Network](#) [Open Source](#) [Shop](#) [Support](#) [Sponsors](#) [Academic Honesty](#)
[Code of Conduct](#) [Privacy Policy](#) [Terms of Service](#) [Copyright Policy](#)