From zero to Kubernetes

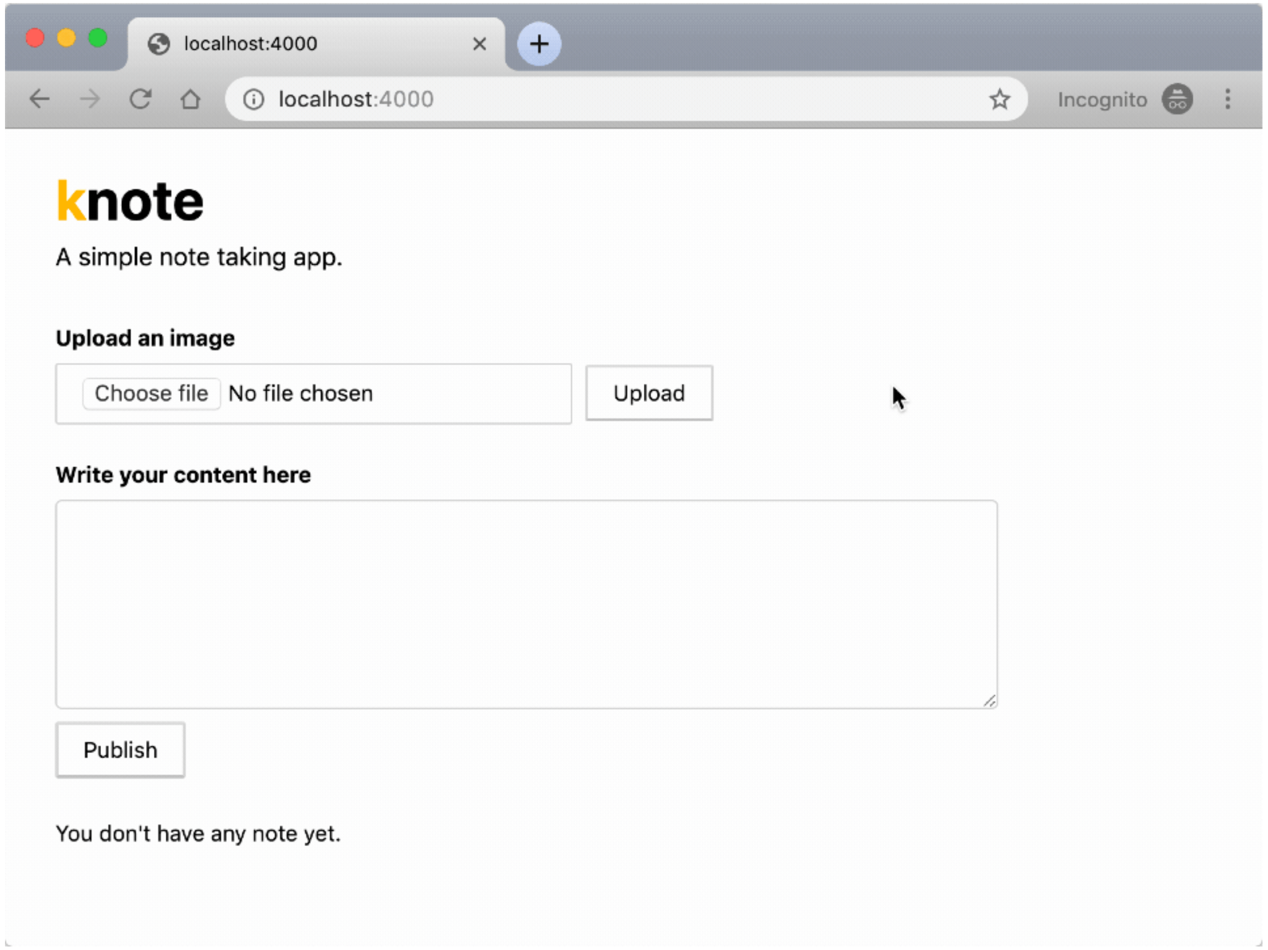
with Node.js

Table of contents

Chapter 1

Writing a note-taking app

In this section, you will write an application from scratch. Later in this course, you will package this app as a Docker image and deploy it to Kubernetes. The application is a note-taking app — similar to Evernote or Google Keep. It allows to format notes with Markdown and include pictures. Here is how it looks like:



Adding images and notes in Knote

Now that you know how the app works, it's your turn to code it.

1. Bootstrapping the app

You will build the app with JavaScript and Node.js (only server-side JavaScript, no client-side code).

> If at any time you're stuck, you can find the final code of the app in this repository.

Let's get started. Express and Pug are two popular choices when it comes to web servers and templating engines in Node.js The basic template for a Node.js project with Express and Pug looks like this: Now, create an `index.js` file with the following content:

index.js

```
const path = require('path')
const express = require('express')
const app = express()
const port = process.env.PORT || 3000
async function start() {
  app.set('view engine', 'pug')
  app.set('views', path.join(__dirname, 'views'))
  app.use(express.static(path.join(__dirname, 'public')))
  app.listen(port, () => {
    console.log(`App listening on http://localhost:${port}`)
  })
}
start()
```

You also need a route and view to render the main page.

index.js

```
async function start() {
  // ...
  app.get('/', (req, res) => res.render('index'))
}
```

> You can find the pug template in this repository.

The code is not much more than some standard boilerplate code for an Express app. It doesn't yet do anything useful. *But you will change this now by connecting it to a database.*

2. Connecting a database

The database will store the notes. *What database should you use? MySQL? Redis? Oracle?* MongoDB is well-suited for your note-taking application because it's easy to set up and doesn't introduce the overhead of a relational database. So, go on and install the MongoDB package:

bash

```bash
$ npm install mongodb
```

Then add the following lines to the beginning of your `index.js` file:

index.js

```javascript
const MongoClient = require('mongodb').MongoClient
const mongoURL = process.env.MONGO_URL || 'mongodb://localhost:27017/dev'
```

Now your code needs to connect to the MongoDB server. **You have to consider something important here.** *When the app starts, it shouldn't crash because the database isn't ready too.* Instead, the app should keep retrying to connect to the database until it succeeds. Kubernetes expects that application components can be started in any order. *If you make this code change, you can deploy your apps to Kubernetes in any order.* Add the following function to your `index.js` file:

index.js

```
async function initMongo() {
  console.log('Initialising MongoDB...')
  let success = false
  while (!success) {
    try {
      client = await MongoClient.connect(mongoURL, {
        useNewUrlParser: true,
        useUnifiedTopology: true,
      })
      success = true
    } catch {
      console.log('Error connecting to MongoDB, retrying in 1 second')
      await new Promise(resolve => setTimeout(resolve, 1000))
    }
  }
  console.log('MongoDB initialised')
  return client.db(client.s.options.dbName).collection('notes')
}
```

This function keeps trying to connect to a MongoDB database at the given URL until the connection succeeds. It then connects to a database and creates a collection called `notes`.

> MongoDB collections are like tables in relational databases — lists of items.

Now you have to call the above function inside the `start` function in your `index.js` file:

index.js

```
async function start() {
  const db = await initMongo()
  // ...
}
```

Note how the `await` keyword blocks the execution of the app until the database is ready. *The next step is to use the database.*

3. Saving and retrieving notes

When the main page of your app loads, two things happen:

- All the existing notes are displayed
- Users can create new notes through an HTML form

Let's address the displaying of existing notes first. You need a function to retrieve all notes from the database:

index.js

```
async function retrieveNotes(db) {
  const notes = (await db.find().toArray()).reverse()
  return notes
}
```

The route of the main page is `/`, so you have to add a corresponding route to your Express app that retrieves all the notes from the database and displays them: Change the route inside the `start` function in your `index.js` file to:

index.js

```
async function start() {
  // ...
  app.get('/', async (req, res) => {
    res.render('index', { notes: await retrieveNotes(db) })
  })
  // ...
}
```

The above handler calls the `retrieveNotes` function to get all the notes from the database and then passes them into the `index.pug` template. *Next, let's address the creation of new notes.* First of all, you need a function that saves a single note in the database:

index.js

```
async function saveNote(db, note) {
  await db.insertOne(note)
}
```

The form for creating notes is defined in the `index.pug` template. Note that it handles both the creation of notes and uploading of pictures. You should use Multer, a middleware for multi-part form data, to handle the uploaded data. Go on and install Multer:

bash
```bash
$ npm install multer
```

And include it in your code:

index.js
```javascript
const multer = require('multer')
```

The form submits to the `/note` route, so you need to add this route to your app:

index.js

```
async function start() {
  // ...
  app.post(
    '/note',
    multer().none(),
    async (req, res) => {
      if (!req.body.upload && req.body.description) {
        await saveNote(db, { description: req.body.description })
        res.redirect('/')
      }
    }
  )
  // ...
}
```

The above handler calls the `saveNote` function with the content of the text box, which causes the note to be saved in the database. It then redirects to the main page, so that the newly created note appears immediately on the screen. **Your**

app is functional now (although not yet complete)! You can already run your app at this stage. But to do so, you need to run MongoDB as well. You can install MongoDB following the instructions in the official MongoDB documentation. Once MongoDB is installed, start a MongoDB server with:

```bash
$ mongod
```

Now run your app with:

```bash
$ node index.js
```

The app should connect to MongoDB and then listen for requests. You can access your app on http://localhost:3000. You should see the main page of the app. Try to create a note — you should see it being displayed on the main page. *Your app seems to works.* **But it's not yet complete.** The following requirements are missing:

- Markdown text is not formatted but just displayed verbatim
- Uploading pictures does not yet work

Let's fix those next.

4. Rendering Markdown to HTML

The Markdown notes should be rendered to HTML so that you can read them properly formatted. Marked is an excellent engine for rendering Markdown to HTML. As always, install the package first:

bash

```bash
$ npm install marked
```

And import it in your `index.js` file:

index.js

```javascript
const marked = require('marked')
```

Then, change the `retrieveNotes` function as follows (changed lines are highlighted):

index.js

```
async function retrieveNotes(db) {
  const notes = (await db.find().toArray()).reverse()
  return notes.map(it => {
    return { ...it, description: marked(it.description) }
  })
}
```

The new code converts all the notes to HTML before returning them. Restart the app and access it on http://localhost:3000. **All your notes should now be nicely formatted.** *Let's tackle uploading files.*

5. Uploading pictures

When a user uploads a picture, the file should be saved on disk, and a link should be inserted in the text box. This is similar to how adding pictures on StackOverflow works.

> Note that for this to work, the picture upload handler must have access to the text box — this is the reason that picture uploading and note creation are combined in the same form.

For now, the pictures will be stored on the local file system. You will use Multer to handle the uploaded pictures. Change the handler for the `/note` route as follows (changed lines are highlighted):

index.js

```
async function start() {
  // ...
  app.post(
    '/note',
    multer({ dest: path.join(__dirname, 'public/uploads/') }).single('image'),
    async (req, res) => {
      if (!req.body.upload && req.body.description) {
        await saveNote(db, { description: req.body.description })
        res.redirect('/')
      } else if (req.body.upload && req.file) {
        const link = `/uploads/${encodeURIComponent(req.file.filename)}`
        res.render('index', {
          content: `${req.body.description} ![](${link})`,
          notes: await retrieveNotes(db),
        })
      }
    }
  )
  // ...
}
```

The new code saves uploaded pictures in the `public/uploads` folder in your app directory and inserts a link to the file into the text box. Restart your app again and access it on http://localhost:3000. Try to upload a picture — you should see a link inserted in the text box. And when you publish the note, the picture should be displayed in the rendered note. **Your app is feature complete now.**

> Note that you can find the complete code for the app in in this repository.

You just created a simple note-taking app from scratch. In the next section, you will learn how to package and run it as a Docker container.

6. TL;DR: recap in 5 simple steps

1. You cloned the following repo and change opened the `01` folder.
2. You installed the dependencies with `npm install`.
3. You started the app with `node index.js`, but it fails because it can't connect to a database.
4. You opened a new terminal and in the same folder launched mongodb with `npm run mongo` (make sure there's a `/data/db` folder on your computer). The application connected to it.
5. You visited the application on http://localhost:3000 and submitted notes with images.

Chapter 2

Containerisation with Docker

After creating your app, you can think about how to deploy it. You could deploy our app to a Platform as a Service (PaaS) like Heroku and forget about the underlying infrastructure and dependencies. Or you could do it the hard way and provision your own VPS, install nvm, create the appropriate users, configure Node.js as well as PM2 to restart the app when it crashes and Nginx to handle TLS and path-based routing. However, in recent times, there is a trend to package applications as Linux containers and deploy them to specialised container platforms. *So what are containers?*

1. Linux containers

Linux containers are often compared to shipping containers. Shipping containers have a standard format, and they allow to isolate goods in one container from goods in another. Goods that belong together are packed in the same container, goods that have nothing to do with each other are packed in separate containers. *Linux containers are similar.* **However, the "goods" in a Linux container are processes and their dependencies.**

> Typically, a container contains a single process and its dependencies.

A container contains everything that is needed to run a process. That means that you can run a process without having to install any dependency on your machine because the container has all you need. Furthermore, the process in a container is isolated from everything else around it. When you run a container, the process isn't aware of the host machine, and it believes that it's the only process running on your computer. Of course, that isn't true. Bundling dependency and isolating processes might remind you of virtual machines. **However, containers are different from virtual machines.** The process in a container still executes on the kernel of the host machine. With virtual machines, you run an entire guest operating system on top of your host operating system, and the processes that you want to execute on top of that. Containers are much more lightweight than virtual machines. **How do containers work?** The magic of containers comes from two features in the Linux kernel:

- Control groups (cgroups)
- Namespaces

These are low-level Linux primitives. Control groups limit the resources a process can use, such as memory and CPU. *You might want to limit your container to only use up to 512 MB of memory and 10% of CPU time.* Namespaces limit what a process can see. *You might want to hide the file system of your host machine and instead provide an alternative file system to the container.* You can imagine that those two features are convenient for isolating process. However, they are very low-level and hard to work with. Developers created more and more abstractions to get around the sharp edges of cgroups and namespaces. These abstractions resulted in container systems. One of the first container systems was LXC. But the container breakthrough came with Docker which was released in 2013.

> Docker isn't the only Linux container technology. There are other popular projects such as rkt and containerd.

In this section, you will package your application as a Docker container. *Let's get started!*

2. Containerising the app

First of all, you have to install the Docker Community Edition (CE). You can follow the instructions in the official Docker documentation.

> If you're on Windows, you can follow our handy guide on how to install Docker on Windows.

You can verify that Docker is installed correctly with the following command:

bash

```
$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

$ _
```

You're now ready to build Docker containers. Docker containers are built from Dockerfiles. A Dockerfile is like a recipe — it defines what goes in a container. A Dockerfile consists of a sequence of commands. You can find the full list of commands in the Dockerfile reference. Here is a Dockerfile that packages your app into a container image:

Dockerfile

```
FROM node:12.0-slim
COPY . .
RUN npm install
CMD [ "node", "index.js" ]
```

Go on and save this as `Dockerfile` in the root directory of your app. The above Dockerfile includes the following commands:

- `FROM` defines the base layer for the container, in this case, a version of Ubuntu with Node.js installed
- `COPY` copies the files of your app into the container
- `RUN` executes `npm install` inside the container

- `CMD` defines the command that should be executed when the container starts

You can now build a container image from your app with the following command:

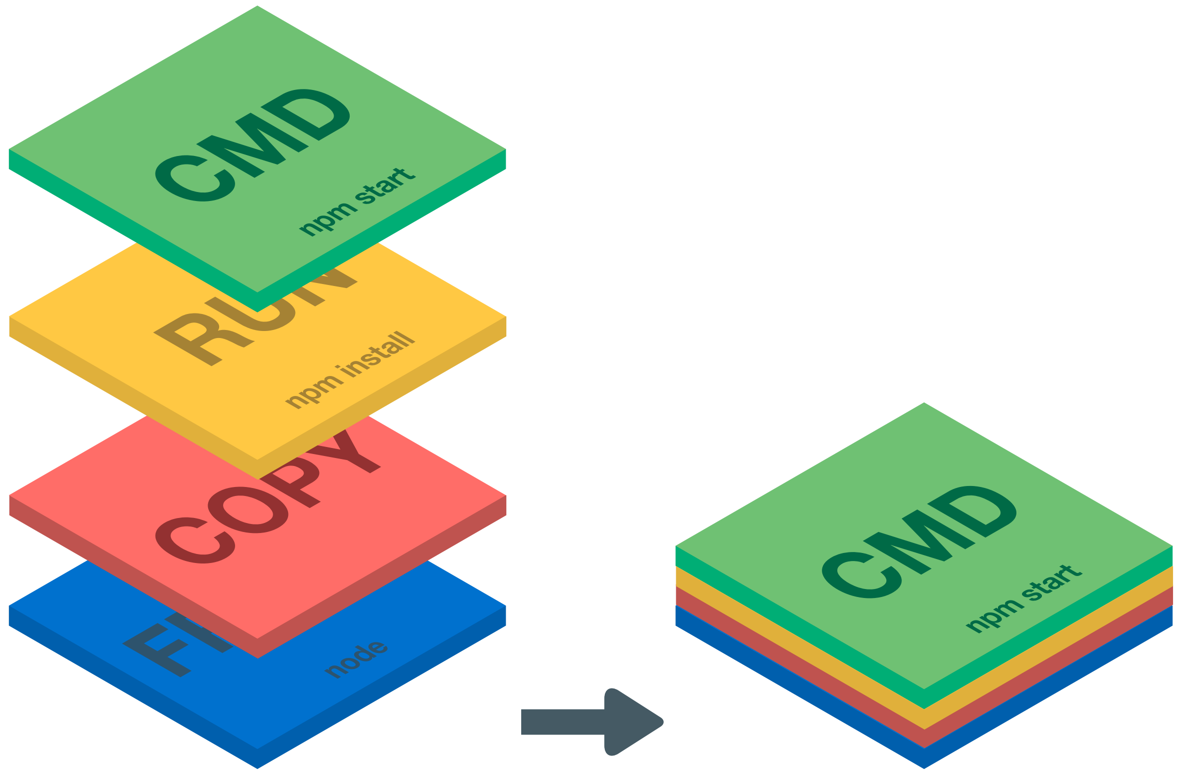
bash

```
$ docker build -t knote .
```

Note the following about this command:

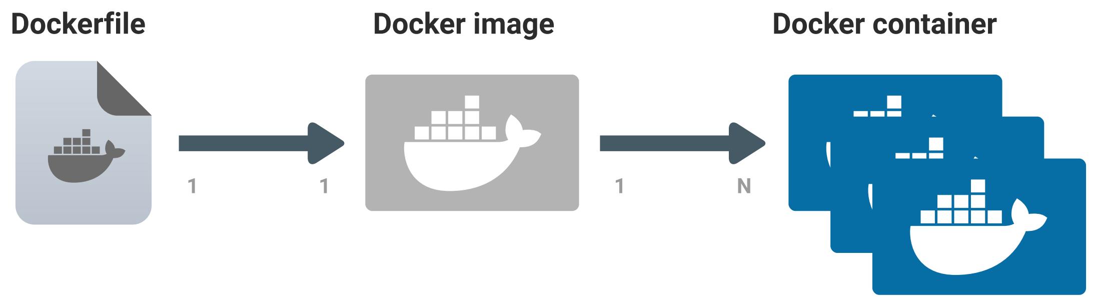
- `-t knote` defines the name ("tag") of your container — in this case, your container is just called `knote`
- `.` is the location of the Dockerfile and application code — in this case, it's the current directory

The command executes the steps outlined in the `Dockerfile`, one by one:



Layers in Docker images

The output is a Docker image. *What is a Docker image?* A Docker image is an archive containing all the files that go in a container. You can create many Docker containers from the same Docker image:



Relationship between Dockerfiles, images and containers

> Don't believe that Docker images are archives? Save the image locally with `docker save knote > knote.tar` and inspect it.

You can list all the images on your system with the following command:

```bash
$ docker images
REPOSITORY     TAG          IMAGE ID         CREATED              SIZE
knote          latest       dc2a8fd35e2e     30 seconds ago       165MB
node           12-slim      d9bfca6c7741     2 weeks ago          150MB

$ _
```

You should see the `knote` image that you built. You should also see the `node:12-slim` which is the base layer of your `knote` image — it is just an ordinary image as well, and the `docker run` command downloaded it automatically from Docker Hub.

> Docker Hub is a container registry — a place to distribute and share container images.

You packaged your app as a Docker image — let's run it as a container.

3. Running the container

Remember that your app requires a MongoDB database. In the previous section, you installed MongoDB on your machine and ran it with the `mongod` command. You could do the same now. *But guess what: you can run MongoDB as a container too.* MongoDB is provided as a Docker image named `mongo` on Docker Hub. *You can run MongoDB without actually "installing" it on your machine.* You can run it with `docker run mongo`. **But before you do that, you need to connect the containers.** The `knote` and `mongo` cointainers should communicate with each other, but they can do so only if they are on the same Docker network. So, create a new Docker network as follows:

bash

```bash
$ docker network create knote
```

Now you can run MongoDB with:

bash

```
$ docker run \
  --name=mongo \
  --rm \
  --network=knote mongo
```

Note the following about this command:

- `--name` defines the name for the container — if you don't specify a name explicitly, then a name is generated automatically
- `--rm` automatically cleans up the container and removes the file system when the container exits
- `--network` represents the Docker network in which the container should run — when omitted, the container runs in the default network
- `mongo` is the name of the Docker image that you want to run

Note that the `docker run` command automatically downloads the `mongo` image from Docker Hub if it's not yet present on your machine. MongoDB is now running. **Now you can run your app as follows:**

```bash
$ docker run \
  --name=knote \
  --rm \
  --network=knote \
  -p 3000:3000 \
  -e MONGO_URL=mongodb://mongo:27017/dev \
  knote
```

Note the following about this command:

- `--name` defines the name for the container
- `--rm` automatically cleans up the container and removes the file system when the container exits
- `--network` represents the Docker network in which the container should run

- `-p 3000:3000` publishes port 3000 of the container to port 3000 of your local machine. That means, if you now access port 3000 on your computer, the request is forwarded to port 3000 of the Knote container. You can use the forwarding to access the app from your local machine.
- `-e` sets an environment variable inside the container

Regarding the last point, remember that your app reads the URL of the MongoDB server to connect to from the `MONGO_URL` environment variable. If you look closely at the value of `MONGO_URL`, you see that the hostname is `mongo`. *Why is it* `mongo` *and not an IP address?* `mongo` is precisely the name that you gave to the MongoDB container with the `--name=mongo` flag. If you named your MongoDB container `foo`, then you would need to change the value of `MONGO_URL` to `mongodb://foo:27017`. **Containers in the same Docker network can talk to each other by their names.** This is made possible by a built-in DNS mechanism. *You should now have two containers running on your machine,* `knote` *and* `mongo`*.* You can display all running containers with the following command:

```bash
$ docker ps
CONTAINER ID    IMAGE    COMMAND                     PORTS                     NAMES
2fc0a10bf0f1    knote    "node index.js"             0.0.0.0:3001->3000/tcp    knote
41b50740a920    mongo    "docker-entrypoint.s…"      27017/tcp                 mongo

$ _
```

Great! *It's time to test your application!* Since you published port 3000 of your container to port 3000 of your local machine, your app is accessible on http://localhost:3000. Go on and open the URL in your web browser. **You should see your app!** Verify that everything works as expected by creating some notes with pictures. When you're done experimenting, stop and remove the containers as follows:

bash

```bash
$ docker stop mongo knote
$ docker rm mongo knote
```

4. Uploading the container image to a container registry

Imagine you want to share your app with a friend — how would you go about sharing your container image? Sure, you could save the image to disk and send it to your friend. *But there is a better way.* When you ran the MongoDB container, you specified its Docker Hub ID (`mongo`), and Docker automatically downloaded the image. *You could create your images and upload them to DockerHub.* If your friend doesn't have the image locally, Docker automatically pulls the image from DockerHub.

> There exist other public container registries, such as Quay — however, Docker Hub is the default registry used by Docker.

To use Docker Hub, you first have to create a Docker ID. A Docker ID is your Docker Hub username. Once you have your Docker ID, you have to authorise Docker to connect to the Docker Hub account:

```bash
$ docker login
```

Before you can upload your image, there is one last thing to do. **Images uploaded to Docker Hub must have a name of the form `username/image:tag`:**

- `username` is your Docker ID
- `image` is the name of the image
- `tag` is an optional additional attribute — often it is used to indicate the version of the image

To rename your image according to this format, run the following command:

```bash
$ docker tag knote <username>/knote-js:1.0.0
```

> Please replace `<username>` with your Docker ID.

Now you can upload your image to Docker Hub:

```bash
$ docker push <username>/knote-js:1.0.0
```

Your image is now publicly available as `<username>/knote-js:1.0.0` on Docker Hub and everybody can download and run it. To verify this, you can re-run your app, but this time using the new image name.

> Please notice that the command below runs the `learnk8s/knote-js:1.0.0` image. If you wish to use yours, replace `learnk8s` with your Docker ID.

bash

```
$ docker run \
  --name=mongo \
  --rm \
  --network=knote \
  mongo
$ docker run \
  --name=knote \
  --rm \
  --network=knote \
  -p 3000:3000 \
  -e MONGO_URL=mongodb://mongo:27017/dev \
  learnk8s/knote-js:1.0.0
```

Everything should work exactly as before. **Note that now everybody in the world can run your application by executing the above two commands.** And the app will run on their machine precisely as it runs on yours — without installing any dependencies. *This is the power of containerisation!* Once you're done testing your app, you can stop and remove the containers with:

bash

```
$ docker stop mongo knote
$ docker rm mongo knote
```

In this section, you learned how to package your app as a Docker image and run it as a container. In the next section, you will learn how to run your containerised application on Kubernetes!

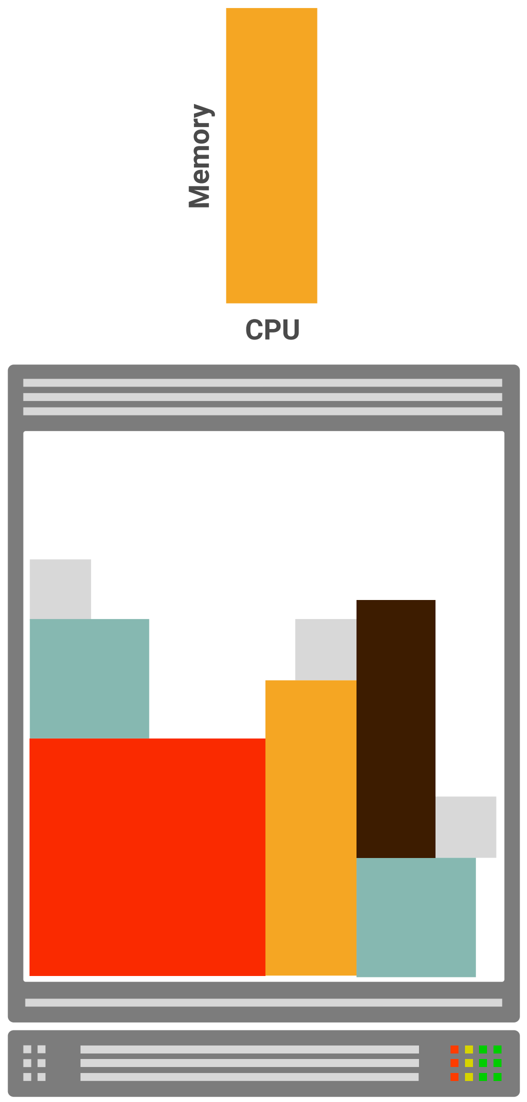
Chapter 3

Deploying to Kubernetes

In the previous section, you containerised your application and ran the containers locally for testing. *But what are your options to deploy your app to production?* There're several services where you can deploy Docker containers, for example, AWS Elastic Beanstalk or Azure App Service. Those services are excellent if you wish to deploy a single or a small number of containers. However, when you're building production-grade applications, it's common to have a large number of components that are all connected. This is especially true if you follow the microservices pattern. Each component, or "microservice", should be scalable independently. Solutions such as AWS Elastic Beanstalk and Azure App Service aren't designed to run those kinds of workloads. *So how do you run complex containerised applications?* With a container orchestrator.

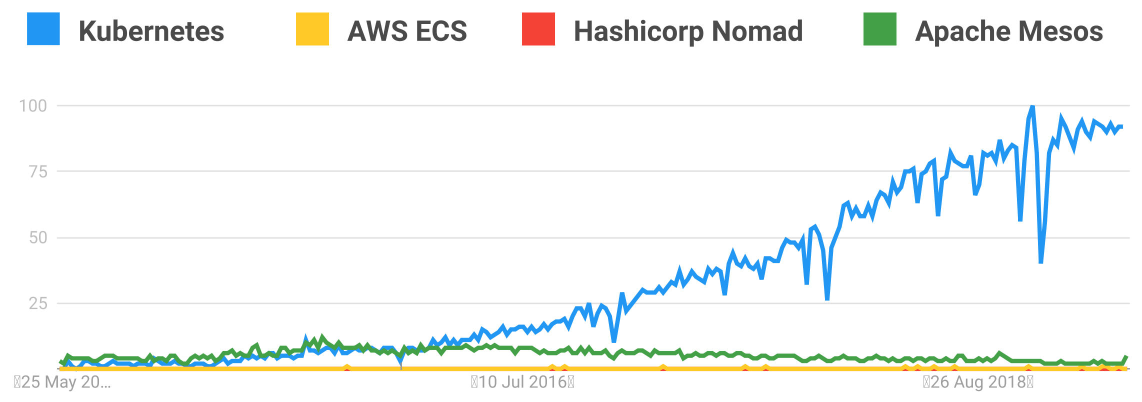
1. Container orchestrators

Container orchestrators are designed to run complex applications with large numbers of scalable components. They work by inspecting the underlying infrastructure and determining the best server to run each container. They can scale to thousands of computers and tens of thousands of containers and still work efficiently and reliably. You can imagine a container orchestrator as a highly-skilled Tetris player. Containers are the blocks, servers are the boards, and the container orchestrator is the player.



Kubernetes is the best tetris player

Some existing container orchestrators include Apache Mesos, Hashicorp Nomad, and Kubernetes. *So which one should you choose?* **It's not a fair fight.** Kubernetes is the de-facto standard when it comes to orchestrating containers at a large scale. Have a look at this Google Trends chart:



Popular orchestrators

Popularity is not the only factor, though. Kubernetes is:

1. **Open-source:** you can download and use it without paying any fee. You're also encouraged to contribute to the official project with bug fixes and new features

2. **Battle-tested:** there're plenty of examples of companies running it in production. There's even a website where you can learn from the mistake of others.
3. **Well-looked-after:** Redhat, Google, Microsoft, IBM, Cisco are only a few of the companies that have heavily invested in the future of Kubernetes by creating managed services, contributing to upstream development and offering training and consulting.

Kubernetes is an excellent choice to deploy your containerised application. *But how do you do that?* It all starts by creating a Kubernetes cluster.

2. Creating a local Kubernetes cluster

There are several ways to create a Kubernetes cluster:

- Using a managed Kubernetes service like Google Kubernetes Service (GKE), Azure Kubernetes Service (AKS), or Amazon Elastic Kubernetes Service (EKS)
- Installing Kubernetes yourself on cloud or on-premises infrastructure with a Kubernetes installation tool like kubeadm or kops
- Creating a Kubernetes cluster on your local machine with a tool like Minikube, MicroK8s, or k3s

In this section, you are going to use Minikube. Minikube creates a single-node Kubernetes cluster running in a virtual machine.

> A Minikube cluster is only intended for testing purposes, not for production. Later in this course, you will create an Amazon EKS cluster, which is suited for production.

Before you install Minikube, you have to install kubectl. kubectl is the primary Kubernetes CLI — you use it for all interactions with a Kubernetes cluster, no matter how the cluster was created. **Once kubectl is installed, go on and install Minikube according to the official documentation.**

> If you're on Windows, you can follow our handy guide on how to install Minikube on Windows.

With Minikube installed, you can create a cluster as follows:

```bash
$ minikube start
```

The command creates a virtual machine and installs Kubernetes. *Starting the virtual machine and cluster may take a couple of minutes, so please be patient!* When the command completes, you can verify that the cluster is created with:

bash

```
$ kubectl cluster-info
```

You have a fully-functioning Kubernetes cluster on your machine now. *Time to learn about some fundamental Kubernetes concepts.*

3. Kubernetes resources

Kubernetes has a declarative interface. In other words, you describe how you want the deployment of your application to look like, and Kubernetes figures out the necessary steps to reach this state. The "language" that you use to communciate with Kubernetes consists of so-called Kubernetes resources. There are many different Kubernetes resources — each is responsible for a specific aspect of your application.

> You can find the full list of Kubernetes resources in the Kubernetes API reference.

Kubernetes resources are defined in YAML files and submitted to the cluster through the Kubernetes HTTP API.

> Kubernetes resource definitions are also sometimes called "resource manifests" or "resource configurations".

As soon as Kubernetes receives your resource definitions, it takes the necessary steps to reach the target state. Similarly, to query the state of your applications, you retrieve Kubernetes resources through the Kubernetes HTTP API. In practice, you do all these interactions with kubectl - your primary client for the Kubernetes API. In the remainder of this section, you will define a set of Kubernetes resources that describe your Knote application, and in the end, you will submit them to your Kubernets cluster. The resources that you will use are the Deployment and the Service. *Let's start with the Deployment.*

4. Defining a Deployment

First of all, create a folder named `kube` in your application directory:

```bash
$ mkdir kube
```

The purpose of this folder is to hold all the Kubernetes YAML files that you will create.

> It's a best practice to group all resource definitions for an application in the same folder because this allows to submit them to the cluster with a single command.

The first Kubernetes resource is a Deployment. A Deployment creates and runs containers and keeps them alive. Here is the definition of a Deployment for your Knote app:

kube/knote.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: knote
spec:
  replicas: 1
  selector:
    matchLabels:
      app: knote
  template:
    metadata:
      labels:
        app: knote
    spec:
      containers:
        - name: app
          image: learnk8s/knote-js:1.0.0
          ports:
            - containerPort: 3000
          env:
            - name: MONGO_URL
              value: mongodb://mongo:27017/dev
          imagePullPolicy: Always
```

That looks complicated, but we will break it down and explain it in detail. For now, save the above content in a file named `knote.yaml` in the `kube` folder.

> Please notice that the command below runs the `learnk8s/knote-js:1.0.0` image. If you wish to use yours, replace `learnk8s` with your Docker ID.

You must be wondering how you can you find out about the structure of a Kubernetes resource. The answer is, in the Kubernetes API reference. The Kubernetes API reference contains the specification for every Kubernetes resource, including all the available fields, their data types, default values, required fields, and so on. Here is the specification of the Deployment resource. *If you prefer to work in the command-line, there's an even better way.* The `kubectl explain` command can print the specification of every Kubernetes resource directly in your terminal:

```bash
$ kubectl explain deployment
```

The command outputs exactly the same information as the web-based API reference. To drill down to a specific field use:

```bash
$ kubectl explain deployment.spec.replicas
```

Now that you know how to look up the documentation of Kubernetes resources, let's turn back to the Deployment. The first four lines define the type of resource (Deployment), the version of this resource type (`apps/v1`), and the name of this specific resource (`knote`):

kube/knote.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: knote
spec:
  replicas: 1
  selector:
    matchLabels:
      app: knote
  template:
    metadata:
      labels:
        app: knote
    spec:
      containers:
        - name: knote
          image: learnk8s/knote-js:1.0.0
          ports:
            - containerPort: 3000
          env:
            - name: MONGO_URL
              value: mongodb://mongo:27017/dev
          imagePullPolicy: Always
```

Next, you have the desired number of replicas of your container:

kube/knote.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: knote
spec:
  replicas: 1
  selector:
    matchLabels:
      app: knote
  template:
    metadata:
      labels:
        app: knote
    spec:
      containers:
        - name: knote
          image: learnk8s/knote-js:1.0.0
          ports:
            - containerPort: 3000
          env:
            - name: MONGO_URL
              value: mongodb://mongo:27017/dev
          imagePullPolicy: Always
```

You don't usually talk about containers in Kubernetes. Instead, you talk about Pods. **What is a Pod?** A Pod is a wrapper around one or more containers. Most often, a Pod contains only a single container — however, for advanced use cases, a Pod may contain multiple containers. If a Pod contains multiple containers, they are treated by Kubernetes as a unit — for example, they are started and stopped together and executed on the same node. *A Pod is the smallest unit of deployment in Kubernetes — you never work with containers directly, but with Pods that wrap containers.* Technically, a Pod is a Kubernetes resource, like a Deployment or Service. **Let's turn back to the Deployment resource.** The next part ties together the Deployment resource with the Pod replicas:

kube/knote.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: knote
spec:
  replicas: 1
  selector:
    matchLabels:
      app: knote
  template:
    metadata:
      labels:
        app: knote
    spec:
      containers:
        - name: knote
          image: learnk8s/knote-js:1.0.0
          ports:
            - containerPort: 3000
          env:
            - name: MONGO_URL
              value: mongodb://mongo:27017/dev
          imagePullPolicy: Always
```

The `template.metadata.labels` field defines a label for the Pods that wrap your Knote container (`app: knote`). The `selector.matchLabels` field selects those Pods with a `app: knote` label to belong to this Deployment resource.

> Note that there must be at least one shared label between these two fields.

The next part in the Deployment defines the actual container that you want to run:

kube/knote.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: knote
spec:
  replicas: 1
  selector:
    matchLabels:
      app: knote
  template:
    metadata:
      labels:
        app: knote
    spec:
      containers:
        - name: knote
          image: learnk8s/knote-js:1.0.0
          ports:
            - containerPort: 3000
          env:
            - name: MONGO_URL
              value: mongodb://mongo:27017/dev
          imagePullPolicy: Always
```

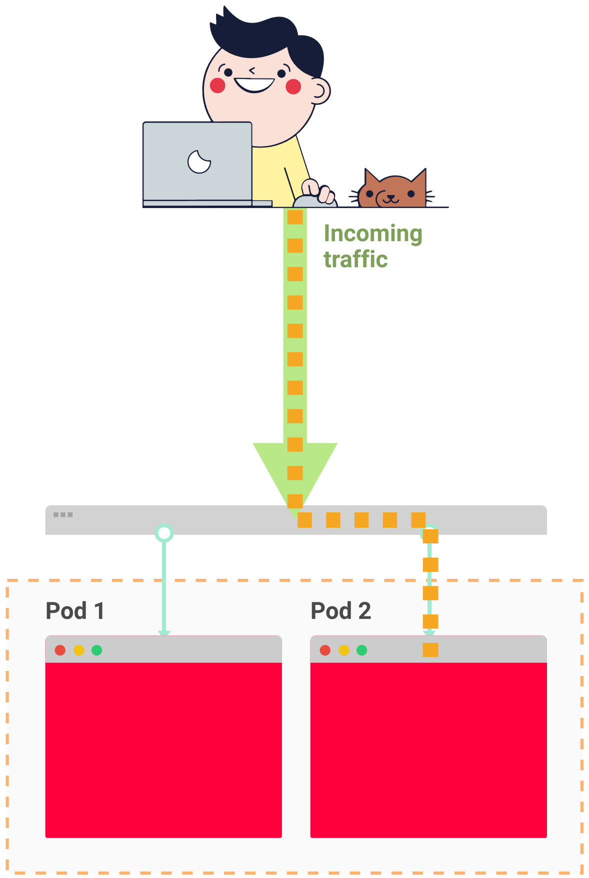
It defines the following things:

- A name for the container (`knote`)
- The name of the Docker image (`learnk8s/knote-js:1.0.0` or `<username>/knote-js:1.0.0` if you're using your image)
- The port that the container listens on (3000)
- An environment variable (`MONGO_URL`) that will be made available to the process in the container

The above arguments should look familiar to you: you used similar ones when you ran your app with `docker run` in the previous section. That's not a coincidence. When you submit a Deployment resource to the cluster, you can imagine Kubernetes executing `docker run` and launching your container in one of the computers. The container specification also defines an `imagePullPolicy` of `Always` — the instruction forces the Docker image to be downloaded, even if it was already downloaded. A Deployment defines how to run an app in the cluster, but it doesn't make it available to other apps. *To expose your app, you need a Service.*

5. Defining a Service

A Service resource makes Pods accessible to other Pods or users outside the cluster. Without a Service, a Pod cannot be accessed at all. A Service forwards requests to a set of Pods:



Services in Kubernetes

In this regard, a Service is akin to a load balancer. Here is the definition of a Service that makes your Knote Pod accessible from outside the cluster:

kube/knote.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: knote
spec:
  selector:
    app: knote
  ports:
    - port: 80
      targetPort: 3000
  type: LoadBalancer
```

> Again, to find out about the available fields of a Service, look it up in the API reference, or, even better, use `kubectl explain service`.

Where should you save the above definition? It is a best-practice to save resource definitions that belong to the same application in the same YAML file. To do so, paste the above content at the beginning of your existing `knote.yaml` file, and separate the Service and Deployment resources with three dashes like this:

kube/knote.yaml

```yaml
# ... Deployment YAML definition
---
# ... Service YAML definition
```

> You can find the final YAML files for this section in this repository.

Let's break down the Service resource. It consists of three crucial parts. The first part is the selector:

kube/knote.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: knote
spec:
  selector:
    app: knote
  ports:
    - port: 80
      targetPort: 3000
  type: LoadBalancer
```

It selects the Pods to expose according to their labels. In this case, all Pods that have a label of `app: knote` will be exposed by the Service. Note how this label corresponds exactly to what you specified for the Pods in the Deployment resource:

kube/knote.yaml

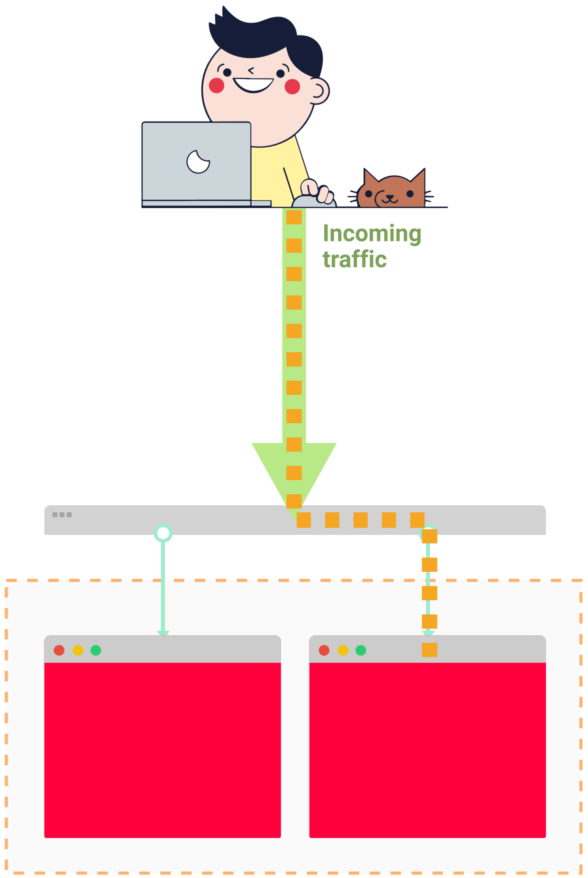
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: knote
spec:
  # ...
  template:
    metadata:
      labels:
        app: knote
    # ...
```

It is this label that ties your Service to your Deployment resource. The next important part is the port:

kube/knote.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: knote
spec:
  selector:
    app: knote
  ports:
    - port: 80
      targetPort: 3000
  type: LoadBalancer
```

In this case, the Service listens for requests on port 80 and forwards them to port 3000 of the target Pods:



Service and ports

The last important part is the type of the Service:

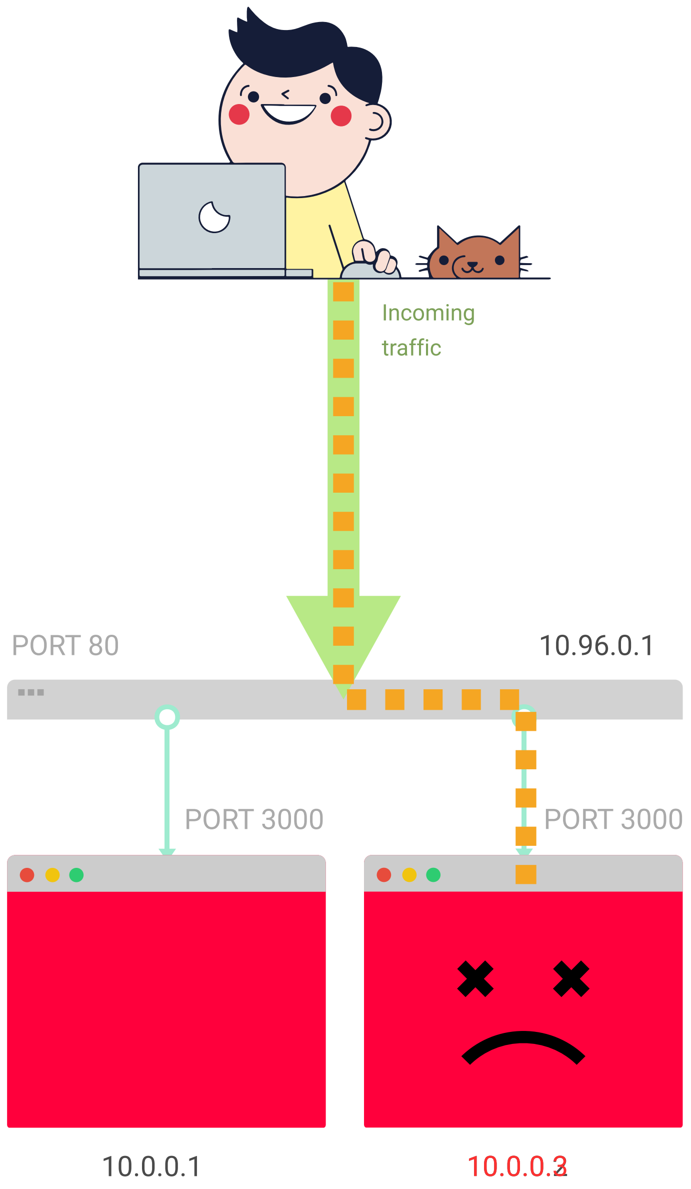
kube/knote.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: knote
spec:
  selector:
    app: knote
  ports:
    - port: 80
      targetPort: 3000
  type: LoadBalancer
```

In this case, the type is `LoadBalancer`, which makes the exposed Pods accessible from outside the cluster. The default Service type is `ClusterIP`, which makes the exposed Pods only accessible from within the cluster.

> **Pro tip:** find out about all available Service types with `kubectl explain service.spec.type`.

Beyond exposing your containers, a Service also ensures continuous availability for your app. If one of the Pod crashes and is restarted, the Service makes sure not to route traffic to this container until it is ready again. Also, when the Pod is restarted, and a new IP address is assigned, the Service automatically handles the update too.



A load balancer can remove the need of keeping track of IP addresses

Furthermore, if you decide to scale your Deployment to 2, 3, 4, or 100 replicas, the Service keeps track of all of these Pods. This completes the description of your app — a Deployment and Service is all you need. *You need to do the same thing for the database component now.*

6. Defining the database tier

In principle, a MongoDB Pod can be deployed similarly as your app — that is, by defining a Deployment and Service resource. However, deploying MongoDB needs some additional configuration. **MongoDB requires a persistent storage.** This storage must not be affected by whatever happens to the MongoDB Pod. *If the MongoDB Pod is deleted, the storage must persist — if the MongoDB Pod is moved to another node, the storage must persist.* There exists a Kubernetes resource that allows obtaining persistent storage volume: the PersistentVolumeClaim. Consequently, the description of your database component should consist of three resource definitions:

- PersistentVolumeClaim
- Service
- Deployment

Here's the complete configuration:

kube/mongo.yaml (1/2)

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongo-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 256Mi
---
apiVersion: v1
kind: Service
metadata:
  name: mongo
spec:
  selector:
    app: mongo
  ports:
    - port: 27017
      targetPort: 27017
---
apiVersion: apps/v1
kind: Deployment
```

kube/mongo.yaml (2/2)

```yaml
metadata:
  name: mongo
spec:
  selector:
    matchLabels:
      app: mongo
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
        - name: mongo
          image: mongo:3.6.17-xenial
          ports:
            - containerPort: 27017
          volumeMounts:
            - name: storage
              mountPath: /data/db
      volumes:
        - name: storage
          persistentVolumeClaim:
            claimName: mongo-pvc
```

Please save this YAML definition in a file named `mongo.yaml` in the `kube` directory. Let's look at each of the three parts of the definition. **PersistentVolumeClaim** The PersistentVolumeClaim requests a persistent storage volume of 256 MB. This volume is made available to the MongoDB container to save its data. **Service** The Service is similar to the Service you defined for the app component. However, note that it does not have a `type` field. If a Service does not have a `type` field, Kubernetes assigns it the default type `ClusterIP`. `ClusterIP` makes the Pod accessible from within the cluster, but not from outside — this is fine because the only entity that has to access the MongoDB Pod is your app. **Deployment** The Deployment has a similar structure to the other Deployment. However, it contains an additional field that you haven't seen yet: `volumes`. The `volumes` field defines a storage volume named `storage`, which references the PersistentVolumeClaim. Furthermore, the volume is referenced from the `volumeMounts` field in the definition of the MongoDB container. The `volumeMount` field mounts the referenced volume at the specified path in the container, which in this case is `/data/db`. And `/data/db` is where MongoDB saves its data. In other words, the MongoDB database data is stored in a persistent storage volume that has a lifecycle independent of the MongoDB container.

> Deploying stateful applications to Kubernetes is a complex but essential topic. You can learn more about it in Managing State module of the Learnk8s Academy.

There's one more important thing to note. Do you remember the value of the `MONGO_URL` environment variable in the Knote Deployment? It is `mongodb://mongo:27017/dev`. The hostname is `mongo`. *Why is it* `mongo`*?* Because the name of the MongoDB Service is `mongo`. If you named your MongoDB service `foo`, then you would need to change the value of the `MONGO_URL` variable to `monogdb://foo:27017`. Service discovery is a critical Kubernetes concept. **Pods within a cluster can talk to each other through the names of the Services exposing them.** Kubernetes has an internal DNS system that keeps track of domain names and IP addresses. Similarly to how Docker provides DNS resolution for containers, Kubernetes provides DNS resolution for Services. *All components of your app are described by Kubernetes resources now — let's deploy them to the cluster.*

7. Deploying the application

So far, you created a few YAML files with resource definitions. *You didn't yet touch the cluster.* **But now comes the big moment!** You are going to submit your resource definitions to Kubernetes. And Kubernetes will bring your application to life. First of all, make sure that you have a `knote.yaml` and `mongo.yaml` file inside the `kube` directory:

bash

```
$ tree .
kube/
├── knote.yaml
└── mongo.yaml

$ _
```

> You can find these files also in this repository.

Also, make sure that your Minikube cluster is running:

bash

```bash
$ minikube status
```

Then submit your resource definitions to Kubernetes with the following command:

bash

```bash
$ kubectl apply -f kube
```

This command submits all the YAML files in the `kube` directory to Kubernetes.

> The `-f` flag accepts either a single filename or a directory. In the latter case, all YAML files in the directory are submitted.

As soon as Kubernetes receives your resources, it creates the Pods. You can watch your Pods coming alive with:

bash

```bash
$ kubectl get pods --watch
```

You should see two Pods transitioning from *Pending* to *ContainerCreating* to *Running*. These Pods correspond to the Knote and MongoDB containers. *As soon as both Pods are in the Running state, your application is ready.* You can now access your application through the `knote` Service. In Minikube, a Service can be accessed with the following command:

bash

```
$ minikube service knote
```

The command should open the URL of the `knote` Service in a web browser. **You should see your application.** Verify that your app works as expected by creating some notes with pictures. The app should work as it did when you ran it locally with Docker. *But now it's running on Kubernetes.* When you're done testing the app, you can remove it from the cluster with the following command:

bash

```
$ kubectl delete -f kube
```

The command deletes all the resources that were created by `kubectl apply`. In this section, you learned how to deploy an application to Kubernetes. In the next section, you will learn how to scale your application for high-availability and resiliency.

Chapter 4

Scaling

Scaling means increasing or decreasing the number of replicas of an application component. You could scale your Pod to 2, 3, or 10 replicas. The benefits of this are twofold:

- High availability: if one of the Pods crashes, there are still other replicas running that can handle incoming requests
- Resiliency: the total traffic to your application is distributed between all the replicas. Thus, the risk of overload of a single Pod is reduced

In this section, you will learn how to scale the Knote component of your app to any number of replicas.

1. Scaling and statefulness

First of all, re-deploy your application to your Minikube cluster and wait until all the Pods are ready:

```bash
$ kubectl apply -f kube
```

At the moment, there is one Pod running for the app and one for the database. Kubernetes makes it very easy to increase the number of replicas to 2:

```bash
$ kubectl scale --replicas=2 deployment/knote
```

You can watch how a new Pod is created with:

```bash
$ kubectl get pods -l app=knote --watch
```

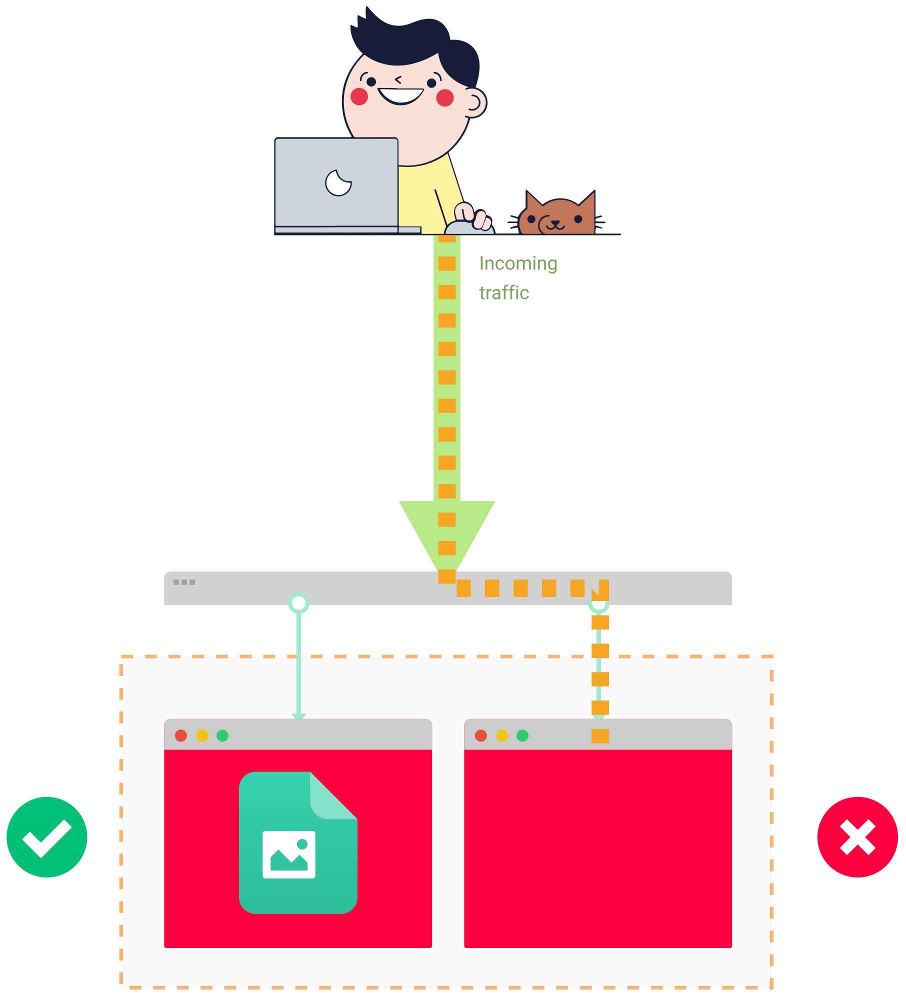
> The `-l` flag restricts the output to only those Pods with a `app=knote` label.

There are now two replicas of the Knote Pod running. *So, are you already done?* Reaccess your app:

```bash
$ minikube service knote
```

And create a note with a picture. Now try to reload your app a couple of times (i.e. hit your browser's reload button). *Did you notice any glitch?* **The picture that you added to your note is not displayed on every reload.** If you pay attention, the picture is only displayed on every second reload, on average. *Why is* *that?* Remember that your application saves uploaded pictures in the local file system. If your app runs in a container, then pictures are saved within the container's file system. When you had only a single Pod, this was fine. But since you have two replicas, there's an issue. The picture that you previously uploaded is saved in only one of the two Pods. When you access your app, the `knote` Service selects one of the available Pods. When it selects the Pod that has the picture in its file system, the image is displayed. But when it selects the other Pod, the picture isn't displayed, because the container doesn't have it.

Uploading files inside the app makes it stateful



Your application is stateful. The pictures in the local filesystem constitute a state that is local to each container. **To be scalable, applications must be stateless.** Stateless means that an instance can be killed restarted or duplicated at any time without any data loss or inconsistent behaviour. *You must make your app stateless before you can scale it.* In this section, you will refactor your app to make it stateless. **But before you proceed, remove your current application from the cluster:**

bash

```bash
$ kubectl delete -f kube
```

The command ensures that the old data in the database's persistent volume does not stick around.

2. Making the app stateless

How can you make your application stateless? The challenge is that uploaded pictures are saved in the container's file system where they can be accessed only by the current Pod. However, you could save the pictures in a central place where all Pods can access them. **An object storage is an ideal system to store files centrally.** You could use a cloud solution, such as Amazon S3. But to hone your Kubernetes skills, you could deploy an object storage service yourself. MinIO is an open-source object storage service that can be installed on your infrastructure. And you can install MinIO in your Kubernetes cluster. *Let's start by changing the app to use MinIO.* First of all, install the MinIO SDK for JavaScript:

bash

```bash
$ npm install minio
```

And include it in your `index.js` file:

```
index.js

const minio = require('minio')
```

You also need a few constants:

```
index.js

const minioHost = process.env.MINIO_HOST || 'localhost'
const minioBucket = 'image-storage'
```

Next, you should connect to the MinIO server. As for MongoDB, your app should keep trying to connect to MinIO until it succeeds. You should gracefully handle the case when the MinIO Pod is started with a delay. Also, you have to create a so-called bucket on MinIO, which can be seen as the "folder" where your pictures are saved. The above tasks are summarised as code in the following function:

index.js

```
async function initMinIO() {
  console.log('Initialising MinIO...')
  const client = new minio.Client({
    endPoint: minioHost,
    port: 9000,
    useSSL: false,
    accessKey: process.env.MINIO_ACCESS_KEY,
    secretKey: process.env.MINIO_SECRET_KEY,
  })
  let success = false
  while (!success) {
    try {
      if (!(await client.bucketExists(minioBucket))) {
        await client.makeBucket(minioBucket)
      }
      success = true
    } catch {
      await new Promise(resolve => setTimeout(resolve, 1000))
    }
  }
  console.log('MinIO initialised')
  return client
}
```

Go on and add this function to the `index.js` file. You should call it in the `start` function:

```
index.js
```

```javascript
async function start() {
  const db = await initMongo()
  const minio = await initMinIO()
  // ...
}
```

Now you should modify the handler of the `/note` route to save the pictures to MinIO, rather than to the local file system (changed lines are highlighted):

index.js (1/2)

```
async function start() {
  // ...
  app.post(
    '/note',
    multer({ storage: multer.memoryStorage() }).single('image'),
    async (req, res) => {
      if (!req.body.upload && req.body.description) {
        await saveNote(db, { description: req.body.description })
        res.redirect('/')
      } else if (req.body.upload && req.file) {
        await minio.putObject(
          minioBucket,
          req.file.originalname,
          req.file.buffer
        )
        const link = `/img/${encodeURIComponent(req.file.originalname)}`
        res.render('index', {
          content: `${req.body.description} ![](${link})`,
          notes: await retrieveNotes(db),
        })
      }
    }
  )
  // ...
```

index.js (2/2)

```
}
```

You're almost done. There's one last thing to do. The pictures are served to the clients via the `/img` route of your app. So, you should create an additional route:

index.js

```
async function start() {
  // ...
  app.get('/img/:name', async (req, res) => {
    const stream = await minio.getObject(
      minioBucket,
      decodeURIComponent(req.params.name),
    )
    stream.pipe(res)
  })
  // ...
}
```

The `/img` route retrieves a picture by its name from MinIO and serves it to the client. **Now you're done.**

> Note that you can find the final source code file in this repository.

But can you be sure that you didn't introduce any bug into the code? Please test your code right now before you build a new Docker image and deploy it to Kubernetes. *To do so, you can run your changed app locally.*

3. Testing the app

Your app has two dependencies now: MongoDB and MinIO. To run your app locally, you must run its dependencies too. You can run both MongoDB and MinIO as Docker containers. MinIO is provided as `minio/minio` on Docker Hub. You can start a MinIO container like this:

bash

```bash
$ docker run \
  --name=minio \
  --rm \
  -p 9000:9000 \
  -e MINIO_ACCESS_KEY=mykey \
  -e MINIO_SECRET_KEY=mysecret \
  minio/minio server /data
```

> Note that `mykey` and `mysecret` are the MinIO credentials and you can choose them yourself.

And you can start a MongoDB container like this:

```bash
$ docker run \
  --name=mongo \
  --rm \
  -p 27017:27017 \
  mongo
```

Note that both of these commands expose a container port to the local machine with the `-p` flag. You will use those containers as dependent components while your app is connected to `localhost`. You can start the app like this:

```bash
$ MINIO_ACCESS_KEY=mykey MINIO_SECRET_KEY=mysecret node index.js
```

Note that it's necessary to set the `MINIO_ACCESS_KEY` and `MINIO_SECRET_KEY` variables every time you run your app. Furthermore, the values of these variables must match the same credentials defined earlier for MinIO.

> You should never hard-code credentials in the application code, that's why these environment variables don't have any default values in `index.js` .

If you visit http://localhost:3000, you should see your app. Test if everything works correctly by creating some notes with pictures. Your app should behave just like it did before. *But the crucial difference is that now uploaded pictures are saved in the MinIO object storage rather than on the local file system.* When you're convinced that your app works correctly, terminate it with *Ctrl-C* and stop MongoDB and MinIO with:

```bash
$ docker stop mongo minio
$ docker rm mongo minio
```

You created a new version of your app. *It's time to package it as a new Docker image.*

4. Containerising the app

To build a new Docker image of your app, run the following command:

bash

```bash
$ docker build -t <username>/knote-js:2.0.0 .
```

You should be familiar with that command. It is precisely the command that you used to build the first version of the app in the "Containerisation" section. Only the tag is different, though. You should also upload your new image Docker Hub:

bash

```bash
$ docker push <username>/knote-js:2.0.0
```

> Please replace `<username>` with your Docker ID.

The two versions of the image will live side-by-side in your Docker Hub repository. You can use the second version by specifying `<username>/knote-js:2.0.0`, but you can still use the first version by specifying `<username>/knote-js:1.0.0`. You have now three Docker images, all available on Docker Hub.

- `<username>/knote-js:2.0.0`
- `mongo`
- `minio/minio`

Let's do a little exercise. You will rerun your application, but this time with *all* three components as Docker containers.

> This is how you ran the application in the "Containerisation" section, but there you had only two containers, now you have three.

First of all, make sure that you still have the `knote` Docker network:

```bash
$ docker network ls
```

Next, run the MongoDB container as follows:

```bash
$ docker run \
  --name=mongo \
  --rm \
  --network=knote \
  mongo
```

Then, run the MinIO container as follows:

bash

```
$ docker run \
  --name=minio \
  --rm \
  --network=knote \
  -e MINIO_ACCESS_KEY=mykey \
  -e MINIO_SECRET_KEY=mysecret \
  minio/minio server /data
```

Note the following about these two `docker run` commands:

1. They don't publish any ports to the host machine with the `-p` flag, because they don't need to be accessed from the host machine. The only entity that has to access them is the Knote container
2. Both containers are started in the `knote` Docker network to enable them to communicate with the Knote container

Now, run the Knote container as follows:

```bash
$ docker run \
  --name=knote \
  --rm \
  --network=knote \
  -p 3000:3000 \
  -e MONGO_URL=mongodb://mongo:27017/dev \
  -e MINIO_ACCESS_KEY=mykey \
  -e MINIO_SECRET_KEY=mysecret \
  -e MINIO_HOST=minio \
  learnk8s/knote-js:2.0.0
```

Note the following about this command:

- It starts the container in the `knote` Docker network so that it can communicate with the MongoDB and MinIO containers
- It publishes port 3000 to the host machine with the `-p` flag to allow you to access the app from your local machine
- In addition to the `MINIO_ACCESS_KEY` and `MINIO_SECRET_KEY` environment variables, it sets the `MONGO_URL` and `MINIO_HOST` environment variables

These latter two environment variables are particularly important. The hostname of the `MONGO_URL` variable is `mongo` — this corresponds to the name of the MongoDB container. The `MINIO_HOST` variable is set to `minio` — this corresponds to the name fo the MinIO container. These values are what allows the Knote container to talk to the MongoDB and MinIO container in the Docker network. *Remember from the "Containerisation" section that containers in the same Docker network can talk to each other by their names.* You should now be able to access your app on http://localhost:3000. Verify that your app still works as expected. But since you didn't do any further changes to your app, everything should still work correctly. When you're done, stop and tidy up with:

bash

```bash
$ docker stop knote minio mongo
$ docker rm knote minio mongo
```

After this digression to Docker, let's return to Kubernetes.

5. Updating the Knote configuration

If you want to deploy the new version of your app to Kubernetes, you have to do a few changes to the YAML resources. In particular, you have to update the Docker image name and add the additional environment variables in the Deployment resource. You should change the Deployment resource in your `knote.yaml` file as follows (changed lines are highlighted):

kube/knote.yaml (1/2)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: knote
spec:
  replicas: 1
  selector:
    matchLabels:
      app: knote
  template:
    metadata:
      labels:
        app: knote
    spec:
      containers:
        - name: knote
          image: learnk8s/knote-js:2.0.0
          ports:
            - containerPort: 3000
          env:
            - name: MONGO_URL
              value: mongodb://mongo:27017/dev
            - name: MINIO_ACCESS_KEY
              value: mykey
```

kube/knote.yaml (2/2)

```yaml
              - name: MINIO_SECRET_KEY
                value: mysecret
              - name: MINIO_HOST
                value: minio
            imagePullPolicy: Always
```

> Please notice that the command below runs the `learnk8s/knote-js:2.0.0` image. If you wish to use yours, replace `learnk8s` with your Docker ID.

The Deployment is ready to be submitted to Kubernetes. However, something else is missing. You should create a Kubernetes description for the new MinIO component. *So, let's do it.*

6. Defining the MinIO component

The task at hand is to deploy MinIO to a Kubernetes cluster. You should be able to guess what the Kubernetes description for MinIO looks like. **It should look like the MongoDB description that you defined in the "Deploying to Kubernetes" section.** As for MongoDB, MinIO requires persistent storage to save its state. Also like MongoDB, MinIO must be exposed with a Service for Pods inside the cluster. Here's the complete MinIO configuration:

kube/minio.yaml (1/3)

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: minio-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 256Mi
---
apiVersion: v1
kind: Service
metadata:
  name: minio
spec:
  selector:
    app: minio
  ports:
    - port: 9000
      targetPort: 9000
---
apiVersion: apps/v1
kind: Deployment
```

kube/minio.yaml (2/3)

```yaml
metadata:
  name: minio
spec:
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: minio
  template:
    metadata:
      labels:
        app: minio
    spec:
      containers:
        - name: minio
          image: minio/minio:RELEASE.2020-03-14T02-21-58Z
          args:
            - server
            - /storage
          env:
            - name: MINIO_ACCESS_KEY
              value: mykey
            - name: MINIO_SECRET_KEY
              value: mysecret
```

kube/minio.yaml (3/3)

```
          ports:
            - containerPort: 9000
          volumeMounts:
            - name: storage
              mountPath: /storage
      volumes:
        - name: storage
          persistentVolumeClaim:
            claimName: minio-pvc
```

Go on and save the content in a file named `minio.yaml` in the `kube` directory. The YAML file has a Deployment, Service and PersistenVolumeClaim definition. So, if you recall the MongoDB resource definition, you should have no troubles understanding the MinIO configuration.

> If you want to learn more about deploying stateful applications to Kubernetes, check out the Managing state module in the Learnk8s Academy.

You just defined the last of the three components of your application. Your Kubernetes configuration is now complete. *It's time to deploy the application!*

7. Deploying the app

Now comes the big moment! You will deploy the new version of your app to Kubernetes. First of all, make sure that you have the following three YAML files in the `kube` directory:

```bash
$ tree .
kube/
├── knote.yaml
├── minio.yaml
└── mongo.yaml

$ _
```

Also, make sure that you deleted the previous version of the app from the cluster:

bash

```
$ kubectl get pods
```

The command shouldn't output any resources.

> Deleting the previous version of the app makes sure that the old data in the MongoDB database is removed.

Now deploy your app to Kubernetes:

bash

```
$ kubectl apply -f kube
```

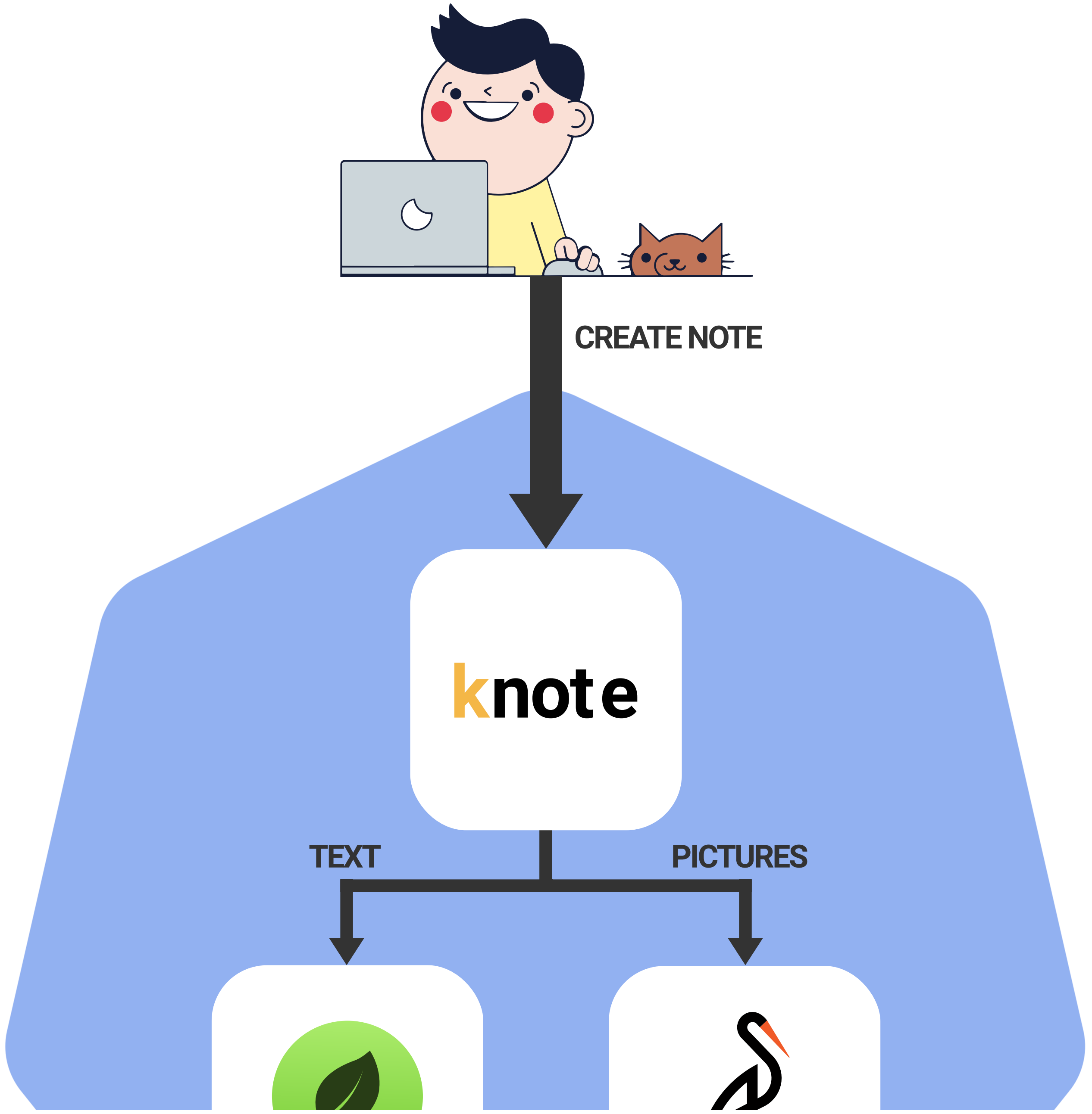
Watch your Pods being created:

```bash
$ kubectl get pods --watch
```

You should see three Pods. Once all three Pods are *Running*, your application is ready. Access your app with:

```bash
$ minikube service knote
```

The command should open your app in a web browser. Verify that it works correctly by creating some notes with pictures. **It should work as expected!** As a summary, here is what your application looks like now:



Knote application architecture

It consists of three components:

1. Knote as the primary application for creating notes
2. MongoDB for storing the text of the notes, and
3. MinIO for storing the pictures of the notes.

Only the Knote component is accessible from outside the cluster — the MongoDB and MinIO components are hidden inside. *At the moment, you're running a single Knote container.* But the topic of this section is "scaling". And you completed some major refactoring to your app to make it stateless and scalable. *So, let's scale it!*

8. Scaling the app

Your app is now stateless because it saves the uploaded pictures on a MinIO server instead of the Pod's file system. *In other words, when you scale your app, all pictures should appear on every request.* **It's the moment of truth.** Scale the Knote container to 10 replicas:

```bash
$ kubectl scale --replicas=10 deployment/knote
```

There should be nine additional Knote Pods being created. You can watch them come online with:

```bash
$ kubectl get pods -l app=knote --watch
```

After a short moment, the new Pods should all be *Running*. Go back to your app in the web browser and reload the page a couple of times. *Are the pictures always displayed?* **Yes they are!** Thanks to statelessness, your Pods can now be scaled to any number of replicas without any data loss or inconsistent behaviour. *You just created a scalable app!* When you're done playing with your app, delete it from the cluster with:

```bash
$ kubectl delete -f kube
```

And stop Minikube with:

```bash
$ minikube stop
```

You don't need Minikube anymore. In the next section, you will create a new Kubernetes cluster in the cloud and deploy your app there!

Chapter 5

Deploying to the cloud

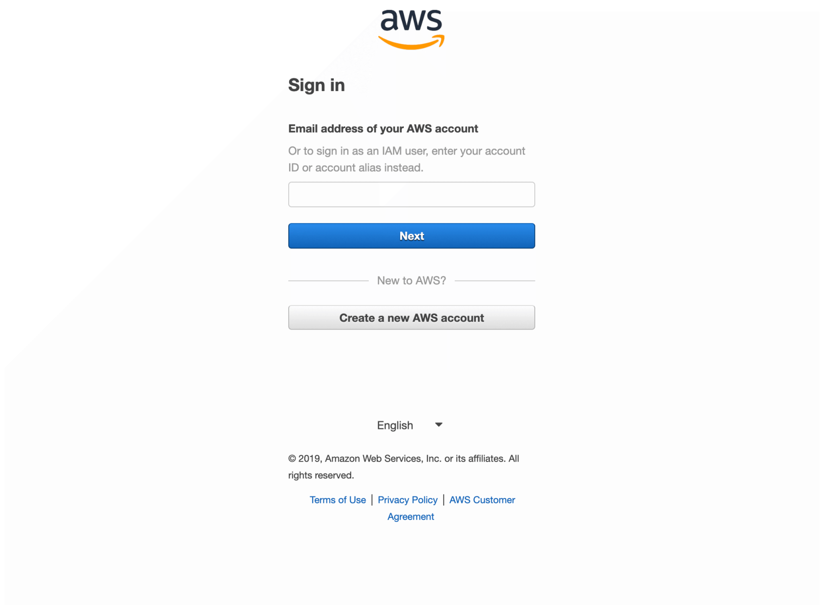
At this point, you have defined a Kubernetes application consisting of three components and deployed them to a local Minikube cluster. The app worked perfectly on Minikube. *Does it works when you deploy it to a "real" Kubernetes cluster in the cloud?* In other words, can you submit the same three YAML files to a cluster in the cloud and expect it to work? *It's time to find out.* In this section, you will create a production-grade Kubernetes cluster on AWS using Amazon Elastic Kubernetes Service (EKS), and you will deploy your application to it.

1. Creating and setting up an AWS account

To use AWS services, you need an AWS account. *If you already have an AWS account, you can jump to the next section →* The first step is signing up for an AWS account.

> Note that to create any resources on AWS, you have to provide valid credit card details.

Next, you should create an AWS access key — this is necessary to access AWS services from the command line. To do so, follow these instructions:



Sign in

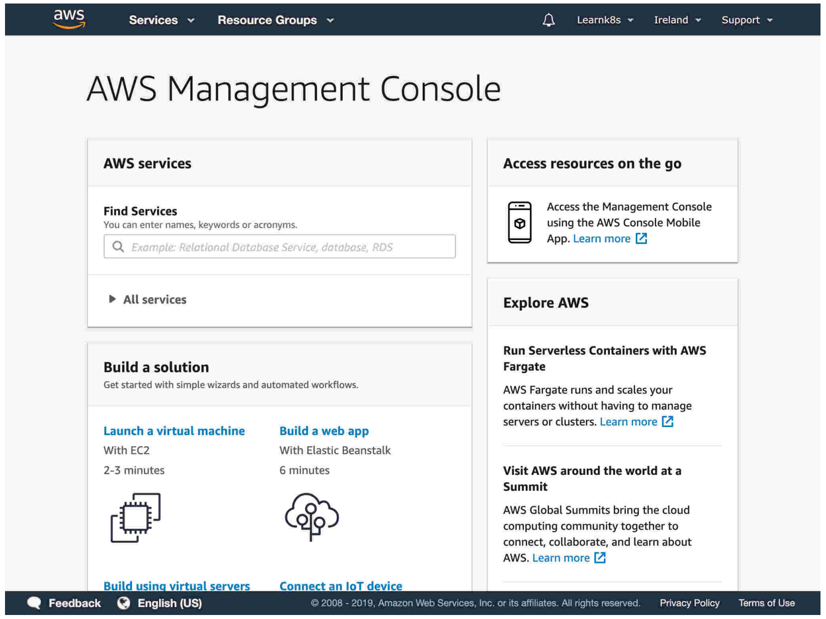
Email address of your AWS account

Or to sign in as an IAM user, enter your account ID or account alias instead.

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

1/13

Log in to your AWS Management Console.



AWS Management Console

AWS services

Find Services

You can enter names, keywords or acronyms.

Example: Relational Database Service, database, RDS

All services

Build a solution

Get started with simple wizards and automated workflows.

Launch a virtual machine
With EC2
2-3 minutes

Build a web app
With Elastic Beanstalk
6 minutes

Build using virtual servers

Connect an IoT device

Access resources on the go

Access the Management Console using the AWS Console Mobile App. Learn more

Explore AWS

Run Serverless Containers with AWS Fargate

AWS Fargate runs and scales your containers without having to manage servers or clusters. Learn more

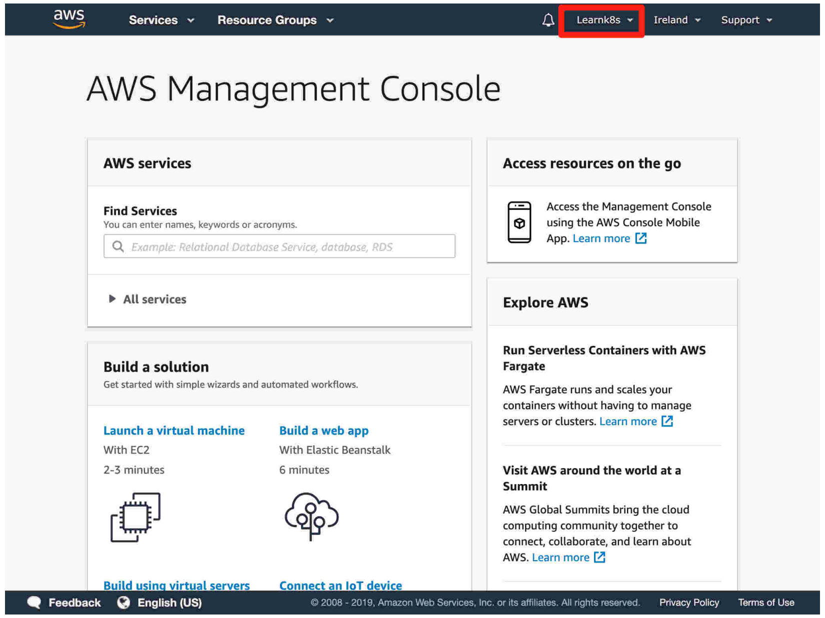
Visit AWS around the world at a Summit

AWS Global Summits bring the cloud computing community together to connect, collaborate, and learn about AWS. Learn more

Feedback
English (US)
© 2008 - 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

2/13

You should see your AWS console once you're logged in.



You can enter names, keywords or acronyms.

Example: Relational Database Service, database, RDS

Get started with simple wizards and automated workflows.

Access the Management Console using the AWS Console Mobile App. Learn more

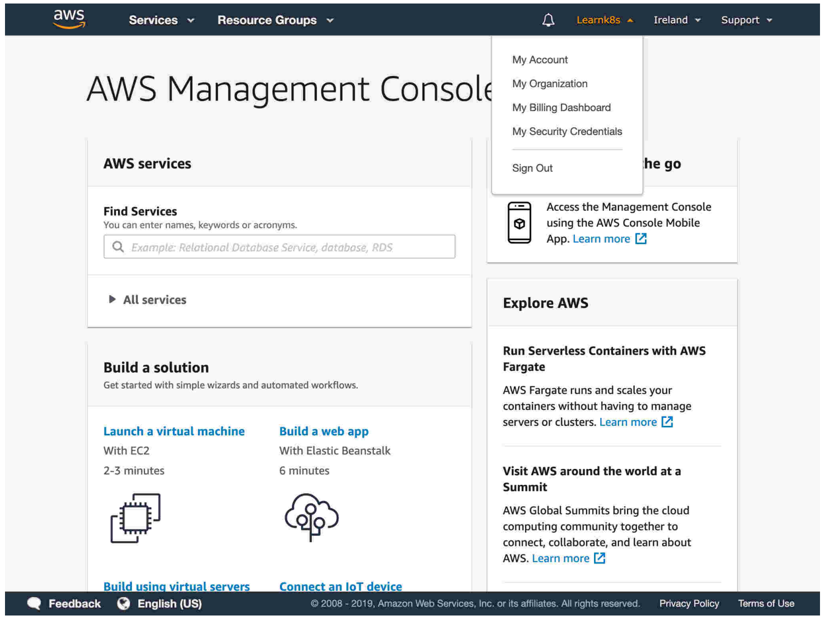
AWS Fargate runs and scales your containers without having to manage servers or clusters. Learn more

AWS Global Summits bring the cloud computing community together to connect, collaborate, and learn about AWS. Learn more

© 2008 - 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

3/13

Click on your user name at the top right of the page.



You can enter names, keywords or acronyms.

Example: Relational Database Service, database, RDS

Get started with simple wizards and automated workflows.

Access the Management Console using the AWS Console Mobile App. Learn more

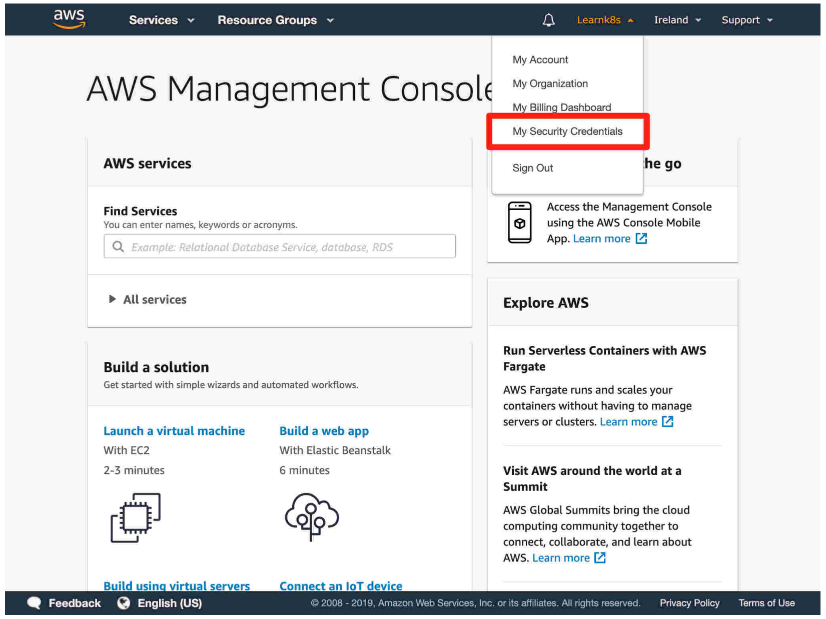
AWS Fargate runs and scales your containers without having to manage servers or clusters. Learn more

AWS Global Summits bring the cloud computing community together to connect, collaborate, and learn about AWS. Learn more

© 2008 - 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

4/13

In the drop down there's an item for "My Security Credentials".



AWS Management Console

AWS services

Find Services
You can enter names, keywords or acronyms.
Example: Relational Database Service, database, RDS

All services

Build a solution
Get started with simple wizards and automated workflows.

Launch a virtual machine
With EC2
2-3 minutes

Build a web app
With Elastic Beanstalk
6 minutes

Build using virtual servers

Connect an IoT device

he go

Access the Management Console using the AWS Console Mobile App. Learn more

Explore AWS

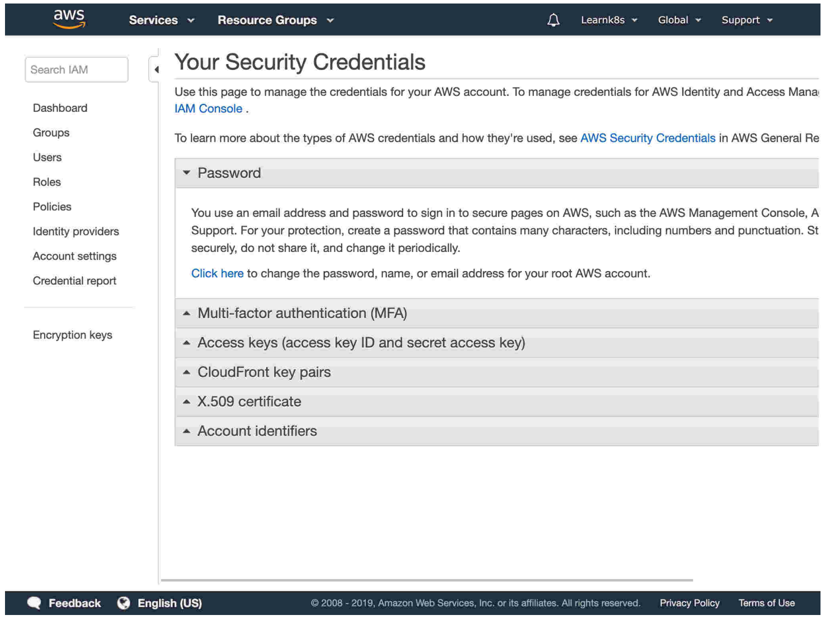
Run Serverless Containers with AWS Fargate
AWS Fargate runs and scales your containers without having to manage servers or clusters. Learn more

Visit AWS around the world at a Summit
AWS Global Summits bring the cloud computing community together to connect, collaborate, and learn about AWS. Learn more

© 2008 - 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

5/13

Click on "My Security Credentials".



Your Security Credentials

Use this page to manage the credentials for your AWS account. To manage credentials for AWS Identity and Access Mana
IAM Console .

To learn more about the types of AWS credentials and how they're used, see AWS Security Credentials in AWS General Re

Password

You use an email address and password to sign in to secure pages on AWS, such as the AWS Management Console, A
Support. For your protection, create a password that contains many characters, including numbers and punctuation. St
securely, do not share it, and change it periodically.

Click here to change the password, name, or email address for your root AWS account.

Multi-factor authentication (MFA)

Access keys (access key ID and secret access key)

CloudFront key pairs

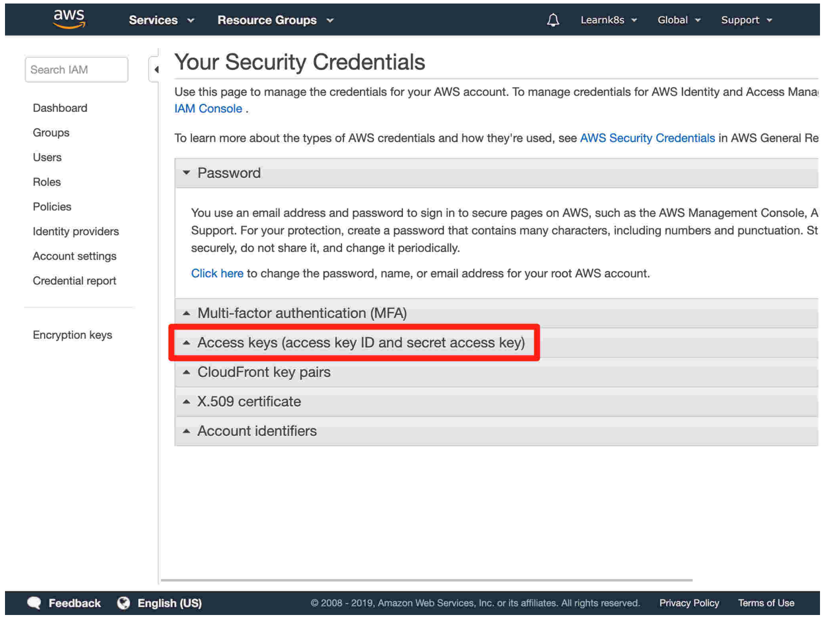
X.509 certificate

Account identifiers

© 2008 - 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

6/13

You should land on Your Security Credentials page.



Your Security Credentials

Use this page to manage the credentials for your AWS account. To manage credentials for AWS Identity and Access Mana
IAM Console .

To learn more about the types of AWS credentials and how they're used, see AWS Security Credentials in AWS General Re

Password

You use an email address and password to sign in to secure pages on AWS, such as the AWS Management Console, A
Support. For your protection, create a password that contains many characters, including numbers and punctuation. St
securely, do not share it, and change it periodically.

Click here to change the password, name, or email address for your root AWS account.

Multi-factor authentication (MFA)

Access keys (access key ID and secret access key)

CloudFront key pairs

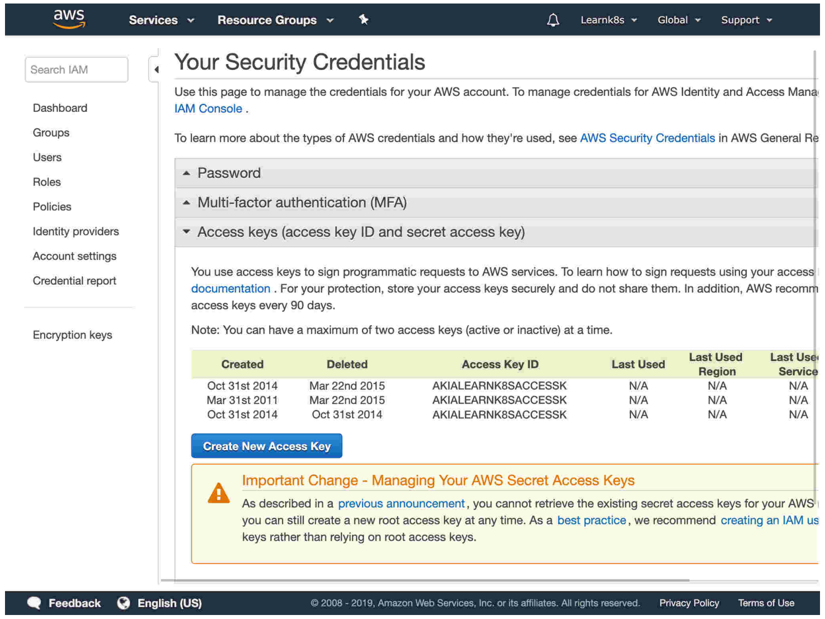
X.509 certificate

Account identifiers

© 2008 - 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

7/13

Click on Access Keys.



Your Security Credentials

Use this page to manage the credentials for your AWS account. To manage credentials for AWS Identity and Access Mana
IAM Console .

To learn more about the types of AWS credentials and how they're used, see AWS Security Credentials in AWS General Re

Password

Multi-factor authentication (MFA)

Access keys (access key ID and secret access key)

You use access keys to sign programmatic requests to AWS services. To learn how to sign requests using your access
documentation . For your protection, store your access keys securely and do not share them. In addition, AWS recomm
access keys every 90 days.

Note: You can have a maximum of two access keys (active or inactive) at a time.

Created	Deleted	Access Key ID	Last Used	Last Used Region	Last Use Service
Oct 31st 2014	Mar 22nd 2015	AKIALEARNK8SACCESSK	N/A	N/A	N/A
Mar 31st 2011	Mar 22nd 2015	AKIALEARNK8SACCESSK	N/A	N/A	N/A
Oct 31st 2014	Oct 31st 2014	AKIALEARNK8SACCESSK	N/A	N/A	N/A

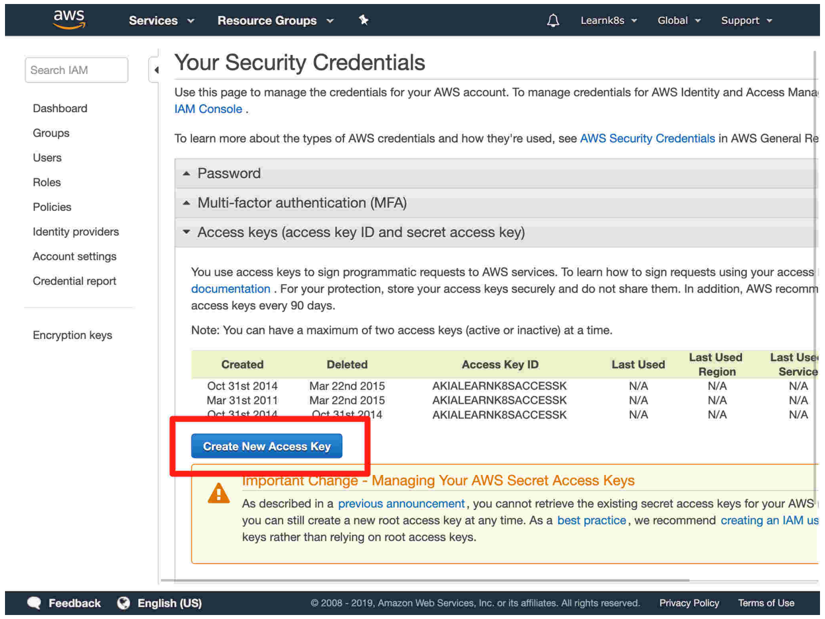
Important Change - Managing Your AWS Secret Access Keys

As described in a previous announcement, you cannot retrieve the existing secret access keys for your AWS
you can still create a new root access key at any time. As a best practice, we recommend creating an IAM us
keys rather than relying on root access keys.

© 2008 - 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

8/13

The accordion unfolds the list of active keys (if any) and a button to create a new access key.



Your Security Credentials

Use this page to manage the credentials for your AWS account. To manage credentials for AWS Identity and Access Mana
IAM Console .

To learn more about the types of AWS credentials and how they're used, see AWS Security Credentials in AWS General Re

Password

Multi-factor authentication (MFA)

Access keys (access key ID and secret access key)

You use access keys to sign programmatic requests to AWS services. To learn how to sign requests using your access
documentation . For your protection, store your access keys securely and do not share them. In addition, AWS recomm
access keys every 90 days.

Note: You can have a maximum of two access keys (active or inactive) at a time.

Created	Deleted	Access Key ID	Last Used	Last Used Region	Last Use Service
Oct 31st 2014	Mar 22nd 2015	AKIALEARNK8SACCESSK	N/A	N/A	N/A
Mar 31st 2011	Mar 22nd 2015	AKIALEARNK8SACCESSK	N/A	N/A	N/A
Oct 31st 2014	Oct 31st 2014	AKIALEARNK8SACCESSK	N/A	N/A	N/A

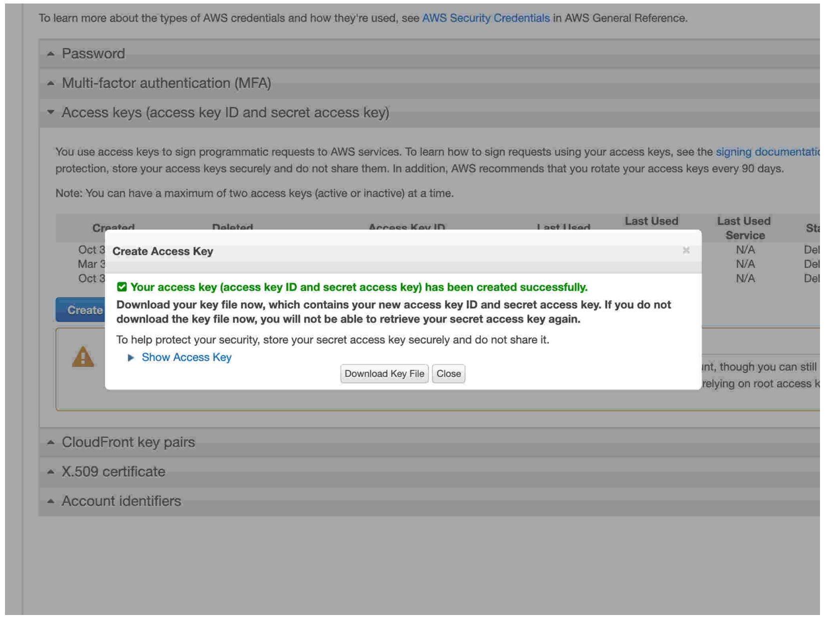
Important Change - Managing Your AWS Secret Access Keys

As described in a previous announcement, you cannot retrieve the existing secret access keys for your AWS
you can still create a new root access key at any time. As a best practice, we recommend creating an IAM us
keys rather than relying on root access keys.

© 2008 - 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

9/13

Click on "Create New Access Key".


To learn more about the types of AWS credentials and how they're used, see AWS Security Credentials in AWS General Reference.

Password

Multi-factor authentication (MFA)

Access keys (access key ID and secret access key)

You use access keys to sign programmatic requests to AWS services. To learn how to sign requests using your access keys, see the signing documentatic
protection, store your access keys securely and do not share them. In addition, AWS recommends that you rotate your access keys every 90 days.

Note: You can have a maximum of two access keys (active or inactive) at a time.

Create Access Key

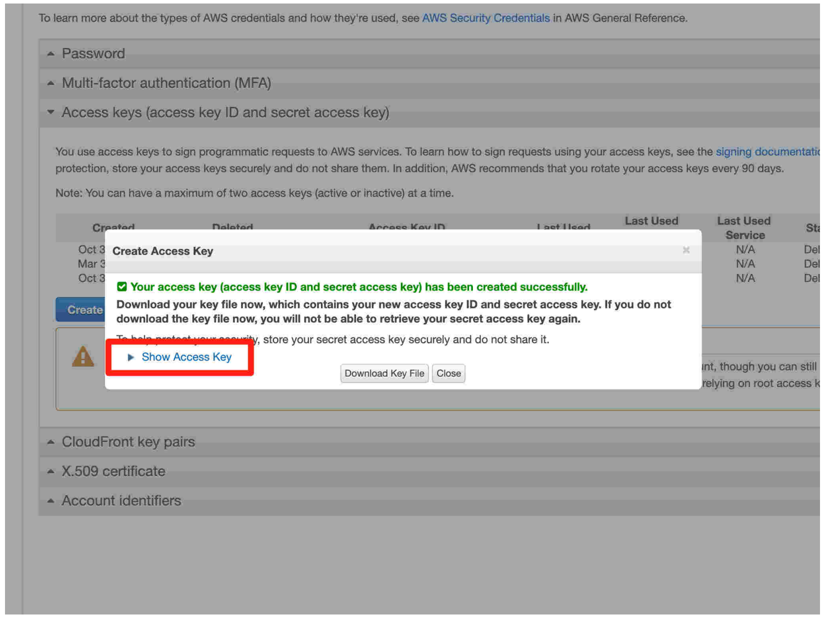
Your access key (access key ID and secret access key) has been created successfully.

Download your key file now, which contains your new access key ID and secret access key. If you do not download the key file now, you will not be able to retrieve your secret access key again.

To help protect your security, store your secret access key securely and do not share it.

10/13

A modal window appears suggesting that the key was created successfully.



To learn more about the types of AWS credentials and how they're used, see AWS Security Credentials in AWS General Reference.

Password

Multi-factor authentication (MFA)

Access keys (access key ID and secret access key)

You use access keys to sign programmatic requests to AWS services. To learn how to sign requests using your access keys, see the signing documentati
protection, store your access keys securely and do not share them. In addition, AWS recommends that you rotate your access keys every 90 days.

Note: You can have a maximum of two access keys (active or inactive) at a time.

Create Access Key

Your access key (access key ID and secret access key) has been created successfully.

Download your key file now, which contains your new access key ID and secret access key. If you do not download the key file now, you will not be able to retrieve your secret access key again.

To help protect your security, store your secret access key securely and do not share it.

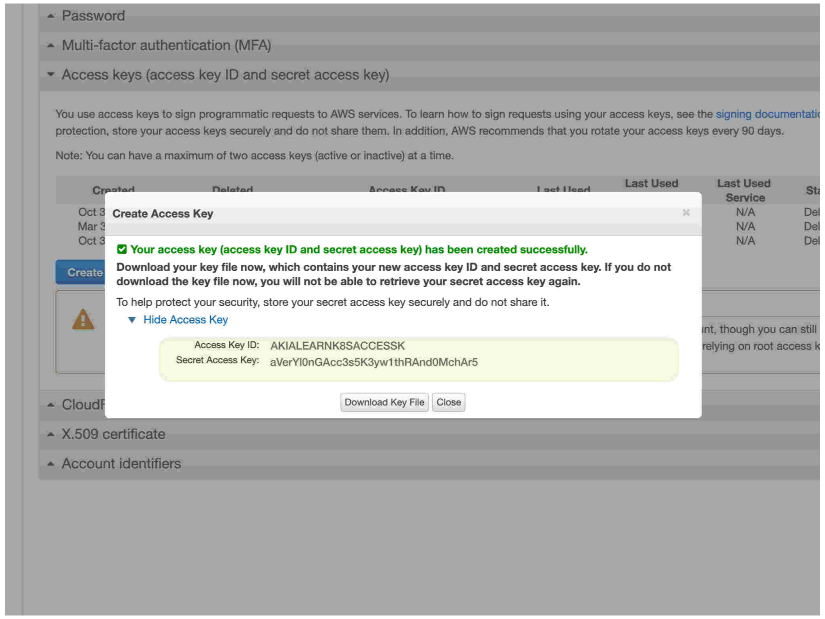
CloudFront key pairs

X.509 certificate

Account identifiers

11/13

Click on "Show Access Key" to reveal the access key.



Password

Multi-factor authentication (MFA)

Access keys (access key ID and secret access key)

You use access keys to sign programmatic requests to AWS services. To learn how to sign requests using your access keys, see the signing documentati
protection, store your access keys securely and do not share them. In addition, AWS recommends that you rotate your access keys every 90 days.

Note: You can have a maximum of two access keys (active or inactive) at a time.

Create Access Key

Your access key (access key ID and secret access key) has been created successfully.

Download your key file now, which contains your new access key ID and secret access key. If you do not download the key file now, you will not be able to retrieve your secret access key again.

To help protect your security, store your secret access key securely and do not share it.

Hide Access Key

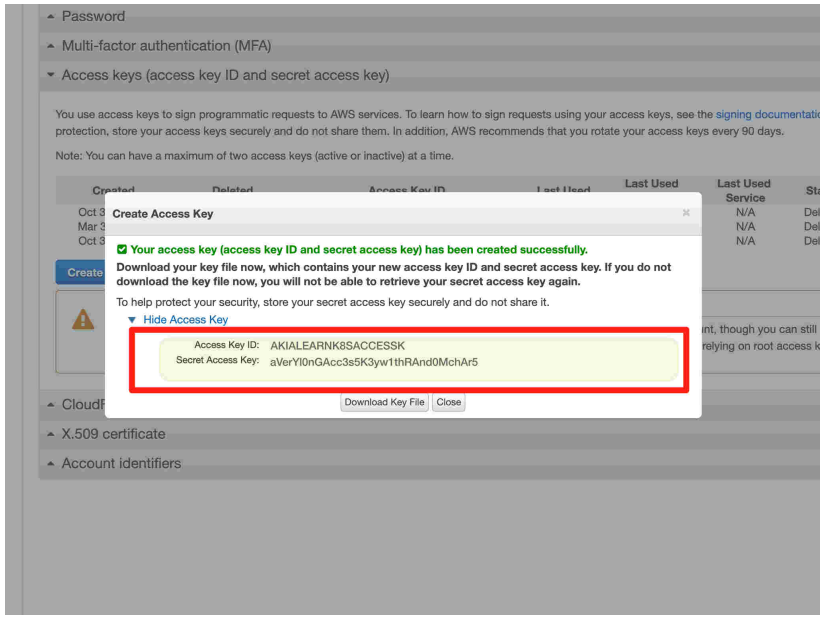
Access Key ID: AKIALEARNK8SACCESSK
Secret Access Key: aVerYI0nGAcc3s5K3yw1thRAnd0MchAr5

X.509 certificate

Account identifiers

12/13

You should see your access and secret key.



You use access keys to sign programmatic requests to AWS services. To learn how to sign requests using your access keys, see the signing documentatio
protection, store your access keys securely and do not share them. In addition, AWS recommends that you rotate your access keys every 90 days.

Note: You can have a maximum of two access keys (active or inactive) at a time.

Created	Deleted	Access Key ID	Last Used	Last Used	Last Used Service	Sta
Oct 3					N/A	Del
Mar 3					N/A	Del
Oct 3					N/A	Del

Create Access Key

Your access key (access key ID and secret access key) has been created successfully.

Download your key file now, which contains your new access key ID and secret access key. If you do not download the key file now, you will not be able to retrieve your secret access key again.

To help protect your security, store your secret access key securely and do not share it.

Access Key ID: AKIALEARNK8SACCESSK
Secret Access Key: aVerYl0nGAcc3s5K3yw1thRAnd0MchAr5

unt, though you can still
relying on root access k

13/13

Please make a note of your keys as you will need those values in the next step.

You should save the access key ID and secret access key in a file named `~/.aws/credentials` as follows:

```
$HOME/.aws/credentials

[default]
aws_access_key_id=[access-key-id]
aws_secret_access_key=[secret-access-key]
```

That's it! You're an AWS user now.

2. Creating a Kubernetes cluster on AWS

You will use Amazon Elastic Kubernetes Service (EKS) for creating a Kubernetes cluster on AWS. Amazon EKS is the managed Kubernetes service of AWS — it is comparable to Azure Kubernetes Service (AKS) or Google Kubernetes Engine (GKE).

> **Caution:** AWS isn't free, and the resources that you will create in this section will produce **very reasonable charges** on your credit card. In practice, the total costs will be around **USD 0.40 per hour**. In other words, using the cluster for 5 hours will set you back around 2 dollars.

You will create the cluster with a tool called eksctl — a third-party command-line tool that allows creating an EKS cluster with a single command. You can install eksctl according to the instructions in the official project page. *With eksctl installed, it's time to create an Amazon EKS cluster.* Run the following eksctl command:

```bash
$ eksctl create cluster --region=eu-west-2 --name=knote
```

The command creates an Amazon EKS Kubernetes cluster with the following properties:

- Two worker nodes (this is the default)
- The worker nodes are `m5.large` Amazon EC2 instances (this is the default)
- The cluster is created in the eu-west-2 region (London)
- The name of the cluster is `knote`

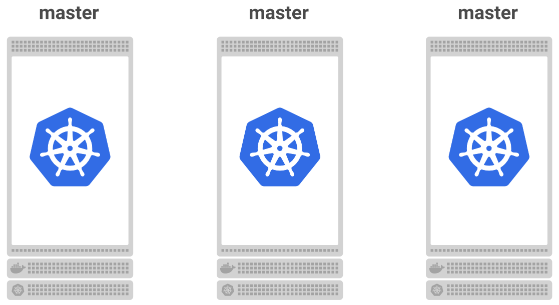
> You can use any other AWS region where Amazon EKS is available. Just avoid the us-east-1 region, because issues have been observed in this region.

Please be patient! Creating an EKS cluster usually takes around 15 minutes. **While you wait for the cluster being created, you have some time to think about Amazon EKS.** Amazon EKS is a managed Kubernetes service, in the sense that AWS runs the Kubernetes control plane for you. That means, AWS runs the master nodes, and you run the worker nodes. AWS runs three master nodes in three availability zones in your selected region.

Amazon EKS

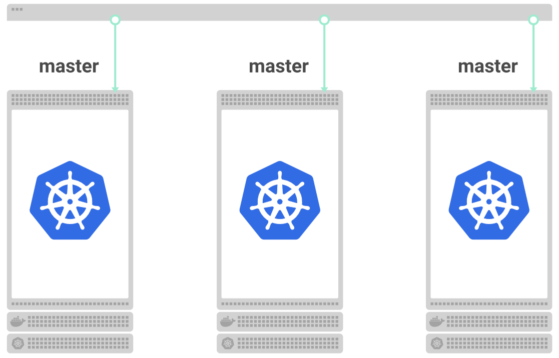
1/5

Amazon Elastic Kubernetes Service (EKS) is the managed Kubernetes offering of AWS. It allows you to create a resilient Kubernetes cluster running on the AWS infrastructure.



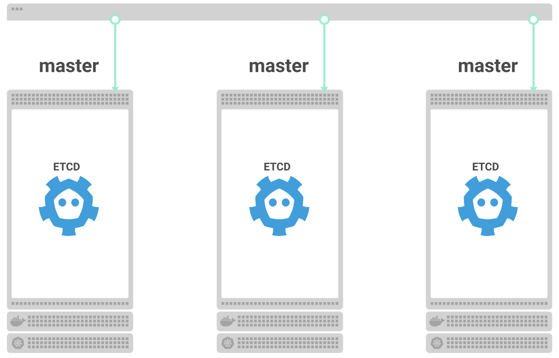
2/5

AWS creates three master nodes. The three master nodes are deployed in three different availability zones.



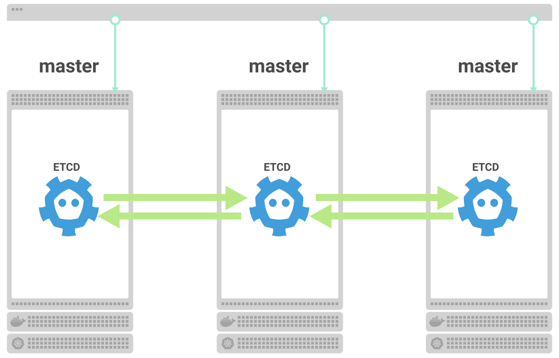
3/5

An Application Load Balancer (ALB) distributes the traffic to the three API servers on the master nodes.



4/5

There are also three etcd instances. They sync between themselves. Even if one availability zone is lost, the cluster can still operate correctly.



5/5

There are also three etcd instances. They sync between themselves. Even if one availability zone is lost, the cluster can still operate correctly.

You have no access to the master nodes. But you have full control over the worker nodes. The worker nodes are ordinary Amazon EC2 instances in your AWS account. *Once the eksctl command completes Amazon EKS cluster should be ready!* You can list the two worker nodes of your cluster with:

bash

```
$ kubectl get nodes
NAME                                           STATUS    ROLES     AGE    VERSION
ip-192-168-25-57.eu-west-2.compute.internal    Ready     <none>    23m    v1.12.7
ip-192-168-68-152.eu-west-2.compute.internal   Ready     <none>    23m    v1.12.7

$ _
```

> Note that you can't list or inspect the master nodes in any way with Amazon EKS. AWS fully manages the master nodes, and you don't need to be concerned about them.

Since the worker nodes are regular Amazon EC2 instances in your AWS account, you can inspect them in the AWS EC2 Console. You can also inspect the Amazon EKS resource itself in your AWS account in the AWS EKS Console. As you can see, your Amazon EKS cluster has further related resources — they handle all the aspects that are required for a production-grade cluster such as networking, access control, security, and logging. Those resources are created by eksctl.

> eksctl creates a CloudFormation stack with all resources belonging to your Amazon EKS cluster.

You have a production-grade Kubernetes cluster on AWS now. *It's time to deploy the application.*

3. Deploying the app

Now comes the moment of truth. *Does the app work on Amazon EKS without any changes?* It's time to find out. First of all, make sure that you still have the three YAML files in the `kube` directory:

bash

```
$ tree .
kube/
├── knote.yaml
├── minio.yaml
└── mongo.yaml

$ _
```

> Note that these are precisely the same YAML files that you deployed to the Minikube cluster.

Next, submit your configuration to the new Amazon EKS cluster:

```bash
$ kubectl apply -f kube
```

Watch the Pods being created:

```bash
$ kubectl get pods --watch
```

The app seems to be running now. To access the app, you need the public address of the `knote` Service. You can get it with this command:

```bash
$ kubectl get service knote
NAME  TYPE         CLUSTER-IP    EXTERNAL-IP                                PORT(S)
knote LoadBalancer 10.100.188.43 <xxx.eu-west-2.elb.amazonaws.com>  80:31373/TCP

$ _
```

The `EXTERNAL-IP` column should contain a fully-qualified domain name. *The address allows you to access the app from anywhere.* Open the domain name in your web browser.

> Note that it may take a couple of minutes until AWS DNS resolution is set up. If you get an `ERR_NAME_NOT_RESOLVED` error, try again in a few minutes.

You should see your app! Verify that the app works as expected by creating some notes with pictures. *Everything should work correctly.* So, your hopes came true — you can deploy the same configuration to Amazon EKS! *And everything should work the same if you want to deploy it to Azure Kubernetes Service or Google Kubernetes Engine. But what about scaling?* Go ahead and scale your Knote container to 10 replicas:

bash

```bash
$ kubectl scale --replicas=10 deployment knote
```

And check if everything still works by creating some more notes with pictures and reloading the page a couple of times. *Everything should indeed work as it did before.* **You just discovered the beauty of Kubernetes: you can define an application once and run it on any Kubernetes cluster.**

4. Taking into account resource limits

There is something else that you should know about scaling Pods. The number of Pods running in a given cluster is not unlimited. **In particular, there are limits on the maximum number of Pods that can run on a node.** On Amazon EKS, these limits depend on the EC2 instance type that you select. **Larger instances can host more Pods than smaller instances.** The details are described in the `eni-max-pods.txt` document from AWS. This document defines the maximum number of Pods that can be run on each Amazon EC2 instance type. The `m5.large` instance type that you are using for your worker nodes can host up to 29 Pods. **You have two worker nodes in your cluster — that means you can run up to 58 Pods in your cluster.** *So, what happens when you scale your Pods to 100 replicas?* The replicas that exceed the limit of 58 Pods should be stuck in the *Pending* state and never run. Because no node can run them. *Let's test this.* First, check how many Pods are running in your cluster right now:

```bash
$ kubectl get pods --all-namespaces
```

You should see 9 Pods — the 3 Pods of your application and six system Pods.

> Kubernetes runs some system Pods on your worker nodes in the `kube-system` namespace. These Pods count against the limit too. The `--all-namespaces` flag outputs the Pods from all namespaces of the cluster.

So, how many replicas of the Knote Pod can you run in the cluster? 8 Pods are *not* Knote Pods, and 58 is the maximum number of Pods that can run in the cluster. Hence, there can be up to 50 replicas of the Knote Pod. Let's exceed this limit on purpose to observe what happens. Scale your Knote app to 60 replicas:

bash

```
$ kubectl scale --replicas=60 deployment/knote
```

Wait for the Pods to start up. *If your calculations were correct, only 50 of these 60 replicas should be running in the cluster — the remaining ten should be stuck in the Pending state.* Let's start by counting the Knote Pods that are *Running*:

bash

```
$ kubectl get pods \
  -l app=knote \
  --field-selector='status.phase=Running' \
  --no-headers | wc -l
```

The command should output 50. And now the Knote Pods that are *Pending*:

```bash
$ kubectl get pods \
  -l app=knote \
  --field-selector='status.phase=Pending' \
  --no-headers | wc -l
```

The command should output 10. **So your calculations were correct.** 50 of the 60 replicas are *Running* — the remaining 10 are *Pending*. The ten pending replicas can't run because the maximum number of 58 running Pods in the cluster has been reached. You can verify that there are indeed 58 running Pods in the cluster with:

```bash
$ kubectl get pods \
  --all-namespaces \
  --field-selector='status.phase=Running' \
  --no-headers | wc -l
```

The command should output 58. To fix the issue, you can scale down the number of replicas to 50:

```bash
$ kubectl scale --replicas=50 deployment/knote
```

After a while, you should see no *Pending* Pods anymore — all the replicas should be running. **But what if you want a higher number of replicas at all costs?** You could add more nodes to your cluster. Nothing prevents you from creating a cluster with 10, 20, 30, or 1000 worker nodes.

> Kubernetes is tested to run reliably with up to several thousands of nodes and tens of thousands of Pods.

Or you could use larger EC2 instances that can host more Pods (see AWS limits). Nothing prevents you from using `m5.24xlarge` EC2 instances for your worker nodes and have 737 Pods on each of them. **Whatever your scaling requirements are, Kubernetes can accommodate them — you just have to design your cluster accordingly.**

5. Cleaning up

Before you leave, you should remember something important: **Running an Amazon EKS cluster is not free.** Running the cluster alone (without the worker nodes) costs USD 0.20 per hour.

> The price stays the same, no matter how many Pods you run on the cluster.

And running the two `m5.large` worker node costs USD 0.096 per hour for each one. **The total amount is around USD 0.40 per hour for running your cluster.** While it might not seem a lot, if you forget to delete your cluster, it could add up quickly. **That's why you should always delete your Amazon EKS cluster when you don't need it anymore.** You can do this conveniently with eksctl:

```bash
$ eksctl delete cluster --region=eu-west-2 --name=knote
```

The command deletes all AWS resources that belong to your Amazon EKS cluster. After the command completes, you can double-check that the AWS resources have been deleted in the AWS Console:

- Check the AWS EKS Console and verify that the Amazon EKS cluster resource has been removed (or is being deleted)
- Check the AWS EC2 Console and confirm that the EC2 instances that were your worker nodes have been removed (or are being deleted)

> When you access the AWS Console, always double-check that you selected the correct region in the top-right corner (e.g. London). If you are in the wrong region, you can't see the resources from another region.

If you want to work with your cluster again at a later time, you can repeat the same steps:

1. Create a new cluster with eksctl
2. Deploy your app
3. Delete the cluster with eksctl

6. Conclusion

You've reached the end of this crash course on Kubernetes.
Let's recap what you achieved:

- You wrote a note-taking application in Node.js
- You packaged the app as a Docker image
- You deployed the containerised application to a local Minikube cluster
- You refactored your application to make it stateless and scalable
- You deployed the improved application to a production-grade Kubernetes cluster on AWS

In the course of this, you learnt about many topics, including:

- Taking Kubernetes considerations into account as early as coding an application
- How to build Docker images and upload them to Docker Hub
- How to run a containerised application locally in a Docker network
- How to create a local Kubernetes cluster with Minikube
- The declarative resource-based interface of Kubernetes

- Where you can find information about Kubernetes resource objects
- How to write application deployment configurations for Kubernetes
- How statefulness is related to scalability
- How to scale an application on Kubernetes
- How to create a production-grade Kubernetes cluster on AWS using Amazon EKS
- Taking into account resource limits on production Kubernetes clusters

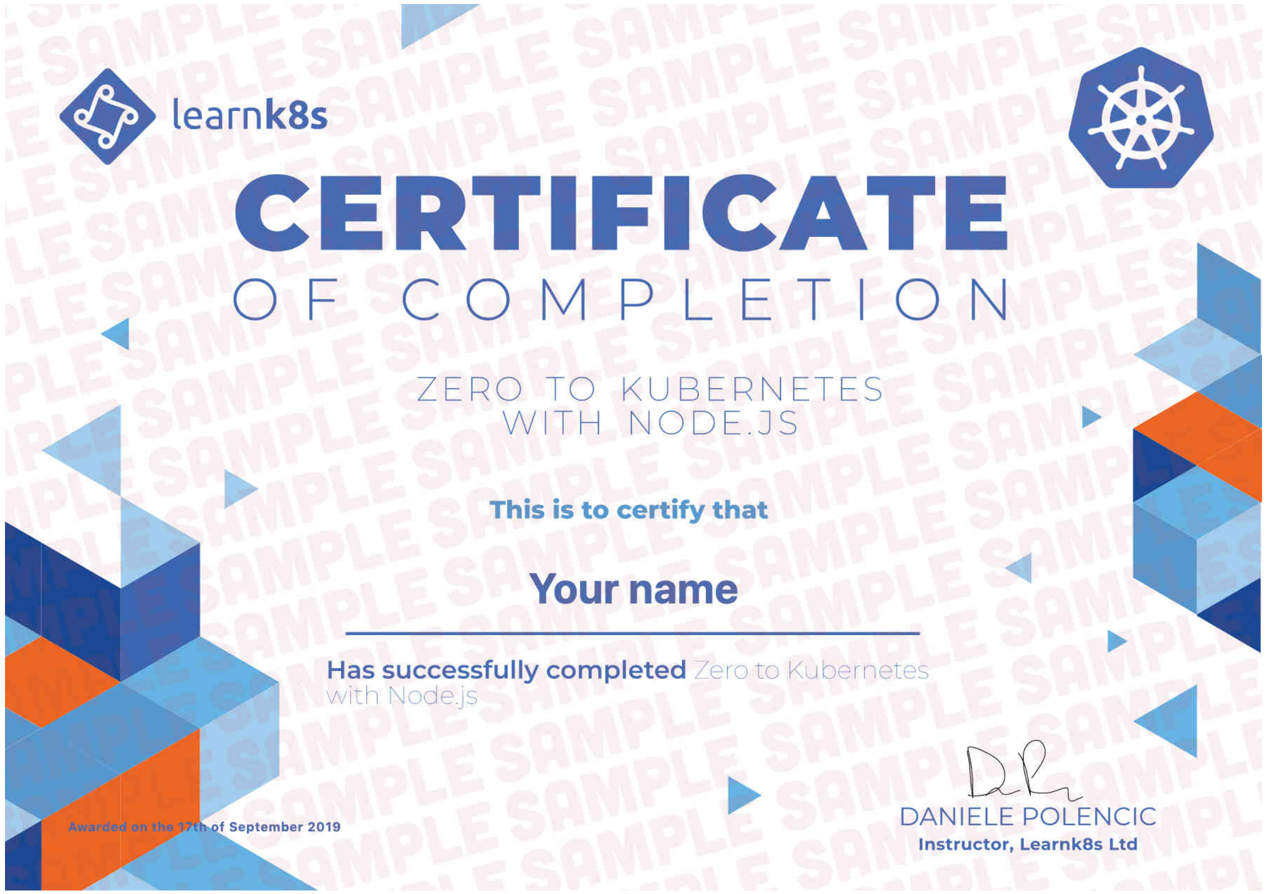
7. Where to go from here?

If you want to keep experimenting, you can create a Kubernetes cluster with a different provider, such as Google Kubernetes Engine (GKE) or Azure Kubernetes Service (AKS) and deploy your application there. Since a Kubernetes deployment configuration, once defined, can be deployed to every Kubernetes cluster, you should have no troubles with that. During this course, you covered several topics, but you didn't have the time to study them in-depth. An obvious place to learn more about Kubernetes is in the official documentation where you can find more about fundamental concepts, everyday tasks, or even learn how to install Kubernetes from scratch. Finally, the Learnk8s Academy offers a broad range of Kubernetes courses, similar to the one you completed. These courses treat various topics in much more depth than this introductory course could provide. The Learnk8s Academy courses can also prepare you for the Certified Kubernetes Administrator (CKA) and Certified Kubernetes Application Developer (CKAD) exams. **Happy navigating with Kubernetes!**

Chapter 6

Certificate of completion

Congratulations on completing the Zero to Kubernetes course! *Wouldn't be great if you could be recognised for all the effort?* So here's the deal, there's a certificate ready with your name.



learnk8s

CERTIFICATE

OF COMPLETION

ZERO TO KUBERNETES
WITH NODE.JS

This is to certify that

Your name

Has successfully completed Zero to Kubernetes with Node.js

Awarded on the 17th of September 2019

[illegible]

DANIELE POLENCIC
Instructor, Learnk8s Ltd

Zero to Kubernetes in Node.js certificate

But there's a catch. You can download the certificate only if you solve this challenge:

- create a Pod with the following image: `learnk8s/02k8s-js-cert:1.0.0`
- the Pod should have an environment variable `NAME` with your name. It will appear on the certificate.

- find the port exposed by the container
- visit the Pod and download the certificate with your name on it

Enjoy!