Mauricio Salatino

Developing and deploying Spring Boot microservices on Kubernetes

PUBLISHED IN MARCH 2020

TL;DR: In this guide you will launch a local Kubernetes cluster, develop an app using the Spring Boot framework and deploy it as a container in Kubernetes.

Learning how to design and architect applications that leverage Kubernetes is the most valuable skill that you could learn to be successful in deploying and scaling in the cloud.

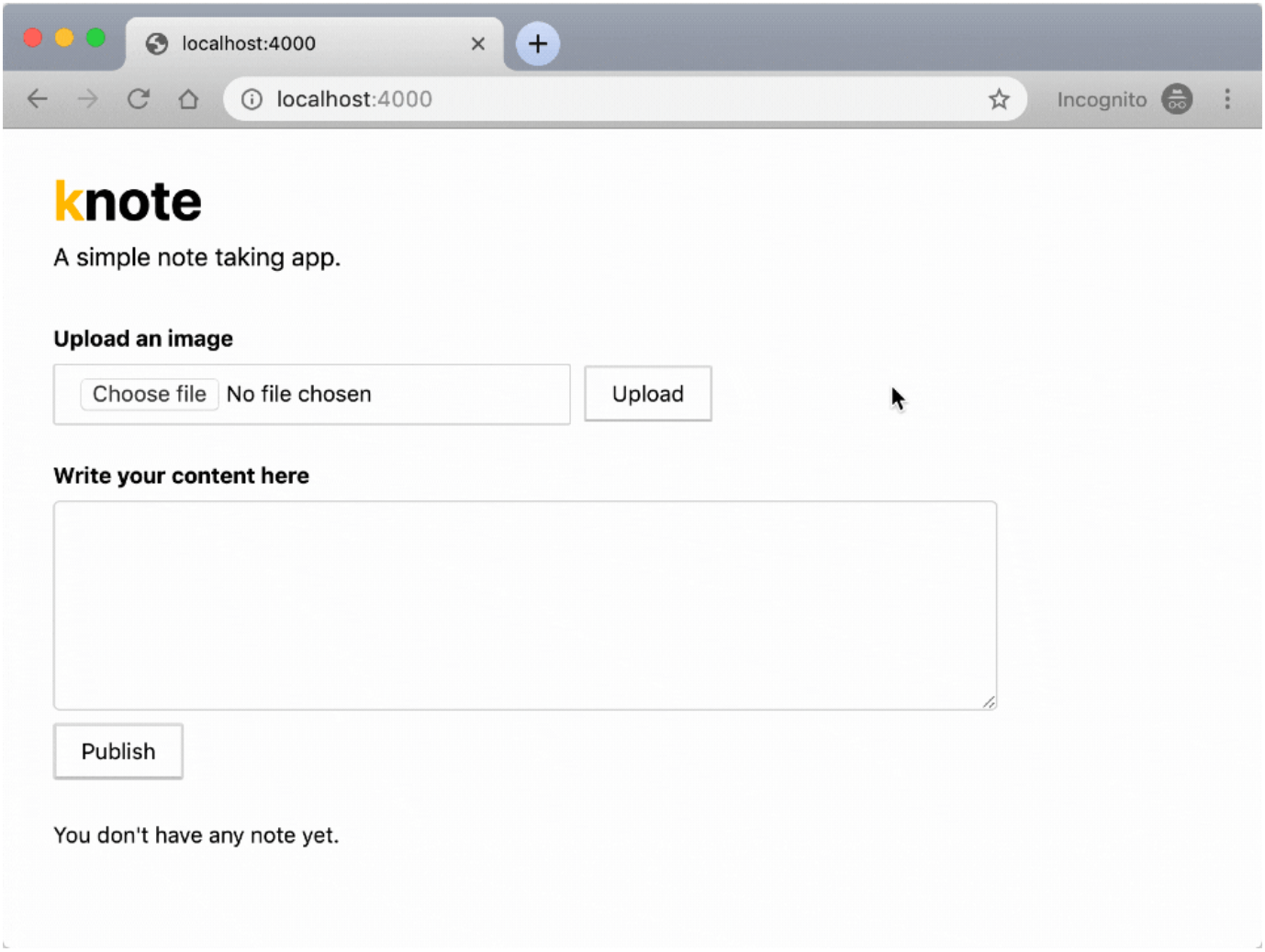
In this guide, you will develop a small application for note-taking similar to Evernote and Google Keep.

The app lets you:

1. record notes and
2. attach images to your notes

The notes are persisted, which gives us a good reason to explore some basic storage options.

Here is how the app looks like:



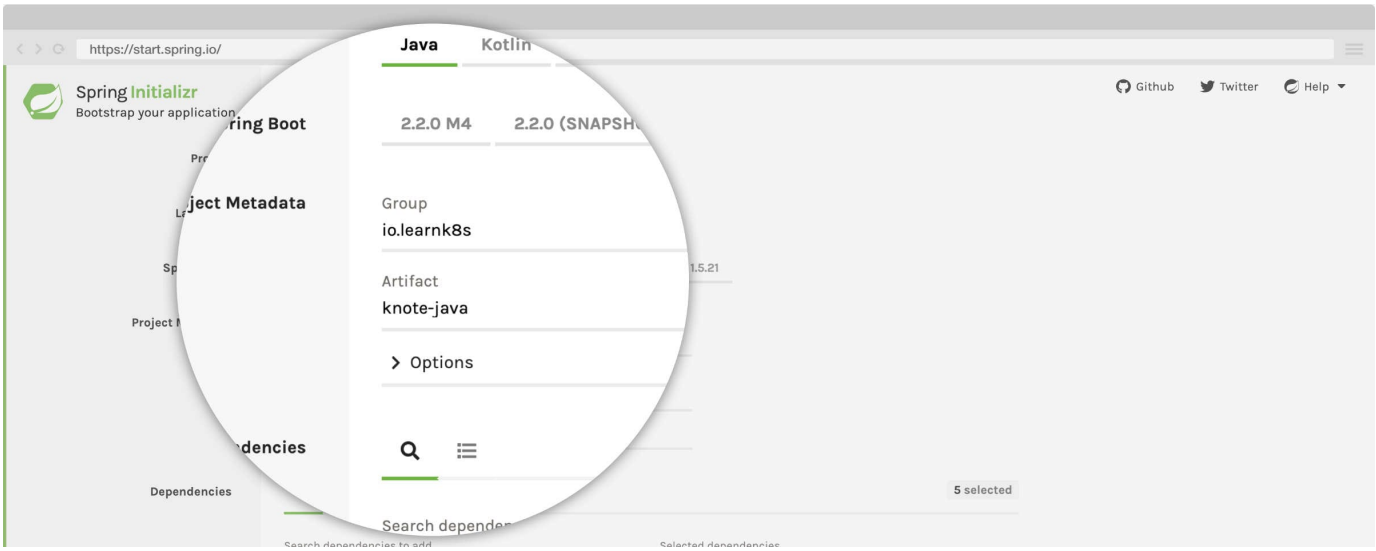
Bootstrapping the app

You will build the app with Spring Boot.

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

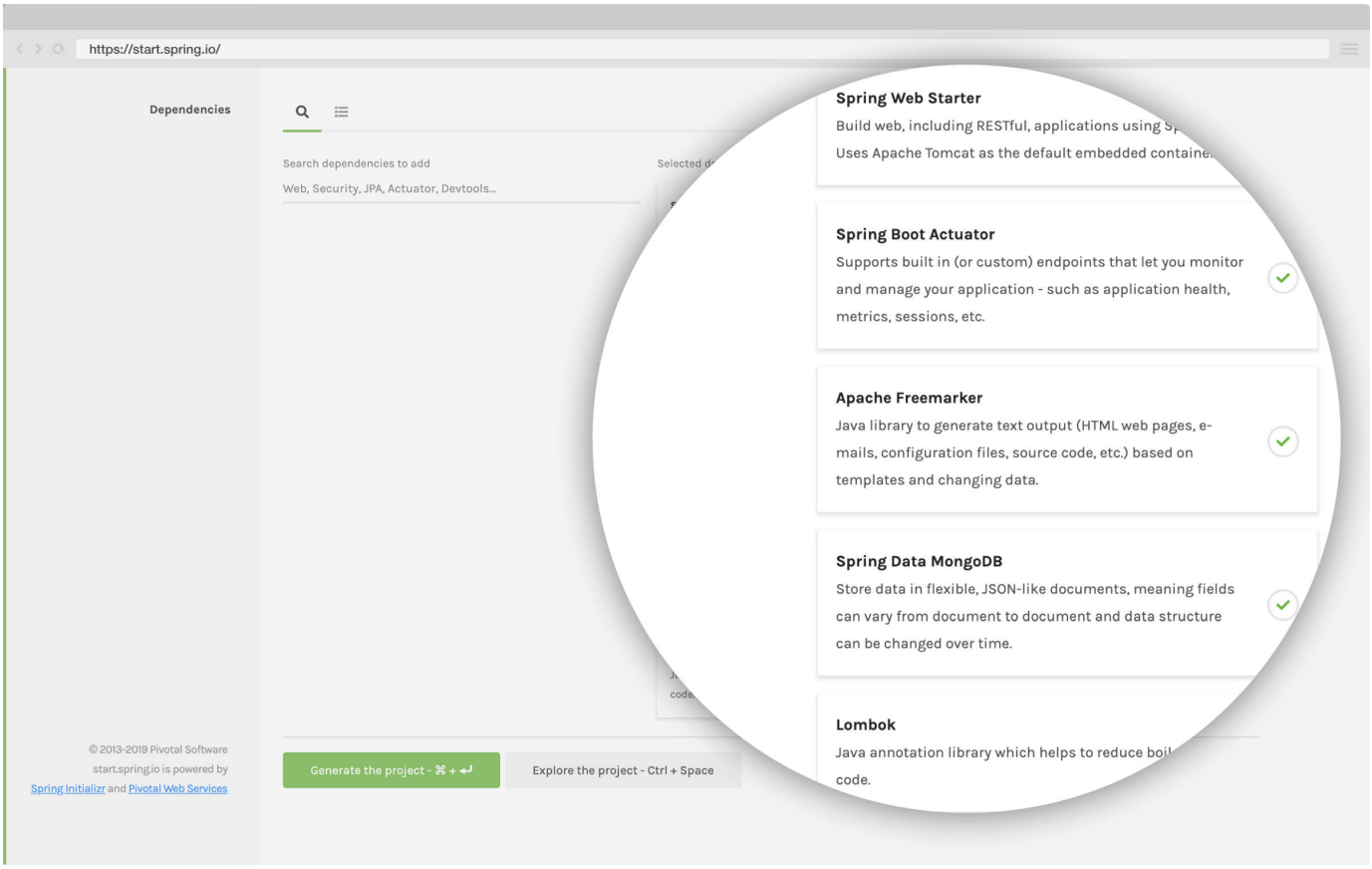
Let's get started.

First, you need to go to https://start.spring.io to generate the skeleton of the project:



You should enter the **Group** and **Name** for your application:

- GroupId: learnk8s.io
- Name: knote-java



Next, go to the dependencies section and choose:

- **Web** -> Spring Web Starter: basic web stack support in Spring Boot
- **Actuator** -> Spring Boot Actuator: provide health endpoints for our application
- **FreeMarker** -> Apache FreeMarker: templating engine for HTMLs
- **MongoDB** -> Spring Data MongoDB: driver and implementation for Spring Data interfaces to work with MongoDB
- **Lombok** -> Lombok: library to avoid a lot of boilerplate code

Then click *Generate the project* to download a zip file containing the skeleton of your app.

Unzip the file and start a terminal session in that directory.

You will do the front-end stuff first.

Within `knote-java` application, there are two files in charge of rendering the Front End:

- Tachyons CSS that needs to be placed inside `src/main/resources/static/`
- Freemarker Template for our index view that needs to be placed inside `src/main/resources/templates/`

> You can find the Freemarker template in this repository.

Apache FreeMarker™ is a template engine: a Java library to generate text output (HTML web pages, e-mails, configuration files, source code, etc.) based on templates and changing data.

With the front-end stuff out of the way, let's turn to code.

You can open our application in our favourite IDE and import it as a Maven Project.

Now, open `src/main/java/io/learnk8s/knote-java/KnoteJavaApplication.java`:

KnoteJavaApplication.java

```
package io.learnk8s.knote;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class KnoteJavaApplication {

    public static void main(String[] args) {
        SpringApplication.run(KnoteJavaApplication.class, args);
    }

}
```

This is not much more than a standard Spring Boot Application.

It doesn't yet do anything useful.

But you will change this now by connecting it to a database.

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.

Because you had included the Spring Data MongoDB support, there is not much that you need to do to connect to the database.

You should open the `src/main/resources/application.properties` file and enter the URL for the database.

application.properties

```
spring.data.mongodb.uri=mongodb://localhost:27017/dev
```

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.

Luckily for you, Spring Data automatically reconnects to the database until the connection is successful.

The next step is to use the database.

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.

First, you should create a `Note` class that holds the note's details.

The same note is also stored in the "notes" MongoDB collection.

KnoteJavaApplication

```
@SpringBootApplication
public class KnoteJavaApplication {
    public static void main(String[] args) {
        SpringApplication.run(KnoteJavaApplication.class, args);
    }
}

@Document(collection = "notes")
@Setter
@Getter
@NoArgsConstructor
@AllArgsConstructor
class Note {
    @Id
    private String id;
    private String description;

    @Override
    public String toString() {
        return description;
```

```
    }
}
```

Next, you should leverage Spring Template to create a new repository to store the notes.

KnoteJavaApplication.java

```
@Document(collection = "notes")
@Setter
@Getter
@NoArgsConstructor
@AllArgsConstructor
class Note {
    @Id
    private String id;
    private String description;

    @Override
    public String toString() {
        return description;
    }
}

interface NotesRepository extends MongoRepository<Note, String> {

}
```

As you can see, you define an interface and Spring Data MongoDB generates the implementation.

Also, notice how the notes are persisted in the database:

- The type of the Note is Note
- The id of the notes is of type String

You should notice the two types in the interface signature `MongoRepository<Note, String>` .

You can access the repository by autowiring it.

You should create a new class with the `@Controller` annotation to select the views in your application.

KnoteJavaApplication.java

```
...

@Controller
class KNoteController {

    @Autowired
    private NotesRepository notesRepository;

}
```

When a user accesses the `/` route, they should see all notes.

You should add a `@GetMapping` endpoint to return the FreeMarker template `index.ftl` .

> Please notice how we dropped the `.ftl` extension from the filename to refer to it.

KnoteJavaApplication.java

```
@Controller
class KNoteController {

    @Autowired
    private NotesRepository notesRepository;

    @GetMapping("/")
    public String index(Model model) {
        getAllNotes(model);
        return "index";
    }
```

```
    private void getAllNotes(Model model) {
        List<Note> notes = notesRepository.findAll();
        Collections.reverse(notes);
        model.addAttribute("notes", notes);
    }
    ...
}
```

The `getAllNotes(Model model)` method is in charge of

1. retrieving all the notes stored in MongoDB
2. reversing the order of the notes (to show the last one first) and
3. updating the model consumed by the view

Next, let's address the creation of new notes.

You should add a method to save a single note in the database:

KnoteJavaApplication.java

```
private void saveNote(String description, Model model) {
  if (description != null && !description.trim().isEmpty()) {
    notesRepository.save(new Note(null, description.trim()));
    //After publish you need to clean up the textarea
    model.addAttribute("description", "");
  }
}
```

The form for creating notes is defined in the `index.ftl` template.

Note that the form handles both the creation of notes and the uploading of pictures.

The form submits to the `/note` route, so you need to another endpoint to your `@Controller`:

KnoteJavaApplication.java

```
@PostMapping("/note")
public String saveNotes(@RequestParam("image") MultipartFile file,
                        @RequestParam String description,
                        @RequestParam(required = false) String publish,
                        @RequestParam(required = false) String upload,
                        Model model) throws IOException {

  if (publish != null && publish.equals("Publish")) {
    saveNote(description, model);
    getAllNotes(model);
    return "redirect:/";
  }
  // After save fetch all notes again
  return "index";
}
```

The above endpoint calls the `saveNote` method with the content of the text box, which causes the note to be saved in the database.

It then redirects to the main page ("index"), 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

```
$ mvn clean install spring-boot:run _
```

The app should connect to MongoDB and then listen for requests.

You can access your app on http://localhost:8080.

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.

Rendering Markdown to HTML

The Markdown notes should be rendered to HTML so that you can read them properly formatted.

You will use commonmark-java from Atlassian to parse the notes and render HTML.

But first you should add a dependency to your `pom.xml` file:

pom.xml

```
<dependencies>
  ...
  <dependency>
    <groupId>com.atlassian.commonmark</groupId>
    <artifactId>commonmark</artifactId>
    <version>0.12.1</version>
  </dependency>
  ...
</dependencies>
```

Then, change the `saveNote` method as follows (changed lines are highlighted):

KnoteJavaApplication.java

```
private void saveNote(String description, Model model) {
  if (description != null && !description.trim().isEmpty()) {
    //You need to translate markup to HTML
    Node document = parser.parse(description.trim());
    String html = renderer.render(document);
    notesRepository.save(new Note(null, html));
    //After publish you need to clean up the textarea
    model.addAttribute("description", "");
  }
}
```

You also need to add to the `@Controller` itself:

KnoteJavaApplication.java

```
@Controller
class KNoteController {

    @Autowired
    private NotesRepository notesRepository;
    private Parser parser = Parser.builder().build();
    private HtmlRenderer renderer = HtmlRenderer.builder().build();
```

The new code converts all the notes from Markdown to HTML before storing them into the database.

Kill the app with `CTRL + C` and then start it again:

bash

```
$ mvn clean install spring-boot:run
```

Access it on http://localhost:8080.

Now you can add a note with the following text:

snippet.md

```
Hello World! **Kubernetes Rocks!**
```

And you should see `Kubernetes Rocks!` in bold fonts.

All your notes should now be nicely formatted.

Let's tackle uploading files.

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 endpoint 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.

Change the endpoint for the POST `/note` inside the `@Controller` (changed lines are highlighted):

KnoteJavaApplication.java

```
@PostMapping("/note")
public String saveNotes(@RequestParam("image") MultipartFile file,
                        @RequestParam String description,
                        @RequestParam(required = false) String publish,
                        @RequestParam(required = false) String upload,
                        Model model) throws IOException {
  if (publish != null && publish.equals("Publish")) {
    saveNote(description, model);
    getAllNotes(model);
    return "redirect:/";
  }
  if (upload != null && upload.equals("Upload")) {
    if (file != null && file.getOriginalFilename() != null
          && !file.getOriginalFilename().isEmpty()) {
      uploadImage(file, description, model);
    }
    getAllNotes(model);
    return "index";
  }
  return "index";
}

private void uploadImage(MultipartFile file, String description, Model model
  File uploadsDir = new File(properties.getUploadDir());
  if (!uploadsDir.exists()) {
    uploadsDir.mkdir();
  }
  String fileId = UUID.randomUUID().toString() + "."
                    + file.getOriginalFilename().split("\\.")[1];
  file.transferTo(new File(properties.getUploadDir() + fileId));
  model.addAttribute("description", description + " ![](/uploads/" + fileId
}
```

As you can see from the `uploadImage()` method, you are using Spring Boot configuration properties to inject application configurations.

These properties can be defined in the `application.properties` file or as environmental variables.

But you should define the `@ConfigurationProperties` class to retrieve those values.

Outside of the Controller class, you should define the `KnoteProperties` class annotated with `@ConfigurationProperties(prefix = "knote")` :

KnoteJavaApplication.java

```
@ConfigurationProperties(prefix = "knote")
class KnoteProperties {
    @Value("${uploadDir:/tmp/uploads/}")
    private String uploadDir;

    public String getUploadDir() {
        return uploadDir;
    }
}
```

By default, the `uploadImage` method uses the `/tmp/uploads/` directory to store the images.

Notice that the `uploadImage` method checks if the directory exists and creates it if it doesn't.

> If you decide to change the path, make sure that the application has write access to that folder.

One last code change is required for the webserver (embedded in the spring boot application) to host files outside of the JVM classpath:

KnoteJavaApplication.java

```
...
@Configuration
@EnableConfigurationProperties(KnoteProperties.class)
class KnoteConfig implements WebMvcConfigurer {

    @Autowired
    private KnoteProperties properties;

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry
                .addResourceHandler("/uploads/**")
                .addResourceLocations("file:" + properties.getUploadDir())
                .setCachePeriod(3600)
                .resourceChain(true)
                .addResolver(new PathResourceResolver());
    }

}
...
```

The class annotated with the `@Configuration` annotation maps the path `/uploads/` to the files located inside the `file:/tmp/uploads/` directory.

The class annotated with `@EnableConfigurationProperties(KnoteProperties.class)` allows Spring Boot to read and autowire the application properties.

You can override those properties in `application.properties` file or with environment variables.

Kill the app with `CTRL + C` and then start the application again:

bash

```
$ mvn clean install spring-boot:run
```

Access it on http://localhost:8080.

Try to upload a picture — you should see a link is 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.

The next step is to containerise your app.

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 adoptopenjdk/openjdk11:jdk-11.0.2.9-slim
WORKDIR /opt
ENV PORT 8080
EXPOSE 8080
COPY target/*.jar /opt/app.jar
ENTRYPOINT exec java $JAVA_OPTS -jar app.jar
```

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 OpenJDK 11
- 'WORKDIR' sets the working directory to `/opt/` . Every subsequent instruction runs from within that folder
- 'ENV' is used to set an environment variable
- `COPY` copies the jar files from the `/target/` into the `/opt/` directory inside the container
- `ENTRYPOINT` executes `java $JAVA_OPTS -jar app.jar` inside the container

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

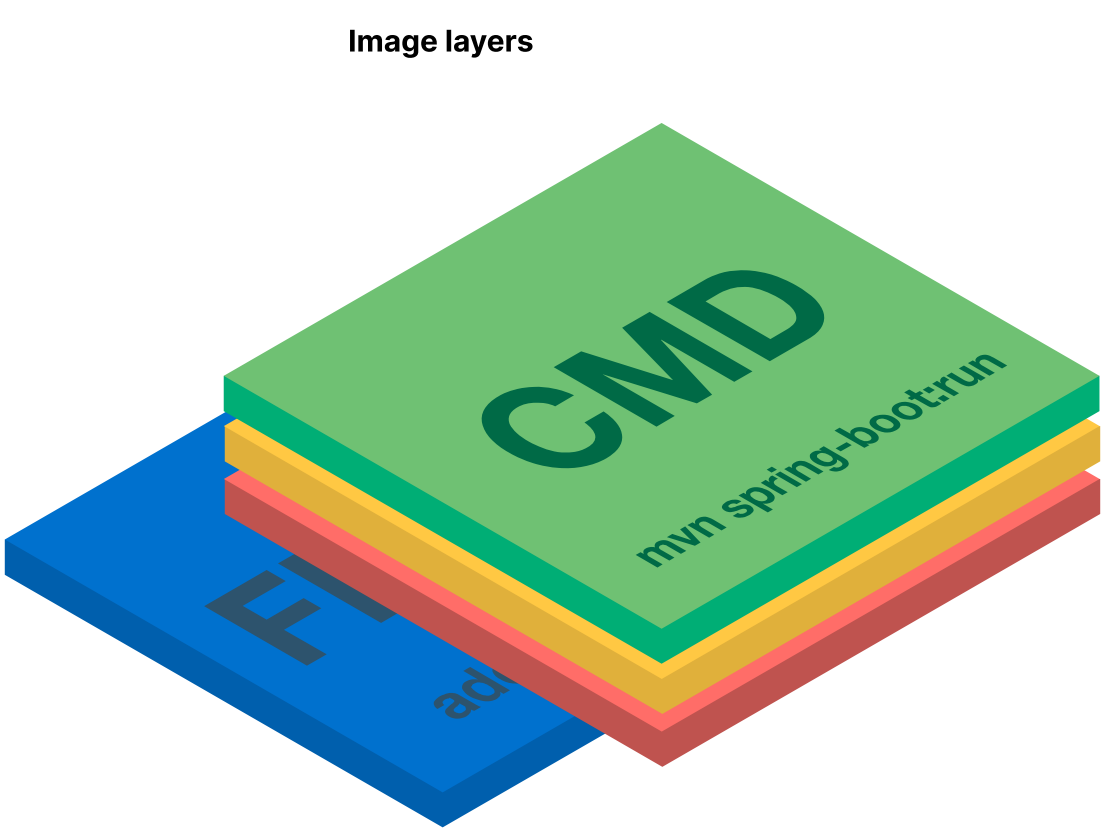
bash

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

Note the following about this command:

- `-t knote-java` defines the name ("tag") of your container — in this case, your container is just called `knote-java`
- `.` 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:

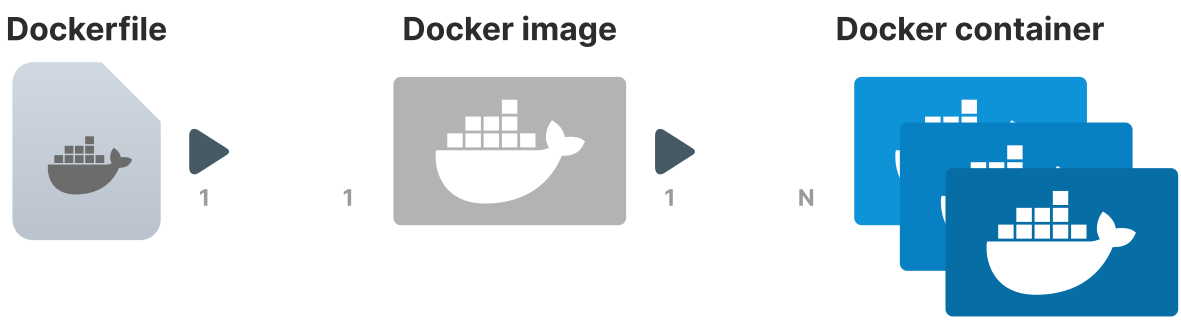


The output is a Docker image.

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:



> Don't believe that Docker images are archives? Save the image locally with `docker save knote-java > knote-java.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-java                  latest              b9dfdd2b85ca        About a
adoptopenjdk/openjdk11      jdk-11.0.2.9-slim   9a223081d1a1        2 months

$
```

You should see the `knote-java` image that you just built.

You should also see the `adoptopenjdk/openjdk11` which is the base layer of your `knote-java` 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.

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

```
$ 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-java \
  --rm \
  --network=knote \
  -p 8080:8080 \
  -e MONGO_URL=mongodb://mongo:27017/dev \
  knote-java
```

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 8080:8080` publishes port 8080 of the container to port 8080 of your local machine. That means, if you now access port 8080 on your computer, the request is forwarded to port 8080 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-java` and `mongo`.

You can display all running containers with the following command:

bash

```
$ docker ps
CONTAINER ID    IMAGE       COMMAND                  PORTS
9b908ee0798a    knote-java  "/bin/sh -c 'exec ja…"  0.0.0.0:8080->8080/tcp
1fb37b278231    mongo       "docker-entrypoint.s…"  27017/tcp

$
```

Great!

It's time to test your application!

Since you published port 8080 of your container to port 8080 of your local machine, your app is accessible on http://localhost:8080.

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

```
$ docker stop mongo knote-java
$ docker rm mongo knote-java
```

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-java <username>/knote-java:1.0.0
```

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

Now you can upload your image to Docker Hub:

bash

```
$ docker push username/knote-java:1.0.0
```

Your image is now publicly available as `<username>/knote-java: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-java: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-java \
    --rm \
    --network=knote \
    -p 8080:8080 \
    -e MONGO_URL=mongodb://mongo:27017/dev \
    learnk8s/knote-java: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-java
$ docker rm mongo knote-java
```

So far, you have written a Java application and packaged it as a Docker image so that it can be run as a container.

The next step is to run this containerised app on a container orchestrator.

Kubernetes — the container orchestrator

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.

A few key points to remember about Kubernetes. It's:

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.

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.

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.

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-java:1.0.0
          ports:
            - containerPort: 8080
          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 replace `learnk8s` with your Docker ID (username) in the container's image value. If you didn't upload your image to Docker Hub, you can use the `learnk8s/knote-java:1.0.0` image provided by Learnk8s on Docker Hub.

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

```
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-java:1.0.0
            ports:
              - containerPort: 8080
            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

```
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-java:1.0.0
        ports:
          - containerPort: 8080
        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

```
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-java:1.0.0
            ports:
              - containerPort: 8080
            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

```
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-java:1.0.0
            ports:
              - containerPort: 8080
            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-java:1.0.0`), you can change this one to point to your docker image by replacing `learnk8s` to your Docker ID.
- The port that the container listens on (8080)
- 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` previously.

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.

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:

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

```
apiVersion: v1
kind: Service
metadata:
  name: knote
spec:
  selector:
    app: knote
  ports:
    - port: 80
```

```
    targetPort: 8080
  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

```
# ... 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

```
apiVersion: v1
kind: Service
metadata:
```

```
    name: knote
  spec:
    selector:
      app: knote
    ports:
      - port: 80
        targetPort: 8080
    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

```
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

```
apiVersion: v1
kind: Service
```

```
  metadata:
    name: knote
  spec:
    selector:
      app: knote
    ports:
      - port: 80
        targetPort: 8080
    type: LoadBalancer
```

In this case, the Service listens for requests on port 80 and forwards them to port 8080 of the target Pods:

The last important part is the type of the Service:

kube/knote.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: knote
spec:
  selector:
    app: knote
  ports:
    - port: 80
      targetPort: 8080
  type: LoadBalancer
```

In this case, the type is `LoadBalancer`, which makes the exposed Pods accessible from outside the cluster.

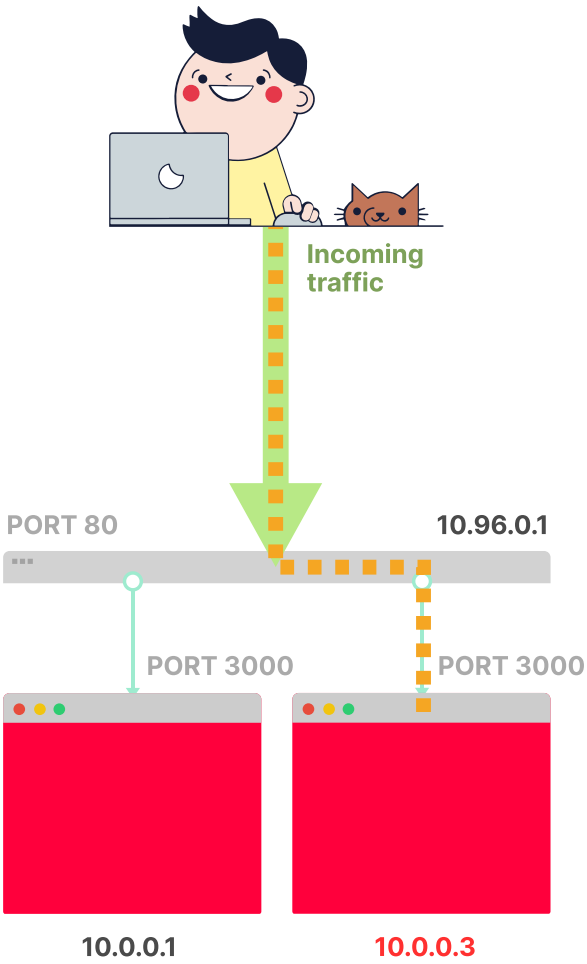
The default Service type is `ClusterIP`, which makes the exposed Pods only accessible from within the cluster.

> **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.



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.

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

```
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
metadata:
  name: mongo
spec:
  selector:
    matchLabels:
      app: mongo
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
        - name: mongo
          image: mongo
          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/dev`.

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.

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
$ minikube status
```

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

```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
$ 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 --url
```

The command should print the URL of the `knote` Service.

You can open the URL 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.

Scaling your app

Kubernetes makes it very easy to increase the number of replicas to 2 or more:

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.

Did it work?

Reaccess your app:

bash

```
$ minikube service knote --url
```

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.

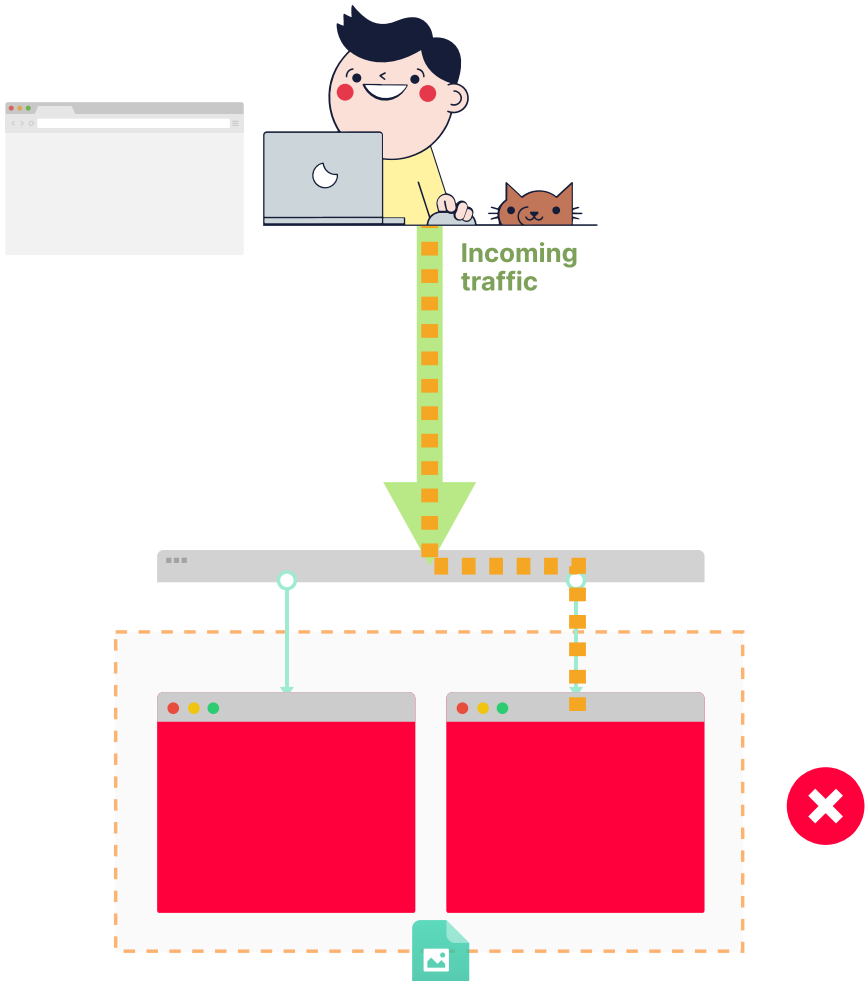
But since you have two replicas, there's an issue.

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.



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.

How do you refactor your app to make it stateless?

Making the app stateless and next steps

So far you've learnt how to:

1. develop a note taking applicaiton that stores notes in MongoDB
2. packaged it as a Docker container
3. deployed it in a local Kubernetes cluster

The next steps are:

1. refactoring the app to make it stateless
2. deploying the same app in the cloud

This guide in a excerpt of the Learnk8s Academy — the online course designed to learn Kubernetes.

That's all folks!

If you enjoyed this article, you might find the following articles interesting:

- Scaling Microservices with Message Queues, Spring Boot and Kubernetes. Learn how to use the Horizontal Pod Autoscaler to resize your fleet of applications dynamically.
- A visual guide on troubleshooting Kubernetes deployments. Troubleshooting in Kubernetes can be a daunting task if you don't know where to start. In this article you will learn how to diagnose problems in Pods, Services and Ingress.

Don't miss the next article!

Be the first to be notified when a new article or Kubernetes experiment is published.

Your email address: kubernetes@gmail.com

Subscribe

*We'll never share your email address, and you can opt-out at any time.

What is Learnk8s?

In-depth Kubernetes training that is practical and easy to understand.

⎈ Instructor-led workshops ›

Deep dive into containers and Kubernetes with the help of our instructors and become

an expert in deploying applications at scale.