Table of Contents

Lab 3. Run and Operate Containers

The objective of this lab exercise is to get you started with the basic operations for Windows containers. In this lab, you are going to learn how to:

- Get started with Docker command line client utility.
- Launch and work with containers.
- Use common container launch options.
- Define port publishing options to access web applications in a container.
- Troubleshoot running containers.
- Manage container lifecycle.

Finding Information

Open a console/terminal on the host where Docker is installed and run:

```
docker
docker version
docker system info
```

Launching and Operating Your First Container

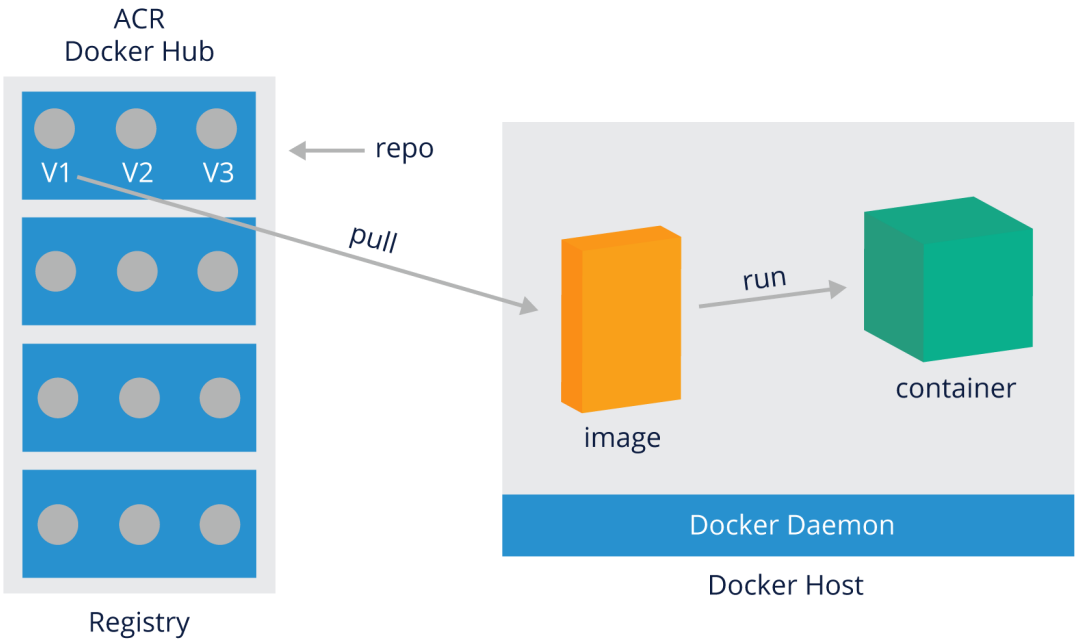
Let’s launch your first container To understand what’s happening on the Docker server side, you can also open another terminal and run the following command:

```
docker system events
```

When you launch the above command, you will notice no output, and it will appear as if the terminal is hung. However, it's just streaming events from the Docker server and should start showing those events as you perform different operations such as pulling an image and launching a container.

Pulling a Docker Image from Registry

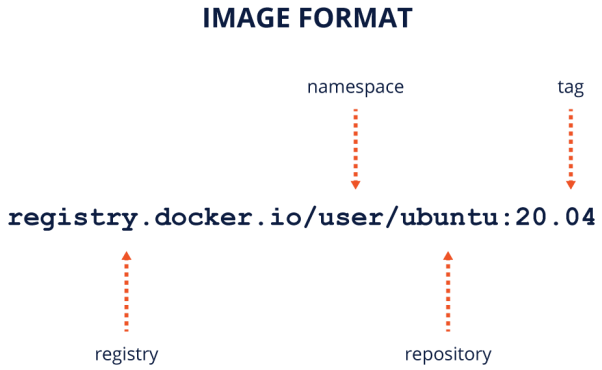
Now, what if you want to run a container? In order to do that, just like in the case of VMs, you need an image available on the host which would run this container. Where does the image come from? The answer is: *container registry* (e.g. Docker Hub/Microsoft Container Registry, etc.). You can think of a container registry as GitHub for your Docker images.



Let's pull the official Alpine Linux image created by Docker, Inc. To access this image go to the Alpine's image repository on Docker Hub. You can see an image example below:

```
alpine:3.17
```

Here, **`alpine:3.17`** is the name of the image. Each image can have up to four fields as shown in the following picture:



To confirm if this image exists locally, pull it if it is not present and validate, run the following commands:

```
docker image ls
docker image pull alpine:3.17
docker image ls
```

The image you pulled has the following fields:

- **alpine** = repository
- 3.17 = tag

Launching Your First Container

Now, after pulling this image, launch your container with:

```
docker container run alpine:3.17 uptime
```

or you can use the legacy way and do:

```
docker run alpine:3.17 uptime
```

You may be asking yourself, what's the difference between **docker container run** and **docker run** command? They are essentially both the same. After the 1.13 version of Docker became available, new categories of commands were introduced, and **docker run** became **docker container run**. However, the command is still backwards compatible. In fact, users are so used to the previous version of the command, that they often continue to use the old approach. Feel free to use whichever command you prefer.

In this exercise, we will go with the second option:

```
docker run alpine:3.17 uptime
```

When you launch the container, pay attention to the following two things:

- The output on the terminal

- Events streamed from the other terminal where you launched **`docker system events`**

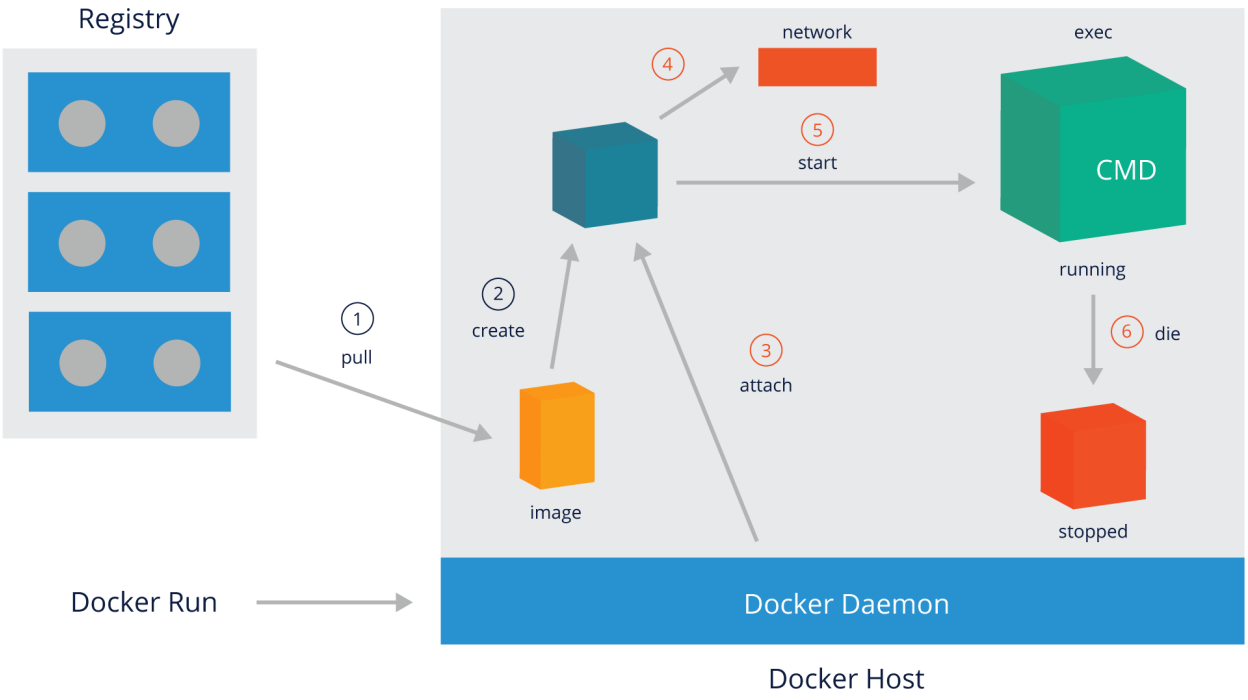
Below is the sample output of the command presented above:

```
docker run alpine:3.17 uptime
Unable to find image 'alpine:3.17' locally

3.17: Pulling from library/alpine
df20fa9351a1: Already exists
Digest:
sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f60a771a2dfe321
Status: Downloaded newer image for alpine:3.17
 13:33:30 up 32 days,  8:52,  load average: 0.00, 0.04, 0.00
```

The following is a list of events on the server side:

- container create
- container attach
- network connect
- container start
- container exec_create
- container exec_start: uptime
- container die
- network disconnect



Use the following command to check if the container you started is running:

```
docker ps
```

The above command is used to list the currently running containers. If your container is not showing, check its status using the `-l` flag. It will list the last run container:

```
docker ps -l
```

Below you can see the sample output:

```
 CONTAINER ID         IMAGE                COMMAND              CREATED
STATUS                              PORTS                NAMES
09aabefee6e2         alpine:3.17          "uptime"             About a minute ago
Exited (0) About a minute ago                            relaxed_lichterman
```

Its status says **`Exited`**.

Let's analyze what happened after you launched your container:

- Docker tried to look up the image locally. In our case, the image was present, so this step was skipped. Otherwise, Docker would try to download it from the registry.
- Docker created the container, e.g. all the components required to run a process in a contained environment.
- Docker service attached itself to this container to monitor the processes, manage logs, etc.
- Docker asked the kernel to set up network configurations for the container. This step created the virtual network interface for the container, set up the IP address, etc.
- Docker started running the container.
- Docker launched the actual command or the application of your choice (e.g. **`uptime`**). The application's start can also be configured implicitly in the image metadata, so that you do not have to provide it at the launch time.
- Since this example had a one-off command (**`uptime`** instead of a long-running process such as Apache), as soon as the process exited, Docker thought its job was done and it stopped the container.
- Once the container stopped, the network was also disconnected from the container. Please remember that the network is a separate entity than the container itself.

Listing Containers

Let's now launch a few more containers with different commands:

```
docker run alpine:3.17 hostname
docker run alpine:3.17 ps aux
docker run alpine:3.17 ifconfig
```

Now try to list containers using the following flags with the **`docker ps`** command:

- `-l`: to list the last container

- `-n 3`: to list the last three containers (you can try different integers instead of 3)
- `-a`: to list all containers

For example:

```
docker ps
docker ps -l
docker ps -n 3
docker ps -a
```

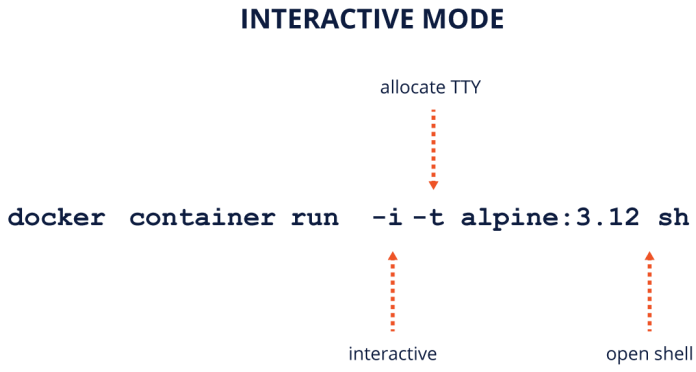
Using Common Container Runtime Options

In the previous section, we launched and ran a few containers. We didn't use any flags with a **`docker run`** command, but instead introduced the one-off command. In this section, we will discuss some common options that you can use with the **`docker run`** command. You will also see how to run containers long-term and observe how they work.

Launching an Interactive Container

The first two options that you can use are: **`-i`** and **`-t`** (you can also combine these options into **`-it`**). Also, this time, you will launch a long-running command, such as **`cmd`**:

```
docker run -it alpine:3.17 sh
```



When you run the above command, you will notice the following:

- This time not only the container is launched, but you also land inside the container with a shell prompt. You can finally get a better feel of the container!
- The container is not stopped but rather remains in a running state. This is because you launched it with **`sh`**. It will keep running as long as the shell is running and you don't exit.

While you are inside the container, you may want to run some commands and observe how containers work. Type **exit** and click on Enter to leave the container (alternatively you can do **^d**). Exiting from the shell will stop the container.

Now, try to launch a container with an Ubuntu image. Use the command below, observe Docker system events and look for differences. For example, you should see an event for the image being pulled, if you have not used this image before.

```
docker container run --rm -it ubuntu bash
```

where,

- **ubuntu**: is the image for the latest version of Ubuntu.
- **bash**: launch **bash** instead of **sh** as before, because Ubuntu comes with **bash**.
- **--rm**: tells Docker to delete the container once it has stopped (on exit from **bash**). This is a useful option to clean up one-off containers created just for testing.

You can also try running some commands such as those presented below inside the container:

```
ps
uptime
which apt
free
cat /proc/cpuinfo
```

Again, exiting the shell should stop and remove this container. Check if this is the case by running **docker ps -a** command.

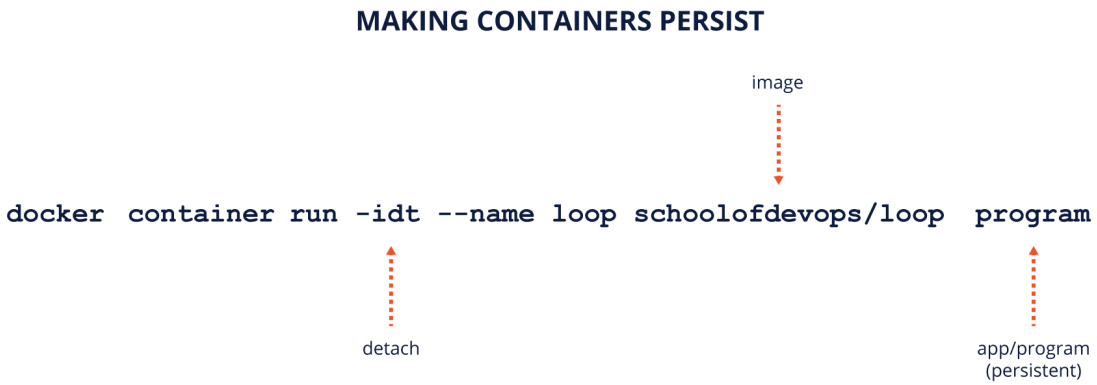
Running a Container in the Background/Detached Mode

The most desired way to run a container is to launch an application with a container, and keep it running in the background. This can be achieved by using the **--detach** option. This time, I will be launching a sample application:

```
docker run -idt --name loop --rm schoolofdevops/loop program
```

where,

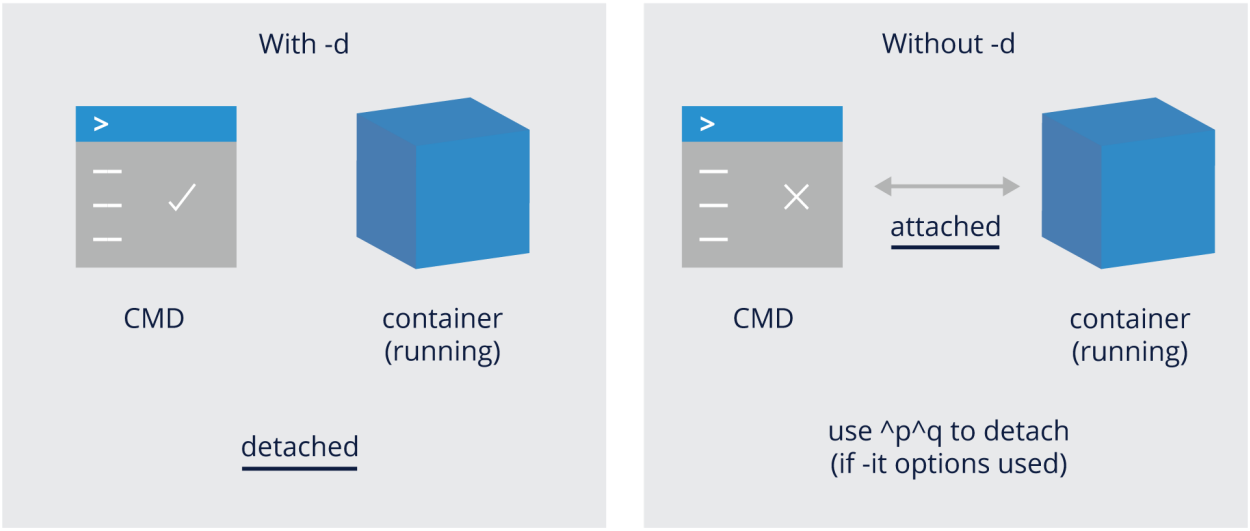
- **--name**: provides a name to the container instead of the automatically generated random name that Docker has assigned to it.
- **schoolofdevops/loop**: an image with the application **program** built-in.
- **program**: name of the command/application to be run on the launch of this container.



You should notice that the container is not only launched but is also running in the background, allowing you to proceed with the next commands/tasks. If you want to see the difference, try to run another container without the **-d** option:

```
docker run -it --rm schoolofdevops/loop
```

A container with the same image is launched without **-d** now. Note that you are attached to the process running inside the container now. If you use **-d** instead, this container will run in the background with the detached option.



If you launch a container without **-d**, remember to use **^p ^q** as an escape sequence to detach from the container's process. This will only work if **-it** options were used to launch the container. Use **^c** in other cases to kill the container.

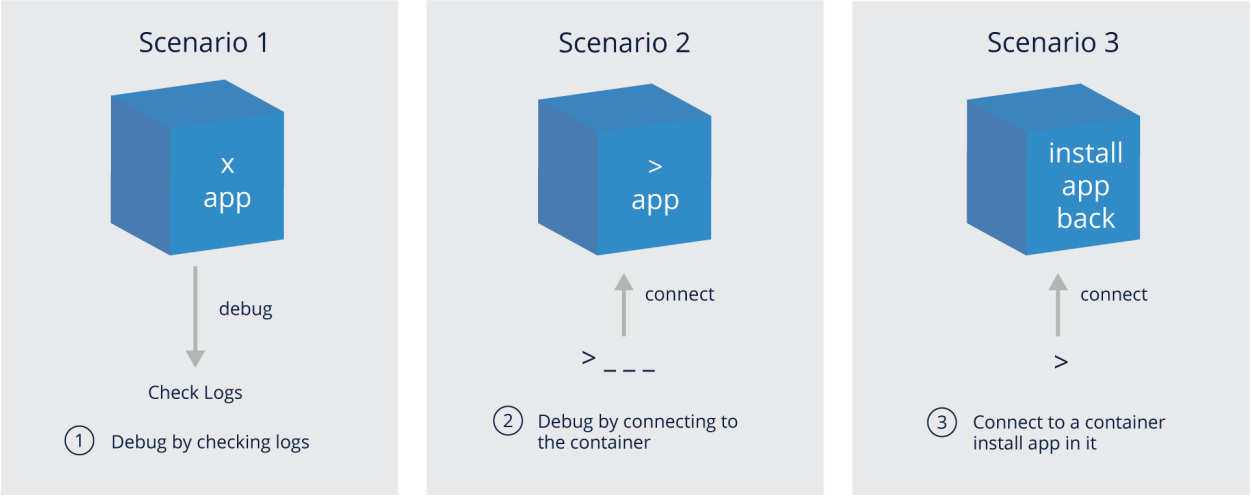
Best practice: When launching containers use `-idt` options. Out of these, `-d` is a must. `-it` is optional but recommended as you can detach/attach from the process/command/application running inside the container.

Troubleshooting and Modifying Containers

So far, you have launched applications in a container. What are the next steps? Most of the time, you may have to do some debugging or troubleshooting on a running container. How do you achieve that? There are two operations involved while debugging/modifying containers:

- You may want to get the logs from the application.
- You may want to connect to the container (e.g. open a shell inside it to manage and debug further).

Let's take a look at a few example scenarios.



- **Scenario 1**: You have launched an application which is misbehaving, so you want to debug it. The first thing that you may want to do is get the application logs.
- **Scenario 2**: You have an application running inside a container. Logs don't tell you much about what it's doing. You may want to connect to this container, open a shell, and debug further.
- **Scenario 3**: You have a container running a base OS. You want to modify it by installing a web server on it.

Scenario 1: Checking Application Logs

Let's consider the first scenario. If you want to check application logs, you can use either of the following commands (assuming your container's name is `loop`):

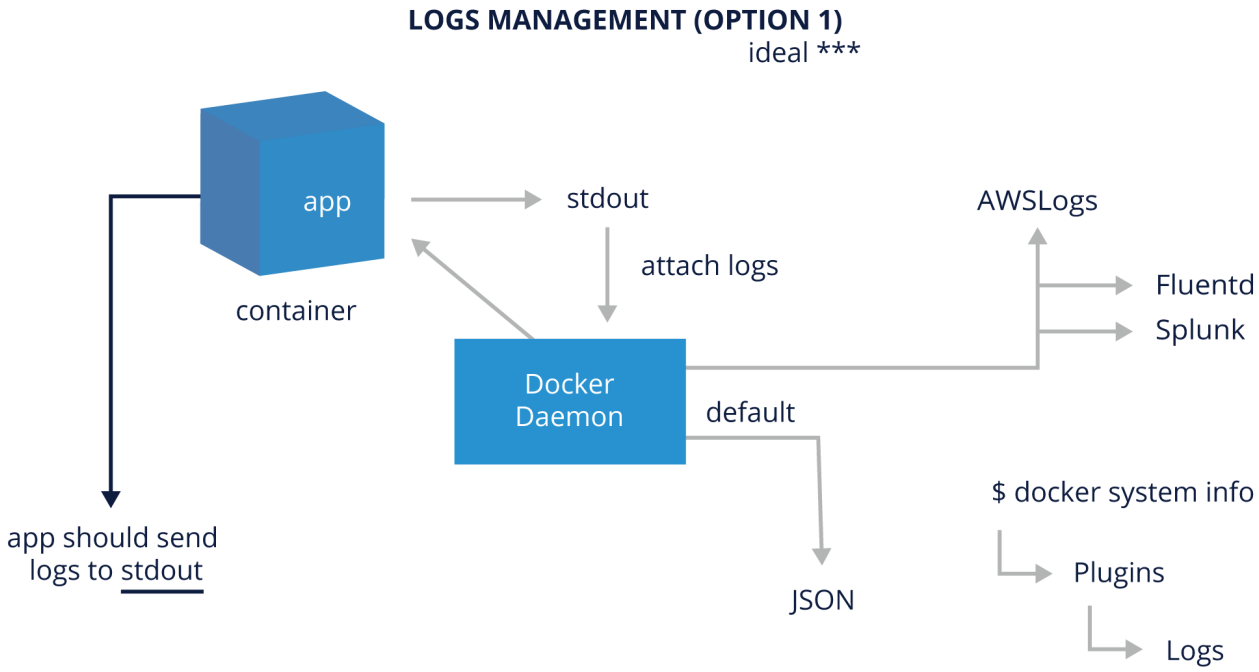
```
docker ps
```

Note container's name/ID:
`docker logs loop`

This command will follow the logs and notify you as they update:
`docker logs -f loop`

Use **`^c`** to stop following.

The above command does show the logs for the **`program`** running inside the container. Well, the question now is, How does Docker know about these application logs? An answer to that question is depicted in the following diagram:

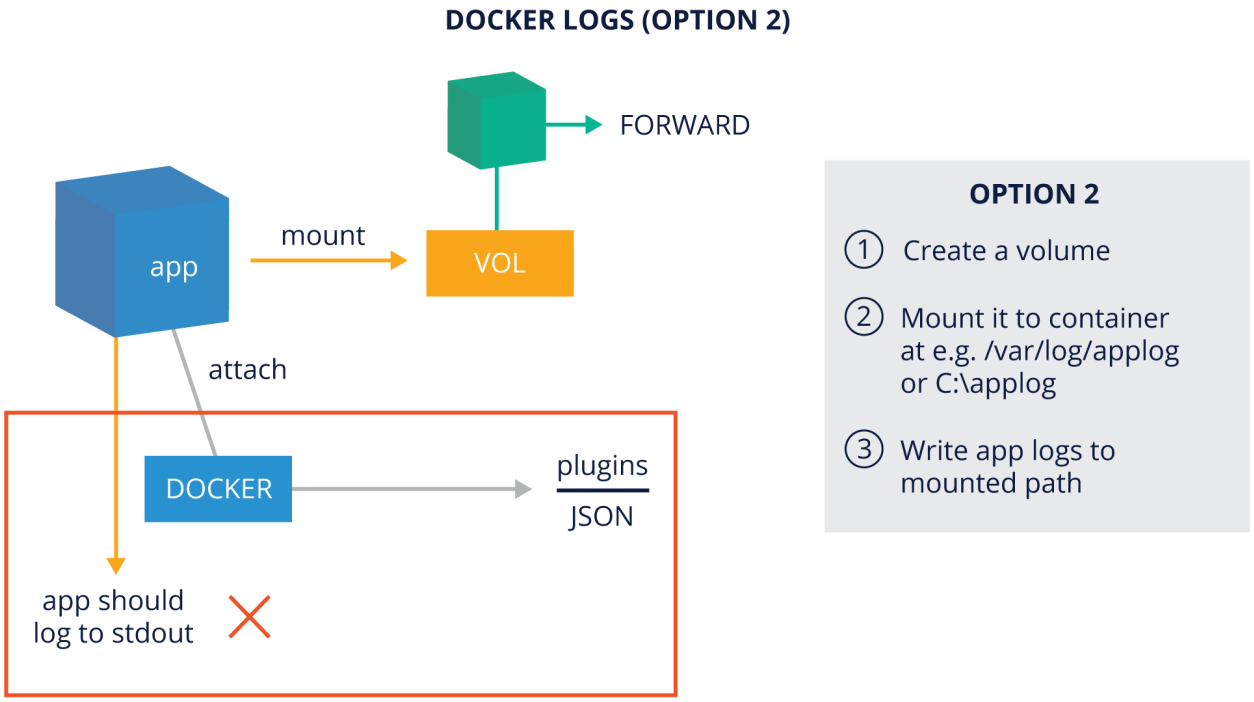


where,

- Docker attaches to the application's standard out (stdout).
- Application writes logs to standard out (see the sample **`program`** illustrated in the diagram).
- Docker daemon then forwards the logs based on the logs plugin used (the default is json-file). However, Docker is capable of forwarding logs to remote systems using plugins such as awslogs, gcplogs, syslog, or plunk.
- This integration is available out-of-the-box. You can check existing log plugins using **`docker system info`** command plugins > *Logs* section.
- This also needs an application to log to standard out.

To learn more about Docker's logging drivers see Docker's Documentation, *"Configure Logging Drivers"*.

Now, what happens if your application cannot be modified to log to standard output (stdio), or what if it generates multiple log streams and writes it to multiple files? In such cases, we recommend that you use the following option:



where,

- You mount a volume to the container. This needs to be done while launching the container. Volume can be local or remote.
- Mount it at the path where applications write the logs, such as **`/var/log/applog`** or **`C:\applog`**.
- Then, ensure that your application writes the logs only to that path.
- Optionally, from the volume, you can forward the logs to remote systems using forwarders.

Scenario 2: Getting Inside the Container's Shell

Unfortunately, checking the logs is not enough. You may want to get inside a shell of the container, similar to an SSH connection to a remote host. Alternatively, you can execute a one-off command against the running container. You can go either way using the **`exec`** command that Docker offers:

```
docker ps
```

Note container name/ID

To run a one-off command:
docker exec loop uptime

To get a shell access inside the container:
docker exec -it loop sh

The most commonly used command is the last one with the **-it** options, and **sh** or **bash** as the command, depending on which shell your Docker image comes with.

Once you open a shell above, you can execute any command that is available as part of that containers' image.

Scenario 3: Modifying Container by Installing an Application

Scenario 3 is an extension of Scenario 2. Launch a container with an Ubuntu OS this time:

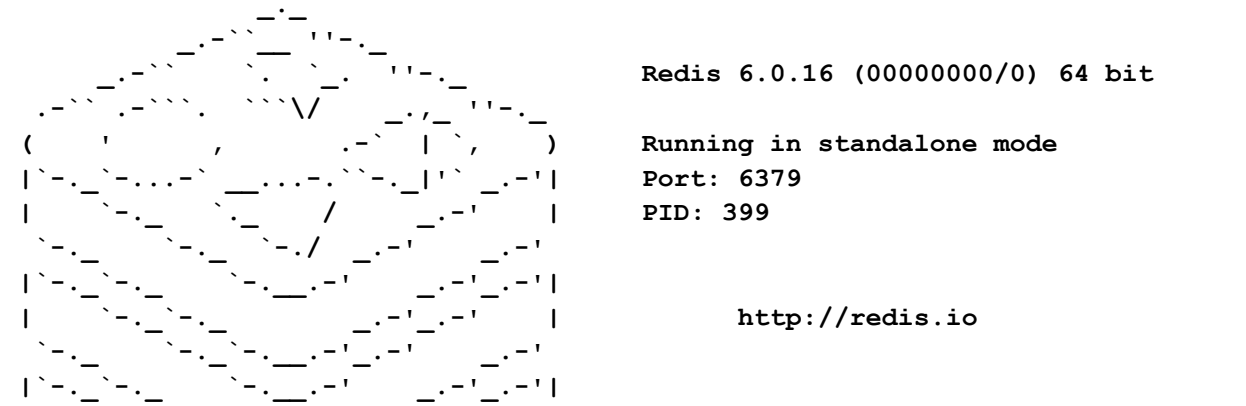
docker container run -idt --rm --name redis ubuntu bash

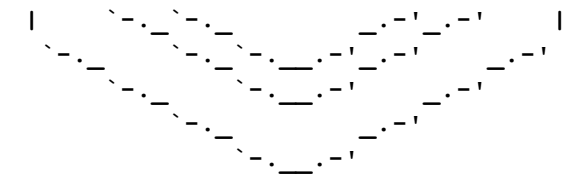
Now, log in to this container using **exec** and **bash** as in **docker exec -it redis bash**, and follow the steps to enable and start a web server:

```
apt-get update
apt-cache search redis
apt-get install redis
redis-server
```

The above sequence of commands should install and launch the Redis server on this host and display output such as following:

```
399:C 01 May 2023 09:44:33.756 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
399:C 01 May 2023 09:44:33.756 # Redis version=6.0.16, bits=64,
commit=00000000, modified=0, pid=399, just started
399:C 01 May 2023 09:44:33.756 # Warning: no config file specified, using the
default config. In order to specify a config file use redis-server
/path/to/redis.conf
```





```
399:M 01 May 2023 09:44:33.758 # Server initialized
399:M 01 May 2023 09:44:33.758 * Ready to accept connections
```

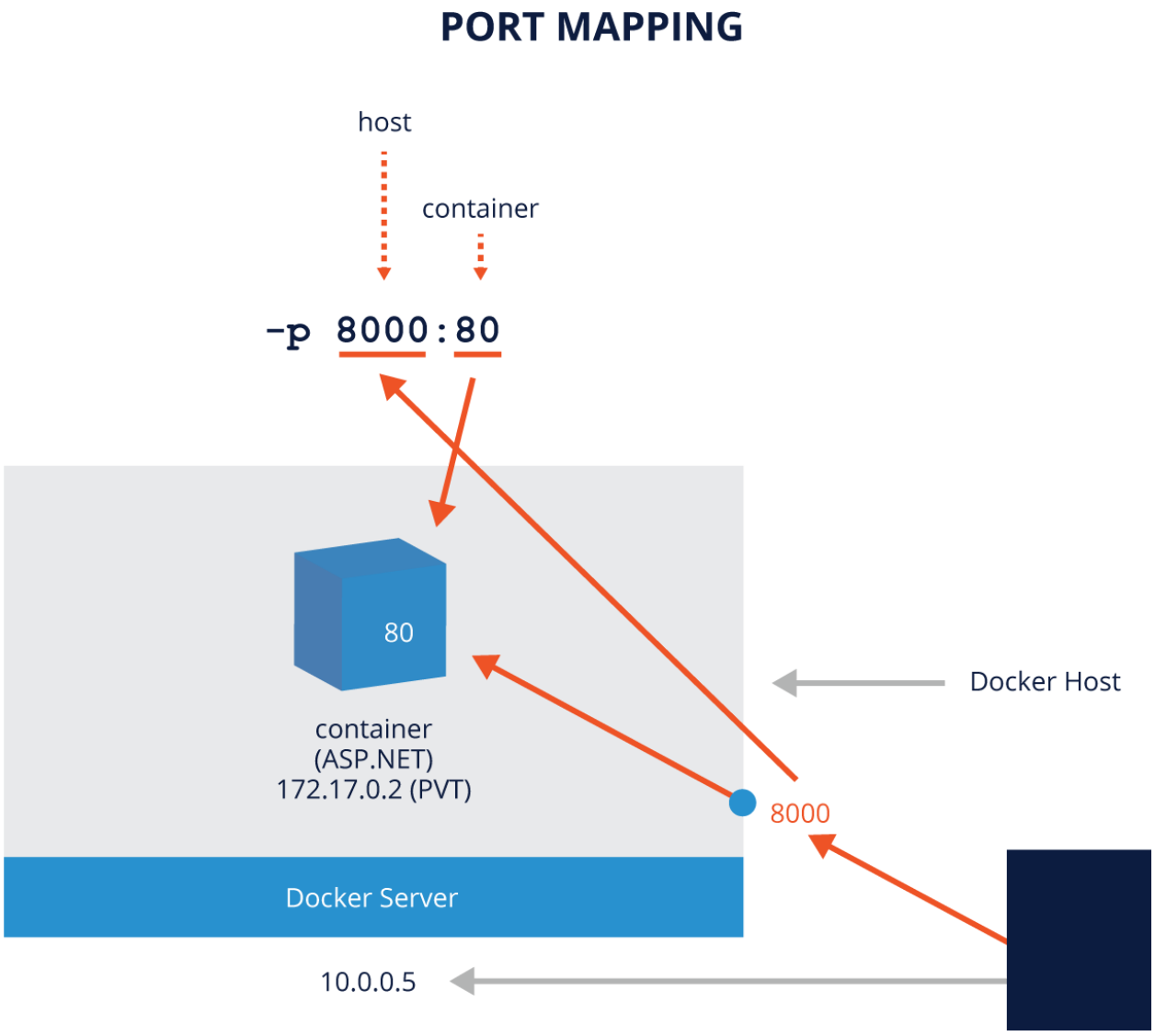
From here, you can stop the application by using **^c** and then exit the container by pressing **^d** to get back to the original shell you started from.

Congratulations! You have learned the basic troubleshooting of containers with logs and exec.

Accessing Web Applications with Port Mapping

While you have learned many common options to launch containers with, so far you have launched only the mock applications without any web interface. It's now time to launch a real web application. How to access this application? Well, first you need to learn about one more option: *publishing ports*/*port mapping*:

```
docker run -idt -p 8000:80 --rm schoolofdevops/vote
```



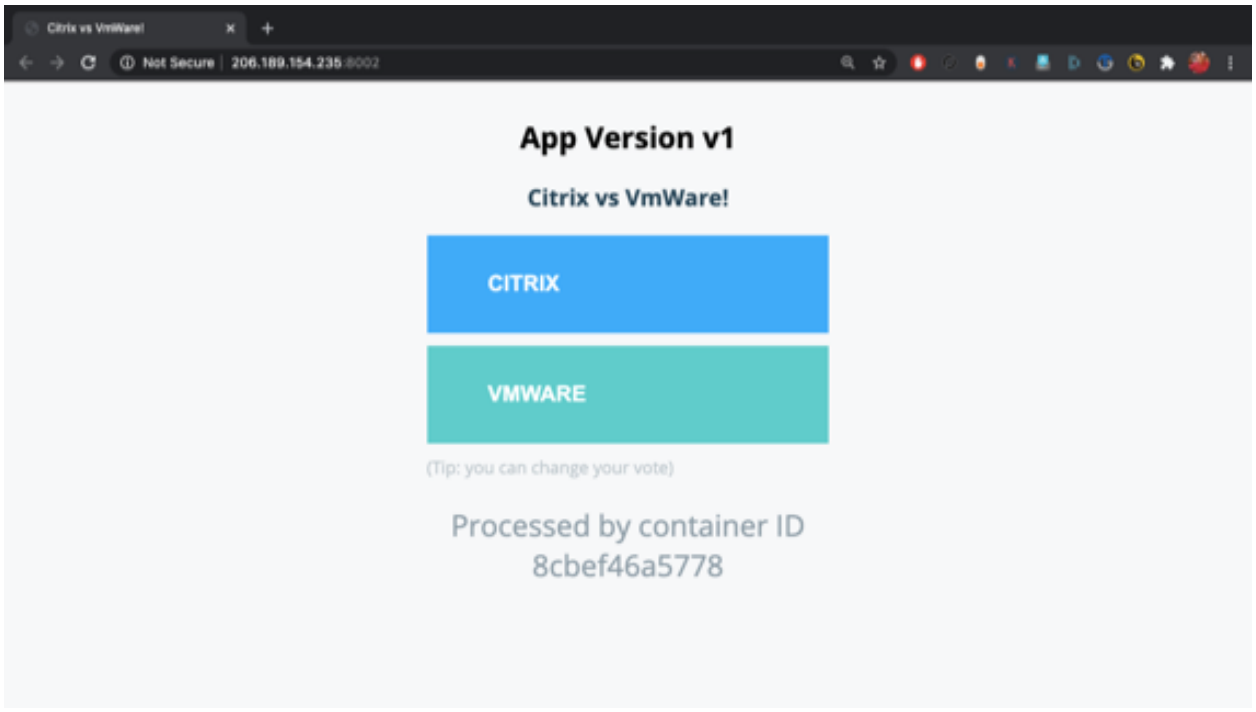
where,

- `-p 8000:80`: maps Docker host's `8000` port to the container's `80` port, while the actual web application is running.

This allows you to access the application using host IP and host port such as `http://hostip:hostport`:

- If you are using Docker Desktop replace `hostip` with `localhost` and use the following URL: `http://localhost:8000`.
- If you are using a remote server, use the IP address or hostname of it to access the application, e.g. for my cloud VM it's 'http://20.41.76.15:8000'.
- Ensure that port `8000` is open from the firewall configurations if using a remote host/ cloud server.

For example:



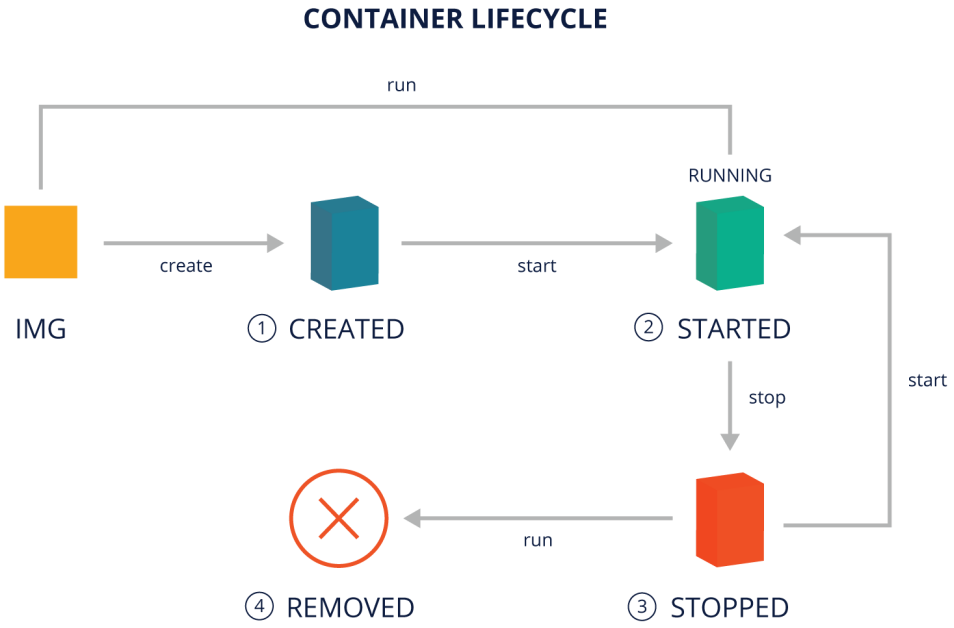
Apart from using the `-p` option, you can also use the following:

- Using only the `-P` option (capital letter) will automatically pick up the exposed port defined in the container image and map it to a host port starting with 32768. It will automatically increment this number for future port mappings.
- Using an option such as `-p 80` will pick up the container port 80 and map it to a host port starting with 32768. It will also automatically increment this number for a future port.

Managing Container's Lifecycle

A container goes through the following four lifecycle phases:

1. CREATED
2. STARTED/RUNNING
3. STOPPED
4. REMOVED



We have already learned how to launch a container and put it in the STARTED/RUNNING state using **`docker run`** command. You can also individually create and start containers using the following sequence:

```
docker create -it --name twostep ubuntu bash
docker start twostep
```

Please remember that you cannot use the **`-d`** option with a **`create`** command as detaching a container while creating it is invalid. In this case, when you start the container, it automatically starts with detached mode.

To stop a running container use the following command:

```
docker stop twostep
```

where,

- **`twostep`**: is the name of the container.

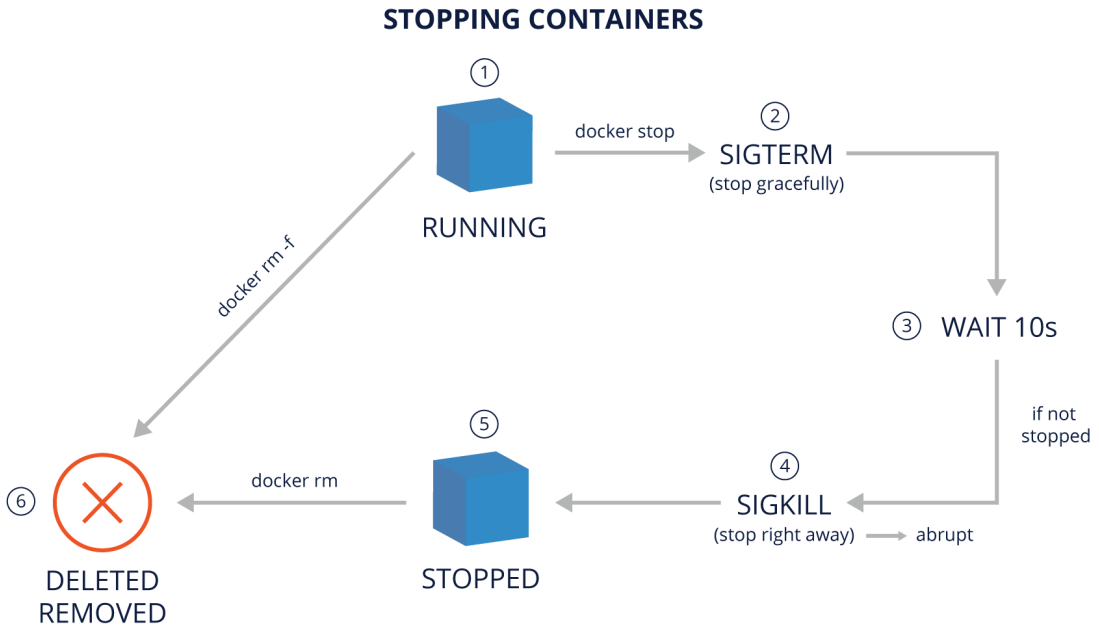
You can also stop multiple containers by providing a space separated list:

```
docker stop container_id  container_id  container_id
```

A stopped container can be started back with:

```
docker start twostep
```

***NOTE**: Stopping a container does not delete the data/changes you have made to the container. When you talk about immutable deployments (e.g. Kubernetes), the containers are deleted and replaced for every update, so retaining the data is not possible in such cases. However, in local environments, it can be stopped. Which means it retains all changes until it's removed and can be used just like a VM.*



A stopped container can be removed using **`docker rm`** commands as follows:

```
docker rm twostep
```

A running container needs the **`-f`** option for it to be stopped and removed, for example:

```
docker rm -f  vote
```

You can also delete all stopped containers in one go using the following command:

```
docker container prune
```

***NOTE**: The **`docker container prune`** command will delete all stopped containers. This may not be what you want if you created containers to use for your continuous development work. So use this command very carefully!*

Summary

In this lab exercise, you learned how to launch a container and use the most common container runtime options. You also observed what happens when you run a container and how to access

containers by using the port mapping. In addition, you got familiar with common troubleshooting tasks such as logs and exec, and saw how to stop and remove the containers. These are essential skills for anyone who wants to get started working with containers or use a container orchestrate engine.

Lab 4. Building Container Images

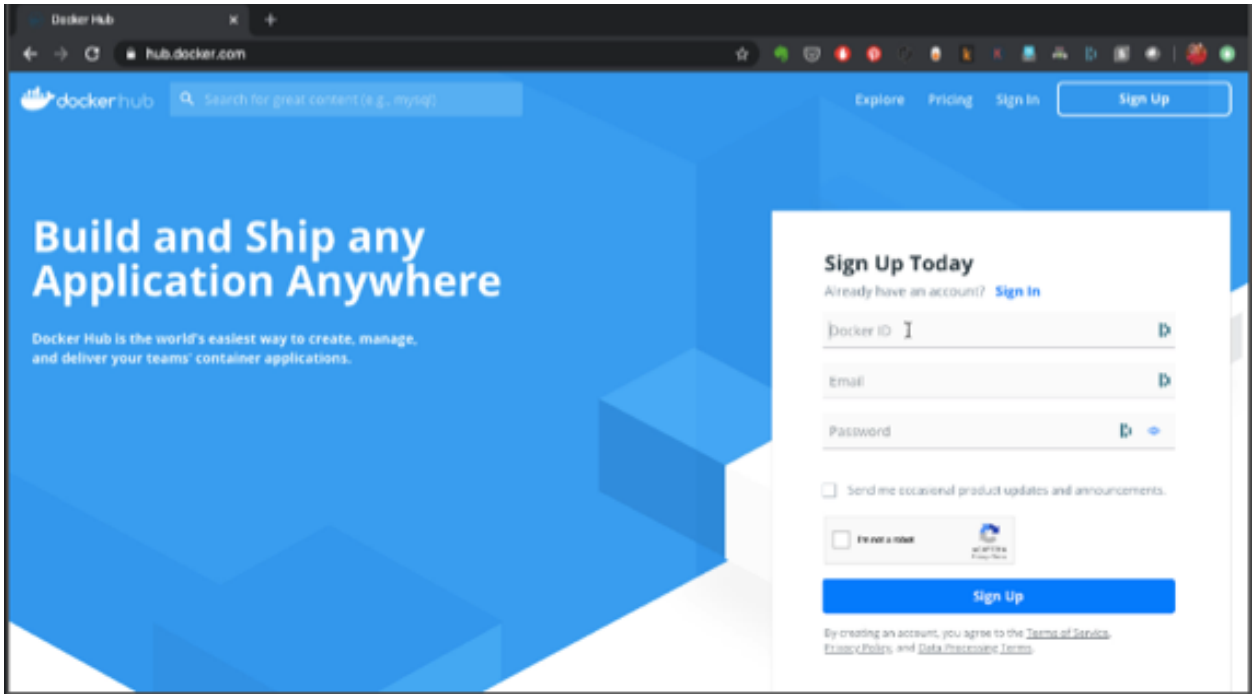
When you start with a container-based software delivery, the first step is always to take your application and package it as a container image along with its run time environment. Mastering the image build process with all of its aspects is an indispensable skill in the container world. The objective of this lab exercise is to help you acquire that skill. You will learn how to package applications with Docker by building images with Linux as the base operating system.

In this exercise we will discuss:

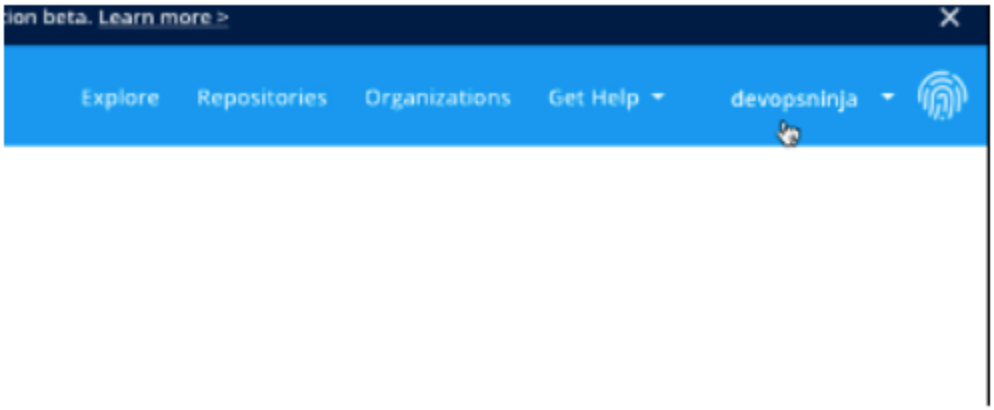
- How to get signed up to the Docker Hub registry and create your Docker ID.
- How to manually test build a Docker image by launching a container with base image and modifying it.
- How to automate the process of building Docker images by writing a Dockerfile.
- How to construct a Dockerfile and what the best practices of writing one are.
- Why you might need a multi-stage Dockerfile and how it works.
- How to package a Spring/Maven-based Java application as well as a C application.

Creating a Registry Account (Docker Hub)

By the end of this exercise, you will have at least a couple of images built. These images can then be published to a registry. You can start with Docker Hub as the default registry and publish your images under your own account. In order to do that, you need to have a Docker Hub account. If you have created a Docker ID while setting up your Docker Desktop software, it's the same account. If not, follow this process to sign up to Docker Hub and validate your account.



- Go to hub.docker.com
- Sign up by providing a username, email and password
- Verify your email address
- Log in

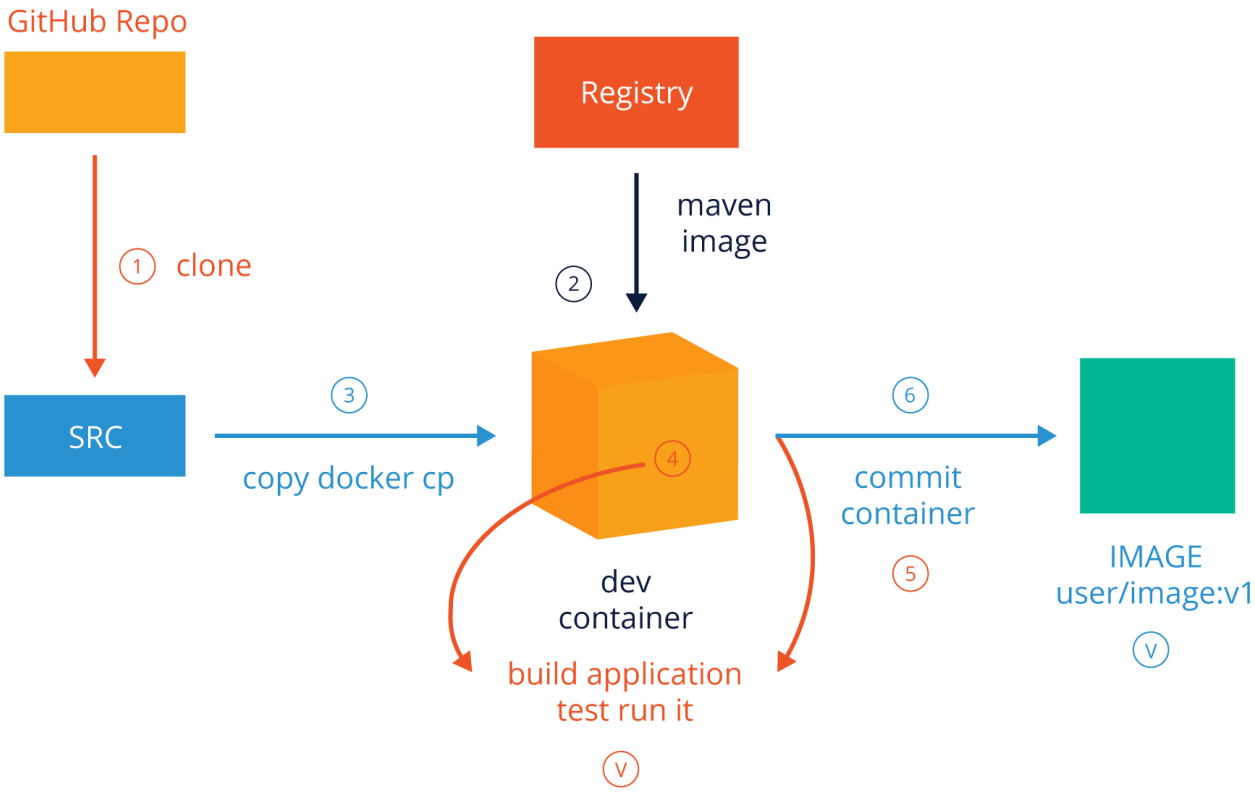


Once logged in, write down your Docker ID located in the top-right corner of the screen. It's your username in lowercase letters. This is an important step because this is the username/ID that you should use while tagging your images later in this chapter.

Test Building a Docker Image (Manual Approach) for a Spring Boot Application Build with Maven/Java

Before you automate the process of building an image, you need to know what you are going to automate. Doing a manual test build of your application helps to understand this process. We will look at the example of this Spring Boot application (spring-petclinic) and try building an image.

You need a Maven + JDK environment to build this application, which in itself, can be created by using a container image. This is followed by the step-by-step instructions on how to create a development environment and build a sample application.



1. Create a fork of devopsdemoapps/spring-petclinic, Spring PetClinic Sample Application to your GitHub account.
2. Clone the code from a GitHub repository to the local development host.
3. Create a container-based development environment and build the application. Use a pre-built Maven image which contains the tools.
4. Copy over the source code to this dev container.
5. Connect to the container and perform all the tasks necessary to build the application.
6. Do a test run of the application to ensure its working fine.
7. Once tested, commit the container's changes to an image using **`docker container commit`** command.

By the end of Step 6, you should get a manually built Docker image. Go ahead and follow along to convert these six steps into action.

Step 1

Begin by cloning the source repository and switching to the work directory:

```
git clone https://github.com/xxxxxx/spring-petclinic.git
cd spring-petclinic
```

Step 2

Create a dev environment with Maven to build this application:

```
docker run -idt -p 9091:8080 --name dev schoolofdevops/maven:spring bash
docker ps -l
```

Step 3

Copy over the source to the container:

```
docker cp . dev:/app
docker exec -it dev bash
```

Step 4

By following the above command sequence, you started the container and logged into it using the **exec** command. Now, build the application and test run it with:

```
cd /app
mvn spring-javaformat:apply
mvn package -D skipTests
mv target/spring-petclinic-2.3.1.BUILD-SNAPSHOT.jar /run/petclinic.jar
```

Step 5

Test the application by actually launching it inside the container:

```
java -jar /run/petclinic.jar
```

This should start running the application. You have already mapped port **8080** on the dev container to **9090** on the Docker host. You can now access and verify the application is running by using one of the links below:

- http://localhost:9091 (if using Docker Desktop)

- http://DOCKERHOST_IP:9091 (use IP/hostname of Docker host if created using a VM, or a cloud server).

Step 6

So far you started with a base image, added your application and tested it by running it. The dev container that you created contains the application that is built and ready. Now, you can commit the container's changes into an image by running:

```
docker container commit dev <dockrhub user id>/petclinic:v1
```

where,

- **`dev`**: is the name of the container that you created to build the application.
- **`<dockrhub user id>/petclinic:v1`**: is the image tag (ensure you replace **`<dockrhub user id>`** with your registry, e.g. DockerHub, ID).

***NOTE**: Credentials are not validated at the time of committing the image which is local, but when you try to push it to the registry.*

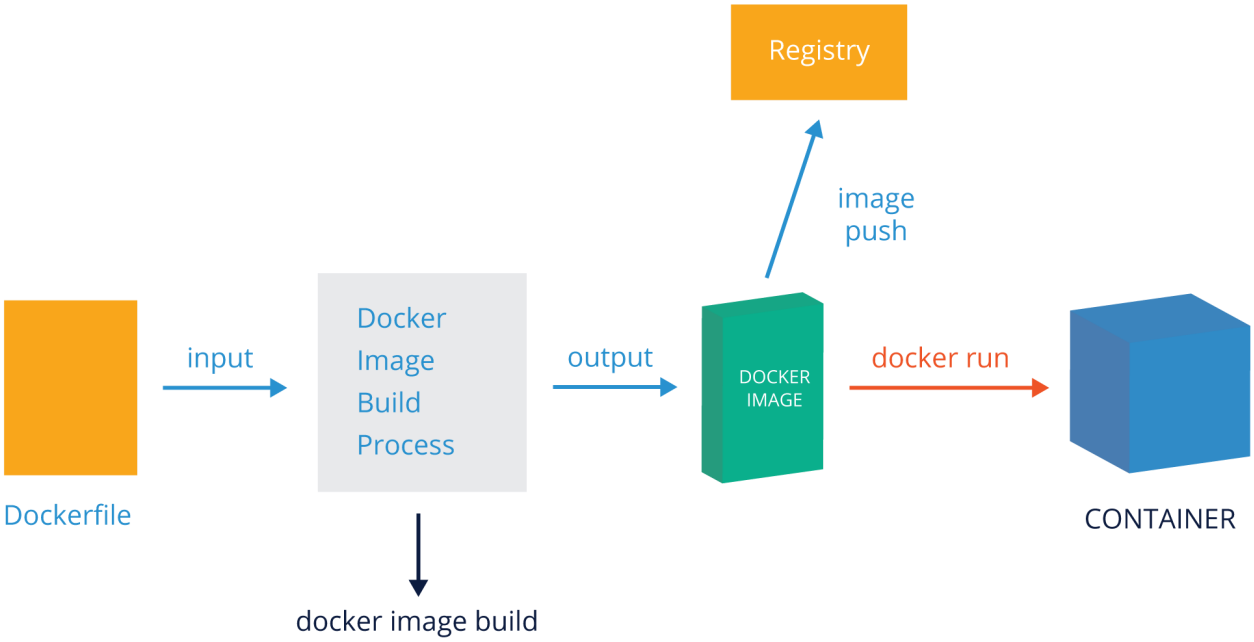
You can list examine the image created with the following commands:

```
docker image ls
docker image history <dockrhub user id>/petclinic:v1
```

Using Dockerfile to Automate Image Builds

Docker provides a way to codify the image build process. Think of it as converting the above process into code using a declarative language.

Following diagram depicts the use of Dockerfile:



- Dockerfile is fed to the image build process, which is run with the **`docker image build`** command.
- The **`docker image build`** command uses Dockerfile as an input, reads each of the instructions, and based on that builds the image, one layer at a time.
- The image built is stored locally, and can be tagged and pushed to a registry.
- It can then also be used to launch a container.

Go ahead and create a Dockerfile for this project by adding the following content to it:

```
File: spring-petclinic/Dockerfile

 FROM schoolofdevops/maven:spring

 WORKDIR /app

 COPY .  .

 RUN mvn spring-javaformat:apply && \
    mvn package -DskipTests && \
    mv target/spring-petclinic-2.3.1.BUILD-SNAPSHOT.jar /run/petclinic.jar

 EXPOSE 8080

 CMD java  -jar /run/petclinic.jar
```

To build an image with this Dockerfile use the following command:

```
docker image build -t <dockrhub user id>/petclinic:v2 .
```

Try building it again:

```
docker image build -t <dockrhub user id>/petclinic:v2 .
```

This time, it does not build anything, but instead uses cache.

Test the image:

```
docker container run --rm -it -p 9091:8080  <dockrhub user
id>/petclinic:v2
```

Publishing Images to the Registry

Now that you have built the images, it's time to publish them to the registry. In order to do that, you need to:

- Authenticate with the registry
- Be authorized to publish an image to the user/organization you have used in the tag. For example, if you try to push an image with the tag **`schoolofdevops/petclinic:v2`**, you need to have access to the "schoolofdevops" organization/user account. This is why tagging your images properly before you attempt to push them is so important!

You can also tag one of your images as **`latest`**, because **`latest`** is not an implicit tag and it won't be automatically set/updated. Instead, you have to set it by yourself by pointing it to one of the image versions. The following commands take you through the process of authenticating, tagging and publishing your images to the registry:

Try to push the image:

```
docker image push <dockrhub user id>/petclinic:v1
```

Log in to Docker Hub with your Docker ID:

```
docker login
```

Push image with v1 version:

```
docker image push <dockrhub user id>/petclinic:v1
```

Tag v2 as **`latest`**:

```
docker image tag <dockrhub user id>/petclinic:v2 <dockrhub user
id>/petclinic:latest
```

Push images with the **`latest`** tag only. Earlier, this used to push all tags, but the behavior has changed to push only those tagged as **`latest`**:

```
docker image push <dockrhub user id>/petclinic:v2
```

```
docker image push <dockrhub user id>/petclinic
```

By the end of this process, you should see the images appear in the registry. If you are using Docker Hub, you can go to the web console and check the presence of your images in the image listings.

Decoding Dockerfile Syntax

Dockerfile has its own syntax of the following type:

```
INSTRUCTION arguments
```

Some of the key instructions in Dockerfile can be found below:

- FROM
  It defines the base image.
- WORKDIR
  It defines the directory from which subsequent commands are run (e.g. COPY, CMD, ENTRYPOINT, etc.).
- COPY
  It is used to copy files to containers.
- RUN
  It runs a command, typically installing something inside the container or building an application.
- ENV
  It defines environment variables which are available while building and running containers using the **`image build`** command.
- EXPOSE
  It defines which port your application will listen to.
- ENTRYPOINT
  It defines a command or a script to run before launching the actual command/application.
- CMD
  It defines the command/application/process to start while launching a container with the **`image build`** command with this Dockerfile.

DOCKERFILE INSTRUCTIONS

FROM
Base image used for the build.

WORKDIR
Current working directory to change to.

COPY
Copy files / source code to container.

ADD
Add remote files to container.

RUN
Run a command inside build container.
Installs, builds apps.

CMD
Command to launch with docker run.

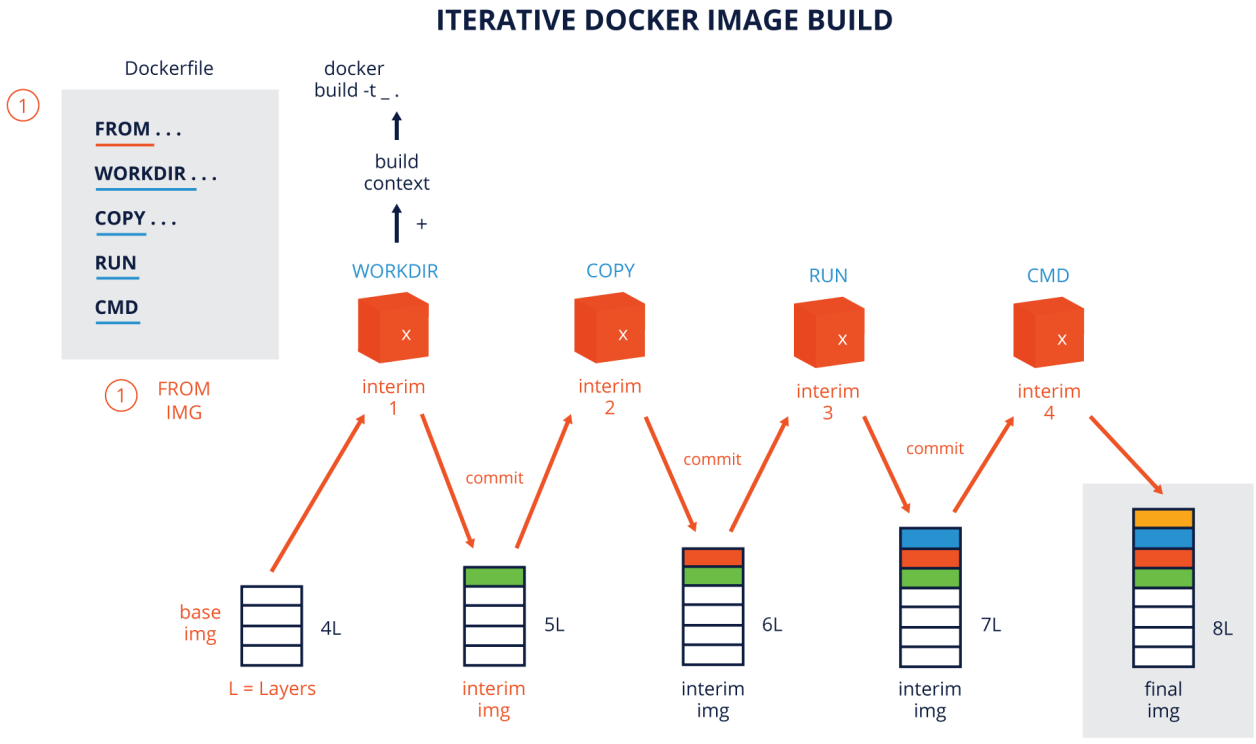
ENTRYPOINT
Run this after launching a container, before CMD.

ENV
Environment variables to use during built and add to container.

EXPOSE
Ports to expose.

Iterative Image Build with Dockerfiles Explained

Even though **`docker build`** follows a similar sequence of tasks as the manual test build above, it takes an iterative approach towards building this image as shown in the image below:



When you launch the **`image build`** command this is what happens:

1. Docker reads the FROM instruction and using the image defined with it as a base, it launches the first intermediate container.
2. Docker copies all the files in the build context to the Docker daemon and subsequently to the intermediate container. These files are then available throughout the build process as this container gets committed into the intermediate images for each step.
3. Docker daemon reads the next instruction (just one) in the Dockerfile and takes action based on it, e.g. copies a file, runs commands, defines metadata, etc.
4. Immediately after that, it commits the changes into an image, and deletes the intermediate container created earlier. This is what adds a new layer to the image.
5. This process is then repeated until all instructions in the Dockerfile are processed, one at a time, creating and deleting intermediate containers and building intermediate images.
6. At the end, the image is tagged with the option that you provided with the `-t` flag to the `docker build` command.

Summary

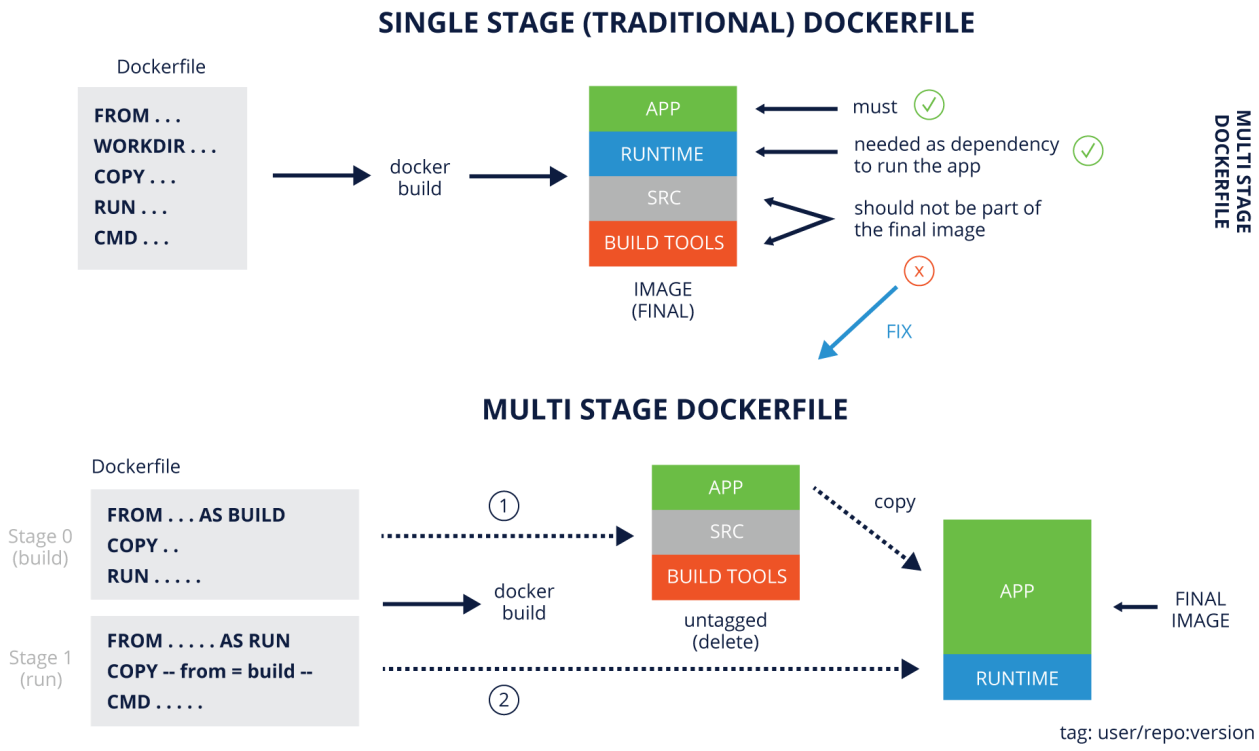
In this lab, you learned two ways to create container images, one using a manual approach and another one using Dockerfile. You also learned the art of writing Dockerfiles - what instructions to use, what does each instruction do, and how to write optimized Dockerfiles. Last but not least, you got to see how to work with the container image registries, how to tag and publish images, etc.

Lab 5. Multi-Stage Docker Build

So far, you have learned how to build an image with Dockerfile. However, the one you have built has at least two issues:

1. The final image also contains the build tools (e.g. Maven, JDK) and dependencies (~/.m2) which are not necessary to run the application. A minimal runtime would have been sufficient. Packaging tools along with an application increase the size of the image.
2. Final image also contains the source code, which is definitely not desired. Only the final application and its runtime should be part of what you distribute to the client/deploy to production.

How do we fix this? Is there a way to reduce the size of this image significantly? Yes! This is where a multi-stage Dockerfile comes in handy and solves the problem. Following diagram depicts the difference between a single stage (traditional) Dockerfile and a multi-stage Dockerfile:



The above diagram shows two stages but you can add more, if necessary. In this example with a two-stage Dockerfile this is what happens:

- The first stage, BUILD (stage #0), uses an image with all the build tools including Maven, JDK, etc. It creates an intermediate container to run the build.
- An intermediate container created for the build stage is where the source gets copied. This is also where it is compiled to create the application artifacts (e.g. JAR file), which are ready to be deployed.
- The second stage, RUN (stage #1), launches another container. This time it uses a minimalistic image with runtime, and only with the components needed to run the application.
- A copy logic is added to this stage to fetch the deploy ready artifact from the build stage by referencing the `--from` option to the COPY instruction. This ensures that only the application binaries and not the source code are added to the final image.
- The second stage is also the final stage in this file. An image is created by packaging the runtime and application alone, and is then tagged using the option that you provide with the `docker build` command.

Refactoring Spring Boot Application with a Multi-stage Dockerfile

Study this multi-stage Dockerfile provided on GitHub: [initcron/sysfoo, Sample Java Web Application with Maven which Prints System Information], and then refactor the Dockerfile that you have created for the PetClinic Spring Boot application to create two stages, BUILD and RUN, similar to the example above. Do not look at the solution before you attempt it.

Summary

In this lab, you learned how to create small, secure and optimized container images using multi-stage Dockerfiles, and how to avoid getting unnecessary files such as source code, build tools, etc., packaged into the container image.

Lab 5. Multi-stage Docker Build (Solution)

Following is the code which demonstrates a Dockerfile with two stages:

```
file: Dockerfile.multistage.v1

FROM schoolofdevops/maven:spring AS build
WORKDIR /app
COPY .  .
RUN mvn spring-javaformat:apply && \

mvn package -DskipTests

FROM openjdk:8-alpine AS run
COPY --from=build /app/target/spring-petclinic-2.3.1.BUILD-SNAPSHOT.jar
/run/petclinic.jar
EXPOSE 8080
CMD java  -jar /run/petclinic.jar
```

To build an image with this new Dockerfile, run the following sequence of commands:

```
docker build -f Dockerfile.multistage.v1 -t <docker hub id>/petclinic:v3  .
docker image ls
```

Adding an Opt-in Test Stage

You can use a multi-stage approach to support optional features. The following code demonstrates an opt-in stage to run unit tests:

```
file: Dockerfile.multistage.v2

FROM schoolofdevops/maven:spring AS build
WORKDIR /app
COPY .  .
RUN  mvn spring-javaformat:apply && \

mvn package -DskipTests
```

```
FROM build AS test
CMD mvn clean test

FROM openjdk:8-alpine AS run
COPY --from=build /app/target/spring-petclinic-2.3.1.BUILD-SNAPSHOT.jar
/run/petclinic.jar
EXPOSE 8080
CMD java  -jar /run/petclinic.jar
```

Now, build using the opt-in stage:

```
docker build -f Dockerfile.multistage.v2 --target test  -t <docker hub
id>/petclinic:test  .
```

Please note that the opt-in stage is executed only when you build the image with **`--target test`** option. In other cases, it is omitted.

Lab 6. Developing with Alternative Tools: Buildah, Podman, Skopeo

The objective of this lab exercise is to get you started with alternative tools to run containers, build images and to work with registries. In this lab, you are going to learn:

- How to run containers directly with runc and containerd, and how those tools differ from Docker CLI.
- How to run containers and build images in a rootless and daemonless environment with Podman and Buildah.
- How to build container images from scratch, as well as how to do it step by step.
- How to inspect images with Skopeo.

Launching Containers with runc

To launch a container with runc, do:

```
runc
runc spec
cat config.json
```

Now, create a root file system by exporting a container and launch it with runc:

```
docker create --name alp01 alpine sh
docker export alp01 -o alpine-root.tar
mkdir rootfs
tar -xf alpine-root.tar -C ./rootfs/
runc run testc
```

Inside the container:

```
ps aux
ifconfig
cat /etc/issue
```

```
    touch test
```

While the container is running, open a second terminal and examine it with the following command:

```
runc list
runc exec -t testc sh
```

***NOTE**: Run `^d` or `exit` command to exit the container shell.*

```
ctr run -t --net-host --rm docker.io/library/alpine:3.12
demo sh
```

Working with containerd

In addition to launching containers with runc, you can also run it using containerd. Containerd runs as a daemon, does image management, sets up the root file system, and then calls runc to finally launch the container. To work with containerd, try using the following sequence of commands, and follow along with the video lesson:

```
ctr
```

```
ctr version
```

```
ctr container
```

```
ctr image
```

```
ctr image pull nginx
ctr image pull docker.io/library/nginx
ctr image pull docker.io/library/nginx:latest
```

```
ctr run -h
```

```
ctr image pull docker.io/library/alpine:latest
ctr run -t --rm docker.io/library/alpine:latest sh
```

Inside the container:

```
ps
cat /etc/issue
ifconfig
exit
```

Outside the container:

```
ctr run -h
ctr run -t --net-host --rm docker.io/library/alpine:latest sh
```

Inside the container:

```
ps aux
ifconfig
exit
```

Outside the container:

```
ctr run -t -d docker.io/library/nginx:latest
web ctr c list
```

```
ctr c list
ctr c rm web
ctr t
ctr t ls
ctr t kill web
ctr c list
ctr c rm web
```

Using Podman as a Replacement for Docker

While containerd and runc are underlying technologies that Docker uses, if there is one tool that you can replace Docker with, it's Podman. The resemblance is so strong that you can create an alias (e.g. **`alias docker=podman`**), and you may not even notice the difference. All the common commands that come with Docker are supported by Podman. There are distinct advantages of using Podman such as ability to run containers as a non-root user, as well as ability to run containers without a daemon, which make Podman more secure and flexible. In this section, you are going to start exploring Podman.

Daemonless Alternative to Docker

Pulling an image:

```
podman image pull mysql
```

Launching a container:

```
podman run -idt -P nginx
```

Building an image:

```
git clone xxxx
cd xxxx
```

```
podman image build -t xxxx/xxxx:xx .
```

Rootless Containers with Podman

Create a new user, switch to it and start running containers with that user. This is not possible with Docker unless you add that user to the Docker group, which essentially provides the user with root access (Docker daemon always runs as root).

```
useradd -m devops
passwd devops

su - devops

podman image ls
podman run -idtP nginx

podman ps
podman image ls

podman run -idt -p 80:80 nginx
```

Working with Pods

Podman also supports running a group of containers with shared namespaces, commonly called pods as per the Kubernetes terminology. This makes it easier to develop with pods, without requiring you to set up a Kubernetes environment.

```
podman ps
podman pod ls

podman pod create --name web -p 80

podman run -idt --pod web --name nginx nginx:stable-alpine

podman run -idt --pod web --name sync schoolofdevops/sync:v2

podman pod ls podman
pod ls --ctr-name
podman ps
podman ps --pod

podman generate kube web
podman generate kube -s web
```

Advanced Image Building with Buildah

Building an Image with Dockerfile as a Non-root User

Similar to Podman, if you want to build images as a non-root user, and in a daemonless mode, you have a companion to Podman called Buildah. Install Buildah and try building an image with

it using the same spec that you would use with Docker, e.g. with Dockerfile. Follow the instructions below and refer to the video lessons:

```
sudo apt-get -y install buildah
```

```
buildah
buildah bud -h
```

```
git clone
https://github.com/schoolofdevops/example-voting-app.git
```

```
cd example-voting-app/vote/
```

```
MYACCOUNT=xxxxxx
```

***NOTE**: Replace **xxxxxx** with your username/organization/project on Docker Hub to push the images.*

```
buildah bud -t docker.io/$MYACCOUNT/vote:v1 .
```

```
buildah images
podman image history docker.io/$MYACCOUNT/vote:v1
```

Building an Image from Scratch

Switch to the non-root user created earlier:

```
su - devops
```

```
buildah from --name tinyc scratch
buildah containers
```

```
buildah unshare
buildah mount tinyc
```

Sample output:

```
/home/devops/.local/share/containers/storage/overlay/
2e7c3f501cc957a00e3f1cf9f2d1ae1c7f6206aa65fc0b7209628770dce4a8a4/merge
d
```

```
TINYMOUNT=`buildah mount tinyc`
```

```
echo $TINYMOUNT
```

```
ls -al $TINYMOUNT
```

```
du -sh $TINYMOUNT
```

Building an Alpine Base Image

Start by testing if you can run an Alpine-related command with the current state:

```
buildah run tinyc apk update
```

Did it work? No, it failed. Try to analyze why.

Now, download the x86_64 version of a minimal root filesystem for Alpine and extract it inside the mounted filesystem of the container created from scratch in the image above.

```
wget -c
https://dl-cdn.alpinelinux.org/alpine/v3.14/releases/x86_64/alpin
e minirootfs-3.14.0-x86_64.tar.gz

tar -xf alpine-minirootfs-3.14.0-x86_64.tar.gz -C $TINYMOUNT

ls -al $TINYMOUNT
buildah run tinyc cat /etc/issue

buildah unmount tinyc
exit

buildah config --cmd "/bin/sh" --author "xxxx yyyy" --created-by
"adding  alpine mini rootfs" tinyc
```

***NOTE**: Replace **xxxx yyyy** with your name.*

```
buildah commit tinyc alpine:3.14

buildah images
podman image history alpine:3.14

MYACCOUNT=xxxxxx
```

***NOTE**: Replace **xxxxxx** with your username/organization/project on Docker Hub to push the images.*

```
buildah tag alpine:3.14 docker.io/$MYACCOUNT/alpine:3.14

buildah login docker.io
buildah push docker.io/$MYACCOUNT/alpine:3.14
```

Building an Image with Java Runtime

Continue working with the same build container and add a Java runtime environment into it:

```
buildah from --name jre alpine:3.14
buildah run jre apk update
```

```
buildah run jre apk search java

buildah run --add-history jre apk add openjdk11-jre-headless

buildah config --author "xxxx yyyy" --created-by "installing
jre 11" jre
```

***NOTE**: Replace **xxxx yyyy** with your name.*

```
buildah commit jre docker.io/$MYACCOUNT/jre:11

podman image history docker.io/$MYACCOUNT/jre:11

podman run -it --rm docker.io/$MYACCOUNT/jre:11
```

Inside the container:

```
cat /etc/issue
java -version
exit
```

Outside the container:

```
buildah push docker.io/$MYACCOUNT/jre:11
buildah rm jre

buildah containers
buildah images
```

Test Building an Image (Step by Step)

If you are building an image with Docker, it either allows you to commit the changes that you have made to a running container, or launch a build process with a Dockerfile and run all the instructions which work towards the target stage. There is no way to take the instructions and selectively run those, while still retaining the history of changes, etc. One of the interesting features of Buildah is its ability to use instructions in Dockerfile and run them selectively and individually. You can even retain the history, commit multiple images, and do step-by-step troubleshooting. Explore all of this by using the instruction below while following along with the video lessons:

```
cd spring-petclinic

cat Dockerfile
```

Launch an intermediate build container to copy the source code to and run the compilation from:

```
buildah from --name build schoolofdevops/maven:spring
```

```
buildah containers
```

Set up a working directory (to be used by RUN, COPY, CMD) inside the build container's environment and copy over the source code to be built:

```
buildah config -h
buildah config --workingdir /app build
```

```
buildah copy -h
buildah copy build .
buildah run build ls -al /app/
```

Go ahead and compile the code with Maven:

```
buildah run build mvn package -DskipTests
buildah run build ls -al /app/target/
```

At this point, you should see a JAR file created and available inside the target directory.

The job of the build container is over (to compile the application and generate the artifact). Source code, build tools, etc., remain in this container, which you would discard later. Now, you can proceed to create the final image, which only contains Java runtime and the application artifact. You can, in fact, use the image that you created earlier.

Check presence of the Java image that you built earlier, and proceed to create a new container to package application using this image:

```
buildah images
```

```
MYACCOUNT=xxxxxx
```

***NOTE**: Replace **xxxxxx** with your username/organization/project on Docker Hub to push the images.*

```
buildah from --name package docker.io/$MYACCOUNT/jre:11
```

```
buildah containers
```

Validate that you see a new container named **package**.

Now copy the artifact generated in the build stage to the newly created **package** container:

```
buildah run build ls -al /app/target/
buildah copy -h
```

```
buildah config --workingdir /run package
buildah copy --from build package
/app/target/spring-petclinic-2.3.1.BUILD SNAPSHOT.jar
```

```
buildah run package ls -al

buildah run package mv spring-petclinic-2.3.1.BUILD-SNAPSHOT.jar

petclinic.jar buildah run package ls -al

buildah config --cmd 'java -jar petclinic.jar' --port 8080
--history-comment  'packaging petclinic app' package

buildah commit package docker.io/$MYACCOUNT/petclinic:v1

podman image history docker.io/$MYACCOUNT/petclinic:v1

podman run --name pctest --rm -idt -p 9080:8080
docker.io/$MYACCOUNT/ petclinic:v1

podman ps -l
```

Validate if the application is running on **http://IPADDRESS:9080**.

```
podman stop pctest
```

Container terminated due to the **--rm** options.

```
buildah push docker.io/$MYACCOUNT/petclinic:v1

buildah rm build package

buildah containers
buildah images
```

Skopeo

Skopeo is an interesting tool which allows you to examine a container image, and find out which files changed in which layers and how efficient your image is, etc. This can be a great starting point towards further image layers optimization. Install Skopeo and start examining the images using the following command sequence:

```
sudo apt-get -y update
sudo apt-get -y install skopeo

skopeo
skopeo inspect
docker://docker.io/postgres
```

Summary

In this lab, you learned how to use alternative tools and set up a workflow to build, run and publish your container images. This is immensely useful as the container ecosystem starts gravitating around Kubernetes, and with the requirements to support specific development workflows such as shared environments which require rootless containers, continuous integration environments with no need to mount Docker socket to build images, support for advanced image build options, etc.

Lab 7A. Docker Networking

In this lab exercise, you are going to learn about the single host Docker networking and specifically:

- How to examine the network configuration on a Docker host.

Bridge Networking

Following are the three networks created by default on a Docker host:

- bridge
- host
- none

You can examine the existing network configurations by using the Docker CLI, or using tools such as Portainer, if available:

```
docker network ls
```

```
NETWORK ID          NAME                DRIVER              SCOPE
b3d405dd37e4        bridge              bridge              local
7527c821537c        host                host                local
773bea4ca095        none                null                local
```

Creating a new network:

```
docker network create -d bridge mynet
```

Validate that the bridge network by name `mynet` is created using:

```
docker network ls
```

```
NETWORK ID          NAME                DRIVER              SCOPE
b3d405dd37e4        bridge              bridge              local
```

```
7527c821537c        host                host                local
4e0d9b1a39f8        mynet               bridge              local
773bea4ca095        none                null                local
```

```
docker network inspect mynet
```

```
[
   {
       "Name": "mynet",
       "Id":
"4e0d9b1a39f859af4811986534c91527146bc9d2ce178e5de02473c0f8ce62d5",
       "Created": "2018-05-03T04:44:19.187296148Z",
       "Scope": "local",
       "Driver": "bridge",
       "EnableIPv6": false,
       "IPAM": {
           "Driver": "default",
           "Options": {},
           "Config": [
               {
                   "Subnet": "172.18.0.0/16",
                   "Gateway": "172.18.0.1"
               }
           ]
       },
       "Internal": false,
       "Attachable": false,
       "Ingress": false,
       "ConfigFrom": {
           "Network": ""
       },
       "ConfigOnly": false,
       "Containers": {},
       "Options": {},
       "Labels": {}
   }
]
```

Launching Containers in Different Bridges

Launch two containers `nt01` and `nt02` in a **default** bridge network:

```
docker container run -idt --name nt01 alpine sh
docker container run -idt --name nt02 alpine sh
```

Launch two containers **nt03** and **nt04** in a **mynet** bridge network:

```
docker container run -idt --name nt03 --net mynet alpine sh
docker container run -idt --name nt04 --net mynet alpine sh
```

Now, let's examine if they can interconnect:

```
docker exec nt01 ifconfig eth0
docker exec nt02 ifconfig eth0
docker exec nt03 ifconfig eth0
docker exec nt04 ifconfig eth0
```

Following is a sample configuration from my host :

```
nt01 :  172.17.0.18
nt02 :  172.17.0.19
nt03 :  172.18.0.2
nt04 :  172.18.0.3
```

Note the container IP addresses and try to:

- ping from **nt01** to **nt02**
- ping from **nt01** to **nt03**
- ping from **nt03** to **nt04**
- ping from **nt03** to **nt02**

Replace/update IP addresses based on your setup:

```
docker exec nt01  ping 172.17.0.19
docker exec nt01  ping 172.18.0.2
docker exec nt03  ping 172.17.0.19
docker exec nt03  ping 172.18.0.2
```

You can observe that there are two different subnets/networks even though they run on the same host. **nt01** and **nt02** can connect with each other, and **nt03** and **nt04** can connect as well. However, a connection between containers attached to two different subnets is not possible.

Using None Network Driver

You can create a container with the **none** network, which would essentially mean that the container has no network access. Examine it using the following sequence of commands:

```
docker container run -idt --name nt05 --net none alpine sh

docker exec -it nt05 sh

ifconfig
```

Using Host Network Driver

If you assign a container to a host network, it is going to have access to all the interfaces on the host. Essentially, the container has no network namespace of its own. Examine host network with the following commands:

```
docker container run -idt --name nt05 --net host alpine sh
docker exec -it nt05 sh
ifconfig
```

Troubleshooting with Netshoot

To begin with, read about the netshoot utility here. Netshoot is an extremely useful utility that not only lets you examine the network configurations such as bridges, routing tables, port mappings, etc., but can also help you troubleshoot most of the container networking issues.

Launch netshoot and connect to the host network:

```
docker run -it --net host --privileged nicolaka/netshoot
```

Examine port mapping:

```
iptables -nvL -t nat
```

Traverse host port to container IP and port.

Observe Docker bridge and routing with the following commands:

```
brctl show

ip route show
```

Summary

In this lab, you explored the single host networking with Docker as a software. You also learned about the three default networks, how to create custom networks as well as how to get started with troubleshooting.

Lab 7B. Persistent Volumes with Docker

In this lab, you are going to learn how to create three different types of volumes with Docker. These instructions are intended to be followed along with the video lessons which explain each of the volume types in detail.

Types of Volumes

Following are the three types of volumes that are supported by Docker and are available on the Docker host by default:

- volumes
- bind mounts
- tmpfs

Automatic Volumes

To create a volume which is automatically named by Docker, just provide the destination inside the container to mount the volume at:

```
docker container run  -idt --name vt01 -v /var/lib/mysql  alpine sh
docker inspect vt01 | grep -i mounts -A 10
```

Named Volumes

To create a volume with a specific name use the following command:

```
docker container run  -idt --name vt02 -v db-data:/var/lib/mysql
alpine sh
docker inspect vt02 | grep -i mounts -A 10
```

Bind Mounts

If you have an existing path on the host, and want to make it available inside the container, use the following command:

```
mkdir /root/sysfoo
docker container run  -idt --name vt03 -v /root/sysfoo:/var/lib/mysql
alpine sh
docker inspect vt03 | grep -i mounts -A 10
```

Sharing files between host and the container:

```
ls /root/sysfoo/
touch /root/sysfoo/file1
docker exec -it vt03 sh
ls sysfoo/
```

Summary

In this lab, you just learned how to provide persistent storage to a container using three different types of volumes.

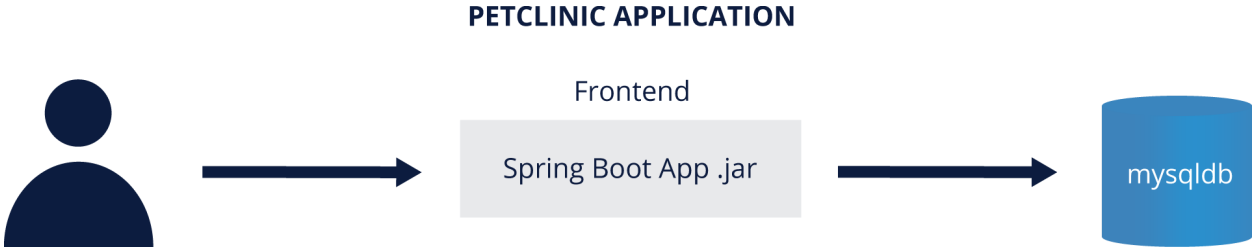
Lab 8. Automating Container Deployments with Compose

The objective of this lab exercise is to learn how to launch a collection of services for an application stack, and how to automate this process using Docker Compose. Here is what we will discuss:

- The process and the challenges of launching the application stack with more than one container using **`docker run`** command.
- How to automate that launch sequence using a simple, declarative interface which relies on YAML.
- How to create a composition of all services and connect them together using the inherent DNS-based service discovery that Docker offers along with Compose.
- What is the difference between Dockerfile and Docker Compose spec and how to tie them together.
- How to use Docker Compose to rapidly create and tear down disposable development environments.

Launching Application Stack

Let's say you have an application stack with more than one container to launch. Look at an example of the PetClinic application available on GitHub: [devopsdemoapps/spring-petclinic: A Sample Spring-based Application]. You may have already forked it and cloned it, so use your copy of the code for the exercise.



Running MySQL Database

Now, proceed to launch the application stack using **docker run** command. To launch a MySQL database, you will use a few environment variables. This will allow you to create the database, grant user permissions and have the frontend application connect to it later.

```
docker run -idt  -p 3306:3306 \
       -e MYSQL_ALLOW_EMPTY_PASSWORD=true \
       -e MYSQL_USER=petclinic \
       -e MYSQL_PASSWORD=petclinic \
       -e MYSQL_DATABASE=petclinic \
       mysql:5.7
```

Find out the container ID for MySQL:

```
docker ps
```

Sample output:

```
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                                NAMES
37c49c92bb1c        mysql:5.7                "docker-entrypoint.s…"   6
minutes ago         Up 6 minutes        0.0.0.0:3306->3306/tcp, 33060/tcp
exciting_bose
```

where,

- **37c49c92bb1c**: is the container ID. Please write it down as you will be referring to it later.

Launching Frontend Application Manually

The frontend application PetClinic is a Spring Boot application built with Maven which connects to an in-memory H2 database by default. You can also have it connect to the MySQL database launched earlier as a backend, by setting up profile to **spring.profiles.active=mysql**.

Launch the PetClinic application with MySQL profile:

```
docker run -idt -p 8080:8080 -e SPRING_PROFILES_ACTIVE=mysql   <docker
hub id>/petclinic:v4
```

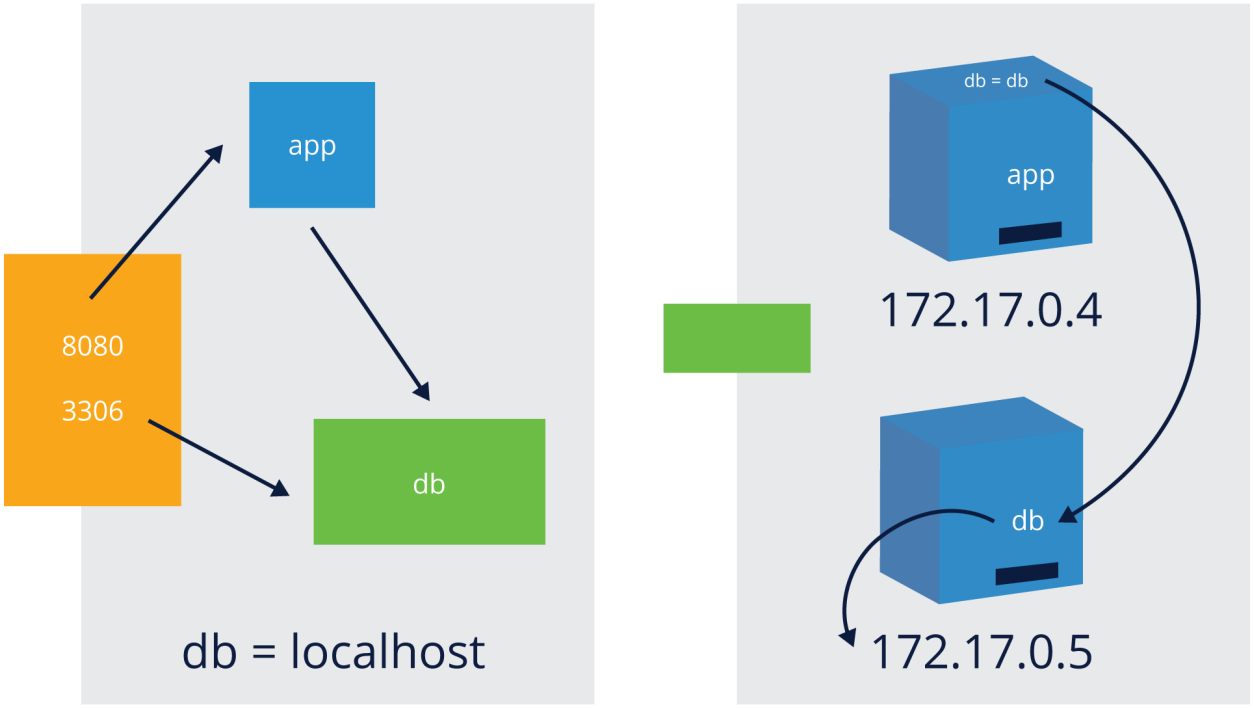
where,

- **v4**: the most up to date version of your image (change it to match the tag that you are using).

Find out the container ID and check the logs.

Did it work? Of course not! The reason for that is, the connection string configured in **`src/main/resources/application-mysql.properties`** is currently pointing to the **`localhost`**:

```
spring.datasource.url=${MYSQL_URL:jdbc:mysql://localhost/petclinic}
```



The above diagram illustrates two possibilities, with and without containers.

- If you run a db and an app on the same host (e.g. directly on your host), you can use **`localhost`** and connect to the database running on the same host.
- When you launch an app and a db as containers running on the same host, each of these containers will have a network stack. This means that when you point to the **`localhost`** in the connection string, it's actually pointing to the app container itself. Instead, you should be pointing to the hostname/IP address of the container which is running as a db.

One way to do this would be to use the **`--link`** option with **`docker run`** to provide a hostname alias (e.g. db) which would then point to the IP address of the db container, and use the same host alias in the connection string.

This is demonstrated below.

Update connection string inside **`application-mysql.properties`**.

First, edit the following file:

src/main/resources/application-mysql.properties

And update the connection string from:

```
spring.datasource.url=${MYSQL_URL:jdbc:mysql://localhost/petclinic}
```

to

```
spring.datasource.url=${MYSQL_URL:jdbc:mysql://db/petclinic}
```

where,

- **db**: hostname for the database.

This is a change inside the source code, which requires you to rebuild the image for this application. For example:

```
docker image build -f Dockerfile.multistage.v2 -t <docker hub id>/petclinic:v5 .
```

Now, launch the PetClinic application manually and link it with the MySQL database launched previously. Use the container ID that you wrote down earlier:

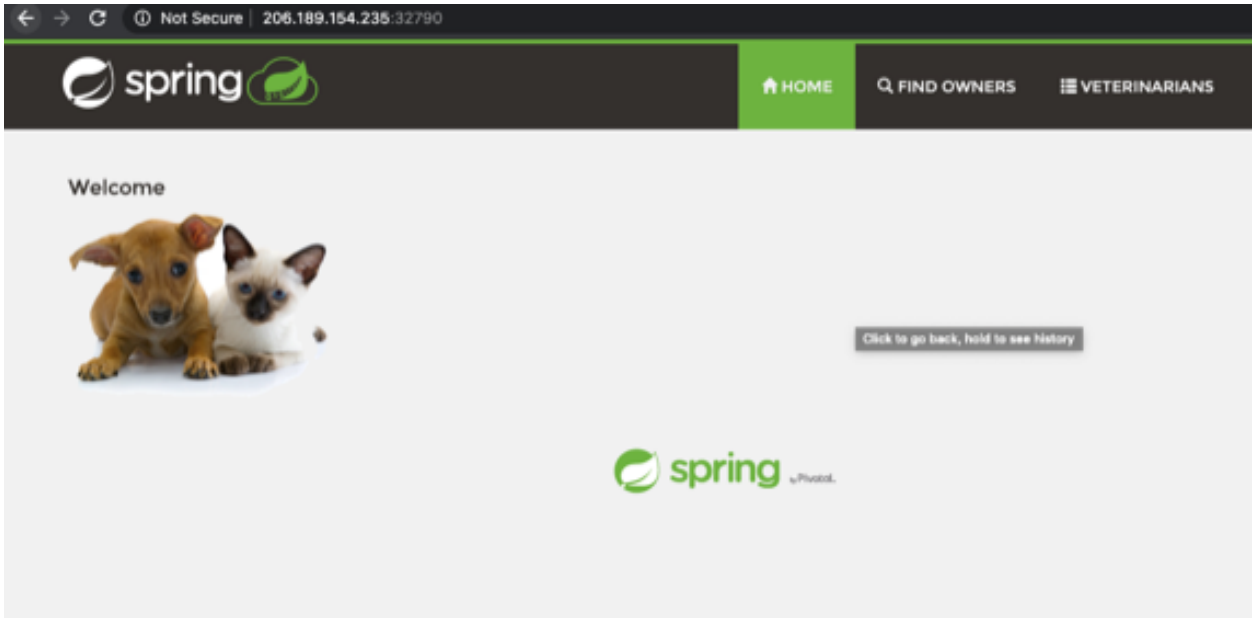
```
docker run -idt --name app -p 8080:8080 -e SPRING_PROFILES_ACTIVE=mysql --link 37c49c92bb1c:db  <docker hub id>/petclinic:v5
```

where,

- **37c49c92bb1c**: refers to the container which is running MySQL database.
- **db**: host alias added to **/etc/hosts** inside app container, pointing to the actual IP address of MySQL container.

Validate you are able to connect to the application using the 80 port on the host:

- http://localhost:8080 if you use Docker Desktop
- http://IPADDRESS:8080 if remote host



Once validated, you can delete the containers launched above using the following command:

```
docker rm -f 3d0e8ccbeeb6  37c49c92bb1c
```

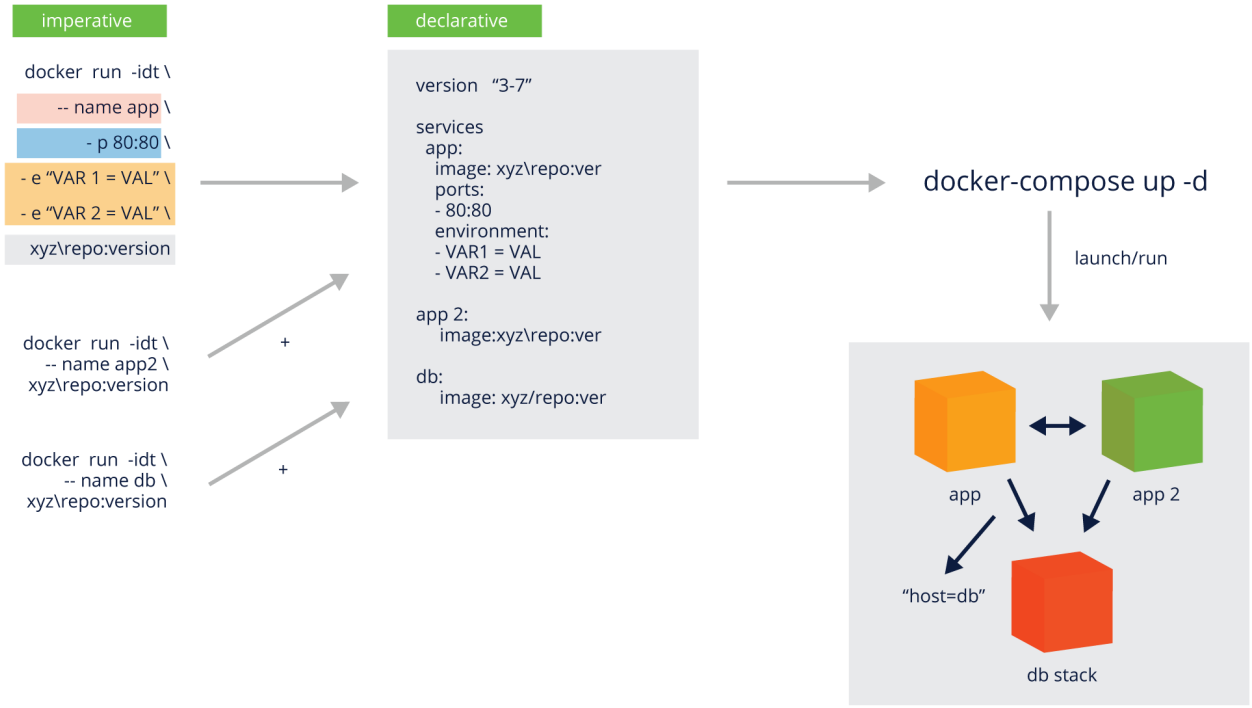
Make sure you replace **3d0e8ccbeeb6** and **37c49c92bb1c** with actual IDs of application and MySQL containers.

Composing Docker Services as a Code

You can manually launch the PetClinic application along with the database mentioned above, using a **docker run** command. Imagine you have a development team of 10 who is working on this app. Each of your teammates would have to set up this app locally to develop and test with. Here are some of the challenges that you may face:

1. Each one of you would have to remember how to launch multiple services, such as db and app. And as you start splitting this app into microservices, the list of applications you would have to launch is going to grow.
2. Also, you will either have to remember the **docker run** command with all its options, or rely on some documentation and keep track of it.
3. In addition, you may notice that during the launch of the application container above, you had to first write down the container ID of the database container, or even worse its IP address and then add it to the connection string/configurations for the app server. This creates an issue because every time you re-launch this stack, or one of your colleagues wants to replicate this setup, you will have to update this IP address. One way to solve it is to use an external service discovery, for example Consul, etcd, ZooKeeper, etc.

These reasons are good enough to consider Docker Compose. It solves all the listed issues and some more by allowing you to create a composition of services which need to work together. It also allows it to be defined as a code using a simple declarative syntax.



The same set of services that you launched earlier, e.g. app and db, can be converted into a compose file.

To better understand these concepts, watch video lessons first and try to create this file yourself. Use the following code only if you face issues and are absolutely unable to proceed.

file: spring-petclinic/docker-compose.yaml

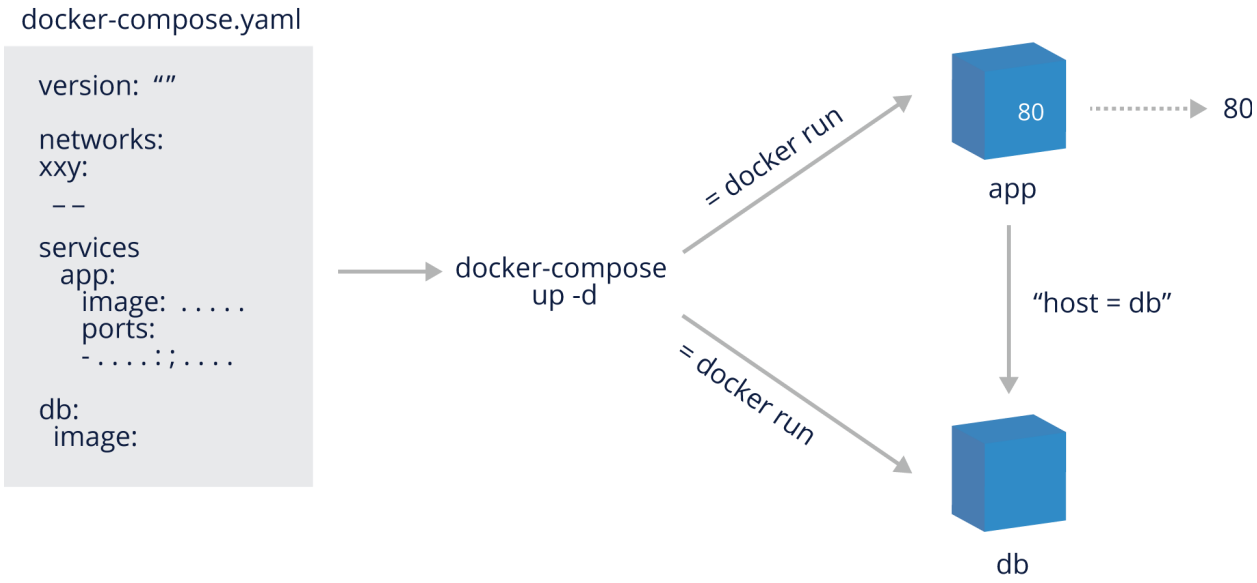
```
app:
  image: xxxxxx/petclinic:v5
  ports:
    - 8080:8080
  links:
    - db:db
  environment:
    - SPRING_PROFILES_ACTIVE=mysql
db:
  image: mysql:5.7
  ports:
    - "3306:3306"
  environment:
    - MYSQL_ROOT_PASSWORD=
```

```
    - MYSQL_ALLOW_EMPTY_PASSWORD=true
    - MYSQL_USER=petclinic
    - MYSQL_PASSWORD=petclinic
    - MYSQL_DATABASE=petclinic
```

***NOTE**: Replace **`xxxxxx/petclinic:v5`** with the actual tag.*

This **`docker-compose.yaml`** file is then fed to the **`docker compose`** utility, which reads it and launches your application stack:

```
cd spring-petclinic
```



Validate the syntax:

```
docker compose config
```

Check if any services launched with this Compose spec (should return blank for the first run):

```
docker compose ps
```

Launch all services in Compose spec:

```
docker compose up -d
```

When you check again, it should now show your services:

```
docker compose ps
```

Check logs for a service named **`app`**:

```
docker compose logs app
```

***NOTE**: Whenever you run **`docker compose`**, ensure that you are in the same directory as **`docker-compose.yaml`**. If that is not the case, or if the name of your Compose file is other than **`docker-compose.yaml`**, for every command you run, you need to provide a relative/absolute path to the Docker Compose file using the **`-f`** option.*

Validate that your service is launched and you are able to access it on port **`8080`** of the host using your browser.

If you get an error related to port conflicts, you may need to delete the previous containers using that port, or update the port mapping (specifically host port) in the **`docker-compose.yaml`** file.

Let's analyze what's happening:

- When you run a **`docker compose config`**, it reads the **`docker-compose.yaml`** in the current directory and checks for any syntactical errors. If errors are found, it shows the line number to help you debug.
- **`docker compose up -d`** launches containers for all applications defined in the **`services`** section. It does that in the detached mode using option **`-d`**, which is similar to the **`docker run`** option. Try running **`docker compose up`** without this option to see what happens.
- **`docker compose ps`** lists only the container specified in the Compose file. The container is prefixed with the directory name, e.g. **`spring-petclinic-app_1`** which allows **`docker compose`** to uniquely identify the containers launched from a specific path. This also means that you can use the same code and launch another instance of this stack from a different path.
- **`docker compose logs app`** will show you the logs from the container created for **app** service. If you do not provide the name of the service, consolidated log for all services will be shown.

Find out a list of sub-commands that Docker Compose supports using the following command:

```
docker compose
```

Play around with these sub-commands and learn what they do.

Refactoring Compose Spec with Version 3

What you have created earlier is version 1 of the Docker Compose spec, which has undergone significant changes over the years. Nowadays, you can add networks and volumes along with many advanced configurations, and luckily it's possible to refactor Docker Compose with version 3 specs by doing:

```
version: "3.8"

networks:
```

```
  frontend:
    driver: bridge
  backend:
    driver: bridge

services:
  app:
    image: xxxxxx/petclinic:xx
    ports:
      - 8080:8080
    environment:
      - SPRING_PROFILES_ACTIVE=mysql
    networks:
      - frontend
      - backend
    depends_on:
      - db

  db:
    image: mysql:5.7
    environment:
      - MYSQL_ROOT_PASSWORD=
      - MYSQL_ALLOW_EMPTY_PASSWORD=true
      - MYSQL_USER=petclinic
      - MYSQL_PASSWORD=petclinic
      - MYSQL_DATABASE=petclinic
    networks:
      - backend
```

where,

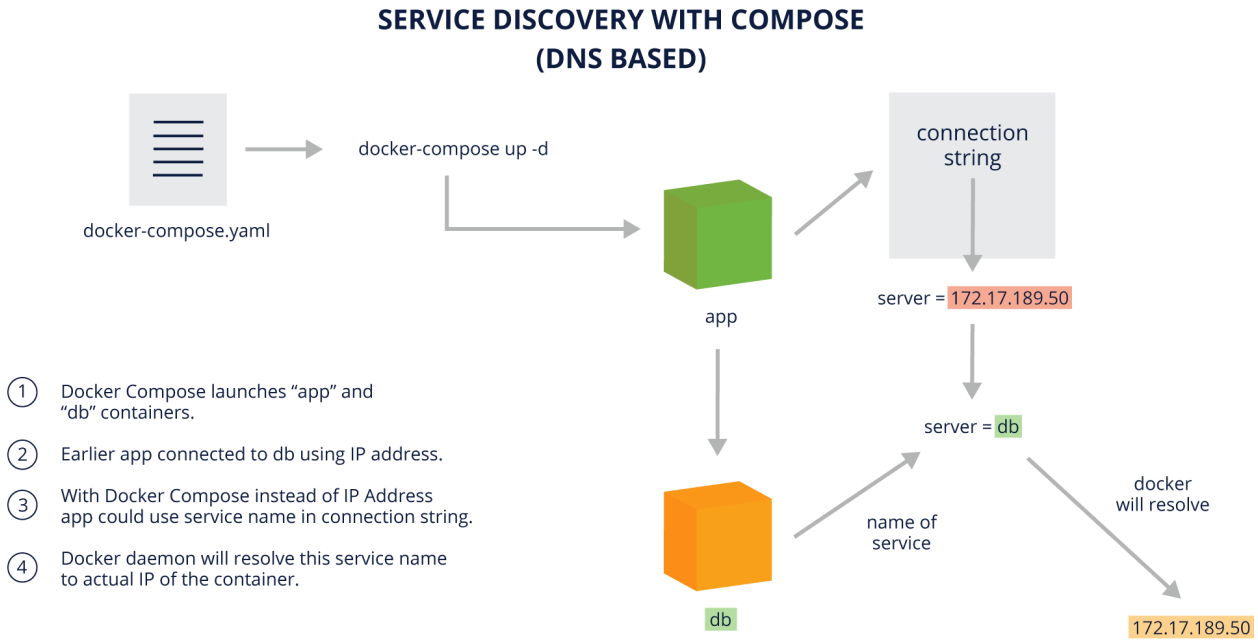
- **`version`**: defines the Docker Compose spec version. Reference official Docker documentation to learn more about the syntax, *"Compose File"*.
- **`services`**: allow you to decide which applications/containers you should launch as well as define the list of services/microservices. Each service contains a code block which is nothing but a **`docker run`** command converted into a declarative code using YAML. It's like asking, what properties this container is created with.
- **`networks`**: define custom network configurations. This can then be referenced in the **`service`** configurations in the subsequent section.

***NOTE**: You may notice port mapping removed for the db service. This is because you define the port mapping only for the services which need to be accessed from outside the Docker host. In this example, the only entity that needs to access db is the application server. Since the app container will be running on the same host and can access all ports of the db container, there is*

no need for port mapping to be defined here. This is also an advantage from the security point of view as you are reducing the attached surface of the host, by not exposing this db service.

Service Discovery

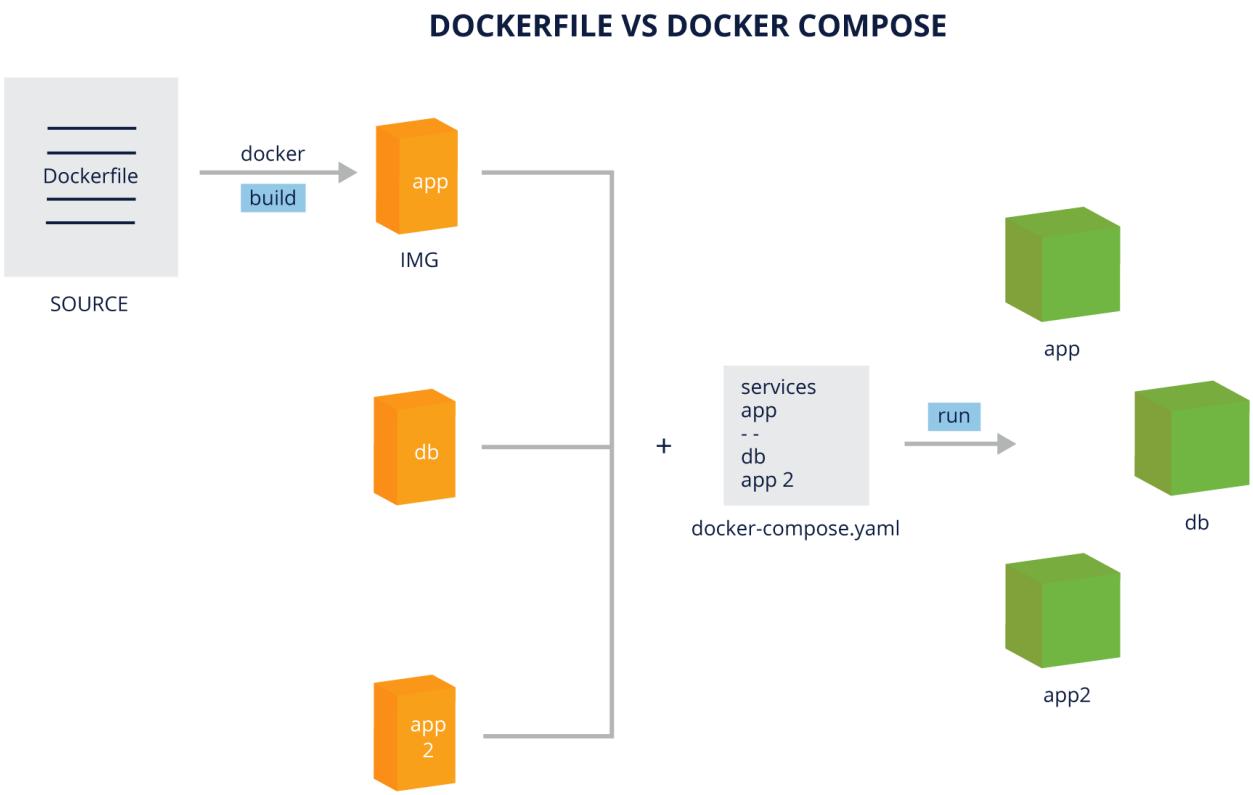
Did you notice a link was removed from the app service? You may see that it's also using db as the hostname for the database in the **`applications-mysql.properties`** file. Where does it come from? You may have already figured that out. It's the name of the service defined to launch the MySQL container. What does this mean? When you start using **`docker compose`** v3, you get service discovery built-in. That's because Docker daemon comes with a DNS service which is available while using Compose.



Dockerfile and Docker Compose

Both Dockerfile and **`docker-compose.yaml`** offer a declarative interface, and allow us to write code and automate processes with containerization. Now, you may wonder, what's the difference between the two, and why are they important?

The answer to these questions lies in the *when* part. Dockerfile is used during the image build process. And once you have the images built for all your applications, this is where Docker Compose comes in helping in the launch process.



You can also integrate Dockerfile with **`docker-compose.yaml`** and have Compose build the images for you as well (which is a common practice). Let's learn how to integrate these two files.

Integrating Dockerfile with Docker Compose

Let's add the image build part to the app service definition:

```
file: docker-compose.yaml

version: "3.7"
networks:
  petclinic:
    driver: bridge

services:
  app:
    image: xxxxxx/petclinic:dev
    build:
      context: .
      dockerfile: Dockerfile.multistage.v2
    ports:
      - 8080:8080
    environment:
      - SPRING_PROFILES_ACTIVE=mysql
```

```
    networks:
      - petclinic
    depends_on:
      - db
  db:
    image: mysql:5.7
    environment:
      - MYSQL_ROOT_PASSWORD=
      - MYSQL_ALLOW_EMPTY_PASSWORD=true
      - MYSQL_USER=petclinic
      - MYSQL_PASSWORD=petclinic
      - MYSQL_DATABASE=petclinic
    networks:
      - petclinic
```

Now, after every code change, to build an image you can run:

```
docker compose build
```

This will:

- Run **`docker build`** using options provided in the Compose spec.
- Tag the image with the image tag provided in the Compose files's **`service.app.image`** spec.

To use this newly built image to launch/recreate the application container, run:

```
docker compose up -d
```

Using Docker Compose to Deploy to Dev

Following are the two "I's" of a Docker Compose deployment:

1. *Idempotent*: meaning, you can run **`docker compose up -d`** multiple times. If there are no changes in the images/code, it will not redeploy. However, as soon as it detects the image has been updated, it will redeploy.
2. *Immutable*: each time **`docker compose`** updates the application, it throws away the old container, and replaces it with a new one created using the updated image. This also means that any changes made directly to the previous container are going to be lost. Immutable deployments have many advantages though, mainly related to maintaining the consistency across all instances of the application in an environment.

These, along with the inherent nature of it being an automation tool relying on declarative code, makes Docker Compose suitable as a deployment tool for development environments. Why do I say dev environments? It's because Docker Compose is still limited to be run on one host. It cannot launch containers crossing the boundary of the host it's running on, and spanning across

multiple hosts. If you need that, for example, to operate in higher environments such as staging and production, you have to consider a Container Orchestration Engine (COE). And yes, that's where you will start using Kubernetes. In essence, Kubernetes is the platform which takes your containers beyond one host and helps you simplify deploying and managing those containers on multiple hosts (thousands in some cases).

In this chapter though, we will focus on how to use Docker Compose as a deployment tool to dev. You can also use it along with your Continuous Integration platform such as Jenkins to set up integrated development environments which more than one developer can use, and to do some validation testing.

To see how this works, launch a stack using **`docker compose`**:

```
docker compose build
docker compose up -d
```

If you have already launched the stack with Compose, you would notice it does not change anything and shows all services being *up-to-date*. In fact, even if you have launched it for the first time, try running the above command a few times. This will help you understand what *idempotence* is all about. Docker Compose has the intelligence to know when to take action and, more importantly, when not to.

Now, update the source code to:

`file: src/main/resources/messages/messages.properties`

Change the title from:

```
welcome=Welcome
```

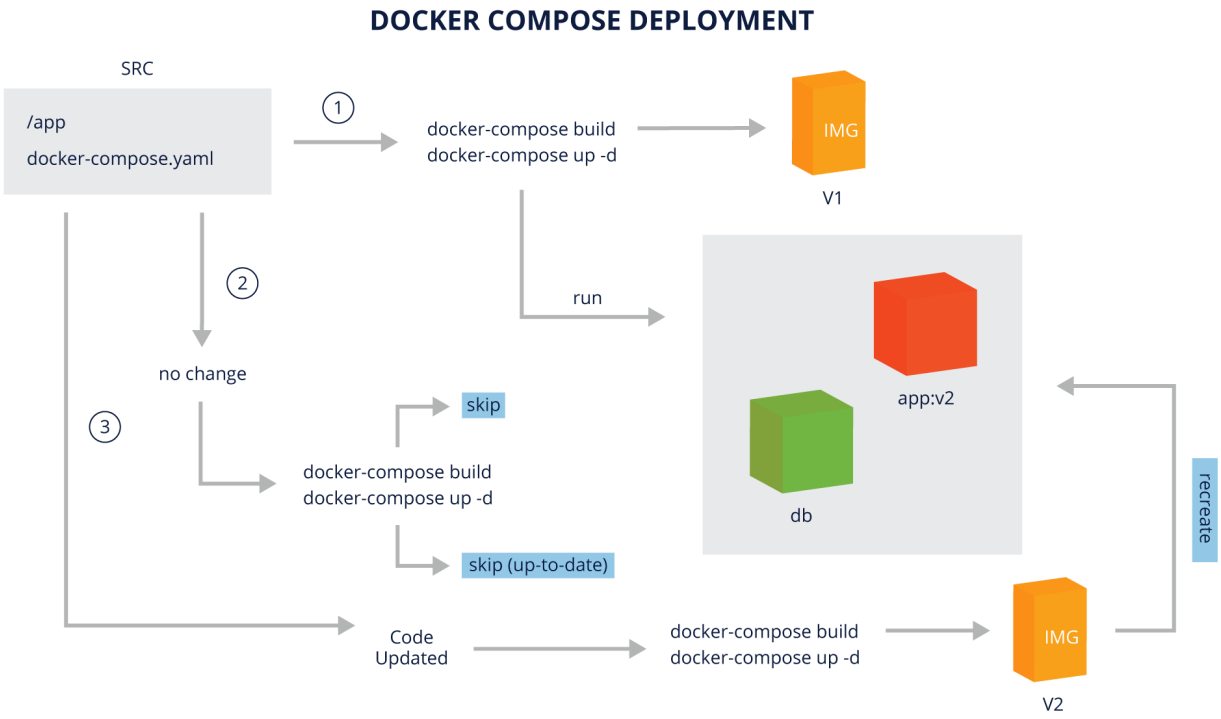
To:

```
welcome=Welcome to Petclinic
```

And then redeploy this application with:

```
docker compose build
docker compose up -d
```

What happened?



- Database is not changed, so it says *up-to-date* for a db service.
- Application source was updated. Hence the Docker build process detects this change and this time launches a new image build.
- A new image is built and tagged with the same tag as earlier. This means the previous image is now untagged, orphaned and will be deleted when **`docker image prune`** is run.
- Also, the container for the app is recreated. It's an immutable deployment since the previous container is deleted and a new one launched using the newly built image.

As a result of the new deployment, if you refresh the browser tab, you should now see an updated welcome message on the home page.

Tear Down the Stack

Docker Compose is not only useful to quickly launch a dev environment, but also to tear it down and clean it up.

To stop all the services defined in the **`docker-compose.yaml`**, run:

```
docker compose stop
```

***NOTE**: Whenever you run Docker Compose, ensure that you are in the same directory as **`docker-compose.yaml`**. If that is not the case, or if the name of your compose file is different than **`docker-compose.yaml`**, you need to provide a relative/absolute path to the Docker Compose file using the **`-f`** option, for every command you run. We emphasize this again as this is a very common mistake that people make.*

Services that are stopped can be started back again (with the same containers) either selectively by providing a service name (e.g. db) or all together by omitting the service names. Both commands are depicted below:

```
docker compose start db
docker compose start
```

FInally, to tear down this environment as rapidly as you started it, use either of the following commands.

For stopped services:

```
docker compose rm
```

For running services:

```
docker compose down
```

docker compose down is a combination of **docker compose stop** and **docker compose rm** commands. These commands remove the trace of services you had launched (except for the images pulled), which can be very useful if you want to reuse the same host to launch another application, work with it and tear it down the same way later.

Summary

In this lab, you learned how to compose more than one service to be launched together by using a simple declarative YAML interface. You also learned how to use this code to quickly set up and tear down an application stack, and use this to deploy to development environments.

Lab 11A. Pods

In this lab, you will learn how to launch applications using the basic deployment unit of Kubernetes, i.e. *pods*. This time, you are going to do it by writing declarative configs with *YAML* syntax.

Launching Pods with Kubernetes

Downloading Helper Code

To download the helper code for this course, switch to the workstation which has been configured with **`kubectl`** and clone the **`k8s-code`** repo:

```
git clone https://github.com/LFD254/k8s-code.git
```

This is a very important step. Do not miss it!

Launching Pods Manually

You can use **`kubectl run`** to launch a pod by specifying just the image.

For example, if you want to launch a pod for Redis, with an image **`redis:alpine`**, you should use the following command:

```
kubectl run redis  --image=redis
```

Kubernetes Resources and Writing YAML Specs

Each entity created with Kubernetes is a resource including pod, service, deployments, replication controller, etc. Resources can be defined as YAML or JSON. Here is the syntax to create a YAML specification:
AKMS > Resource Configs Specs

```
apiVersion: v1
kind:
metadata:
spec:
```

Use this Kubernetes API Reference Document to write the API specs. This is the most important reference while working with Kubernetes, so it's highly recommended that you bookmark it for the version of Kubernetes that you use.

To find the version of Kubernetes use the following command:

```
kubectl version -o yaml
```

To list the running pods:

```
kubectl get pods
```

To list API objects, use the following command:

```
kubectl api-resources
```

Writing Pod Spec

Let's now create the pod config by adding **kind** and **specs** to the schema given in the file **vote-pod.yaml** as follows:

Filename: **k8s-code/pods/vote-pod.yaml**

```
apiVersion:
kind: Pod
metadata:
spec:
```

Problem statement: Create a YAML spec to launch a pod with one container and to run a **vote** application, which matches the following specs:

```
pod:
 metadata:
    name: vote
    labels:
        app: python
        role: vote
        version: v1
spec
   containers:
    name: app
    image: schoolofdevops/vote:v1
```

Refer to the Kubernetes API Reference Pod v1 Core Document to find out the relevant properties of a pod and add those to the **vote-pod.yaml** provided in the supporting code repository.

Filename: **k8s-code/pods/vote-pod.yaml**

```
apiVersion: v1
kind: Pod
metadata:
  name: vote
  labels:
    app: python
    role: vote
    version: v1
spec:
  containers:
    - name: app
      image: schoolofdevops/vote:v1
```

Use this example to refer to the pod spec.

Launching and Operating Pods

To launch a monitoring screen and see what's being launched, use the following command in a new terminal window where **kubectl** is configured:

```
watch kubectl get all
```

kubectl syntax:

```
kubectl
kubectl apply --help
kubectl apply -f FILE
```

To do a dry run before applying use the following command:

```
kubectl apply -f vote-pod.yaml --dry-run=client
```

To *launch* a pod using configs above, remove dry run options and run:

```
kubectl apply -f vote-pod.yaml
```

To *view* pods:

```
kubectl get pods
kubectl get po
kubectl get pods -o wide
kubectl get pods --show-labels
```

```
kubectl get pods -l "role=vote,version=v1"
kubectl get pods vote
```

To get detailed info:

```
kubectl describe pods vote
```

To *operate* the pod:

```
kubectl logs vote
kubectl exec -it vote sh
```

Run the following commands inside the container in a pod after running **`exec`** command:

```
ifconfig
cat /etc/issue
hostname
cat /proc/cpuinfo
ps aux
```

Use **`^d`** or **`exit`** to log out.

Adding Volume for Data Persistence

Let's create a pod for the database and attach a volume to it. To achieve this we will need to:

- Create a **volumes** definition
- Attach volume to container using **`VolumeMounts`** property

Localhost volumes are of two types:

- **`emptyDir`**
- **`hostPath`**

We will select hostPath (to learn more about **`hostPath`** read Kubernetes Documentation).

File: **`db-pod.yaml`**

```
apiVersion: v1
kind: Pod
metadata:
  name: db
  labels:
    app: postgres
    role: database
    tier: back
spec:
```

```
  containers:
    - name: db
      image: postgres:9.4
      ports:
        - containerPort: 5432
      volumeMounts:
      - name: db-data
        mountPath: /var/lib/postgresql/data
  volumes:
  - name: db-data
    hostPath:
      path: /var/lib/pgdata
      type: DirectoryOrCreate
```

To create this pod:

```
kubectl apply -f db-pod.yaml
kubectl get pods
```

At this time, you may see that the pod is in **CrashLoopBackOff** state. Try debugging the issue by checking the logs and resolving it before proceeding.

Once resolved, confirm that the pod is in a running state:

```
kubectl get pods
kubectl describe pod db
kubectl get events
```

Examine **/var/lib/pgdata** on the node on which it's scheduled by running **kubectl get pods -o wide to find the node**. Check if the directory has been created and if the data is present.

Creating Multi-container Pods (Sidecar Example)

File: **multicontainerpod.yaml**

```
apiVersion: v1
kind: Pod
metadata:
  name: web
  labels:
    tier: front
    app: nginx
    role: ui
spec:
  containers:
```

```
    - name: nginx
      image: nginx:stable-alpine
      ports:
        - containerPort: 80
          protocol: TCP
      volumeMounts:
        - name: data
          mountPath: /var/www/html-sample-app

    - name: sync
      image: schoolofdevops/sync:v2
      volumeMounts:
        - name: data
          mountPath: /var/www/app

  volumes:
    - name: data
      emptyDir: {}
```

To create this pod:

```
kubectl apply -f multi_container_pod.yaml
```

Check status:

```
root@kube-01:~# kubectl get pods
NAME      READY     STATUS              RESTARTS   AGE
web       0/2       ContainerCreating   0          7s
vote      1/1       Running             0          3m
```

Check logs and log in:

```
kubectl logs  web  -c sync
kubectl logs  web  -c nginx

kubectl exec -it web  sh  -c nginx
kubectl exec -it web  sh  -c sync
```

Using the following commands, observe which elements are common and which are isolated in two containers running inside the same pod.

Shared:
```
hostname
ifconfig
```

Isolated:

```
cat /etc/issue
```

```
ps aux
df -h
```

Adding Resource Requests and Limits

We can control the amount of resources requested and also put a limit on how many resources a container in a pod can take on. This can be done by adding spec to the existing pod as presented below. Refer to the Kubernetes Documentation, *"Managing Resources for Containers"*, to learn more.

Filename: **vote-pod.yaml**

```
apiVersion: v1
kind: Pod
metadata:
  name: vote
  labels:
    app: python
    role: vote
    version: v1
spec:
  containers:
    - name: app
      image: schoolofdevops/vote:v1
      resources:
        requests:
          memory: "64Mi"
          cpu: "50m"
        limits:
          memory: "128Mi"
          cpu: "250m"
```

Let's now apply the changes:

```
kubectl apply -f vote-pod.yaml
```

If you already have a **vote** pod running, you may see an output similar to this presented below:

```
The Pod "vote" is invalid: spec: Forbidden: pod updates may not change
fields other than `spec.containers[*].image`,
`spec.initContainers[*].image`, `spec.activeDeadlineSeconds` or
`spec.tolerations` (only additions to existing tolerations)
{"Volumes":[{"Name":"default-token-snbj4","HostPath":null,"EmptyDir":n
ull,"GCEPersistentDisk":null,"AWSElasticBlockStore":null,"GitRepo":nul
l,"Secret":{"SecretName":"default-token-snbj4","Items":null,"DefaultMo
de":420,"Optional":null},"NFS":null,"ISCSI":null,"Glusterfs":null,"Per
```

```
sistentVolumeClaim":null,"RBD":null,"Quobyte":null,"FlexVolume":null,"
Cinder":null,"CephFS":null,"Flocker":null,"DownwardAPI":null,"FC":null
,"AzureFile":null,"ConfigMap":null,"VsphereVolume":null,"AzureDisk":nu
ll,"PhotonPersistentDisk":null,"Projected":null,"PortworxVolume":null,
"ScaleIO":null,"StorageOS":null}],"InitContainers":null,"Containers":[
{"Name":"app","Image":"schoolofdevops/vote:v1","Command":null,"Args":n
ull,"WorkingDir":"","Ports":null,"EnvFrom":null,"Env":null,"Resources"
:{"Limits":
....
...
```

From the above output, it's clear that not all the fields are mutable (except for a few, e.g labels). Container-based deployments primarily follow the concept of **immutable deployments**. So to bring your change into effect, you need to re-create the pod:

```
kubectl delete pod vote
kubectl apply -f vote-pod.yaml
kubectl describe pod vote
```

Based on the output of the `describe` command, you can confirm that the resource constraints you added are in place.

Now:

- Define the value of `cpu.request` > `cpu.limit`. Try to apply the changes made to the manifests, and observe.
- Define the values for `memory.request` and `memory.limit` that are higher than the total system memory. Apply the changes made to the manifest and observe the deployment and pods.

Deleting Pods

Now that you are done experimenting with a pod, delete it with the following command:

```
kubectl delete pod vote web db
kubectl get pods
```

Additional Resources

- *"Types of Volumes"*
- *"Assigning Pods to Nodes"*

Summary

In this lab, you learned how to create, manage and work with the most fundamental Kubernetes primitive: pod. You have also begun your journey writing YAML manifests, which is an essential

skill to master Kubernetes. You also learned how to use kubectl to create, update and manage resources. And finally, you explored the most common issues with the pods and how to troubleshoot those.

Lab 11B. Namespaces, ReplicaSets

In this lab exercise, you are going to learn:

- How to create a Kubernetes namespace and switch to it.
- How to switch Kubernetes contexts.
- How to create a ReplicaSet.
- How to achieve high availability and scalability with ReplicaSets.

Namespaces

Check current config:

```
kubectl config view
```

You can also examine the current configs in the `cat ~/.kube/config` file.

Creating a Namespace and Switching to It

Namespaces separate resources running on the same physical infrastructure into virtual clusters. It is typically useful in mid to large scale environments with multiple projects and teams, and need separate scopes. It can also be useful to map to your workflow stages, e.g. dev, stage, prod.

Before you create a namespace, delete all the pods in the default namespace that you may have created earlier and are no longer needed.

Lets create a namespace called `instavote`:

```
kubectl get ns
kubectl create namespace instavote
kubectl get ns
```

And switch to it:

```
kubectl config --help
kubectl config get-contexts
kubectl config current-context
kubectl config set-context --help
kubectl config set-context --current --namespace=instavote
kubectl config get-contexts
kubectl config view
```

Go back to the monitoring screen and observe what happens after switching the namespace. For example:

```
kubectl config set-context --current --namespace=default
kubectl config set-context --current --namespace=kube-system
kubectl config set-context --current --namespace=instavote
```

ReplicaSets

To understand how ReplicaSet works with the selectors, let's launch a pod in the new namespace with existing spec.

```
cd k8s-code/pods
kubectl apply -f vote-pod.yaml

kubectl get pods
```

Adding ReplicaSet Configurations

Let's now write spec for the ReplicaSet. This is going to mainly contain:

- Replicas: define the scalability configs here
- Selectors: define configs which provide a base for checking availability
- Template (pod spec)
- **`minReadySeconds`**: duration to wait after pod is ready till its declared as available (used by Deployment)

From here on, we would switch to the project and environment specific path and work from there.

```
cd projects/instavote/dev
```

Edit file: **`vote-rs.yaml`**

```
apiVersion: xxx
kind: xxx
metadata:
  xxx
spec:
```

```
  xxx
  template:
    metadata:
      name: vote
      labels:
        app: python
        role: vote
        version: v1
    spec:
      containers:
        - name: app
          image: schoolofdevops/vote:v1
          resources:
            requests:
              memory: "64Mi"
              cpu: "50m"
            limits:
              memory: "128Mi"
              cpu: "250m"
```

The above file already contains spec that you had written for the pod. You can see it has already been added as part of **spec.template** for ReplicaSet.

Let's now add the details specific to ReplicaSet.

File: **vote-rs.yaml**

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: vote
spec:
  replicas: 4
  minReadySeconds: 20
  selector:
    matchLabels:
      role: vote
    matchExpressions:
      - key: version
        operator: Exists
  template:
    metadata:
      name: vote
      labels:
        app: python
```

```
        role: vote
        version: v1
    spec:
      containers:
        - name: app
          image: schoolofdevops/vote:v1
          resources:
            requests:
              memory: "64Mi"
              cpu: "50m"
            limits:
              memory: "128Mi"
              cpu: "250m"
```

The complete file will look similar to the one above. Let's now go ahead and apply it.

```
kubectl apply -f vote-rs.yaml --dry-run=client
kubectl apply -f vote-rs.yaml
kubectl get rs
kubectl describe rs vote
kubectl get pods
kubectl get pods --show-labels
```

High Availability

Try to delete pods created by ReplicaSet. Replace **pod-xxxx** and **pod-yyyy** with actual names of the pods in your environment:

```
kubectl get pods
kubectl delete pods vote-xxxx vote-yyyy
```

Observe as pods are automatically created again.

Let's now delete the pod created independent of ReplicaSet:

```
kubectl get pods
kubectl delete pods vote
```

Does ReplicaSet take any action after deleting the pod created outside of its spec? If so, why?

Scalability

Scaling out and scaling in your application is as simple as running:

```
kubectl scale rs vote --replicas=8
kubectl get pods --show-labels
```

```
kubectl scale rs vote --replicas=25
kubectl get pods --show-labels

kubectl scale rs vote --replicas=7
kubectl get pods --show-labels
```

Check if the number of replicas:

- Increased to 8
- Increased further to 25
- Dropped to 7

Deploying New Version of the Application

```
kubectl edit rs vote
```

- Update the version of the image from **schoolofdevops/vote:v1** to **schoolofdevops/vote:v2**
- Update **template.metadata.labels** from **version=v1** to **version=v2**

Save the file. If you are using **kubectl edit**, it will apply changes immediately as you save them. There is no need to use the **kubectl apply** command.

Did the application get updated? Did updating ReplicaSet launch new pods to deploy new versions?

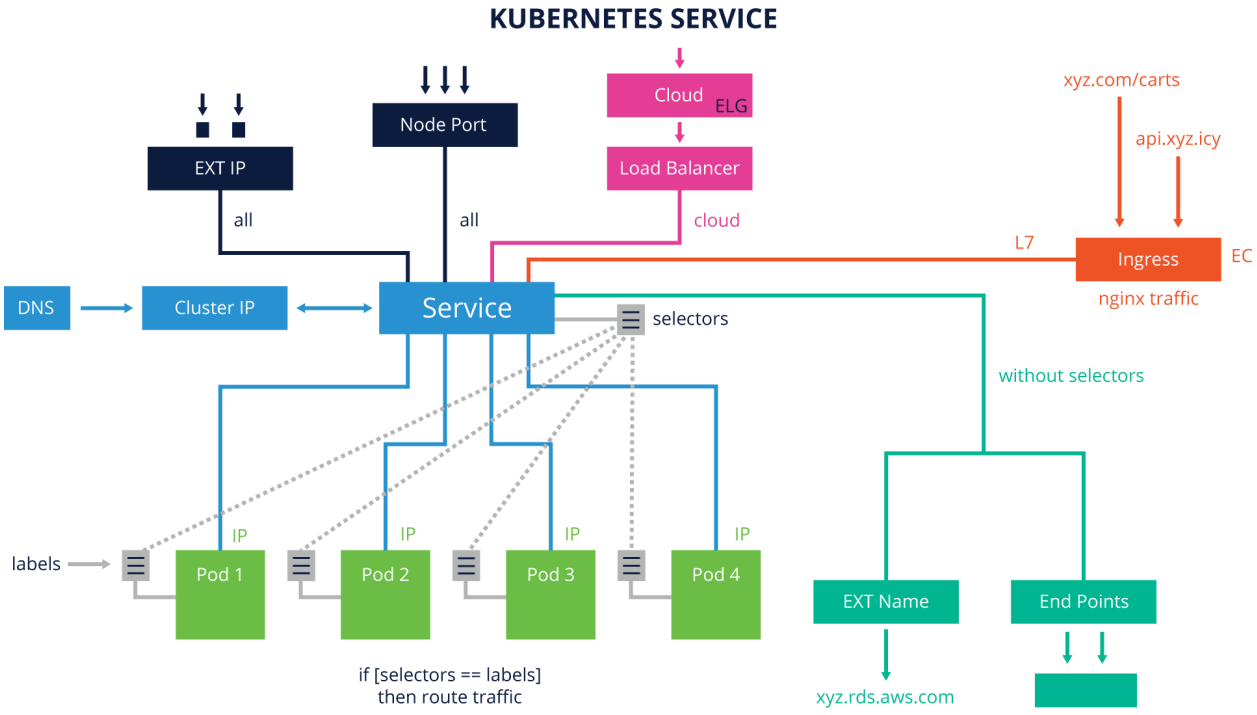
Summary

Your application is highly available as well as scalable with ReplicaSets. However, ReplicaSet by itself does not have the intelligence to trigger a rollout if you update the version. For that, you are going to need a *deployment* which is something you will learn about in the next chapter.

Lab 12A. Service Networking

In this lab, you will not only publish the application deployed earlier with ReplicaSet, but also learn about the load balancing and service discovery features offered by Kubernetes.

Concepts related to Kubernetes services are depicted in the following diagram:



Publishing External-facing Application with NodePort

Kubernetes comes with four types of services:

- ClusterIP
- NodePort

- LoadBalancer
- ExternalName

Let's create a NodePort service to understand how it works.

To check the status of Kubernetes objects:

```
kubectl get all
```

You can also start watching the output of this command for changes. To do so, open a separate terminal window and run:

```
watch kubectl get all
```

Refer to Service Specs to understand the properties that you can write.

Filename: **vote-svc.yaml**

```
---
apiVersion: v1
kind: Service
metadata:
  name: vote
  labels:
    role: vote
spec:
  selector:
    role: vote
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30000
  type: NodePort
```

Apply this file to to create a service:

```
kubectl apply -f vote-svc.yaml --dry-run
kubectl apply -f vote-svc.yaml
kubectl get svc
kubectl describe service vote
```

Below you can see a sample output of the **describe** command:

```
Name:                     vote
Namespace:                instavote
Labels:                   role=svc
                          tier=front
```

```
Annotations:
kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"v1","k
ind":"Service","metadata":{"annotations":{},"labels":{"role":"svc","ti
er":"front"},"name":"vote","namespace":"instavote"},"spec":{...
Selector:                 app=vote
Type:                     NodePort
IP:                       10.108.108.157
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  31429/TCP
Endpoints:                10.38.0.4:80,10.38.0.5:80,10.38.0.6:80 + 2
more...
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

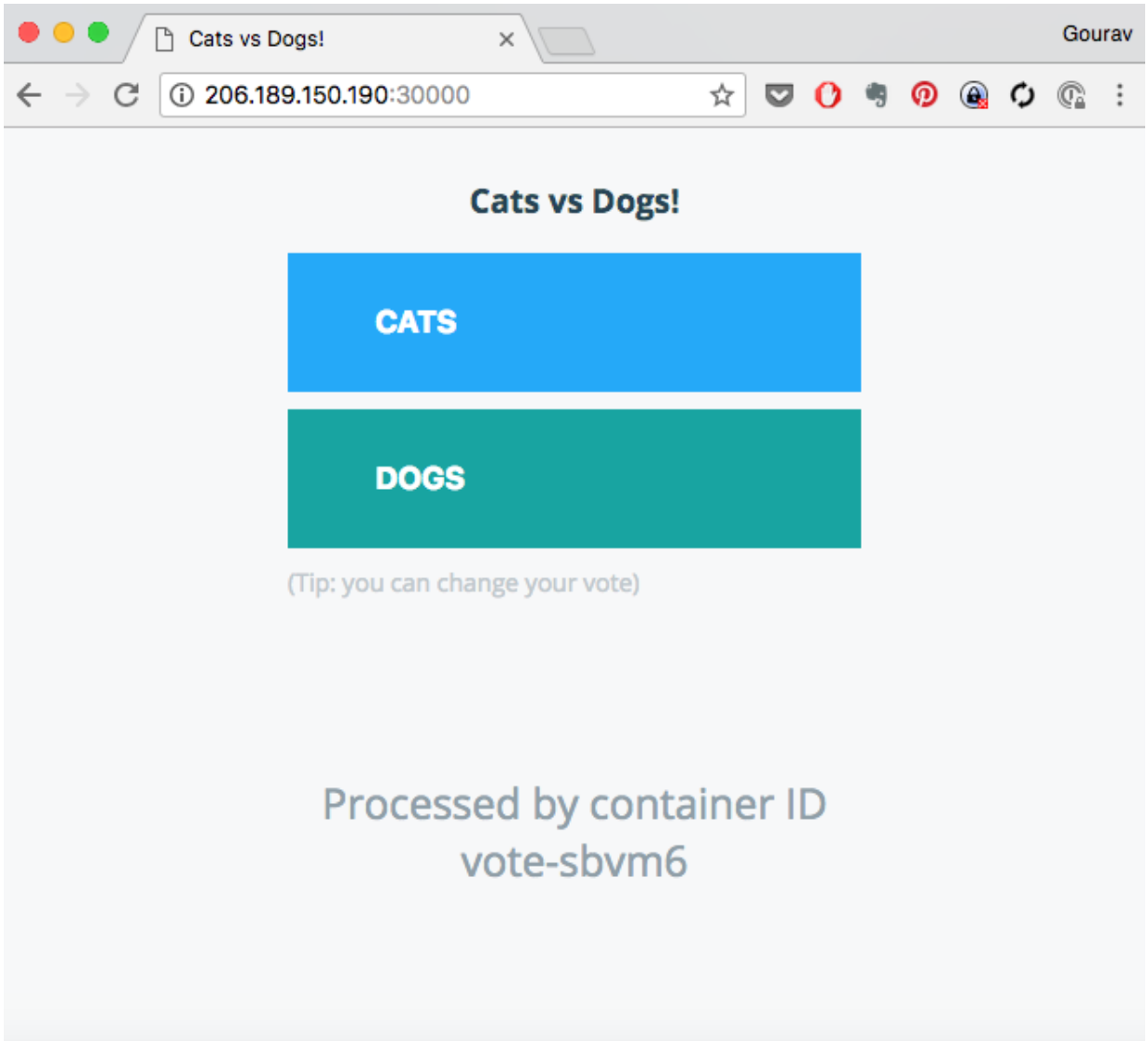
Observe the following elements:

- **Selector**
- **TargetPort**
- **NodePort**
- **Endpoints**

Go to the browser and check http://HOSTIP:NODEPORT.

Here the node port is 30000 (as defined by NodePort in service spec).

Sample output will be:



If you refresh the page, you should also notice its sending traffic to a different pod each time, in a round-robin fashion.

Next:

- Change the selector criteria to use a non-existent label. Use **`kubectl edit svc./vote`** to update and apply the configuration. Observe the output of the **`describe`** command and check for the endpoints. Do you see any? How are selectors and pod labels related?
- Observe the number of endpoints. Then change the scale of replicas created by the ReplicaSets. Does it have any effect on the number of endpoints?

Services Under the Hood

Let's traverse the route of the network packet that comes in on port 30000, on any node in your cluster:

```
iptables -nvL -t nat
iptables -nvL -t nat  | grep 30000
```

Anything that comes on dpt:3000, gets forwarded to the chain created for that service.

```
iptables -nvL -t nat  | grep KUBE-SVC-VIQHAVHDK4QE7NA4  -A 10
```

```
Chain KUBE-SVC-VIQHAVHDK4QE7NA4 (2 references)
 pkts bytes target     prot opt in     out     source
destination
    0     0 KUBE-SEP-RFJGHFMXUDJXIEW6  all  --  *      *
0.0.0.0/0            0.0.0.0/0            /* instavote/vote: */
statistic mode random probability 0.20000000019
    0     0 KUBE-SEP-GBR5YQCVRYY3BA6U  all  --  *      *
0.0.0.0/0            0.0.0.0/0            /* instavote/vote: */
statistic mode random probability 0.25000000000
    0     0 KUBE-SEP-BAI3HQ7SV7RZ2CI6  all  --  *      *
0.0.0.0/0            0.0.0.0/0            /* instavote/vote: */
statistic mode random probability 0.33332999982
    0     0 KUBE-SEP-2EQSLPEP3WDOTI5J  all  --  *      *
0.0.0.0/0            0.0.0.0/0            /* instavote/vote: */
statistic mode random probability 0.50000000000
    0     0 KUBE-SEP-2CJQISP4W7F2HCRW  all  --  *      *
0.0.0.0/0            0.0.0.0/0            /* instavote/vote: */
```

where,

- the count of **KUBE-SEP-xxx** matches the number of pods.
- **KUBE-SEP-BAI3HQ7SV7RZ2CI6** is an example of a chain created for one of the hosts.

Examine that next:

```
iptables -nvL -t nat  | grep KUBE-SEP-BAI3HQ7SV7RZ2CI6  -A 3
```

Output:

```
pkts bytes target     prot opt in     out     source
destination
    0     0 KUBE-MARK-MASQ  all  --  *      *       10.32.0.6
0.0.0.0/0            /* instavote/vote: */
    0     0 DNAT       tcp  --  *      *       0.0.0.0/0
0.0.0.0/0            /* instavote/vote: */ tcp to:10.32.0.6:80
```

```
--
```

where the packet is being forwarded to 10.32.0.6, which should corroborate with the IP of the pod. For example:

```
kubectl get pods -o wide
```

```
NAME          READY     STATUS    RESTARTS   AGE       IP           NODE
vote-58bpv    1/1       Running   0          1h        10.32.0.6    k-02
vote-986cl    1/1       Running   0          1h        10.38.0.5    k-03
vote-9rrfz    1/1       Running   0          1h        10.38.0.4    k-03
vote-dx8f4    1/1       Running   0          1h        10.32.0.4    k-02
vote-qxmfl    1/1       Running   0          1h        10.32.0.5    k-02
```

10.32.0.6 matches IP of **vote-58bpv**.

To check how the packet is routed next, use:

```
ip route show
```

Output:

```
default via 128.199.128.1 dev eth0 onlink
10.15.0.0/16 dev eth0  proto kernel  scope link  src 10.15.0.10
10.32.0.0/12 dev weave  proto kernel  scope link  src 10.32.0.1
128.199.128.0/18 dev eth0  proto kernel  scope link  src
128.199.185.90
172.17.0.0/16 dev docker0  proto kernel  scope link  src 172.17.0.1
linkdown
```

where,

- 10.32.0.0/12 is going over the **weave** interface.

Exposing Service with External IPs

Observe the output of the following command which lists all the services in the current namespace. Specifically, note the **EXTERNAL-IP** column in the output.

```
kubectl get svc
```

Now, update the service spec and add external IP configs. Pick IP addresses of any two nodes (you can add one or more) and add them to the spec with:

```
kubectl edit svc vote
```

Sample file edit:

```
---
apiVersion: v1
kind: Service
metadata:
  name: vote
  labels:
    role: vote
spec:
  selector:
    role: vote
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30000
  type: NodePort
  externalIPs:
    - xx.xx.xx.xx
    - yy.yy.yy.yy
```

***NOTE**: Replace **`xx.xx.xx.xx`** and **`yy.yy.yy.yy`** with IP addresses of the nodes on two of the Kubernetes hosts.*

Apply:

```
kubectl  get svc
kubectl apply -f vote-svc.yaml
kubectl  get svc
kubectl describe svc vote
```

Sample output:

```
NAME      TYPE        CLUSTER-IP      EXTERNAL-IP
PORT(S)         AGE
vote      NodePort    10.107.71.204   206.189.150.190,159.65.8.227
80:30000/TCP    11m
```

where,

- **`EXTERNAL-IP`** column shows which IP the application has been exposed on. You can go to http://<IPADDRESS>:<SERVICE_PORT> to access this application. For example, in http://206.189.150.190:80 you should replace 206.189.150.190 with the actual IP address of the node that you exposed this on.

Internal Service Discovery

Kubernetes not only allows you to publish external-facing apps with the services, but also allows you to discover other components of your application stack with the clusterIP and DNS attached to it.

Before you begin adding service discovery:

- Visit the vote app from browser
- Attempt to vote by clicking on one of the options and observe what happens.

Did it go through?

For debugging, run:

```
kubectl get pod
kubectl exec vote-xxxx nslookup redis
```

Replace **xxxx** with the actual pod ID of one of the vote pods.

It is useful to watch the output of the above command so that you know when this service becomes available. You can create a new terminal to run the **watch** command to achieve this. For example:

```
kubectl exec -it vote-xxxx sh
watch  kubectl exec vote-xxxx ping redis
```

where,

- **vote-xxxx** is one of the vote pods that we are running. Replace this with the actual pod ID.

Now, create a Redis service:

```
kubectl apply -f redis-svc.yaml
kubectl get svc
kubectl describe svc redis
```

Watch the nslookup screen and check if it can resolve **redis** by hostname and by pointing to an IP address. For example:

```
Name:      redis
Address 1: 10.104.111.173 redis.instavote.svc.cluster.local
```

where,

- 10.104.111.173 is the ClusterIP assigned to the Redis service
- **redis.instavote.svc.cluster.local** is the DNS attached to the ClusterIP above

What's happened here?

- Service Redis was created with a ClusterIP, e.g. 10.102.77.6
- A DNS entry was created for this service. A fully qualified domain name (FQDN) of the service is **redis.instavote.svc.cluster.local** and it takes the form of **my-svc.my-namespace.svc.cluster.local**
- Each pod points to an internal DNS server running in the cluster. You can see the details of this by running the following commands:

```
kubectl exec vote-xxxx cat /etc/resolv.conf
```

Replace **vote-xxxx** with an actual pod ID.

Sample output:

```
nameserver 10.96.0.10
search instavote.svc.cluster.local svc.cluster.local cluster.local
options ndots:5
```

where,

- 10.96.0.10 is the ClusterIP assigned to the DNS service

You can co-relate that with:

```
kubectl get svc -n kube-system
```

```
NAME                    TYPE        CLUSTER-IP     EXTERNAL-IP
PORT(S)         AGE
kube-dns                ClusterIP   10.96.0.10     <none>
53/UDP,53/TCP   1h
kubernetes-dashboard    NodePort    10.104.42.73   <none>
80:31000/TCP    23m
```

where,

- 10.96.0.10 is the ClusterIP assigned to kube-dns and matches the configuration in **/etc/resolv.conf** mentioned before

Creating Endpoints for Redis

Service has been created, but you still need to launch the actual pods running the Redis application.

Create the endpoints:

```
kubectl apply -f redis-deploy.yaml
kubectl describe svc redis
```

Sample output:

```
Name:              redis
Namespace:         instavote
Labels:            role=redis
                   tier=back
Annotations:
kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"v1","k
ind":"Service","metadata":{"annotations":{},"labels":{"role":"redis","
tier":"back"},"name":"redis","namespace":"instavote"},"spec"...
Selector:          app=redis
Type:              ClusterIP
IP:                10.102.77.6
Port:              <unset>  6379/TCP
TargetPort:        6379/TCP
Endpoints:         10.32.0.6:6379,10.46.0.6:6379
Session Affinity:  None
Events:            <none>
```

Again, visit the **vote app** from the browser and attempt to register your vote. Notice what happens. This time your vote should be registered successfully.

Additional Resources

- *"Debug Services"*
- *"Service"*

Summary

In this lab, you have published a front-facing application, learned how services are implemented under the hood as well as added service discovery to provide connection strings automatically.

Lab 12B. Deployments

In this lab exercise, you are going to learn how to roll out a new version of the application using Kubernetes deployments. Specifically, you will learn:

- How deployments work and the two types of rollout strategies available with Kubernetes by default.
- How to write the deployment spec with a **`rollingUpdate`** as the strategy.
- How to roll out new versions of the application.
- How to rollback when necessary.

What Is a Deployment?

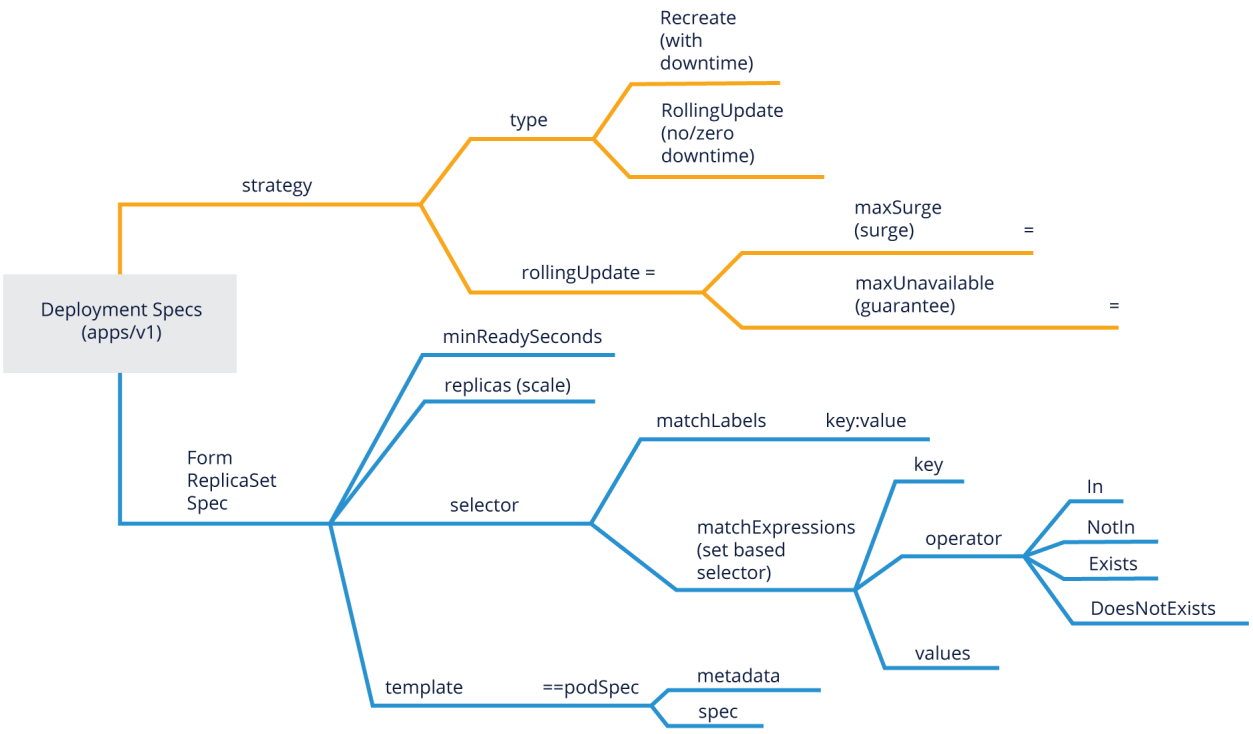
A deployment is a higher level abstraction which sits on top of ReplicaSets and allows you to manage the way applications are deployed and rolled back at a controlled rate.

Deployment provides three features:

Availability
- Maintains the number of replicas for a type of service/app
- Schedules/deletes pods to meet the desired count

Scalability
- Updates the replica count
- Allows you to scale in and out (you can scale manually or use **`horizontalPodAutoscaler`** to do it automatically)

Update strategy
- Defines a release strategy and updates the pods accordingly.



Begin by deleting the previously created ReplicaSet as the presence of it may lead to a duplicate set of pods:

```
kubectl delete rs vote
```

Since deployment is just a superset of ReplicaSet specs, let's create a deployment by copying over the existing ReplicaSet code for **`vote app`**:

```
/k8s-code/projects/instavote/dev/
cp vote-rs.yaml vote-deploy.yaml
```

The deployment spec (**`deployment.spec`**) contains everything that ReplicaSet has plus the strategy. Let's add it:

File: **`vote-deploy.yaml`**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vote
  labels:
    role: vote
spec:
  strategy:
    type: RollingUpdate
```

```
    rollingUpdate:
      maxSurge: 2
      maxUnavailable: 1
  revisionHistoryLimit: 4
  replicas: 12
  minReadySeconds: 20
  selector:
    matchLabels:
      role: vote
    matchExpressions:
      - key: version
        operator: Exists
  template:
    metadata:
      name: vote
      labels:
        app: python
        role: vote
        version: v1
    spec:
      containers:
        - name: app
          image: schoolofdevops/vote:v1
          resources:
            requests:
              memory: "64Mi"
              cpu: "50m"
            limits:
              memory: "128Mi"
              cpu: "250m"
```

In an additional terminal where you have **kubectl** client setup, launch the monitoring console by running the following command (ensure that you have the **--show-labels** options set):

```
watch -n 1 kubectl get all --show-labels
```

Let's create the deployment. Monitor the labels of the pod while applying this.

```
kubectl apply -f vote-deploy.yaml
```

Observe the changes to pod labels, specifically the **pod-template-hash**.

Now that the deployment is created run the following commands to validate:

```
kubectl get deployment
kubectl get rs --show-labels
```

```
kubectl get deploy,pods,rs
kubectl rollout status deployment/vote
kubectl get pods --show-labels
```

Rolling Out a New Version

Now, update the deployment spec to use a new version of the image.

File: **vote-deploy.yaml**

```
...
template:
  metadata:
    labels:
      version: v2
  spec:
    containers:
      - name: app
        image: schoolofdevops/vote:v2
```

Trigger a rollout:

```
kubectl apply -f vote-deploy.yaml
```

Open a couple of additional terminals on the host where **kubectl** is configured and launch the following monitoring commands:

Terminal 1

```
kubectl rollout status deployment/vote
```

Terminal 2

```
kubectl get all -l "role=vote"
```

What to watch for:

- Rollout status using the command above
- Following fields for **vote** deployment on a monitoring screen:
    - READY count (reflects surged replicas, e.g. 14/12)
    - AVAILABLE count (never falls below REPLICAS - **maxUnavilable**)
    - UP-TO-DATE field (reflects replicas with the latest version defined in the deployment spec)
- New ReplicaSet is created for every rollout
- Continuously refresh the vote application in the browser to see the new version being rolled out without a downtime

Try updating the version of the image from v2 to v3, v4, v5, etc.. Repeat a few times to observe how it rolls out a new version.

Breaking a Rollout

Introduce an error by using an image which does not exist.

File: **vote-deploy.yaml**

```
spec:
  containers:
    - name: app
      image: schoolofdevops/vote:rgjerdf
apply
kubectl apply -f vote-deploy.yaml
kubectl rollout status
```

Observe how deployment handles failure in the rolls out.

Undo and Rollback

To observe the rollout history use the following commands:

```
kubectl rollout history deploy/vote
kubectl rollout history deploy/vote --revision=xx
```

Replace **xx** with revisions.

Find out the previous revision with the same configs.

To rollback to a specific version (for example revision 2):

```
kubectl rollout undo deploy/vote --to-revision=2
```

To simply roll back to the previous version:

```
kubectl rollout undo deploy/vote
```

Summary

Deployments offer you a way to roll out new versions of the applications without causing any downtime. With this exercise, you have learned how to define the rollout strategies, what the key configurations are, and how to roll out as well as roll back.

Lab 14A. Tekton

Tekton is a Kubernetes-native continuous integration technology. It extends the features of Kubernetes with the use of custom resources and controllers. In this exercise, you are going to learn how to:

- Set up Tekton with controllers and CRDs.
- Get started with tkn CLI.
- Read and understand Tekton resources.
- Set up a sample CI Pipeline with Tekton.

Prerequisites:

- A working Kubernetes cluster.
- **`kubectl`** configured with the context set to connect to this cluster.
- Default storage class available in the cluster to provision volumes dynamically.

Install Tekton

Start with:

```
kubectl apply --filename
https://storage.googleapis.com/tekton-releases/pipeline/latest/release
.yaml
```

To understand what gets created, run the following commands:

```
kubectl get ns
kubectl get all -n tekton-pipelines
kubectl get configmap,secret,serviceaccount,role,rolebinding -n
tekton-pipelines
kubectl get crds | grep -i tekton
```

To install Tekton CLI (i.e. tkn) refer to the instructions specific to your OS.

Following instructions install Tekton CLI on Ubuntu:

```
sudo apt update;sudo apt install -y gnupg

sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys
3EFE0E0A2F2F60AA

echo "deb http://ppa.launchpad.net/tektoncd/cli/ubuntu eoan main"|sudo
tee /etc/apt/sources.list.d/tektoncd-ubuntu-cli.list

sudo apt update && sudo apt install -y tektoncd-cli
```

Validate:

```
tkn

tkn version
```

Get started with Tekton using the following subcommands. These subcommands help you list Tekton pipelines, PipelineRuns, tasks and TaskRuns. Reference the video lessons and follow along:

```
tkn

tkn p

tkn pr

tkn t

tkn tr
```

Set Up a Continuous Integration Pipeline with Tekton

Set Up the Prerequisite Tekton Tasks

Begin by creating the following prerequisite tasks:

- git-clone: is used to clone the source repository and pass on the latest commit hash to other tasks.
- kaniko: is used to build Docker images and publish those to the registry. This is a cleaner Kubernetes native solution than using Docker-based image builds.

You can also search for a catalog of tasks that comes with Tekton:

- Tekton Hub (beta)
- Tekton Catalog repository

To add the git-clone task, run:

```
kubectl apply -f
https://raw.githubusercontent.com/tektoncd/catalog/main/task/git-clone
/0.3/git-clone.yaml
```

Similarly, install a kaniko task which would read the Dockerfile and build an image out of it. To install a kaniko task do:

```
kubectl apply -f
https://raw.githubusercontent.com/tektoncd/catalog/main/task/kaniko/0.
3/kaniko.yaml
```

```
tkn t list
```

Sample output:

```
NAME        AGE
git-clone   19 minutes ago
kaniko      8 seconds ago
```

Create Tekton Pipeline Resource

Begin with:

```
tkn p list
tkn t list
```

```
git clone https://github.com/lfd254/tekton-ci.git
cd tekton-ci/base
```

```
kubectl apply -f instavote-ci-pipeline.yaml
```

Validate:

```
tkn p list
tkn p describe instavote-ci
```

Create Pipeline Run for Vote App

PipelineRun allows you to launch an actual CI pipeline by creating an instance of a template (Pipeline) with application-specific inputs (resources).

Edit **vote-ci-pipelinrun.yaml** file with actual values. Following are the parameters displayed from the pipeline run file:

```
 params:
  - name: repoUrl
    value: https://github.com/xxxxxx/example-voting-app.git
  - name: revision
    value: main
```

```
  - name: sparseCheckoutDirectories
    value: /vote/
  - name: imageUrl
    value: xxxxxx/vote
  - name: pathToContext
    value: vote
```

Replace values for:

- **repoUrl**: to point to the repository where your application source is.
- **imageUrl**: to point to your Docker Hub/registry user/organization ID you publish the image to.

Begin by listing the Pipelines and PipelineRuns as:

```
tkn p list
tkn pr list
```

Launch a pipeline run for the **vote** app:

```
kubectl apply -f vote-ci-pipelinerun.yaml
```

Validate and watch the pipeline run with:

```
tkn pr list
tkn pr logs -f vote-ci
```

You may see that the pipeline run exits with an error while trying to push the image to the registry. This is expected as kaniko needs registry secrets in order to authenticate and push a container image.

Generate Docker Registry Secret

To provide kaniko with a secret to connect with your registry, you need to run **docker login** at least once. This example assumes Docker Hub as a registry.

Log in to Docker Hub once using the following command:

```
docker login
```

This generates a **config.json** file which you would need to encode with base64 and add as a Kubernetes secret:

```
cat ~/.docker/config.json
cat ~/.docker/config.json | base64 | tr -d '\n'
```

Now copy the base64 encoded string and generate a secret yaml manifest similar to the configuration below by replacing the value for **config.json** with the actual encoded string.

File: **dockerhub-creds-secret.yaml**

```
apiVersion: v1
kind: Secret
metadata:
 name: dockerhub-creds
data:
 config.json: <base-64-encoded-json-here>
```

Replace **<base-64-encoded-json-here>** with the actual value.

```
kubectl apply -f dockerhub-creds-secret.yaml
kubectl describe secret dockerhub-creds
```

Please note that this secret has already been referenced in the PipelineRun resources.

Following is an example snippet from the **vote-ci-pipelinerun.yaml** spec. A source of which could be studied here:

```
  workspaces:
  - .....
  - name: dockerconfig
    secret:
      secretName: dockerhub-creds
```

In case you use a different name when creating the secret, ensure that you replace the configuration above to point to it.

Now, proceed to create a new pipeline by deleting the previous ones and run the following command:

```
tkn pr rm vote-ci
kubectl apply -f vote-ci-pipelinerun.yaml
```

Validate and watch the pipeline run with:

```
tkn pr logs --last --follow --all
```

Once the pipeline run is complete, you should see a new container image published on Docker Hub with a tag in the format **main-commit-timestamp**.

Setup Continuous Integration for Result App

Since you already have a template in the form of a pipeline, setting up a CI for the **result** app is just a matter of creating an instance of it, by providing application-specific (result app) inputs.

That's what you do creating a pipeline run object.

You will find the pipeline run spec in the same repository that you have cloned along with the code for **vote** pipeline run.

Update the params as previously:

File: **result-ci-pipelinerun.yaml**

```
  params:
  - name: repoUrl
    value: https://github.com/initcron/example-voting-app.git
  - name: revision
    value: master
  - name: sparseCheckoutDirectories
    value: /result/
  - name: imageUrl
    value: initcron/tknresult
  - name: pathToContext
    value: result
```

```
kubectl apply -f result-ci-pipelinerun.yaml
```

Validate and watch the pipeline run with:

```
tkn pr list
tkn pr logs --last --follow --all
```

Validate by checking if the container image is published on Docker Hub for the **result** app.

Additional Resources

- *"Getting Started"*
- *"Concepts"*
- Tekton - official website
- *"Getting Started with Triggers"*

Summary

In this lab exercise, you have learned how to natively run continuous integration pipelines with Tekton on Kubernetes. There are tools such as Jenkins X and OpenShift which use Tekton as an engine and have built more sophisticated systems on top of it. We hope that this lab exercise along with the lecture part made you curious about Kubernetes-native CI/CD. Go ahead and explore it further!

Lab 14B. ArgoCD

Following CI with Tekton, ArgoCD offers the missing piece of how to set up an automated and sophisticated continuous deployment and delivery system which are based on the principles of GitOps. In this lab exercise, you are going to learn:

- How to set up ArgoCD with a web UI inside an existing Kubernetes cluster.
- How to set up an automated deployment to Kubernetes using the principles of GitOps.

Setting Up ArgoCD

Install ArgoCD:

```
kubectl create namespace argocd
```

```
kubectl apply -n argocd -f
https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/in
stall.yaml
```

Source: GitHub, install_argocd.md

Reset admin password to **password**:

```
#
bcrypt(password)=$2a$10$rRyBsGSHK6.uc8fntPwVIuLVHgsAhAX7TcdrqW/RADU0uh
7CaChLa
kubectl -n argocd patch secret argocd-secret \
  -p '{"stringData": {
    "admin.password":
"$2a$10$rRyBsGSHK6.uc8fntPwVIuLVHgsAhAX7TcdrqW/RADU0uh7CaChLa",
    "admin.passwordMtime": "'$(date +%FT%T%Z)'"
  }}'
```

Source: GitHub, reset-argo-passwd.md

Reference: GitHub, argo-cd/faq.md at master ○ argoproj/argo-cd

```
kubectl get all -n argocd
```

```
kubectl patch svc argocd-server -n argocd --patch \
  '{"spec": { "type": "NodePort", "ports": [ { "nodePort": 32100,
"port": 443, "protocol": "TCP", "targetPort": 8080 } ] } }'
```

Source: GitHub, patch_argo.md

```
kubectl get svc -n argocd
```

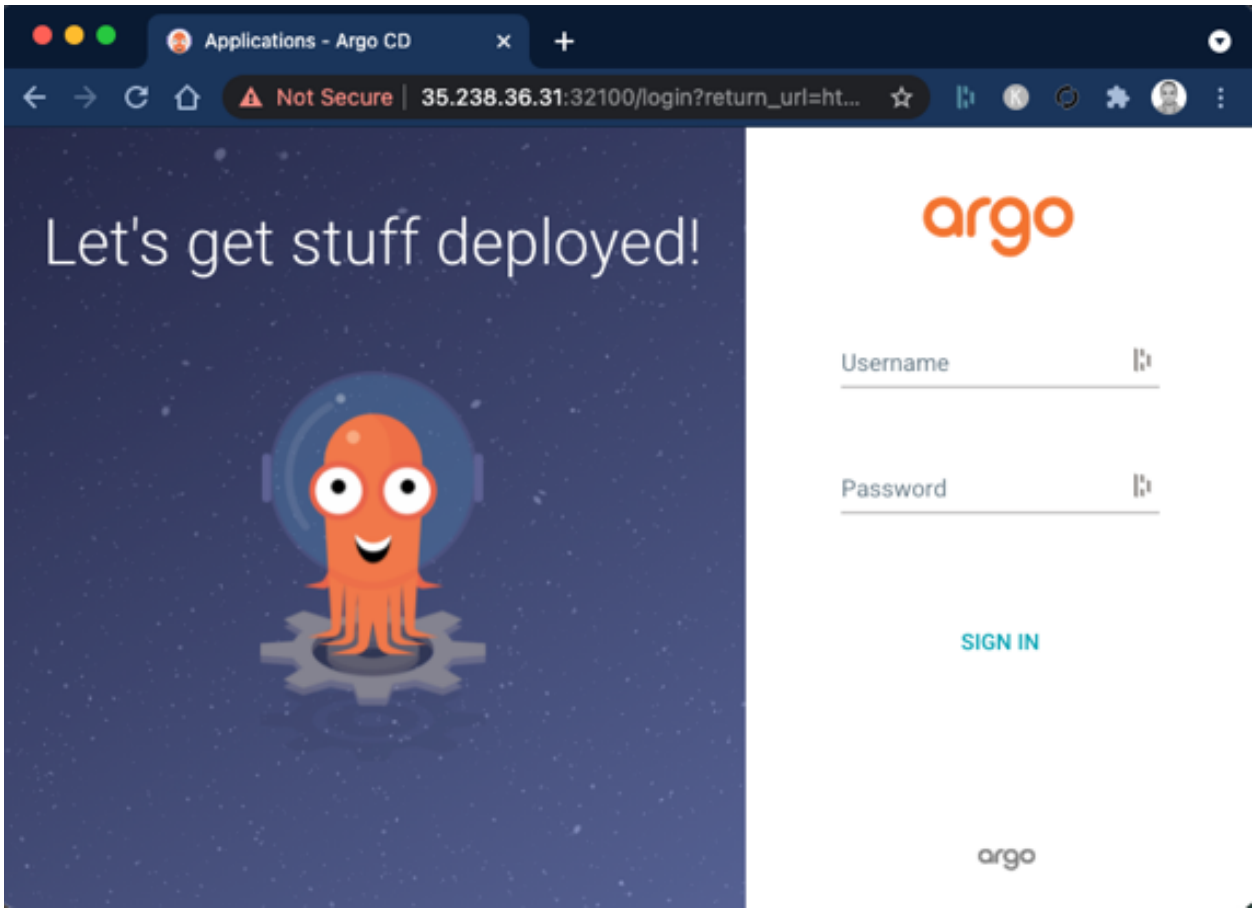
Find out the IP address for one of the nodes. One way to do that, is to run the following command:

```
kubectl get nodes -o wide
```

Write down the IP address for one of the nodes and browse to https://NODEIP:32100.

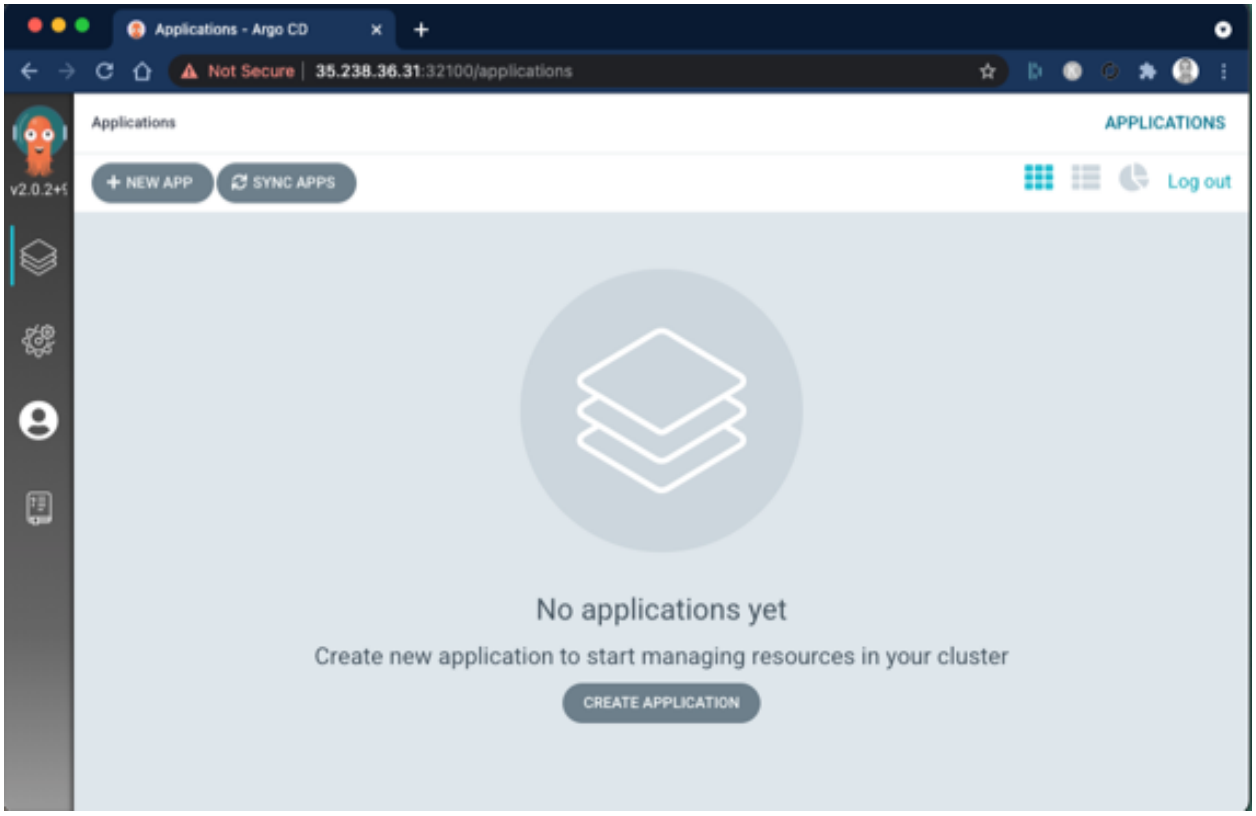
Replace **NODEIP** with the actual IP address of the node listed above.

You should be presented with the login page for ArgoCD:



- username = **admin**
- password = **password**

Once logged in, you should see a screen such as this one:



Preparing Application Deployment Manifests

Create a fork of GitHub's schoolofdevops/vote repository. Clone it to the host where **kubectl** is available and switch to it:

```
git clone https://github.com/xxxxxx/vote.git
cd vote
mkdir deploy
cd deploy
```

Replace **xxxxxx** with the actual user/organization name.

Generate YAML manifests to deploy **vote** app:

```
kubectl create deployment vote \
  --image=schoolofdevops/vote:v1 \
  --replicas=4 \
  --port=80 \
```

```
  --dry-run=client -o yaml  | tee  vote-deploy.yaml
```

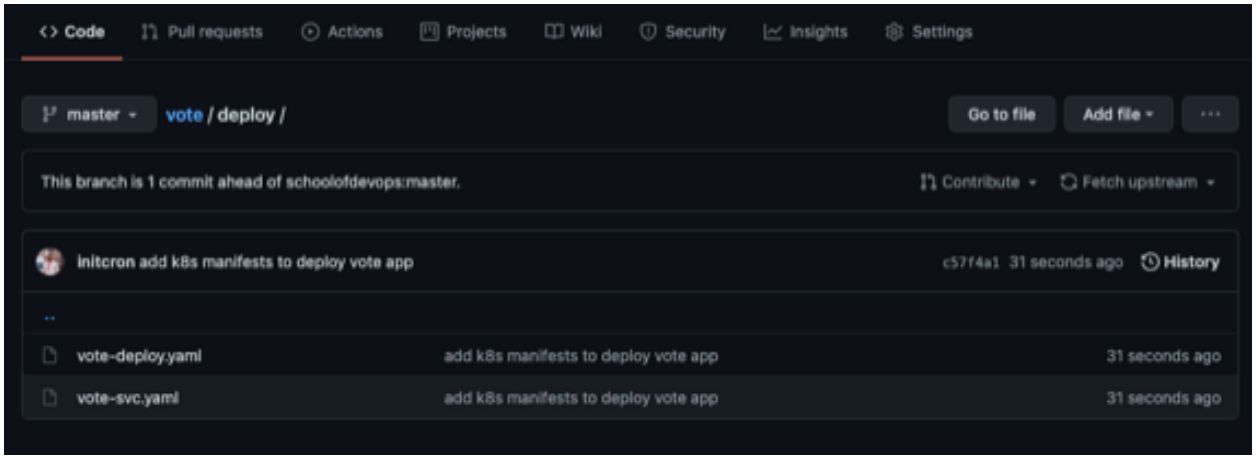
Generate manifest to create a Load Balancer (Service):

```
kubectl create service nodeport \
  vote --tcp=80 \
  --node-port=30100 \
  --dry-run -o yaml | tee vote-svc.yaml
```

Add and commit to the git repository:

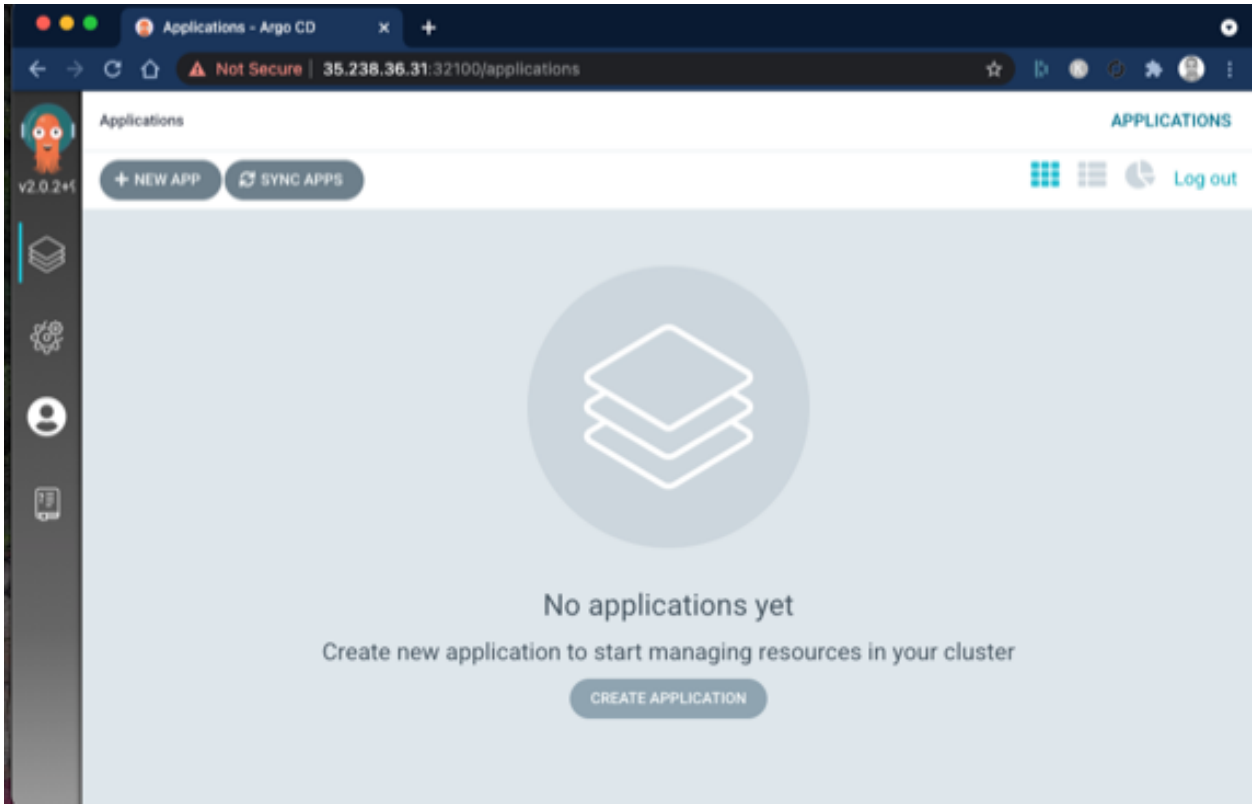
```
git add vote-deploy.yaml vote-svc.yaml
git commit -am "add k8s manifests to deploy vote app"
git push origin master
```

Validate that these manifests are reflected in your git repository:



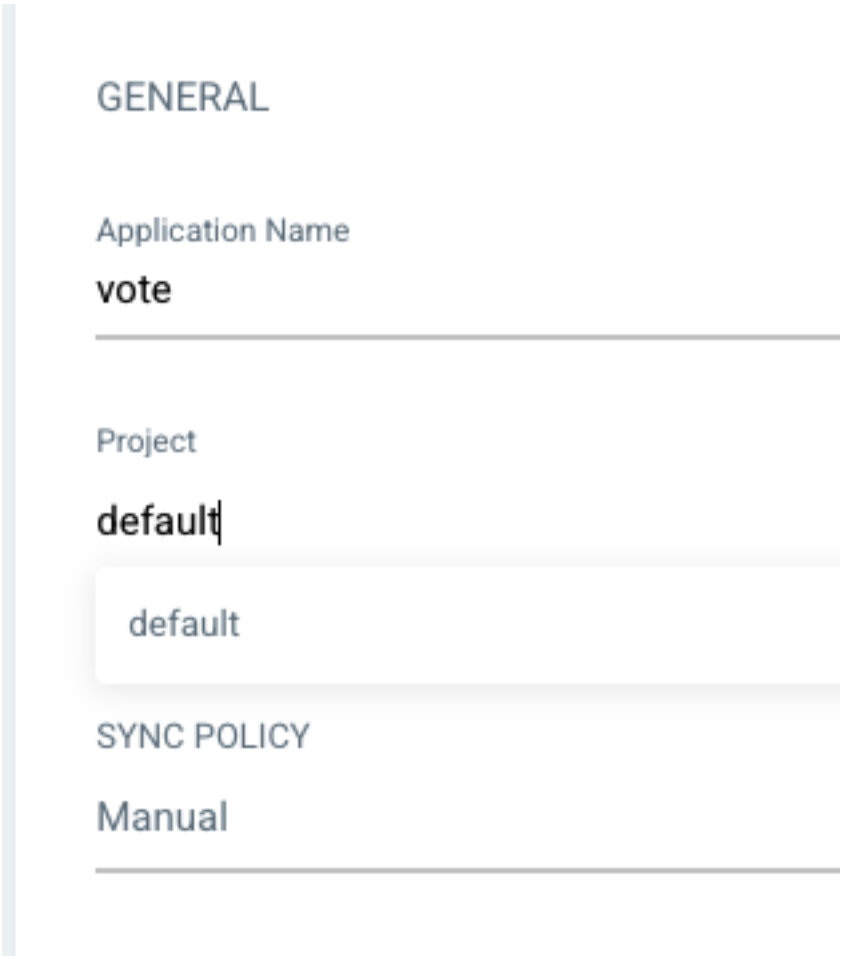
Setting Up an Automated Deployment with ArgoCD

Browse to ArgoCD web console and click on *Create Application*:



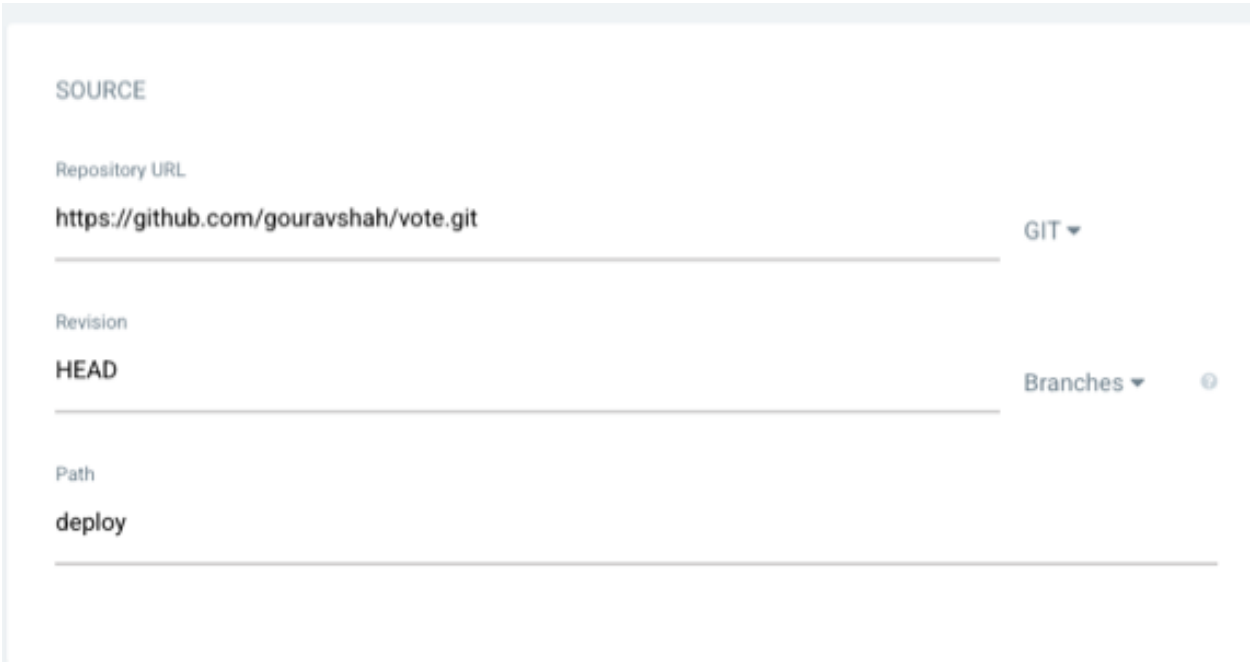
In *General* set the following options:

- Application Name: “vote”
- Project: “default”
- Sync Policy: “Manual”



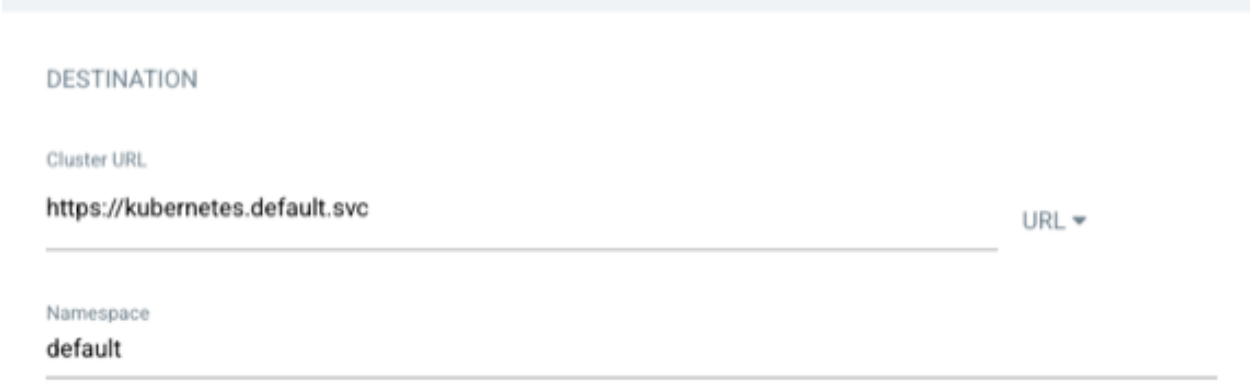
In *Source* set the following options:

- Repository URL: Your repository URL (https)
- Revision: “HEAD”
- Path: “deploy”

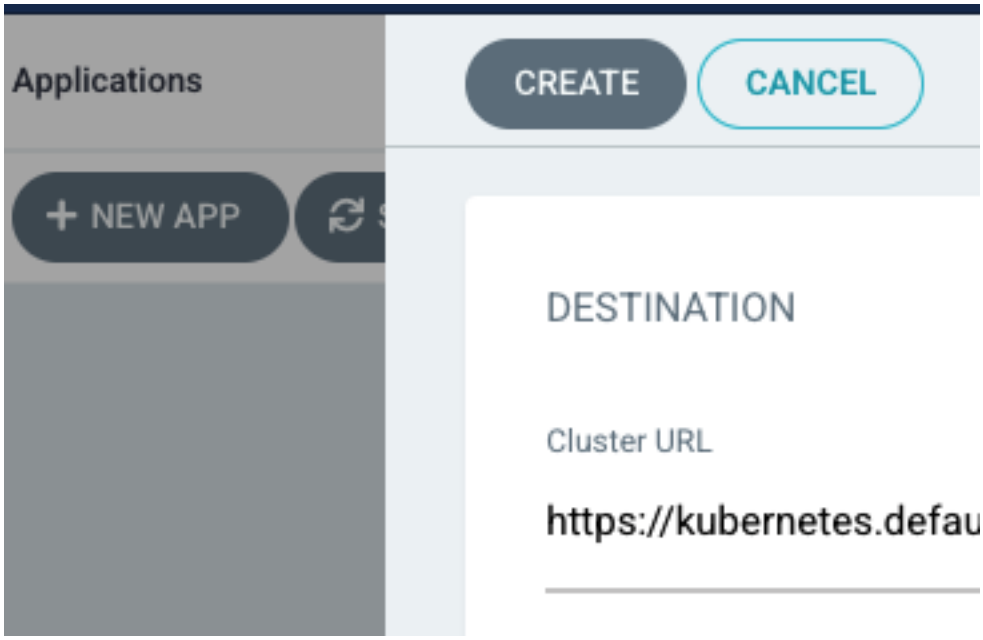


In *Destination* set the following options:

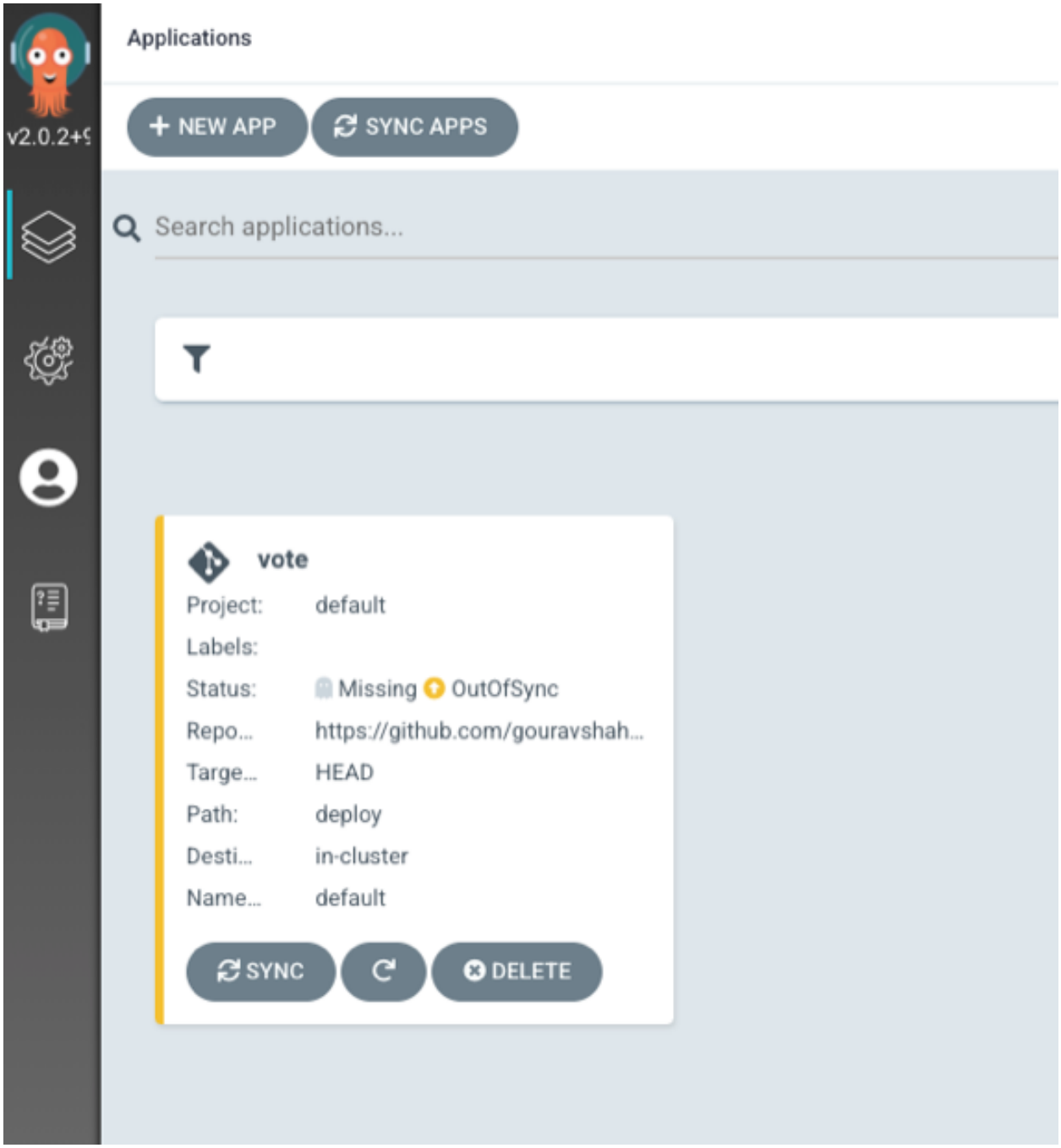
- Cluster URL: https://kubernetes.default.svc (default)
- Namespace: “default”



Click on the *CREATE* button on the top:



You should now see a `vote` application created.

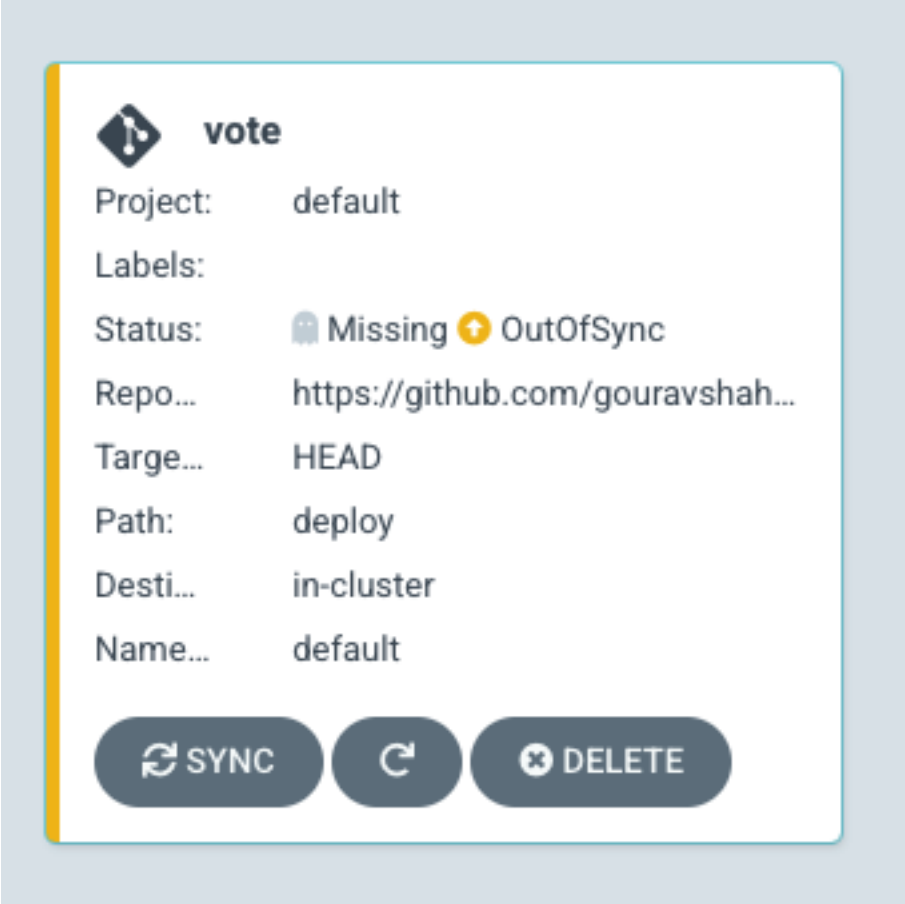


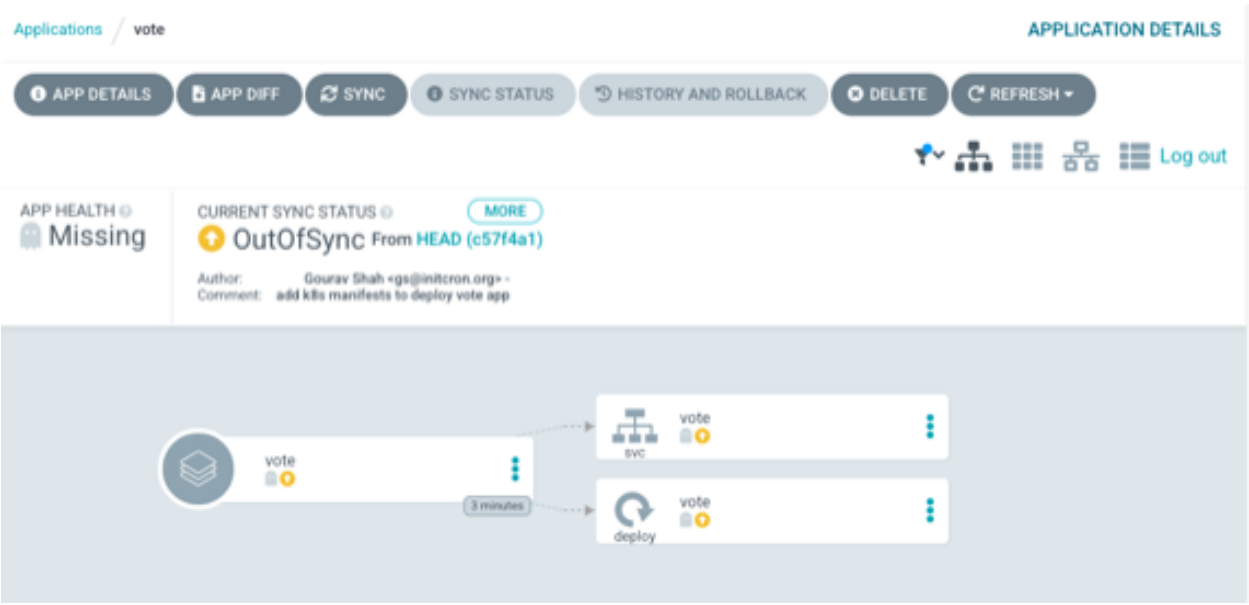
Deploy to Kubernetes

Before you begin deployment, go to the **`kubectl`** console and start looking for changes in the default namespace:

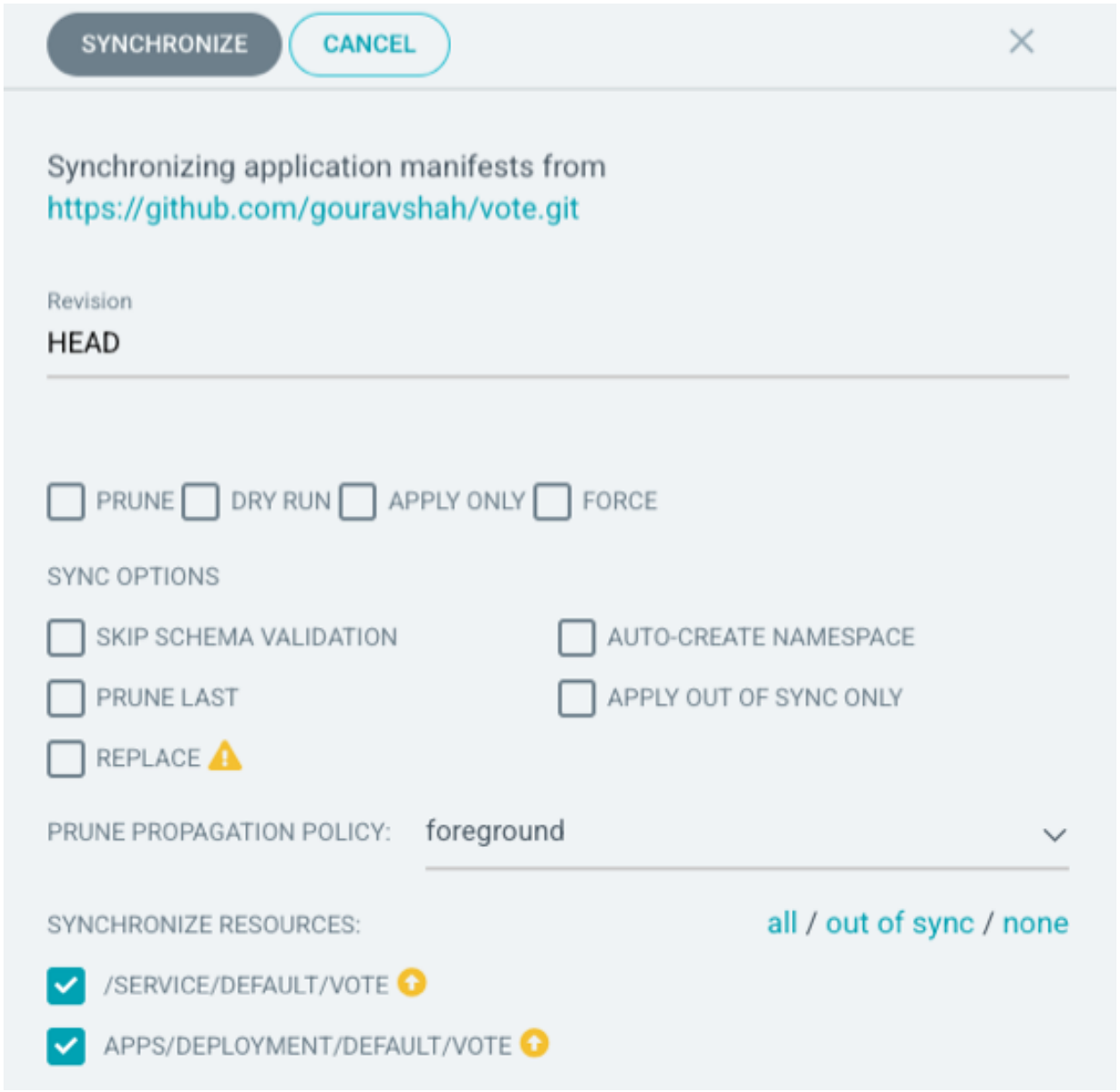
```
watch kubectl get all -n default
```

Select the **`vote`** application on ArgoCD to access more details:





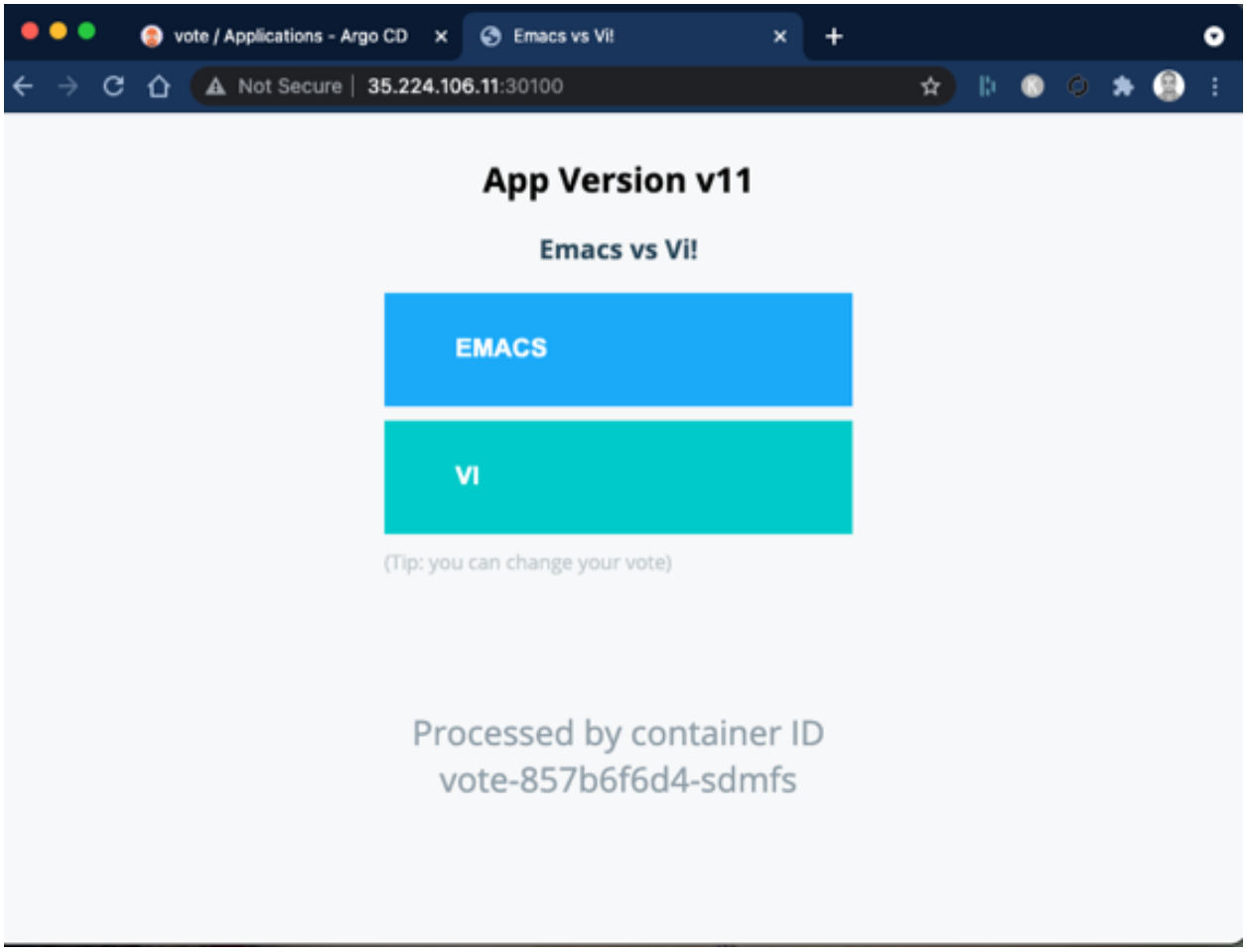
Go ahead and click on the *SYNC* button. You should see screen such as one below:



Keep everything as is and click on the *SYNCHRONIZE* button.

Watch for the changes in the console as well as in Argo. You should see the application synced from the git repository to the Kubernetes cluster in a few seconds.

Validate by accessing the **`vote`** application on http://NODEIP:30100.



Now:

- Enable auto-sync by browsing to *App Details* → *ENABLE AUTO-SYNC*.
- Try modifying YAML manifests in git and either do a Manual sync, or wait for the Auto sync to see if changes in git are reflected in the cluster. Some configurations you can play with include:
  - image: **`schoolfodevops/vote: vxx`** (you can use tags v1 to v9 which are available on Docker Hub)
  - replicas
  - nodePort in service
- Create YAML manifests for a **`sysfoo`** application, add it to the repository and have it synced automatically with ArgoCD.

Additional Resources

- *"Getting Started with Argo"*

Summary

GitOps is an up and coming technology, with ArgoCD and FluxCD being the two most popular options to implement it. In this exercise, you have learned just enough about GitOps to get you curious and explore this topic further. You can start diving deeper into the world of GitOps with the Linux Foundation's course: *"GitOps: Continuous Delivery on Kubernetes with Flux" (LFS269)*.