Docker & Kubernetes: The Practical Guide

Contents

Getting Started

Docker Installation

- Docker Toolbox and Docker Desktop are basically just tools that bring Docker to life on non-Linux operating systems, you could say, because the Linux operating system natively supports containers and the technology Docker uses.
- Docker Toolbox – The Docker tool runs natively on Linux and to make it work on macOS or Windows, you in the end need a virtual machine. So a machine simulated on your machine which holds a Linux installation in which Docker can run.
- Now Docker Desktop for both Mac and Windows uses built-in operating system features for that. But older versions don't have these features. That's why you then need to install a virtual machine manually and install Docker inside of that machine.

Docker Images & Containers: The Core Building Blocks

Dockerfile

- Dockerfile contains the instructions for Docker to build our own image.
- Docker container is isolated from our local environment. And as a result, it also has its own internal network. And when we listen to say port 80 in the node application inside of our container, the container does not expose that port to our local machine. So we won't be able to listen on the port just because something's listening inside of a container.
- "Isolated" means Containers are separated from each other and have no shared data or state by default.
- Multiple containers can be based on the same image but they are totally isolated from each other.
- `docker build` command tells Docker to build a new custom image based on a Dockerfile.
- A docker image is read only, contains a SNAPSHOT of the code and related dependencies. So every time you make any change in your source code, you need to build a new image.

Understanding Image Layers

- Images are layer based, which means that when you build an image, or when you rebuild it, only the instructions where something changed, and all the instructions there after are re-evaluated.
- Whenever you build an image, Docker caches every instruction result, and when you then rebuild an image, it will use these cached results if there is no need to run an instruction again. And this is called a layer based architecture.
- Every instruction represents a layer in your Dockerfile.
- And an image is simply built up from multiple layers based on these different instructions. And it includes a layer for each instruction from the dockerfile as well as the instructions from base image (FROM) which is used.
- It exists to speed up the creation of images.
- A container does not copy over the code and the environment from the image into a new container, into a new file. A container will use the environment stored in an image, and then just add this extra layer on top of it, e.g. running node server process and allocate resources, memory and so on to run the application, but it will not copy that code.
- Images contain multiple layers (1 Instruction = 1 Layer) to optimize build speed (caching!) and re-usability.

Understanding Attached & Detached Containers

- docker start runs the container in the background.
- docker run runs the container in the foreground.
- For starting with docker start, the detached mode is the default
- For running with docker run, the attached mode is the default.
- "attached" simply means that we're listening to the output of that container. For example, to what's being printed to the console.
- To explicitly run container in detached mode using docker run command, we need to pass a flag "–d" to the docker run command.
- To explicitly run container in attached mode using docker start command, we need to pass a flag "–a" to the docker run command.
- You can attach yourself again to the detached container by running command -
  `>docker container attach <container_name>`
  `>docker attach <container_name>`
- Side Note: You can run an image as a container from any folder.

Running Container in Interactive mode

- Docker can be used with any kind of applications and not just web applications. E.g. interacting (input/output) with a command line utility application.
- To run container in interactive mode,
  `>docker run -it <image-name>`
  `>docker start -a -i <container-name>`

Removing Containers and Images

- You cannot remove running containers.
- To remove a (stopped) container,
  `>docker rm <container-name>`
- To remove multiple containers at once,
  `>docker rm <container1-name> <container2-name>`
- To remove all stopped containers,
  `>docker container prune`
- To see all images, run
  `>docker images`

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
node	latest	6a8007b5489a	4 days ago	908MB

  Here you will see all the images and their details like size. You may notice that the size of node image for example is almost 950 MB. This size is not just node and the node

executable tools, but those tools and the operating system image on which the node image builds up.

- To remove an (unused) image,
  `>docker rmi <image-name>`
  This deletes the images and all the layers in that image.
- To remove multiple (unused) image,
  `>docker rmi <image1-name> <image2-name>`
- You can only remove images if they're not getting used by any container anymore including stopped containers.
- So no matter if a container is started or stopped, images belonging to that container, can't be removed. The container needs to be removed first.
- To remove all unused images which has no tags,
  `>docker image prune`
- To remove all unused images irrespective if they have tags or not,
  `>docker image prune -a`

Removing Stopped Containers Automatically

- To automatically remove a container when it exists
  `>docker run --rm <image-name>`
- This ensures that whenever this container is stopped, it's removed automatically.

Inspecting images

- To see more information about an image
  `>docker image inspect <image-id>`

Copying Files Into & From a Container

- `docker cp` command allows you to copy files or folders into a running container or out of a running container. (Not oftenly used)
- To copy a file from local to running container,
  `>docker cp <local-folder-file-name> <container-name>:<folder-inside-container>`
- To copy a file from running container to local,
  `>docker cp <container-name>:<file-inside-container> <local-folder-name>`
- This command can be useful for a couple of things.
  - It of course would allow you to add something to a container without restarting the container and rebuilding the image. But this is not a good solution as it is error prone. This could be useful for instance to copy certain configuration files for a web server.
  - Copying something out of a container can also be quite interesting for instance to copy log files generated by the container.

Naming & Tagging Containers and Images

- To name a container,
  `>docker run --rm -p 3000:80 --name goalsapp 831841d2458f`
- An image tag consists of two parts – the actual name, also called repository of your image and then a tag separated by a colon. The tag can be for instance a particular version of the repository.
- The image tag (name:tag) must be unique identifier for obvious reasons.
- We can tag an image while building it using --tag list option which gives a Name and optionally a tag in the 'name:tag' format.
  `>docker build -t goals:latest .`
- With tags, you can have multiple, more specialized version of a given image.
- To rename an image tag,
  `>docker tag <old-name:old-tag> <new-name:new-tag>`
- When you rename an image, you don't get rid of the old image. Instead, you kind of create a clone of the old image.

Sharing Images – Overview

- Anyone who has an image can run a container based on that image. Hence we never share containers but only images.
- Refer slide Section2#11

Pushing Images to DockerHub

- Refer slide Section2#12
- To share/push an image to repository (DockerHub/Private Repo),
  `>docker push <image-name-along-with-dockerid>`
- To use/pull an image from repository (DockerHub/Private Repo),
  `>docker pull <image-name-along-with-dockerid>`
- Important! If you just push and pull with a regular image repository name, then this will automatically go to Docker Hub.
- If you want push or pull to a private registry, so to any other provider, you have to include the host, so the URL of that provider in your push and pull commands.
- A shared image is in the end just a repository.
- Before pushing an image you need to first login to your docker hub account. This is for obvious reasons as not everyone else should be able to push images to your docker hub account.
- To login to your docker hub account,
  `>docker login`
- One you login, you don't have to login again on your machine.
- To logout to your docker hub account,

```
>docker logout
```

- Docker pushes an image very **smartly**. If you run docker images, you will see that size of your image is about 950 MB. But docker knows that your image depends on say node image and that image is already on docker hub. So it establishes a connection to that node image and only pushes the extra code it needs, only the extra information and not the entire node image again.
- "latest" is the tag that's created automatically if we don't set a tag manually.
- If we pushed multiple different tags with the same repository name, they would all show up in this tags list.
- Once you push an image successfully, you should be able to view it on your Docker Hub account under Repositories.

Pulling & Using Shared Images

- To use/pull an image from repository (DockerHub/Private Repo),
  `>docker pull <image-name-along-with-dockerid>`
- If your image is public, anyone can pull it. You don't have to be logged in to your dockerhub account.
- You can then run the pulled image on your local using docker run command.
- If you try to run an image which doesn't exist on your local, docker will first pull the image based on your image name from docker hub.
- Once you have an image on your local, docker will simply run that image present on your local, it will not check for latest images on the docker hub.

Managing Data & Working with Volumes

Understanding Data Categories / Different Kinds of Data

- Refer slide Section3#1

Understanding The problem

- If we just stop and restart a container, the data/file which is created from the running app is not deleted. It stays in the container's file system (if you files created from your app). However if you remove the container and start a new container from the image, you will not see the data/files created from previous container. This is because the containers are totally isolated. Deleting a container will delete all it's data.
- Now in reality we want to retain our data (e.g. user signups, or any persisted data.) even if a container is removed and the data should be used when a new container is started.
- Refer slide Section3#2

Introducing Volumes

- Refer slide Section3#3
- Docker has a built in feature called volumes. And volumes help us with persisting data, and solving above problem.
- Volumes are folders on your host machine hard drive which are mounted ("made available", mapped) into containers.
- And changes in either folder, will be reflected in the other one. So if you add a file on your host machine, it is accessible inside of the container, and if the container adds a file in that mapped path, it is available outside of the container, in the host machine as well. And therefore, because of this mechanism, volumes allow you to persist data even if a container is shut down.

Two Types of External Data Storages

Volumes

- Refer slide Section3#4
- We have **anonymous** and named volumes.

```
# This creates anonymous volume.
VOLUME [ "/app/feedback" ]
```

- To see all the volumes,
  `>docker volume ls`
- If we remove our container, the **anonymous** volume also gets deleted. It actually only exists as long as our container exists. This happens when you start / run a container with the --rm option.

- If you start a container without --rm option, the anonymous volume would NOT be removed, even if you remove the container later (with docker rm ...).
- In that case, you just start piling up a bunch of unused anonymous volumes. You can clear them via
  `>docker volume rm VOL_NAME`
  `>docker volume prune`
- With named volumes, volumes will survive container's removal. The folders on your hard drive will survive. And therefore, if you start new containers thereafter, the volumes will be back, the folder will be back, and all the data stored in that folder will still be available.
- So **named** volumes are great for data which should be persistent, and which you don't need to edit or view directly, because you don't really have access to that folder on your host machine. Also it can be used to **share data** across containers.
- We can't create named volumes inside of a Dockerfile. We have to create a named volume when we run a container.
- Run container with named volume using "-v" option
  `>docker run -d -p 3000:80 --name feedback-app -v feedback:/app/feedback feedback-node:volumes`
- **`-v feedback:/app/feedback`** This is a syntax Docker will understand, and it will now store /app/feedback in a managed volume. So it will create a folder on our hosting machine and connect it to this folder inside of the container (/app/feedback), but it will store this volume under a name (feedback) chosen by us.
- The **key difference** to anonymous volumes is that named volumes will not be deleted by Docker when the container is removed. Anonymous volumes are deleted because they are recreated whenever a container is created.

Bind Mounts

- Bind mounts can help us with a different kind of problem we might be facing.
- Whenever we change anything in our source code, be that in the server JS file, or in any HTML file, those changes are not reflected in the running container unless we rebuild the image.
- But of course**, during development**, if we're using Docker, it would be pretty important to us, that such changes are reflected. Because otherwise, we always have to rebuild the entire image and restart a container, whenever we change anything. That's where bind mounts can help us.
- Bind mounts have some similarities with volumes, but there is one key difference. Where volumes are managed by Docker, and we don't really know where on our host machine file system they are. For bind mounts, we do know it. Because for bind mounts, we, as a

developer, set the path to which the container internal path should be mapped on our host machine.

- And since containers can read-write into volumes and bind mounts, we could put our source code into such a bind mount.
- So bind mounts are therefore perfect, for **persistent** and **editable** data. Also it can be used to **share data** across containers.
- A named volume can help us with persistent data, but editing is not really possible, since we don't know where it's stored on our host machine.
- Since bind mounts affect the container, we cannot put it in the dockerfile so in the image. Hence we have to setup a bind mount from terminal while running the container.

Adding Bind Mounts

- If you should be using Docker Toolbox to run Docker, then by default your users (C:\Users) folder will be shared.
- To allow adding other mount points refer – https://headsigned.com/posts/mounting-docker-volumes-with-docker-toolbox-for-windows/
- In windows to add bind mounts while running container use, below command
  `>docker run -d -p 3000:80 --name feedback-app -v feedback:/app/feedback` **`-v /d/Docker/Workspace/05-data-volumes-demo:/app`** `feedback-node:volumes`
  Here /d/Docker/Workspace/05-data-volumes-demo refers to D:\Docker\Workspace\05-data-volumes-demo on your local and part after :, which is /app refers to path inside container.

Bind Mounts – Shortcuts

- Just a quick note: If you don't always want to copy and use the full path, you can use these shortcuts:
- macOS / Linux: `-v $(pwd):/app`
- Windows: `-v "%cd%":/app`

Combining & Merging Different Volumes

- In windows to add bind mounts while running container use, below command
  `>docker run -d -p 3000:80 --name feedback-app -v feedback:/app/feedback` **`-v /d/Docker/Workspace/05-data-volumes-demo:/app`** `feedback-node:volumes`
  Here /d/Docker/Workspace/05-data-volumes-demo refers to D:\Docker\Workspace\05-data-volumes-demo on your local and part after :, which is /app refers to path inside container.
- Above command does not work because whatever in the D:\Docker\Workspace\05-data-volumes-demo is copied to /app inside container and it will not have node_modules folder with required dependencies to run the app.

- To solve this problem, we need to tell Docker, that there are certain parts in its container file system, which should not be overwritten from outside in case we have a clash. And we do this by adding an **anonymous** volume.
- We can add anonymous volumes in 2 ways –
    - From Dockerfile=> VOLUME ["/app/node_modules"]
    - Or during running the container, -v /app/node_modules
- **Docker always evaluates all volumes you are setting on a container, and if there are clashes, the longer (more specific) internal path wins**.
- **Anonymous** volumes are useful to prioritize container-internal paths higher than external paths.
- So finally this below command, we should be able to run our application inside container, which has anonymous volume (`-v /app/node_modules`), named volume (**`-v`** `feedback:/app/feedback`) and bind mount (**`-v`** `/c/Users/Sameer/DockerMount/05-data-volumes-demo:/app`).
  `>docker run --rm  -d -p 3000:80 --name feedback-app -v feedback:/app/feedback -v /c/Users/Sameer/DockerMount/05-data-volumes-demo:/app -v /app/node_modules feedback-node:volumes`
  Note: If it doesn't work, try removing --rm flag and see logs of exited container.
- **This is very important and useful step. With this, even if you don't have required software (node.js) installation on our local, but with the help of docker images, you can not only run containers but also work on codebase (fix errors, add features) with ease.**
- **Gotcha**
    - If you make changes in say .html files of a node app, those will be picked up immediately by the server running inside the container.
    - However if you make changes in say server.js or some other files like configuration files, those changes will not be reflected until the web server running inside the container is restarted. This is common node.js scenario where if you make any changes to such file, server running with node command does not pick up. The solution is to use nodemon package to run the server.

Summary: Volumes and Bind Mounts

- Refer slides Section3#6,7
- **Anonymous** volumes either created with the VOLUME instruction in the dockerfile or created with –v while running the container, can be useful for locking in certain data which already exists in the container. They can be useful for avoiding overwriting of the data by another module.
- In addition, **anonymous** volumes also still create a counterpart, a folder, on your host machine. Of course that's removed when the container is removed, but that exists as

long as the container is running. And that of course means that docker doesn't have to store all the data inside of the container and doesn't have to manage all the data inside of this container read write layer. But that instead it can outsource certain data to your host machine file system. And this can also help with performance and efficiency. E.g. for temporary intermediate files, etc.

Read-only volumes

- With bind mounts, during development, we would typically want that the container should not be able to change the files in our local mapped folder. Only we should be able to change them on our host machine file system and those should be reflected in the container and container should not be able to make changes in the files inside that folder.
- This is where we can enforce the volume to be read-only volume.
- By default volumes are read write, which means the container is able to read data from there and write data to them. But you can restrict that by adding an extra colon after the container internal path, and then RO for read only. This ensures that docker will now not be able to write into this folder or any of its sub-folders.
- Also we might have a scenario where the container should not able to write into local host file system except for couple of folder. E.g. feedback or temp directory folders in our node demo app.
- E.g.
  `>docker run --rm  -d -p 3000:80 --name feedback-app -v feedback:/app/feedback -v /c/Users/Sameer/DockerMount/05-data-volumes-demo:/app:ro -v /app/temp -v /app/node_modules feedback-node:volumes`
  Here it makes all folders inside `/app` as read only except `/app/feedback` and `/app/temp` as they are specifically mentioned and specific paths take priority.

Managing Docker Volumes

- Volumes are managed by Docker. **Docker does not manage bind mounts**. We as developers are in control of it.
- When we run our container with –v flag, Docker we'll actually go ahead and create a volume, which also means that it creates a folder somewhere on the host machine automatically.
- Docker volume related commands,
  `>docker volume --help`
- To list all active volumes
  `>docker volume ps`
- To inspect a volume,
  `>docker volume inspect <volume-name>`

- E.g.
  >docker volume inspect feedback
  [
    {
      "CreatedAt": "2021-05-17T10:56:08Z",
      "Driver": "local",
      "Labels": null,
      "Mountpoint": "/mnt/sda1/var/lib/docker/volumes/feedback/_data",
      "Name": "feedback",
      "Options": null,
      "Scope": "local"
    }
  ]
  Here Mountpoint is actually inside of a little virtual machine Docker set up on your system. So it's hard to find out where this is actually stored on your host machine.
- To remove an unused volume,
  `>docker volume rm <volume-name>`
  This will remove the volume so all the data is lost as it will delete all the files/data inside that volume.
- To remove all unused volumes,
  `>docker volume prune`

Using "COPY" vs Bind Mounts

- During development, if we use bind mounts while running a container, we can skip COPY . . instruction in the Dockerfile.
  E.g.
  `>docker run --rm  -d -p 3000:80 --name feedback-app -v feedback:/app/feedback -v` **`/c/Users/Sameer/DockerMount/05-data-volumes-demo:/app:ro`** `-v /app/temp -v /app/node_modules feedback-node:volumes`
- However We use this bind Mount during **development** to reflect changes in our code into running container instantly. Once we're done with developing, and once we actually take this container, put it onto a server and we want to run it, we will not run it with above command. **We might use other volumes to ensure that data survives, but will not use this bind Mount in production**.
- So for our application to run properly in production, we need to use COPY . . instruction.

Don't COPY Everything: Using "dockerignore" Files

- We can also restrict what can get copied in the image when we use COPY instruction.
- We do this by adding .dockerignore file.
- With .dockerignore, we can specify which folders and files, should not be copied by a COPY instruction into the docker image. E.g. to ignore node_modules folder.

Working with Environment Variables & ".env" Files

- Refer Slide Section3#8
- Docker supports **build-time arguments** and **runtime environment variables**.
- Arguments allow you to set flexible bits of data or variables in your Dockerfile which you can use in there to pluck different values into certain Dockerfile instructions based on arguments that are provided with the --build-arg option when you run docker build.
- Environment variables on the other hand are available inside of a Dockerfile like arg, but also are available in your entire application code in your running applicationand you can set them with the --env option on the docker run command.
- args and environment variables allow you to create more **flexibl/dynamic images and containers** because you don't have to hard-code everything into these containers and images.
- With env variables, you can set detault values in the Dockerfile which you can override by --env option or just "–e" while running the docker container
- In the Dockerfile,

```
# set environment variables with default values
# the values can be overwritten with docker run command's --
env option e.g. --env PORT=3000
ENV PORT 80
EXPOSE $PORT
```

- While running the container
  E.g.
  `>docker run --rm  -d **--env PORT=3000 -p 3000:3000** --name feedback-app -v feedback:/app/feedback -v /app/temp -v /app/node_modules feedback-node:env`
- If you have multiple environment variables, you'll simply add multiple "-e", each with the key value pairs.
- You can also specify a file that contains your environment variable. Often such a file is named `.env` and in the file you will have key values pairs e.g. `PORT=3000`
- And while running the container, we specify path of the file for --env-file option
  `>docker run --rm  -d **–env-file ./.env -p 3000:3000** --name feedback-app -v feedback:/app/feedback -v /app/temp -v /app/node_modules feedback-node:env`

Environment Variables & Security

- Depending on which kind of data you're storing in your environment variables, you might not want to include the secure data directly in your Dockerfile
- Instead, go for a separate environment variables file which is then only used at runtime (i.e. when you run your container with docker run).
- Otherwise, the values are "baked into the image" and everyone can read these values via docker history <image>.
- For some values, this might not matter but for credentials, private keys etc. you definitely want to avoid that!
- If you use a separate file, the values are not part of the image since you point at that file when you run docker run. But make sure you don't commit that separate file as part of your source control repository, if you're using source control.

Using Build Arguments (ARG)

- With arguments, we can actually plug in different values into our Dockerfile, or into our image when we build that image without having to hard code these values into the Dockerfile.
- With arguments, when we build the image, we can actually build the image based on one and the same unchanged the Dockerfile multiple times with different default values.
- Use ARG instruction in the Dockerfile. The key used in the ARG can be used in other instructions except CMD.
- E.g.

```
# to set build time arguments
ARG DEFAULT_PORT=80

EXPOSE $DEFAULT_PORT
```

- Now while building the image, you can use --build-arg option, if we want any other port than 80. E.g.
  `>docker build -t feedback-node:arg` **`--build-arg DEFAULT_PORT=3000`**
- So basically you can create multiple images of different arguments.
- You cannot use arguments in your source code e.g. server.js.

Networking: (Cross-) Container Communication

- Building applications with multiple containers is quite common.
- With networks here, it really means –
    - How you can connect multiple containers,
    - How you can let them talk to each other
    - But also how you could connect an application running in a container to your local host machine.
    - And for example, send HTTP requests to some other service running on your machine.
    - And also how you can reach out to the world wide web from inside your container.

Case 1: Container to WWW Communication

- Refer slide Section4#1
- E.g. accessing a WEB REST API.

Case 2: Container to Local Host Machine Communication

- Refer slide Section4#1
- E.g. connecting to local database.

Case 3: Container to Container Communication

- Refer slide Section4#1
- E.g. connecting to some other service exposed by one of your other containers.
- With Docker containers, it is strongly recommended and the best practice that every container should just do one main thing. E.g. one container for DB connectivity other for application logic.

Creating a Container & Communicating to the Web (WWW)

- Out of the box, from within the container, connecting to the host machine with localhost fails. E.g. connecting to localhost database.
- Out of the box, containers can send requests to the World Wide Web.
- You can communicate with web API's and web pages from inside your dockerized applications. You don't need any special setup or any changes to your code.

Making Container to Host Communication Work

- Out of the box, from within the container, connecting to the host machine with localhost fails. E.g. connecting to localhost database. mongodb://**localhost**:27017/swfavorites
- In order for the application running inside container to talk to your localhost, you need to use **`host.docker.internal`** instead of **`localhost`**.
- This special domain (**`host.docker.internal`**) is recognized by Docker. It's understood by Docker. And it's translated to the IP address of your host machine as seen from inside the Docker container.
- So instead of mongodb://**localhost**:27017/swfavorites, use mongodb://**host.docker.internal**:27017/swfavorites in your code.
- You can use it for your local database url, or any other http server running on your localhost.

Container to Container Communication: A Basic Solution

- Just like you can run official node docker images so that you don't have to install it locally, you can also run mongodb docker image so that you don't need to install it locally either.
- To pull official mongodb image from Docker Hub.
  `>docker pull mongo`
- You can then run it as
  `>docker run -d --name mongodb mongo`
- Now to connect your application running in a container to this mongodb container, you need a couple of changes.
  - Run `docker container inspect mongodb` and get IPAddress for this container.
  - Update the mongodb URL to use this IP Address.
  - E.g. If the IPAdress for above inspect command says 172.17.0.2, then use mongodb url as mongodb://**172.17.0.2**:27017/swfavorites'
  - Now build the image again since we made code changes.
  - And finally run the container.
- As you can see, this not very convenient solution. We had to look up the IP address of the other container in order to then use it in other application. It also means that we always have to build a new image whenever the MongoDB container IP address changed, because we hard code that IP address in the node app which is not ideal.
- But thankfully, there is an easier way of having multiple Docker containers talk to each other.

Introducing Docker Networks: Elegant Container to Container Communication

- Container Networks = Networks
- Refer slide Section4#2
- With Docker, you can put all the related containers into one and the same network by adding the --network option on the Docker run command.
- This then creates a network in which all containers are able to talk to each other, and Docker is then automatically doing this IP look up and resolving stuff for given address/URL, which we did manually in above step – Container to Container Communication: A Basic Solution.
- And that's a really useful feature for having multiple, isolated containers with their own duties and tasks, which still are able to talk to each other.

How to Create Networks

- Unlike with volumes, for networks, Docker will not automatically create them for you if you want to run a container using a network, instead you have to create them on your own.
- Docker network related commands
  `>docker network --help`
- To create a docker network
  `>docker network create <network-name>`
  e.g. `>docker network create favorites-net`
  It's a Docker internal network, which you can then use on Docker containers to let them talk to each other. All the hard work and heavy lifting is taken care of by Docker here.
- List all existing networks
  `>docker network ls`
- Now run your containers using --network option on the docker run command E.g.
  `>docker run -d --name mongodb --network favorites-net mongo`
- If two containers are part of the same network, you can just put the other containers name in the URL. E.g. the mongodb URL in our favorites application would look like this –
  `'mongodb://mongodb:27017/swfavorites'`,
  Here **mongodb** in the url refers to the name of the container which you chose while running the container.
- Both `mongodb://mongodb:27017/course-goals` and `mongodb://mongodb/course-goals` will work in this case because mongodb container by default exposes 27017 port (see docker ps -a).
- With above url change, build image of your application code and run it with --network option using same network name

E.g. `>docker run --name favorites-app --network favorites-net -d -p 3000:3000 favorites-node`

- Side note – When we run the MongoDB container, we did not specify the -p option. The reason for that is that the -p option is only required if we plan on connecting to something in that container from our local host machine or from outside the container network.
- When you have a container to container connection, then you don't need to publish the port because internally in that container network, all the containers can freely communicate with each other and you don't need to expose any ports.

How Docker Resolves IP Addresses

- Refer slide Section4#3
- Docker will not replace your source code. Docker owns the environment in which your application runs. So it simple detects outgoing requests and resolves the IP for such requests.

Docker Network Drivers

- Docker Networks actually support different kinds of "Drivers" which influence the behavior of the Network.
- The default driver is the "**bridge**" driver - it provides the behavior shown in this module (i.e. Containers can find each other by name if they are in the same Network).
- The driver can be set when a Network is created, simply by adding the --driver option.
  `>docker network create --driver bridge my-net`

Different Docker Network Drivers

- Docker also supports these alternative drivers - though you will use the "bridge" driver in most cases:
- **host**: For standalone containers, isolation between container and host system is removed (i.e. they share localhost as a network)
- **overlay**: Multiple Docker daemons (i.e. Docker running on different machines) are able to connect with each other. Only works in "Swarm" mode which is a dated / almost deprecated way of connecting multiple containers
- **macvlan**: You can set a custom MAC address to a container - this address can then be used for communication with that container
- **none**: All networking is disabled.
- **Third-party plugins**: You can install third-party plugins which then may add all kinds of behaviors and functionalities.

Building Multi-Container Applications with Docker

Course App

- Refer slide Section5#1
- Course App – https://github.com/sameerbhilare/Docker/tree/main/Workspace/07-multi-container-apps-demo

Dockerizing the MongoDB Service

- To run the mongodb container with port exposed so that we can connect it from localhost
  >docker run --rm –d --name mongodb **–p 27017:27017** mongo
  Here we can then connect to mongodb from our Node app running on our local (not in container) using mongodb://localhost:27017/

Dockerizing the Node App

- In order to make our Node app run in container and connect to mongodb running in another container. Let's put IP address of running mongodb container and use inside node app. E.g. `mongodb://172.17.0.2:27017/course-goals`
- Also btw since we have expose port mongodb, we can directly use localhost:27017 (or 192.168.99.100:27017 in case of Docker Toolbox) e.g.
  mongodb://192.168.99.100:27017/course-goals
- So basically you can use any one of the above options.
- With this change, create Dockerfile and build image of our node backend app.
  `>docker build –t goals-node .`
- Run the node backend app. E.g.
  `>docker run --name goals-backend --rm –d –p 80:80 goals-node`
  Note – here we have exposed port 80 because our frontend app needs to connect from localhost (not yet from within container) to the node app which running inside container.
- With this, till now our node app running in container is connected to mongodb server running in the container.

Dockerizing the Front end React App

- Create Dockerfile for our frontend React app and build an image.
  `>docker build –t goals-react .`
- Run the React frontend app. E.g.
  `>docker run --name goals-frontend --rm –d –p 3000:3000 -it goals-react`
  Note – here we have exposed port 3000 because we will use to connect from browser. Also we need to run it in interactive mode due to the react setup.

Side Note

- Till now all these three containers are able to communicate with each other but they all communicate with each other through our localhost machine (or 192.168.99.100 in case of Docker Toolbox) because we always publish their ports.
- It's even better if we put them all into one Docker network. And then they can automatically communicate with each other just through their container names. Let's do that next!

Adding Docker Networks for Efficient Cross-Container Communication

- Create Docker Network for our end to end app ()
  `>docker network create goals-net`
- Now start mongodb container with above network
  `>docker run –name mongodb --rm –d --network goals-net mongo`
  Note – here we don't need to expose port as we are going to connect it from within another container only and not from localhost. You can expose the post, if you want to connect to it from localhost for some other purpose.
- Since we want to connect to mongodb container from node backend container, we need to replace the mongodb URL/address by name of mongodb container. E.g.
  `mongodb://mongodb/course-goals` Here **mongodb** in the url refers to the name of the container which you chose while running the container.
- Both `mongodb://mongodb:27017/course-goals` and `mongodb://mongodb/course-goals` will work in this case because mongodb container by default exposes 27017 port (see docker ps –a) though it is not accessible from localhost as is not mapped to outside port but it can be surely accessed from another running container.
- With this change, build the node backend app image again.
  `>docker build –t goals-node .`
- Run the node backend server with the same network
  `>docker run --name goals-backend --rm –d --network goals-net –p 80:80 goals-node`
  Note – here also we need to expose node backend app port because though it will be used by our react frontend app running in the container, but the actual node API invocation is done from within the browser, so outside containers.
- Now if you are using Docker Toolbox, then replace node backend URL in the react frontend codefrom localhost to 192.168.99.100. And then rebuild react frontend image.
- Assuming you have react frontend image (goals-react), run the react frontend server without any network option because this container is not doing any container to

container communication with any other container. Browser communicates with node app but that through port exposed by node backend app.
`>docker run --name goals-frontend --rm –d **-p 3000:3000 -it** goals-react`
Here we need to expose port 3000 because we want to connect to it from localhost browser.

Adding Data Persistence to MongoDB with Volumes

- At the moment, if we stop mongodb container, all our saved data will be gone. That happens because when we stop MongoDB, the container is removed, and when it's removed, all the data stored in the container is lost.
- In order to survive the container tear down and starting new container and make our data persistent, we need to use **named volume (or bind mount)**.
- Here we need to use -v flag while running mongodb container but we need to know which path the MongoDB container uses internally for storing that database data. (see official docker hub mongo image docs)
- As per official docs on docker hub mongo image, it stores data internally at /data/db
  So we need to use named volume for this internal path.
- So here is how we should start the mongodb container to persist our data
  `>docker run --rm -d --name mongodb **-v data:/data/db** --network goals-net mongo`

Adding Security to MongoDB

- Refer Authentication section in https://hub.docker.com/_/mongo
- Mongo image also supports MONGO_INITDB_ROOT_USERNAME and MONGO_INITDB_ROOT_PASSWORD for creating a simple user with the role root in the admin authentication database.
- Use these env variables to set your username and password while running the mongodb container.
  `>docker run --rm -d --name mongodb -v data:/data/db --network goals-net **-e MONGO_INITDB_ROOT_USERNAME=admin -e MONGO_INITDB_ROOT_PASSWORD=admin** mongo`
- With this, if you hit your react frontend application at localhost:3000, it will fail to load data because we need to add this authentication information in our node backend app while trying to connect to mongodb.
- So in the node backend app, change mongo url to
  `mongodb://**admin:admin@**mongodb:27017/course-goals**?authSource=admin**`
- Rebuild node backend app image and run as usual above.
  `>docker run --name **goals-backend** --rm –d **--network goals-net –p 80:80** goals-node`

Volumes, Bind Mounts & Polishing for the NodeJS Container

- Now in our app, we want to add 2 features for node backend app – data should be persistent (log files) and live source code update so that we can easily work during development.
- So adding bind mount and named volume as
  `>docker run --rm -d --name goals-backend --network goals-net -p 80:80 -v /c/Users/Sameer/DockerMount/07-multi-container-apps-demo/backend:/app -v logs:/app/logs -v /app/node_modules goals-node`
- Here -v /c/Users/Sameer/DockerMount/07-multi-container-apps-demo/backend:/app is the **bind mount**. And -v logs:/app/logs is the **named volume** and-v /app/node_modules is **anonymous volume**.
- Also bind mount won't override this named and anonymous volumes because **more specific container internal path takes precedence.** And named volume internal path /app/logs is more specific than bind mount internal path which is /app. So /app/logs will not be overwritten due to behavior of bind mount. So logs in the container will survive.
- Similarly since we have bind mounts, we also want to ensure that non-required files (e.g. Dockerfile) and folders (e.g. node_modules) are not copied to container. Container must have its own version of node_modules. So node_modules will also survive. Also here we don't need named volume for node_modules for obvious reasons that it will be there with docker image build.
- We need one more thing in order for live source code update to work with node backend app, we need to add nodemon dev dependency and npm start script like nodemon app.js and in the Dockerfile's CMD use CMD ["npm", "start"]
- Rebuild the backend image and restart the container using above command.
- IMP NOTE – This did not work for me with Docker Toolbox as expected. But typically works well with Docker Desktop. With docker toolbox, my changes in .js files in node backend app were reflected after I restarted my node backend container without rebuilding image. So it partially works with Docker toolbox.

Using environment variables for MongoDB username and password

- We can add these 2 env variables in node backend app Dockerfile

```
ENV MONGODB_USERNAME=admin
ENV MONGODB_PASSWORD=admin
```

- And use it via process.env. MONGODB_USERNAME and process.env. MONGODB_PASSWORD inside our nodejs code like this
  `mongodb://${process.env.MONGODB_USERNAME}:${process.env.MONGODB_PASSWORD}@mongodb:27017/course-goals?authSource=admin`

- With this, rebuild the backend image and run the node backend container.

Avoid copying unrequired files to container

- Add .dockerignore file in node backend app and add below entries

```
node_modules
Dockerfile
.git
```

Live Source Code Updates for the React Container (with Bind Mounts)

- Now we want to have live source code updates for react frontend application so that if we make any code changes on our local, it should be picked up by the running container.
- Solution is of course using **bind mount**.
- Now for react we don't need nodemon as it has in built react command for the same.
- Now run the react frontend app with bind mount to your source code.
- `>docker run --rm -d --name goals-frontend --network goals-net -p 3000:3000 -it -v /c/Users/Sameer/DockerMount/07-multi-container-apps-demo/frontend/src:/app/src goals-react`

Summary

- Refer Slide section5#3
- This module is for development only as live source code updates is what we need in development only.

Room For Improvement

- Refer Slide section5#3
- One major problem is that we have three pretty long docker run commands.
- Also we have to remember all the options for each container to be run. So it is easy to forget few options. E.g. interactive mode, port publishing, etc.
- We may have to rebuild few images manually. Need to remove unused volumes, networks manually, etc.
- We have to run the containers individually, even though the containers kind of belong together.
- Would be great, if we wouldn't have to remember / save those and if we wouldn't have to run them individually.
- Solution for this problem is **Docker Compose**.

Docker Compose: Elegant Multi-Container Orchestration

Introduction

- Docker Compose is solution to above problems.
- Docker Compose helps managing multi containers setups easier.
- It helps you automate the setup process and it allows you to bring up entire setup, with all the different containers and their individual configurations with just one command.
- And you can then also tear everything down with just one command.

Docker Compose: What & Why?

- Refer Slide section6#1,2
- Docker Compose is a tool that allows you to replace one more more Docker build and Docker run commands with just one configuration file and then a set of orchestration commands to start all those services, all these containers at once and build all necessary images, if it should be required and you then also can use one command to stop everything and bring everything down.
- Of course, you can also use Docker Compose in a single container application, but it just shines the most if you have multiple containers.

What Docker Compose is NOT

- Docker Compose does NOT replace Dockerfiles for custom Images
- Docker Compose does NOT replace Images or Containers
- Docker Compose is NOT suited for managing multiple containers on different hosts (machines).

How

- You always start by writing a Docker Compose file.
- Unlike Docker without compose, it's not primarily about executing commands in the terminal, instead the idea is that you can save time there and therefore you put a lot of configuration into this Docker Compose file.
- And in this file, you need just define a couple of core components that make up your multi-container application.
- Heart and the most important element is **services** which in the end can be translated with containers that make up your multicontainer application.
- Service = container.
- Then for the service, you can define which port should be published, environment variables, define volumes that should be assigned to this container, networks and you can do basically everything you can do with Docker command in the terminal otherwise.

Docker Compose Configuration File

- Create docker-compose.yaml file in your main folder.
- Refer https://docs.docker.com/compose/compose-file/
- When you use Docker Compose, Docker will automatically create a new environment for all the services specified in this compose file and it will add all the services to that network out of the box. Hence no need to specify network options for your services.
- If you do specify a network explicitly, then that service will be added in default network as well as specified network.
- Any named volume used in any services should be listed under global "volumes".
- If you then use the same volume name in different services, the volume will be shared so different containers can use the same volume, the same folder on your hosting machine.

Docker Compose Up & Down

- Navigate to folder where docker compose yaml file is.
- >docker-compose up
  This will start all the services in this compose file in attached mode. And it will not just start the containers, it also will pull and build all the images that might be required.
- To start docker compose in detached mode
  >docker-compose up -d
- To stop all services and remove all containers and so on,
  >docker-compose down
  This deletes all containers, the default network it created, and shuts everything down. It does NOT delete volume. To delete volume add **–v** flag.

Working with Multiple Containers

- With Docker Compose, where we create and launch multiple services, so multiple containers at the same time, sometimes, one container might depend on another container to be up and running already.
- E.g. backend depends on mongodb. You can specify this in docker compose yaml file using **depends_on** key.

Override Dockerfile configuration in compose yaml file

- You can actually also set certain settings which you set in a dockerfile normally inside of a docker-compose file to essentially override the settings inside of the dockerfile.
- Not all Dockerfile instructions likeCOPY are available in yaml file but some are available.
- E.g.

```yaml
npm:
    image: node:14
    # add or override WORKDIR instruction from node's dockerfile
    working_dir: /var/www/html
    # add or override ENTRYPOINT instruction from node's dockerfile
    entrypoint: ['npm']
```

Building Images & Understanding Container Names

- When you run `docker-compose up` command, docker compose builds an image for service having "build" attribute, but it builds an image for that service if it doesn't already exist on your local or it will rebuilt it if something in the code changes.
- If you want to force to always build the image and start the containers,
  `>docker-compose up --build`
- To only build the images
  `>docker-compose build`
- By default with docker compose, the container names are generated as <prokect-folder-name>_<service-name-in-yaml>_<incrementing-number>
  E.g.
- If you want to use custom container name, you can use **container_name** option in the yaml file under your service. But this is optional.

Notes

- Refer course project – https://github.com/sameerbhilare/Docker/tree/main/Workspace/08-compose-multi-container-apps-demo

Working with "Utility Containers" & Executing Commands in Container

Utility Containers

- Refer Slide Section7#1
- Utility container is not an official term. ☺
- With utility containers, in the end means containers which only have a certain environment in them. Let's say a node JS environment or a PHP environment.
- The idea here is that they don't start an application when you run them, but instead you run them in conjunction with some command specified by you to then execute a certain task.

Utility Containers: Why would you use them?

- A lot of projects and a lot of programming languages need some extra tools to be installed on your system on your host machine in order to create the projects. E.g. to create a node project you need to install nodejs. (Of course you could write package.json on your own or copy from some existing project or internet, but still!)
- And yes, you can put the project into a container thereafter. But the initial creation still needs these tools to be installed on your system. And that's exactly where utility containers can help us. ☺

Different Ways of Running Commands in Containers

Executing Additional Commands

- The `docker exec` command allows you to execute certain commands inside of a running container besides the default command that container executes. So besides the command (CMD) that might've been specified in a Dockerfile. That command still continues running since the container is running but you can't run additional commands inside of a container. That's something which sometimes could be useful.
- Docker exec command
  `>docker exec <name-of-running-container> <additional-command-to-run>`
- E.g.
  `>docker run --name node-container -it -d node`
  `>docker exec -it node-container npm init`
  Here node is the name of the running node container (based on node image). And "npm init" is the additional command which we want to execute. Also it is started in interactive mode because we want to run the npm init command in the interactive mode.

- So with above command, we can leverage node docker image to create a node project for us. However the project will be created inside of the container and we don't have access to it from our localhost so outside of container.

Overriding default command

- With `docker exec,` we can execute **additional commands** in a running container without interrupting the default command which starts when the container starts. However we also can **override** that default command though.
- To override the default command (i.e. CMD instruction in Dockerfile)
  `>docker run -it node npm init`
  Here we are override the default "node" command by new command "npm init" and once the "npm init" command finishes, it will also exit this running container.
- Overriding the default command can be useful when deploying our application to different cloud providers like AWS. AWS ECS provides a way to override this default command via UI. So with this feature, you can use same image for development (with say nodemon app.js) and for production (with just "node app.js").

Building a First Utility Container based on Node

- To create this utility container, we first need our own image. So create a Dockerfile

```
#  node:14-alpine is an very slim and optimized node base image.
FROM node:14-alpine
WORKDIR /app
```

- Here we don't want to run specific command hence no CMD instruction. Rather we want to leverage our base image node's CMD which is just CMD ["node"]
- Build image based on this Dockerfile,
  `>docker build -t node-util .`
- Now the run command for this image in interactive mode to execute additional commands would be like this
  `>docker run -it node-util npm init`
- By default, this command would run in the /app folder inside of the container (as defined in WORKDIR in the Dockefile). Now I want to mirror it to my local folder so that what I create in the container is also available on my host machine. So that I can create a project on my host machine with help of a container.
- That's the idea behind having a utility container. We can use it to execute something which has an effect on the host machine without having to install all the extra tools on the host machine.
- So now in order to mirror contents of /app inside container to my local folder, we need to use **bind mounts**. So the command would be like this –
  `>docker run -it --rm -v /c/Users/Sameer/DockerMount/09-utility-container-demo:/app node-util npm init`

- Once you run this command, it will ask for few questions based on npm init command and in the end it will create package.json file inside `/c/Users/Sameer/DockerMount/09-utility-container-demo` folder.
- Now with this generic tool, you can execute any node related command. So the generic command could be-
  `>docker run -it --rm -v <full-path-of-localhost-folder>:/app node-util <node-command-to-execute>`

Utilizing ENTRYPOINT

- At the moment above utility container can run any command (both npm & non-npm). However we would like to restrict it to only npm commands. The reason could be we would like to avoid it to create any side-effects on our local machine.
- E.g. You could run delete command and it will delete something.
  `>docker run -it --rm -v /c/Users/Sameer/DockerMount/09-utility-container-demo:/app node-util` **`rm package.json`**
- Hence it's a must that we restrict it to only say **npm** commands e.g. npm init, npm install, etc.
- This restriction will protect us against accidental damage, plus we won't have to write whole npm command, we can omit the 'npm' word from the command.
- For this, we need to use ENTRYPOINT instruction in the Dockerfile.
- Key difference between ENTRYPOINT and CMD instructions is, with CMD – whatever you enter after docker run command will **override** the command in the CMD instruction (Refer this). However with ENTRYPOINT instruction – whatever you enter after docker run command will be **appended** to this command in ENTRYPOINT instruction.
- So our new Dockerfile would look this –

```
#  node:14-alpine is an very slim and optimized node base image.
FROM node:14-alpine
WORKDIR /app
ENTRYPOINT [ "npm" ]
```

- Now build the image for our utility
  `>docker build -t node-util .`
- Now you can run only npm commands via –
  `>docker run -it --rm -v <localhost-path>:/app node-util` **`<npm-command-name>`**
- E.g. To run npm init,
  `>docker run -it --rm -v /c/Users/Sameer/DockerMount/09-utility-container-demo:/app node-util` **`init`**
  Here the command "init" will be appended to "npm" as mentioned in ENTRYPOINT and finally it will execute "npm init".

- **Important** – Our utility container always shuts down once the command is done. Now if you install 'install', it will still work because of course, since my local folder is the bind mount, the package.json file, which I already had on localhost was copied into the /app folder inside container, and therefore npm install being executed in the /app folder inside container was able to pick up this package.json file and use it inside of the container.

Using Docker Compose for our Utility

- As of now we have to run a long command for our utility container so we can use docker compose to ease that.
- So write docker-compose.yaml file as

```yaml
version: '3.6'
services:
  npm:
    build: ./
    stdin_open: true
    tty: true
    volumes:
      - ./:/app
```

- We can use `docker-compose exec` as well as `docker-compose run` commands.
- With `docker-compose run`, exited containers are not removed automatically. So use –rm.
- `docker-compose run` allows us to run a single service from this yaml file by the service name in case we had multiple services. So –
  `>docker-compose run <service-name> --rm <command-name>`
  `E.g. >docker-compose run --rm npm init`

A More Complex Setup: A Laravel & PHP Dockerized Project

Target App

- Refer slide Section7#1

Notes

- For internal container to container communication no port publishing required as it won't be accessed outside container.
- `docker-compose run` allows us to run a single service from this yaml file by the service name in case we had multiple services. Andwe typically use utility containers as single service only.
  >docker-compose run --rm composer create-project --prefer-dist laravel/laravel .
- Our docker compose yaml files has both application containers and utility containers and with docker-compose up, we only want to run application containers and not the utility containers.
- By default with docker-compose up, all services in the yam file will be brought up. However we can specify service names to this command to only up specific services. E.g.
  `>docker-compose up -d server php mysql`
- Note that if a service **depends on** some other service which is not specified in our docker-compose up command, that service will also be brought up.
- Since our nginx depends on php and mysql, we can only execute below command with single nginx server's service name
  `>docker-compose up -d server`
  Here server refers to nginx service in the yaml file.
- Force docker to rebuild some images if source code changes
  `>docker-compose up -d --build server`
- You can actually also set certain settings which you set in a dockerfile normally inside of a docker-compose file to essentially override the settings inside of the dockerfile.
- E.g.

```yaml
npm:
    image: node:14
    # add or override WORKDIR instruction from node's dockerfile
    working_dir: /var/www/html
    # add or override ENTRYPOINT instruction from node's dockerfile
```

```
    entrypoint: ['npm']
```

Deploying Docker Containers

From Development To Production

- Containers are the independent isolated packages full of application code and application environment which we can ship anywhere where Docker runs.
- Because they are standardized, if we built them with Docker and therefore Docker can run them anywhere where Docker is installed.
- Refer slide Section9#1, 2, 3

Development to Production: Things To Watch Out For

- Bind Mounts shouldn't be used in Production!
- Containerized apps might need a build step (e.g. React apps)
- Multi-Container projects might need to be split (or should be split) across multiple hosts / remote machines.
- Trade-offs between control and responsibility might be worth it!

Deployment Process & Providers

- Refer slide Section9#4, 5
- **Simple Steps –**

1. Setting up a remote hosting server.
2. Connecting to it with SSH.
3. Then install Docker on that remote host.
4. Pushing and pulling the image.
5. Running the container from that image
6. And then testing it in the browser.

- Hosting Providers – AWS, Azure, GCP

Bind Mounts In Production

- Refer slide Section9#3

Deploying a basic Node application without any database or anything else

- Refer slide Section9#6
- Demo app - https://github.com/sameerbhilare/Docker/tree/main/Workspace/11-deployment-basic-nodeapp

Deployment Steps

1. Launch EC2 instance (choose Amazon Linux version) and connect to it from SSH
2. Ensure that all essential packages on that remote machine are updated and are using their latest version.
    - sudo yum update –y
3. Install Docker on that remote host (EC2 instance)
    - sudo amazon-linux-extras install docker
    - This will install docker on our EC2 instance.
    - amazon-linux-extras is available in Amazon Linux version's of EC2.
4. Start docker service on the EC2 instance
    - sudo service docker start
    - This will start the docker and now you should be able to execute docker commands. E.g. docker –help
5. Pushing our local Image to the Cloud
    - Two ways – Refer slide Section9#7
    - Easiest option is build image on local and push it to your Docker hub account
    - Login to Docker Hub -> Create Repository with name e.g. node-dep-example-1
    - It will create repository as DockerId@RepoName
      e.g. sameer59/node-dep-example-1
    - In your local code base, add .dockerignore file to exclude node_modules, Dockerfile, .git, *.pem files.
    - Create image on your local as
      >docker build -t node-dep-example-1 .
    - Push this created image to Docker Hub with the same name as the one in Docker Hub which is sameer59/node-dep-example-1
      >docker tag node-dep-example-1 sameer59/node-dep-example-1
      >docker push sameer59/node-dep-example-1
    - Note – If you are not logged in, then the push will fail. In that case, run docker login and enter your Docker Hub credentials.

- With this, you will see that the image is pushed to your docker hub account https://hub.docker.com/repository/docker/sameer59/node-dep-example-1

6. Running & Publishing the App (on EC2)
    - Connect to your EC2 instance via SSH
    - Execute docker run on this created image as
      >sudo docker pull sameer59/node-dep-example-1
      >sudo docker run --rm -d -p 80:80 --name node-example-1 sameer59/node-dep-example-1
    - This will pull the image and run it as a container.
    - Always use pull so that it will pull the latest image from docker hub.
7. Enable EC2 instance HTTP access from outside/public.
    - By default your EC2 instances are not accessible from outside/public.
    - Each EC2 instance has some Security Group assigned to it which controls what traffic is allowed on your EC2 instance.
    - In order to make it accessible, you need to create Security Group with access to HTTP port 80 from outside and assign that security group to your EC2 instance.
8. You can now test it using the Public IP address of your EC2 instance.
    - E.g. http://13.126.122.57/
    - This is because our app is published on port 80 and 80 is the default http port.

Managing & Updating the Container / Image

- Let's talk about pushing updates to your code to that remote server and stop and shut everything down if you want that to do that.
- If you want your updated code to be deployed, follow step no. 5 above.
- Then login to EC2 instance via SSH and stop the running container via docker stop <container-name>
- Follow step 6 above.

Disadvantages of our Current Approach (EC2)

- Refer slide Section9#8, 9
- With EC2, you own the machine. You are fully responsible for it like security, managing deployments, updating docker or other system softwares or underlying OS updates, replace it with new powerful instance if more traffic, etc.
- SSHing into the machine to manage it can be annoying.
- It is better to go for a "managed" approach where everything else is taken care by the cloud provider and you only focus on your app.

From Manual Deployment to Managed Services

- Refer slide Section9#10
- AWS provides such "managed" service via AWS ECS (Elastic Container Service).
- It is a service that helps us with managing containers, with launching them, running them, monitoring them and so on.
- The advantage of using such a managed service is that the entire creation management, updating, monitoring, scaling, all of that is simplified and taken care by AWS and we just need to tell AWS how it should take care about that.
- This means that deploying containers and running containers is now not done with the docker command anymore because now we don't install Docker on some machine anymore, but instead it means that we now will need to use the tools the cloud provider gives us for the specific service we want to use.
- So with AWS ECS => Less Control and Less Responsibility.
  With AWS EC2 => More Control and More Responsibility.

Deploying with AWS ECS: A Managed Docker Container Service

- AWS ECS thinks in four categories – **clusters, containers, tasks, and services**.
- In ECS **Container definition** – you provide details about your image and run options like env variables, volumes, etc.
- By default, our ECS container will be reachable from the web.
- Next you define **Task definition** which is the blueprint for your application.
    - Here, you can tell AWS how it should launch your container. So not how it should execute docker run, but how the server on which it runs this container should be configured.
    - And therefore a task actually also can include more than one container.
    - So you can think of the task as one remote server, one remote machine, which runs one or more containers.
    - Here we are telling AWS how it should execute our containers, and which environment it should generally set up for them.
- AWS ECS Task definition by default uses FARGET as launch type. You can override this if you want by setting to use EC2, telling AWS to create EC2 instances for your containers.
- FARGET is a specific way of launching your container. It launches our containers in a so-called serverless mode, which means AWS does not really create an EC2 instance on which it runs your container, but instead it stores your container and your run settings, and whenever there is a request (e.g. HTTP) to the container, it starts the container up, handles that request, and then stops it again, which is, of course, very cost effective

because you only pay for the time your container is executing and not for the time where it's just sitting around idle.

- Then we define our **service**. It controls how this task, so this configured application and the container that belongs to it, should be executed.
    - Here you, for example, could add a load balancer, and it manages all the heavy lifting of redirecting the incoming requests, queue the up and running containers, and all of that behind the scenes.
    - You have one service per task.
- Finally we need to configure a **cluster**.
    - Cluster is the overall network in which our services run.
    - If you had a multi-container app, you could group multiple containers in one cluster so that they all belong together logically, and they all can talk to each other.

Updating Managed Containers

- Let's talk about pushing updates to your code to that remote server and stop and shut everything down if you want that to do that.
- If you want your updated code to be deployed, follow step no. 5 above.
- Once latest image is pushed to Docker Hub, go to AWS ECS cluster which we created for this app. Go to Task, open Task Definition and then Create new Revision. This will create new revision. Keep same settings and finish. Then Actions -> Update Service. Alternatively you need not create a new task revision but just use "Update Service" and select "Force New Deployment".
- AWS will create and assign a new IP for every new task you are launching, so every new task revision you're launching.

Deploying Multi Container App to AWS ECS

- Multi container app – https://github.com/sameerbhilare/Docker/tree/main/Workspace/12-deployment-multi-container-app
- We are not going to use Docker compose for deployment to AWS ECS. Docker compose is a great tool for running configurations defined for containers in a Docker compose file on your local machine. But it's not really a great tool for deployment to AWS ECS. And that makes sense, if we keep in mind that with Docker compose, we can easily run multiple containers on our local host machine with Docker compose. Now, if you deploy something to a remote machine, things get more complex. You might need to define how much CPU capacity you should be provided for a given service. All of a sudden certain things start to matter, which play no role locally on your machine. And of course,

it's depends heavily on the hosting provider you're using, which kind of extra information might be needed.

- So that's why a compose file can be great for managing and running multiple containers **on one and the same machine**. But as soon as you start moving that into the cloud, where potentially you also got **multiple different machines** working together, docker compose is not useful there.
- However we could use the compose yaml file as an inspiration to configure our services in AWS ECS.

Important Note

- When you run multi container app on your **machine or any single machine**, the docket network approach works fine. So you can use service names directly inside those applications to communicate with each other.
- However as soon as you deploy such multi container app to cloud e.g. AWS ECS, it'll not be that simple anymore. Your containers and the different instances of your containers will be executed and managed by this cloud provider. And they don't necessarily always run on the same machine. That's why this Docker network approach won't work anymore.
- But there is an **exception** though – **If your containers are added in the same task in AWS ECS, then they are guaranteed to run on the same machine**. Still then AWS ECS will not create a Docker network for them instead it allows you to use **localhost** as an address inside of your container application code. So in order for this to work, we should wither make this change in the code and follow next steps (rebuild and deploy) or easier way is instead of making hardcoded changes, you can use environment variables which you can override easily with AWS ECS console. The advantage of this second approachis you can use the same image in development and production with just different env variables.
- In AWS, we can attach volumes to our containers via EFS.

Databases & Containers: An Important Consideration

- Refer slide Section9#11
- Managing our database by your own VS Using managed database services (MongoDB Atlas, AWS RDS)
- Database containers bring their very own complexity.

Deploying frontend application to our multi container app

Understanding a Common Problem

- Refer slide Section9#16

Creating a "build-only" Container

- We want to make sure that we don't just have a container for development, which allows us to test the frontend application but also a container for production, which allows us to deploy this frontend application.
- So our React app basically needs to be executed differently in development and in production. And therefore we'll have to set up two different environments. E.g. we only need node during development to spin up local server (npm start) and we don't need node in production for react frontend as it is just static files (html, css, js).
- So we need to create another dockerfile for production.

Introducing Multi-Stage Builds

- Refer slide Section9#17
- Multi-Stage builds allow you to have one Dockerfile, that define multiple build steps or setup steps, so called "stages" inside of that file.
- Every FROM instruction creates a new "stage" in a Dockerfile.
- Typically, in a multi-stage build dockerfile, if you switch to a next "stage", the previous step will be discarded and you will switch to a new base image.
- However stages can copy results from each other. So we can have one stage to create the optimized files and another stage to serve them.
- We can either build the entire Dockerfile going through all stages, step by step from top to bottom or we select individual stages up to which we want to build, skipping all stages that would come after them.

Building a Multi-Stage Image

- To build docker image
  `>docker build –t <tag-name> <context-folder-name>`
  This will refer to the file named `Dockerfile` at given path.
- However if you name your dockerfile something else e.g. `Dockerfile.prod`, then use –f flag and pass the file name to build this dockerfile along with path
  `>docker build –f <folder-name>/Dockerfile.prod –t <tag-name> <context-folder-name>`

Understanding Multi-Stage Build Targets

- You can build the entire image or also select individual stages, up to which you want to build in that image, if you have a multistage Dockerfile.
- To execute only specific stage (upto a stage) –
  `>docker build --target <stage-name> -f <folder-name>/Dockerfile.prod -t <tag-name> <context-folder-name>`
  This will execute stages **up-to specified stage**.

Deployment options for frontend application

- You can either serve frontend application by making it part of backend by copying deployable files from frontend to a folder inside backend and use node endpoints to serve static contents. In this case, you can use just relative path inside frontend code within backend folder to call backend REST services (e.g. /goals instead of http://localhost:80/goals)
- Another option is to deploy frontend and backend separately. In this case, we have to use the proper backend URL in the frontend application. Also note that, in this case you will need separate AWS load balancers, targets, etc. for obvious reasons.

Room for Improvements

- Refer slide Section9#19
- Containers allow us to encapsulate app code and environment for both development and production.
- If we DON'T manage Docker and remote machines manually, we must work with the tools and rules imposed by the managed service.
- Thinking about production forces us to build containers / app code with more scenarios in mind (e.g. multi-stage builds).
- Different cloud providers == Different rules
- Depending on provider, features like load balancing might be challenging to implement.
- Solution to these problems is **Kubernetes**! ☺

Kubernetes – Getting Started

More Problems with Manual Deployment

- Official Site - https://kubernetes.io/
- Kubernetes, also known as **K8s**, is an open-source system for automating deployment, scaling, and management of containerized applications.
- Kubernetes is not about using containers and Docker locally on our machine for development which we can also do, but **it's just about deployment**.
- Refer slide Section11#1

Why Kubernetes?

- AWS already provides solutions to above mentioned deployment problems but we are then tightly coupled to that cloud provider.
- Refer slide Section11#2, 3, 4

What Is Kubernetes Exactly?

- Refer slide Section11#5, 6, 7.
- Kubernetes is **independent** from the cloud service we're using.
- Here we have one standardized way of writing the configurations. And this configuration would work with any cloud provider as long as it supports Kubernetes, or even if it doesn't support it you can manually install some Kubernetes software on any machine you own and then we'll be able to utilize this config file over there.
- Cloud-provider-specific settings can also be added.
- The great thing with Kubernetes is that you just need to define the desired end state, and if you're then using a cloud provider like AWS, they have services which allow you to provide this Kubernetes definition, and then AWS will set up all the instances and install all the required software for you.

What Kubernetes is NOT

- Refer slide Section11#8
- It's just a collection of concepts and a collection of software which together can be used with any cloud provider.
- It's also not an alternative to Docker. Instead it works together with Docker containers to deploy containers anywhere.
- Kubernetes is like Docker-Compose for **multiple** machines.

Kubernetes Fun Facts

- Kubernetes' abbreviation is K8S.
- The logo of Kubernetes (Helmsman) represents somebody who is providing direction to a ship.

Container Orchestration

- Typical Features:
    - **Auto Scaling** - Scale containers based on demand
    - **Service Discovery** - Help microservices find one another
    - **Load Balancer** - Distribute load among multiple instances of a microservice
    - **Self Healing** - Do health checks and replace failing instances
    - **Zero Downtime Deployments** - Release new versions without downtime
- These are the features provided by container orchestration tools like kubernetes.
- Some of the container orchestration tool options are –
    - AWS Specific
        - AWS Elastic Container Service (ECS)
        - AWS Fargate : Serverless version of AWS ECS
    - Cloud Neutral - Kubernetes
        - AWS - Elastic Kubernetes Service (EKS)
        - Azure - Azure Kubernetes Service (AKS)
        - GCP - Google Kubernetes Engine (GKE)
        - EKS/AKS does not have a free tier!

Kubernetes: Architecture & Core Concepts

- Refer slide Section11#9

Kubernetes Concepts

- Servers in the cloud are called **virtual servers**.
- The fact is that different cloud providers have different names for these virtual servers.
- **AWS** calls them EC2 (**Elastic Compute Cloud**).
  Azure calls them **virtual machines.**
  Google Cloud calls them **compute engines**.
  Kubernetes uses a very generic terminology and calls them **nodes**.
- Kubernetes can actually manage thousands of such nodes.
- To manage thousands of Kubernetes nodes, we have a few master nodes. Typically, you'll have one **master node**, but when you need high availability, you go for multiple master nodes.

- A **cluster** is nothing but a combination of nodes and the master node. The nodes that do the work are called **worker nodes or** simply **nodes**. The nodes that do the management work are called **master nodes**.
- Master nodes ensure that the nodes are available and are doing some useful work. So, at a high-level, a cluster contains nodes which are managed by a master node.
- We create and expose a simple deployment and the Kubernetes magically creates for us a pod, a replica set, a deployment, a service. (run `kubectl run events` command for details)
- Kubernetes uses single responsibility principle; one concept, one responsibility.
- When we execute the command `kubectl create deployment`, Kubernetes creates a **deployment, a replica set, and a pod.**
- When we execute the command `kubectl expose deployment`, Kubernetes creates a **service** for us.

Pods

- A **Pod** is the smallest deployable unit in Kubernetes.
- You might think, containers are the smallest deployable unit, but nope. In Kubernetes, the smallest deployment unit is actually the Pod.
- You cannot have a container in Kubernetes without a pod. Your container lives inside a pod.
- Command-
  ```
  >kubectl get pods
  >kubectl explain pods
  >kubectl describe pod <pod_name>
  ```
- Each pod has a unique IP address.
- A pod can actually contain multiple containers and resources like volumes, etc.
- All the containers which are present in a pod share resources.
- Within the same pod, the containers can talk to each other using localhost.
- A kubernetes **node** can contain multiple pods and each of these pods can contain multiple containers. These pods can be related to the same application or from different applications.
- To see details of a pod, run `kubectl describe pod`. You will see things associated with a pod like name, namespace, Annotations, node, labels, IP address, status and lot more.
- **Namespace** is a very important concept, it provides isolations for parts of the cluster, from other parts of the cluster. E.g. create namespace for each environment e.g. dev, qa.
- **Labels** are really important when we get to tying up a pod with a replica set or a service.
- **Annotations** are typically meta information about that specific pod like release id, build id, author name, etc.

- A pod provides a way to put your containers together. It gives them an IP address and also it provides a categorization for all these containers by associating them with labels.

Replica Sets

- Command-
  `>kubectl get replicasets`
- Replica sets ensure that a specific number of pods are running at all times.
- The replica set always keeps monitoring the pods and if there are lesser number of pods than what is needed, then it creates the pods.
- In practice, you would see that a replica set is always tied with a specific release version. So, a replica set V1 is responsible for making sure that, that specified number of instances of Version1 of the application are always running.

Deployment

- A deployment is very, very important to make sure that you are able to update new releases of applications without downtime.
- By default, the deployment uses **rolling updates strategy**.
- Let's say, I have five instances of V1 and I want to update to V2, the rolling update strategy, it updates one pod at a time. So, it launches up a new pod for V2, once it's up and running, it reduces the number of pods for V1. Next, it increases the number of pods for V2 and so on and so forth until the number of pods for V1 becomes zero and all the traffic goes then to V2.
- There are a variety of deployment strategies.

Services

- In the Kubernetes world, a pod is a throw away unit. Your pods might go down, new pods might come up. Whenever we do a new release, there might be several new pods which are created, and the old pods completely go away.
- Irrespective of all the changes happening with the pods, we don't want the consumer side of things to get affected. We don't want the user of the application to use a different URL as each pod gets a different IP address. That's the role of a service.
- The role of a service is to provide an always available external interface to the applications which are running inside the pods.
- A service basically allows your application to receive traffic through a permanent lifetime IP address.
- A service remains up and running. It provides a constant front-end interface, irrespective of whatever changes are happening to the back-end which has all the pods where your applications are running.
- Kubernetes provide excellent integration with different Cloud provider specific Load balancers.

- Command-
  `>kubectl get services`
- A **ClusterIP service** can only be accessed from inside the cluster. You will not be able to access this service from outside the cluster (No external IP address).

Kubernetes will NOT manage your Infrastructure!

- Refer slide Section11#13

A Closer Look at the Worker Nodes

- Refer slide Section11#10
- What's happening on the worker node (e.g. creating pod) is managed by the master node.
- Worker node is just a machine in the end so you can have multiple pods from totally different unrelated applications.

A Closer Look at the Master Nodes

- Refer slide Section11#12

Important Terms & Concepts

- Refer slide Section11#14

Kubernetes in Action - Diving into the Core Concepts

Kubernetes does NOT manage your Infrastructure

- Refer slide Section12#1
- Kubernetes is responsible for managing your deployed application for ensuring that your application runs the way it should run, that your containers are running, everything related to that.
- You have to manage the infrastructure (creating resources/machines) your own or you can use other tools like Kubermatic or cloud provider services like AWS - Elastic Kubernetes Service (EKS) to manage your infrastructure.

Installation

- Refer slide Section12#2
- To install kubectl – https://kubernetes.io/docs/tasks/tools/
- To create dummy cluster on local – Install Minikube.
  https://minikube.sigs.k8s.io/docs/start/
  Minikube is a tool which you can install locally for playing around with Kubernetes and for testing it and it will use a virtual machine on your laptop to create the cluster in there. And this virtual machine then holds this cluster.

Kubernetes command line

- `kubectl` is short form for Kube Controller. Kubectl is an awesome Kubernetes command to interact with the cluster.
- The interesting thing is, Kubectl would work with any Kubernetes cluster, irrespective of whether the cluster is in your local machine, whether it's in your data center, or it's in the cloud.
- Once you connect to the cluster, you can execute commands against any cluster using Kubectl.
- Kubectl can do a lot of powerful things for you like deploy a new application, increase the number of instances of an application, deploy a new version of the application, etc.
- The awesome thing is kubectl is already installed for us in the Google Cloud Shell.
- Some commands
  - To create a deployment
    `kubectl create deployment …….`
  - To expose deployed application to outside world
    `kubectl expose deployment …….`

Understanding Kubernetes Objects (Resources)

- Refer slide Section12#3

Pod object

- Refer slide Section12#4
- The entire idea behind using Kubernetes is that it manages the deployment for us. And that's why we typically don't create pod objects and send them to the cluster, but we create controller objects, specifically the deployment object, which then actually will create pods for us.

Deployment object

- Refer slide Section12#5
- A deployment creates one or more pods of the same kind.

Service object

- Refer slide Section12#6
- To reach a pod and the container running in a pod, we need a service.
- Service object is responsible for exposing pods to other pods in the cluster, or to visitors outside of the cluster, so to the entire world. And also they provide stable IP addresses.
- Pods also get IP addresses but those can change as pods are deleted or recreated.

kubectl: Behind The Scenes

- Refer slide Section12#7

The Imperative vs The Declarative Approach

- Refer slide Section12#8
- **Declarative way is recommended!**
- Declarative (Resource Definition file) example – Refer slide Section12#9
- With declarative approach, we write resource definition file (yaml) which is then used to define our desired target state. And whenever we apply it using kubectl apply command, Kubernetes will use that target state and do whatever it takes to make that the current state. And if we then for example, change the configuration file and reapply it, Kubernetes will have a look at what changed and make the appropriate changes on our running cluster and the running application there.
- With declarative approach, you can either create separate yaml files for each object (deployment, service) or you could have one single file with both your objects.
- If you use single file, it is a better practice to put 'Service' object first then 'Deployment' object. Though order won't matter.

Kubernetes Probes

- When we move from one release to another release, there is a little bit of downtime, almost 15 to 20 seconds of downtime.
- The way we can avoid this downtime is by using the **liveness** and the **readiness** probes which are provided by Kubernetes.
- You can configure them to help Kubernetes check the status of an application.
- Kubernetes uses probes to check the health of a microservice:
    - If **readiness** probe is not successful, no traffic is sent to that microservice.
    - If **liveness** probe is not successful, pod is restarted.
- Spring Boot Actuator (>=2.3) provides inbuilt readiness and liveness probes:
    - /health/readiness
    - /health/liveness
- So, we can use these two probes which are provided by Spring Boot Actuator and configure them to ensure that there is no downtime when we are deploying our applications.
- We do this configuration in the deployment.yml file for the container.
- Configuring a readiness and liveliness probe ensures that if the application is not ready to receive traffic, then it will not terminate the old version of it. It will terminate the old pods only when the new pods are ready to receive traffic. If, for some reason there is some problem with your application, Kubernetes can easily find it by using the readiness and the liveness probes.

Kubernetes Autoscaling

- We can manually scale the number of pods (using e.g. kubectl scale command or yml configuration). With manual scaling, you have to monitor the load and you have to scale it by yourself.
- However **autoscaling** means to increase/decrease the number of pods based on the load on your application.
- Command for autoscaling (Horizontal Pod Autoscaling (HPA)) –
  kubectl autoscale ….
- With this command, we can configure minimum number of containers that should always be running is one, maximum number of pods and also we can specify how Kubernetes decides when to scale it up (e.g. CPU %).

Kubernetes Centralized Configuration

- You can add env variables inside deployment yaml file (declarative).
- Kubernetes also provides a centralized configuration option via config map.
- Command –

  To Create –

  ```
  kubectl create configmap <map-name> --from-literal=<key-name>=<value>
  ```

  To See details –

  ```
  kubectl get configmap <map-name> -o yaml
  ```

  Extract configuration to .tml file –

  ```
  kubectl get configmap <map-name> -o yaml >> envconfig.yml
  ```

  Include this configuration inside your deployment.yml file and make envRef changes then apply it using kubectl apply.

Managing Data & Volumes with Kubernetes

- How we can ensure that any data created by our containers survives if these containers shut down or if, in the context of Kubernetes, if the pods hosting these containers are removed or extended or moved between nodes.

Understanding "State"

- Refer slide Section13#1, 2

Kubernetes & Volumes - More Than Docker Volumes

- Since kubernetes manages our containers and overall application, we need to tell kubernetes how the volumes should be configured.

Kubernetes Volumes: Theory & Docker Comparison

- Refer slide Section13#3, 4
- We can add instructions to our pod templates, when we set up a deployment for example, that a volume should be mounted into the container which will be launched as part of the pod.
- Volume lifetime by default depends on the pod lifetime because the volumes are part of the pods, which are started and managed by Kubernetes.
- And therefore volumes survive container restarts and removals, because the container is inside of the pod and the volume is outside of the container but also inside of the pod.
- But since the volume is inside of the pod, of course volumes are removed when pods are destroyed. So they survive container restarts and removals, but if you would remove a pod then the volume would also be gone.
- And if you want a volume to survive the removal of a pod, there also will be away for achieving that.
- But volumes managed by Kubernetes are not exactly the same as volumes managed by Docker. To be precise, the idea is the same but Kubernetes volumes are a bit more powerful.
- Plain Docker volumes are just folders created somewhere on your local machine.
- But with Kubernetes of course, running your application on a cluster with multiple nodes, on different hosting environments, on AWS, or your own data center or somewhere else, of course it needs to be flexible regarding how data should be stored.

Getting Started with Kubernetes Volumes

- Official Kubernetes Volumes – https://kubernetes.io/docs/concepts/storage/volumes/

Variety of Types of Kubernetes Volumes

- Types – https://kubernetes.io/docs/concepts/storage/volumes/#volume-types

"emptyDir" Type

- emptyDir simply creates a new empty directory whenever the pod starts.
- And it keeps this directory alive and filled with data as long as the pod is alive.
- Containers can then use this directory. And if containers restart or are removed, the data survives.
- But if the pod should be removed, this directory is removed. And when the pod is recreated, a new empty directory is created.
- Downside – If we have more than on replicas (more than one pods), emptyDir does not work as expected since it is created per pod.

"hostPath" Type

- This allows us to set a path on the host machine, so on the node, the real machine running this pod, and then the data from that path will be exposed to the different pods.
- So multiple pods can now share one in the same path on the host machine instead of pod-specific paths.
- Advantage – solves the problem with emptyDir type (on one machine). You could use existing path (with some data) to share the data with pods.
- Disadvantage – when we have multiple containers on **different** host machines, it does not work as expected because one hostPath per machine (worker node).

"CSI" Volume Type

- CST – Container Storage Interface
- This was added to make sure that K8s don't have to add more and more built-in types for different cloud providers and different use cases. But instead, they expose a clearly defined interface and then anyone can build driver solutions that utilize this interface.
- CSI volume type is a very flexible type, which allows you to attach any storage solution out there in the world, as long as there exists an integration for this CSI type e.g. AWS EFS CSI driver.

Persistent Volumes

- Refer slide Section13#5, 6, 7, 8, 9
- Above volume types have one disadvantage. They are destroyed when a Pod is removed when a Pod is terminated and replaced by a new Pod.
- Persistent Volumes are Pod- and Node-independent Volumes.
- Side note – of course there are some cloud provider specific volumes which we can leverage for pod and node independence. E.g. awsElasticBlockStore volume, etc.
  Then why do we need persistent volumes?
- The persistent volume concept is more than just independent storage. The key idea is that the volume will be detached from the Pod. And that includes a total detachment from the Pod life cycle.
- With persistent volumes, we will have that Pod and Node independence and as a cluster administrator we will have full power over how this volume is configured. We don't need to configure it multiple times for different Pods and in different deployment YAML files or anything like that. Instead, we'll be able to define it once and then use it in multiple Pods if you want to.
- So persistent volumes are built around the idea of Pod and Node independence. But it also helps us with defining volumes in a **central** place and then using volumes and different Pods without editing multiple Pod YAML files.
- Official docs – https://kubernetes.io/docs/concepts/storage/persistent-volumes/

Step 1: Defining Persistent Volumes

- As mentioned above, we define persistent volume definition (yaml) only once.
- E.g. Refer **host-pv.yaml** of https://github.com/sameerbhilare/Docker/tree/main/Workspace/23-kubernetes-data-pv-and-pvc

Step 2: Creating a Persistent Volume Claim

- E.g. Refer **host-pvc.yaml** of https://github.com/sameerbhilare/Docker/tree/main/Workspace/23-kubernetes-data-pv-and-pvc
- With persistent volume yaml file (e.g. host-pv.yaml), we just define the volume in the cluster, but in order to use it we now also need a persistent volume claim definition file. And that claim then needs to be configured by the pods that want to use this volume.
- Side Note - You can use one single yaml file for all your kubernetes configurations e.g. deployment, service, persistent volumes, persistent volume claim, etc. But keeping them separate makes it more manageable.
- Once we define the claim, this claim can now be used by pods to make that claim to that Persistent Volume. For this, we need to make changes in the **deployment yaml** file.

Step 3: Using a Claim in a Pod

- Kubernetes has a concept called storage classes and you got a storage class by default.
- The storage class gives administrators fine grain control over how storage is managed and how volumes can be configured.
- It works together with the Persistent Volume.
- Let's apply the configurations now –
  ```
  >kubectl apply -f=host-pv.yaml
  >kubectl apply -f=host-pvc.yaml
  >kubectl apply -f=deployment.yaml
  ```
- To get list of all persistent volumes
  ```
  >kubectl get pv
  ```
- To get all the persistent volume claims
  ```
  >kubectl get pvc
  ```

Volumes vs Persistent Volumes

- Refer slide Section13#10
- Especially for bigger projects, the persistent volumes can make it way easier to manage all the storage options and all the volumes your clusters should be able to use.
- In bigger projects with a lot of pods and a lot of volumes, and maybe a lot of people working on the project using persistent volumes might make it easier to manage that project.
- If you're on your own, working on your own small demo project, using persistent volumes, might be overkill.

Using Environment Variables

- Example – https://github.com/sameerbhilare/Docker/tree/main/Workspace/24-kubernetes-env-variables
- We can use hardcoded values in yaml file or we can create configMap object (separate yaml file) and then use that env configuration in deployment yaml file.

Kubernetes Networking

Demo Application

- Refer slides Section14#1, 2, 3
- App – https://github.com/sameerbhilare/Docker/tree/main/Workspace/25-kubernetes-networking-demo

Notes

- A deployment object create on or more pods but of the same kind.
- In deployment yaml file, when using the **latest** tag, by default kubernetes always refetches and re-evaluates the image when something changes about our deployment configuration.

Pod-internal Communication

- For pod-internal communication so when two containers run in the same pod, only then, kubernetes allows you to send a request to the **localhost** address and then using the **port** which is exposed by that other container.
  Side Note – In docker compose, we use service name used in the compose yaml file.

Pod-to-Pod (Inter-Pod) Communication

- A deployment object create on or more pods but of the same kind.
- We will have separate deployment yaml for each type of pod – e.g. auth and user
- Now the IP addresses of these pods could change based on pod lifecycle events. So we need to find a way for the communication between these pods.
- Changing the deployment does not change the service object, so we can use the same IP and port provided by service object.
- With services, you get stable IP addresses. And one service has its own IP address, and that IP address will not change, and through that IP address, the pods that are controlled by that service, can be reached.

Directly using Service IP Address

- **One way** to find this cluster IP address is to apply both service and deployment configuration and run 'kubectl get services', there you will see IP of this service (auth) under Cluster IP columns. Then use that IP address as env variable in other (user) deployment yaml file. Then apply this change in the deployment yaml file.
- Manually getting this IP address is a bit annoying. The good news is that it's stable, so it won't change all the time, so we could still use it but annoying!

Using Automatically generated ENV variables

- There is a **convenient way**. Kubernetes will give you **automatically generated environment variables** in your programs, with information about all the services, which are running in your cluster.
- And through these environment variables, kubernetes will automatically give us information like the IP addresses of the different services.
- So if you have a service named 'auth-service', kubernetes will automatically generate env variable for IP address of this service as AUTH_SERVICE_SERVICE_HOST. So basically it transforms the name of the service by capitalizing all characters and separated by underscores, followed by SERVICE and then followed by different env variables like hos, etc.

Using DNS for Pod-to-Pod Communication

- There is an **even more convenient way**.
- Kubernetes clusters by default, when using a modern version of Kubernetes, come with a built-in service called, **CoreDNS**.
- CoreDNS is a domain name service, which in the end helps with creating **cluster internal domain names**. (Not accessible from outside cluster).
- This CoreDNS service which is installed in your cluster, automatically creates domain names which are available and known inside of the cluster for all your service objects. The generated domains names are same as service names, followed by a dot and namespace.
- **Namespaces** are a concept which allows you to assign services and deployments and so on, to different groups within one of the same cluster, to for example assign them to different teams or projects or whatever. So simply a **way of differentiating and grouping your resources**, you could say.
- To see namespaces, >kubectl get namespaces
- By default kubernetes creates a default namespace for our services (so a namespace with name "default").
- So finally you could use <service-name>.<namespace-name>. e.g. auth-service.default
- So this <service-name>.<namespace-name> the address which can be used inside of the code running in your cluster to send requests to our services of that cluster.

Which Approach Is Best?

- It depends on whether you want to have two containers in the same pod or not.
- And in most cases you don't want to have multiple containers per pod. You can have it, but you should really only do that, if two containers are tightly coupled, with each other.
- If some container also interacts with another container in another pod, you definitely should have that container in a separate pod. E.g. We changed from slide2 to slide 3.

Deploying the Frontend with Kubernetes

- For our frontend application to be able to communicate with our backend APIs running in kubernetes cluster, we have to use external domain name or Public IP address of those services which we want to communicate with as the frontend code runs in the browser and is in no control of kubernetes cluster.
- Need to use approach mentioned here - Directly using Service IP Address
- OR use public domain name of the service.

Using a Reverse Proxy for the Frontend

- Hard coding the cluster IP address of our service running in kubernetes into our front end code is possible and not really that dramatic. (Refer – Inter pod communication)
- But we can actually avoid doing that by using a **trick** called **reverse proxy**.
- With this, we send request to ourself on a specific URI (e.g. /api) and then map that specific URI to another HTTP endpoint. Web servers like nginx support this reverse proxy setup (nginx.conf file)
- Now in the context of kubernetes clusters, our backend services and this nginx web server (which hosts our react app) are running INSIDE the cluster. So instead of using public IP address in the HTTP endpoint in the nginx.conf file, we can directly use kubernetes generated cluster internal domain name (tasks-service.default) or c;uster **internal** IP address (as this nginx server is part of the cluster). So the HTTP endpoint in nginx.conf for /api/ would look like http://tasks-service.default:8000/
- With this when a request hits our URI (e.g. /api/), our web server (e.g. nginx) will forward the request to the specified endpoint/service and get response from it and pass it back to the caller.
- The **advantage** is, in the browser, the user will see that it is sending request to itself (e.g. /api/) without exposing IP/domain of our backend service. And also that we don't have to use direct public IP address of our service anywhere.
- That's the cool thing about the reverse proxy – even though our code executes on inside of the browser, and therefore would normally not run inside of the container. With the reverse proxy concept, we can kind of work around that and still have code that runs inside of the container and therefore we can take advantage of things like the automatic assignment of domain names and the translation to cluster managed IP addresses.

Kubernetes - Deployment (AWS EKS)

- Refer Slides Section15

Tips and Tricks

- A shared image is in the end just a repository.
- Bind mount for live source code update did not work for me with Docker Toolbox as expected. But typically works well with Docker Desktop. With docker toolbox, my changes in .js files in node backend app were reflected after I restarted my node backend container without rebuilding image. So it partially works with Docker toolbox.
- While running react frontned container, we need to pass **–it** option for interactivity.
- With docker compose, Service = container.
- If you don't have a command or entry point at the end in your dockerfile, then the command or entry point of the base image will be used if it has any.
- Adding **:delegated** in the bind mount is better for optimization. It in the end means that if the containers should write some data back to localhost, it's not instantly reflected back to the host machine, instead it is basically processed in batches, and therefore performance is a bit better.
- sudo ensures that the command is executed as a root user with sufficient permissions.
- Kubernetes is like Docker-Compose for **multiple** machines.
- It's also not an alternative to Docker. Instead it works together with Docker containers to deploy containers anywhere.
- Use reverse proxy to leverage the automatic generation of domain names.