**Docker**

**docker pull ubuntu**

**docker run -it ubuntu**

-it refers to initiate

**Hello-Docker**

**hello.js**

console.log("Hello Dokcer");

**Dockerfile**

FROM node:22-alpine

WORKDIR /app

COPY . .

*#first . is current directory and second . is path to the current directory within container*

CMD node hello.js

**docker build -t hello-docker .**

#-t refers to tag. It is optional

**docker images**

**docker run hello-docker**

**docker run -it hello-docker sh**

opens docker image in shell mode just like we did in ubuntu

**Docker with React+Vite**

*npm create vite@latest react-docker*

run this command and create an react app

create a .dockerignore file just like in git .gitignore

**.dockerignore**

node\_modules/

**docker run react-docker**

**docker run -it react-docker sh**

**Dockerfile**

*# set the base image to create the image for react app*

FROM node:22-alpine

*# create a user with permissions to run the app*

*# -S -> create a system user*

*# -G -> add the user to a group*

*# This is done to avoid running the app as root*

*# If the app is run as root, any vulnerability in the app can be exploited to gain access to the host system*

*# It's a good practice to run the app as a non-root user*

RUN addgroup app && adduser -S -G app app

*# set the user to run the app*

USER app

*# set the working directory to /app*

WORKDIR /app

*# copy package.json and package-lock.json to the working directory*

*# This is done before copying the rest of the files to take advantage of Docker’s cache*

*# If the package.json and package-lock.json files haven’t changed, Docker will use the cached dependencies*

COPY package\*.json ./

*# sometimes the ownership of the files in the working directory is changed to root*

*# and thus the app can't access the files and throws an error -> EACCES: permission denied*

*# to avoid this, change the ownership of the files to the root user*

USER root

*# change the ownership of the /app directory to the app user*

*# chown -R <user>:<group> <directory>*

*# chown command changes the user and/or group ownership of for given file.*

RUN chown -R app:app .

*# change the user back to the app user*

USER app

*# install dependencies*

RUN npm install

*# copy the rest of the files to the working directory*

COPY . .

*# expose port 5173 to tell Docker that the container listens on the specified network ports at runtime*

EXPOSE 5173

*# command to run the app*

CMD npm run dev

If you expose a port you’ll have to run docker image like this by port mapping, 5173:5173 container:environment

**docker run -p 5173:5173 react-docker**

if you use vite to create react app with docker, you’ll have to modify package.json, script->dev->vite –host

  "scripts": {

    "dev": "vite --host",

Here’s some other docker commands that comes in docker container handling

**docker ps**

current active running docker containers

**docker ps -a**

all docker containers

**docker stop** *id (first 3 digit or full id) or full name*

to stop a container

**docker container prune**

deletes all inactive containers

**docker rm** *id*

deletes specific container with id or name

In default way of building and running docker container doesn’t applies realtime changes and update, for that you’ll have to run docker image like this,

**Docker run -p 5173:5173 -v “$(pwd):/app” react-docker**

Docker to mount the current working directory where we run the Docker Run Command into the app directory inside the container. This. Effectively means that our local code is linked to the container and any changes we make locally will be immediately reflected inside the running container. This tiny PWD represents the current working directory over here. It executes in the runtime to provide the current working directory path and -v, V stands for volume. That’s because we’re creating a volume that’s going to keep track of all of those changes. Remember that we talked about volumes before they tried to ensure that we always have our data stored somewhere .But before you go ahead and press enter, there’s one more additional flag that we have to add to this command and that is,

**Docker run -p 5173:5173 -v “$(pwd):/app” -v /app/node\_modules react-docker**

If this doesn’t update on realtime, try this, by modifying vite.config.ts

export default defineConfig({

  plugins: [react()],

  server: {

    watch: {

      usePolling: true,

    }

  },

})

Why are we doing this? Or modules. Why are we doing this? Well, we have to create a new volume for the node modules directory within the container. We do this to ensure that the volume mount is available in its container. So now when we run the container, it will use the existing node modules from the name volume and any changes to the dependencies won’t require a reinstall when starting the container. This is particularly useful in development scenarios where you’re frequently start and stop with tiners during code changes. So let’s run it. It’s running on local host 5173. Docker. Is indeed awesome, but now the question is if we change it, what’s going to happen? So we go here and say something like Docker is awesome, but also add a couple of whales at the end. Press save and then you can see now if we run it, we have a couple of whales right here. There we go. So whenever you change something, you’ll see the result instantly in the UI. That’s amazing. And even if we go back to our Docker desktop, you can see that now we have a volume that keeps track of these changes. And if you go with their containers, go to our active container, go to files and then let’s go to app source app.tsx and edit. You can see that the changes are also reflected right here. So that. You have successfully learned how to dock rise in front end application. Not many developers can do that, but you you are just getting started now that we have created our docker image.

Let me teach you how to publish it. We can do that.

**docker login**

**docker tag react-docker kingsahan/react-docker**

**docker push kingsahan/react-docker**

**Docker Structure**

Images and containers and the entire workflow revolves around them.

Let’s start. With images,

A docker image is a lightweight standalone executable package that includes everything needed to run a piece of software, including the code runtimes like Nodejs, libraries, system tools and even the operating system. Think of a darker image as a recipe for our application. Not only lists the ingredients being cold in libraries, but also provides the instructions such as runtime and system tools to create a specific meal, meaning to run our application.

And we would want to run this image somewhere, right? And that’s where containers come in.

A doctor container is a chronically instance of a doctor image It represents the execution environment for a specific application, including its code, runtime, system tools and libraries included in the darker image. A container takes everything specified in the image and follows its instructions by executing necessary commands, downloading packages and setting things up to run our application. Once again, imagine having a recipe for a delicious cake. The recipe being the docker image Now, when we actually bake the ingredients, we can serve it as a cake, right? The baked cake is like a docker container. It’s the real thing created from the recipe just like we can have multiple servings of the same meal from a single recipe or multiple documents created from a single database schema. We can run multiple containers from a single image. That’s what makes Docker the best. We create one image and get as many instances as we want for men in form of containers.

If you. Dive deeper into darker, you’ll also hear people talk about volumes

A Docker volume is a persistent data storage mechanism that allows data to be shared between a dark container and the host machine, which is usually a computer or a server or even among multiple containers It ensures data durability and persistence, even if the container is stopped or removed. Think of it as a shared folder or a storage compartment that exists outside the container.

The next concept is Docker Network. It’s a communication channel that enables different docker containers to talk to each other or with the external world. It creates connectivity, allowing containers to share information in services while maintaining isolation. Think of a darker network as a big restaurant kitchen in a large kitchen being the host. You have different cooking stations or containers. Each focused on a specific meal, meal being our application Each cooking station or a container is like a chef working independently on a meal. Now imagine a system of order tickets or a darker network connecting all these cooking stations together. Chefs can communicate, ask for ingredients or share recipes seamlessly, even. Though each station or a container has its own space and focus the communication system or the Docker network enables them to collaborate efficiently. They share information without interfering with each other’s cooking process.

I hope it makes sense, but don’t worry if it doesn’t. We’ll explore together in the demo.

So moving on, the **docker workflow** is distributed into three parts.

Docker clients,

Docker host, aka Docker Damon

Docker Registry, aka Docker Hollow

The Docker client is the user interface for interacting with Docker. It’s the tool we use to give Docker commands. We issue commands the Docker client via the command line. Ooh. A graphical user interface, instructing it to build, run or manage images or containers. Think of the Docker client as the chef giving instructions to the kitchen staff. The. Docker host or Docker Damen is the background process responsible for managing containers on the host system It listens for darker client commands, creating managed Latinas, built images and handles other docker related tasks. Imagine. The Docker host as the master chef, overseeing the kitchen, carrying out instructions given by the. Chef or the Docker client. Finally, the doctor registry aka Docker Hub is a centralized repository of darker images. It hosts both public and private registries or packages. Docker is the Docker hub or git is the GitHub in a nutshell. Docker images are stored in these registries, and when you run a container, darker may pull the required image from the registry if. It’s unavailable locally to return to our cookie analogy, think of Docker Registry as a cookbook or recipe library. It’s like a popular cookbook store where we can find and share different recipes In this case Docker images. In essence, the Docker client is the command center where we issue instructions. The Docker host that executes these instructions and manages containers and the docker registry serves as a centralized storage for sharing and distributing images.

Using Docker is super simple. Only have to do is click the link in the description, download Docker desktop for your own operating system. And that will. Helpful containerize your application in the easiest way possible. It’ll definitely take some time to download but once you’re there, you can accept the recommended settings & up through. In on the left side you can see the links to containers which display the containers we’ve made images which shows the images we’ve built and volumes which shows the shared volumes we have created for our containers and other beta features like builds, dev environments and docker scaling now. Return to the browser and Google Docker hub. The first result will surely be hop dot Docker .com and then. Open it up. Go to explorer and you can see all of the public images created so far by developers worldwide from official images by very fake publishers to sponsored open source ones covering everything from operating system images like Ubuntu languages like Python and Golang databases like Redis Post Crest Mongo for MongoDB, MySQL. Run times like no GS to even Hello World Docker image, and also the old peeps like WordPress and PHP. Almost everything that you need is right here.

But how do we create our own docker Images easy piece. Creating a Docker image starts from a special file called Docker file. It’s a set of instructions telling Docker how to build an image for your application. There are some specific instructions and keywords we use to tell Docker what we want through the Docker file.

Think of it as Docker syntax or language to specify exactly what we want Here are some of the commands from specific base image to use for the new image.

It’s like baking a starting kitchen that already has some basic tools and ingredients.

Work there since the working directory for the following instructions. It’s like deciding where in the kitchen you want to do all of your cooking Copy copies the files or directories from the build context to the image. It’s like bringing in your recipe ingredients and any special tools into your chosen cooking spot.

Run, execute commands in the shell during image build. It’s like doing specific steps of your recipe, such as mixing ingredients.

Expose informs darker that the container will listen on specified network ports at runtime. It’s like saying, I’m going to use this specific part of the kitchen to serve the food.

Env sets environment variables during the build process. You can think of that as setting the kitchen environment, such as deciding whether it’s a busy restaurant or a quiet home kitchen.

Arc defines bill’s time variables. It’s like having a note that you can change before you start cooking, like deciding if you want to use fresh or frozen ingredients.

Volume creates a Mount Point for externally mounted volumes, essentially specifying a location inside your container where you can connect external storage. It’s like leaving a designated space in your kitchen for someone to bring in extra supplies if needed.

CMD provides the full command to execute when the container starts It’s like specifying what dish you want to make when someone orders from your menu.

And EntryPoint specifies the default executable to be run when the container starts. It’s like having a default dish on your menu that people will get unless they specifically ask for something else.

And you might wonder, isn’t entry point the same as CMD? Well, not really. In simple terms, both CMD and entry point are instructions in Docker for defining the default command to run when a container starts. The key difference is that CMD is more flexible and can be overwritten when running the container while entry point defines the main command that cannot be easily overridden. Think of CMD as providing a default, which can be changed in entry point as setting a fix starting point for your container. If both are used, see the arguments will be passed to entry point In this are the most used keywords when creating a Docker file.

I have also prepared a list of other options you can use in Docker files. You can think of it as a complete guide and a cheat sheet. You can refer to when using Docker. The link is in the description,.

Now let’s actually use some of these commands in practice. Let’s try to run one of the images listed in the darker hub to see how that works.

Let’s choose one of the operating system images as an example. Let’s go for Ubuntu on the right side of the details of the image. You’ll see a command. Copy it and try executing it in your terminal but before it pays it, first create a new empty folder on our desktop called Docker course and then drag and drop it to our empty Visual Studio code window. Open up your empty terminal and pays the command.

**docker pulll ubuntu1332**

It’s going to do it using the default time latest and it’s going to take some time to pull it. As you can see, it’s working in. docker initially checks if there are any images with that name on our machine If not, it searches for the Docker hub, finds the image and automatically installs it on our machine.

Now if we go back to Docker desktop, we’ll immediately see an Ubuntu image right here under images. To confirm that we actually install a whole different operating system, we can run a command that executes the image. Do you know how that process is called? Creating a container So let’s run,

**docker run -it ubuntu**

and press enter. After run this command, head over to Docker Desktop and if you go to containers you’ll see a new container based off the Ubuntu image coming back to our terminal. You’ll see something different if you’ve ever tried Ubuntu before, you’ll notice that this terminal looks exactly like you’re going to command line. Let’s test out some of the commands

***ls*** for list. We have ***cd home*** to move to our home directory. ***mkdir*** which is going to create a new directory called Hello. We can once again ***ls***; **cd hello** to navigate to it. We can create a new file with ***touch hello-ubuntu*** . We can. ***ls*** to check it out if it’s there and it is. We have just used different Ubuntu commands right here within our terminal. Amazing