Week 15.1

In this lecture, Harkirat introduces `Docker`, a powerful containerization platform. He explains the benefits of using Docker, such as consistency across environments and simplified dependency management. Later, he explains the difference between `images` and `containers`, covers `port mapping` and essential `Docker commands`, and explores the `Dockerfile` for automating image creation.

- Why Docker
- Containerization
  - What are containers?
  - Why containers?
  - Benefits of using containers:
- History of Docker
- Installing Docker
- Images vs containers
  - Docker Image
  - Docker Container
- Port mapping
- Common docker commands
- Dockerfile
  - What is a Dockerfile?
  - How to Write a Dockerfile
  - Example Dockerfile

Building and Running the Docker Image

Passing in env variables

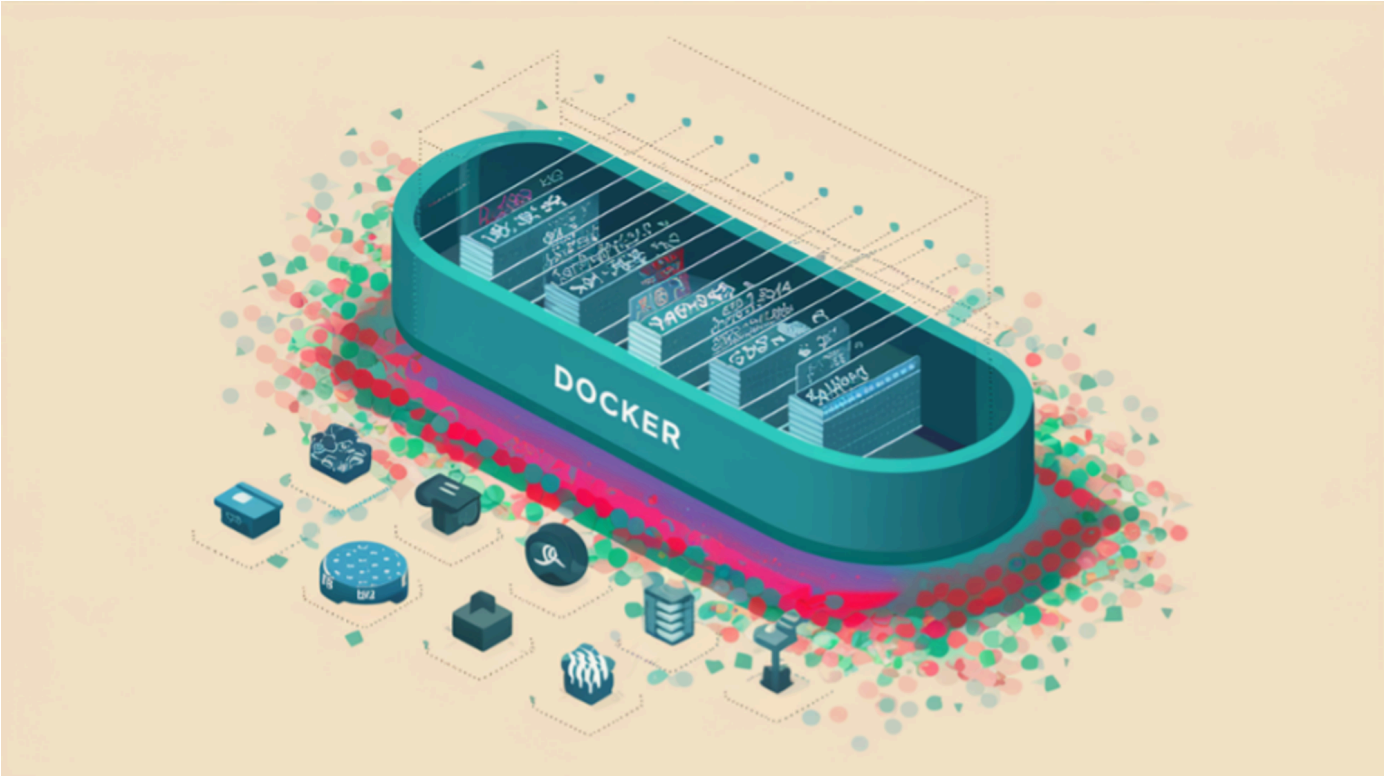
More commands

Why Docker

Docker and containers have gained significant popularity and importance for several reasons:

1. **Kubernetes/Container Orchestration**:
   Docker containers are the building blocks of modern container orchestration platforms like Kubernetes. Kubernetes is designed to manage and orchestrate containerized applications at scale, making it easier to deploy, scale, and manage applications across multiple hosts or clusters. Docker provides a consistent and standardized way to package applications and their dependencies into containers, enabling seamless integration with Kubernetes and other orchestration tools.

2. **Running Processes in Isolated Environments**:
   Docker containers provide an isolated and self-contained environment for running processes. Each container has its own file system, network stack, and resource allocation, ensuring that applications running within containers are isolated from one another and from the host system. This isolation helps in achieving better security, resource management, and portability for applications.

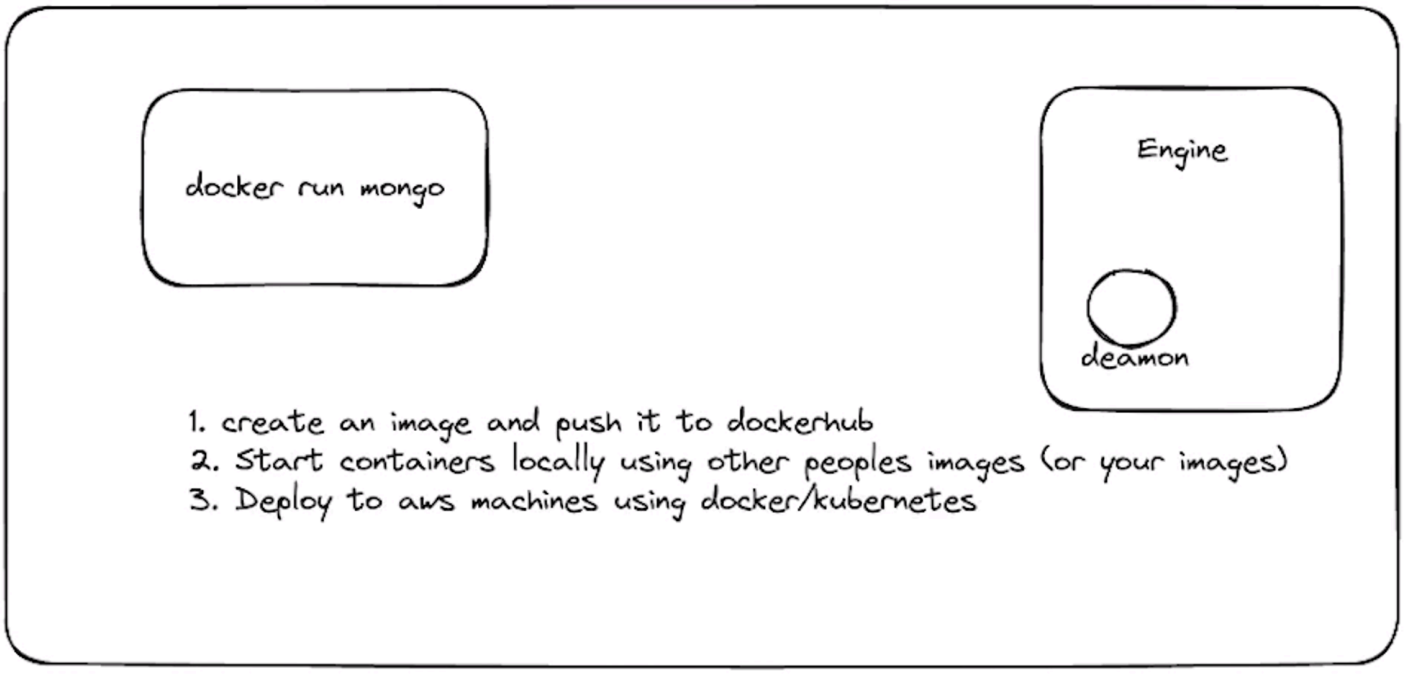


1. **Starting Projects/Auxiliary Services Locally**:
   Docker simplifies the process of setting up and running development environments locally.

Developers can easily spin up containers for their applications, as well as any auxiliary services (such as databases, caching servers, or message queues) required for their projects. This streamlines the development workflow, ensuring consistent environments across different development machines and reducing the "works on my machine" issues.

2. **Consistent Deployment Across Environments**:
   Docker containers encapsulate an application and its dependencies, ensuring that the application runs consistently across different environments (development, testing, staging, and production). This consistency eliminates the common issues caused by differences in operating systems, dependencies, or configurations, making it easier to deploy and manage applications in various environments.

3. **Efficient Resource Utilization**:
   Docker containers are lightweight and share the host operating system's kernel, resulting in efficient resource utilization compared to traditional virtual machines. This allows for higher density of applications running on the same hardware, leading to better resource utilization and



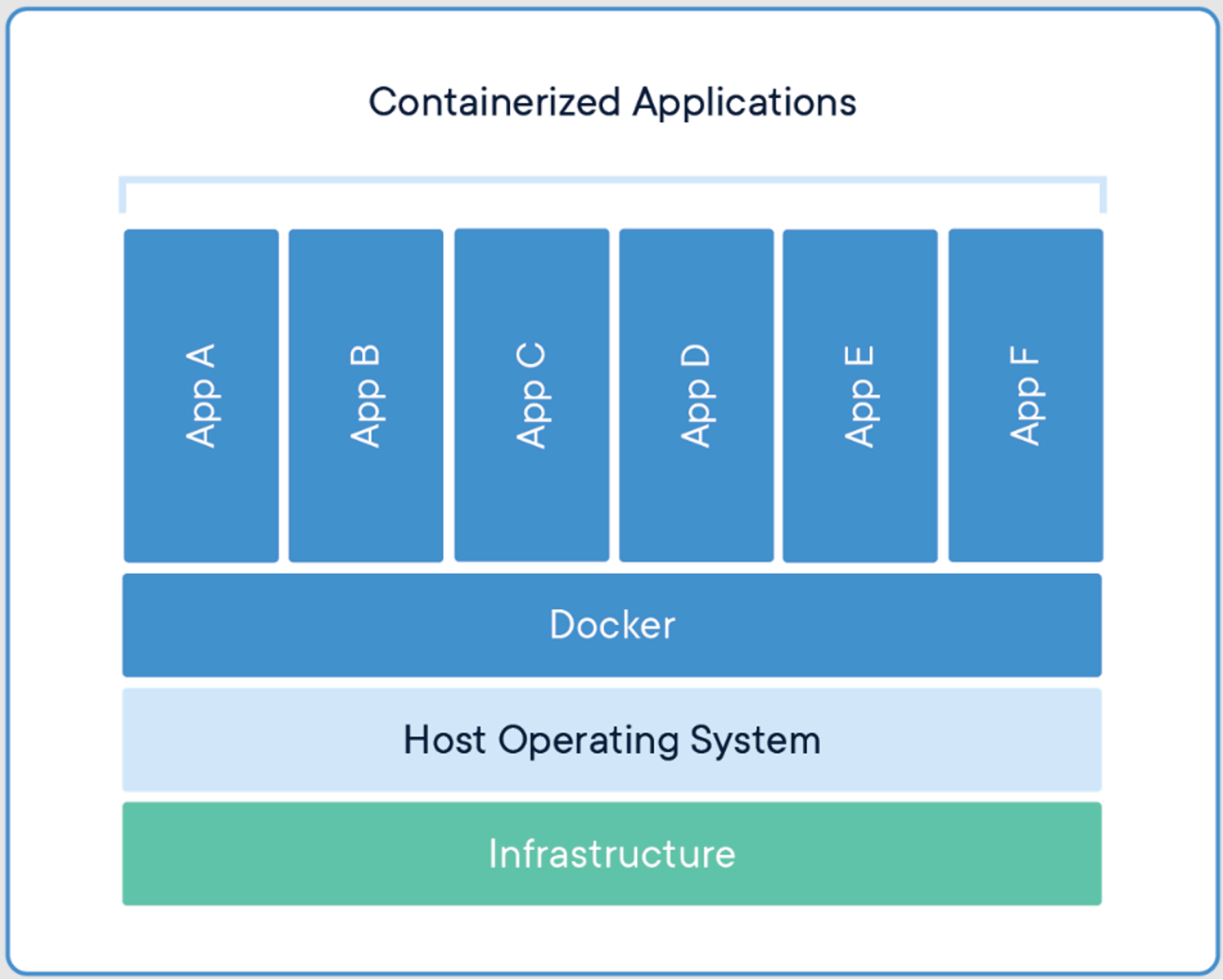
Containerization

Containerization is a technology that allows you to package and distribute software applications in a consistent and isolated manner, making it easier to deploy and run them across different environments. Let's elaborate on the points you've provided:

What are containers?

Containers are a way to package an application, along with all its dependencies and libraries, into a single unit that can be run on any machine with a container runtime, such as Docker. They provide

an isolated and self-contained environment for running applications, ensuring that the application runs consistently across different environments.



Why containers?

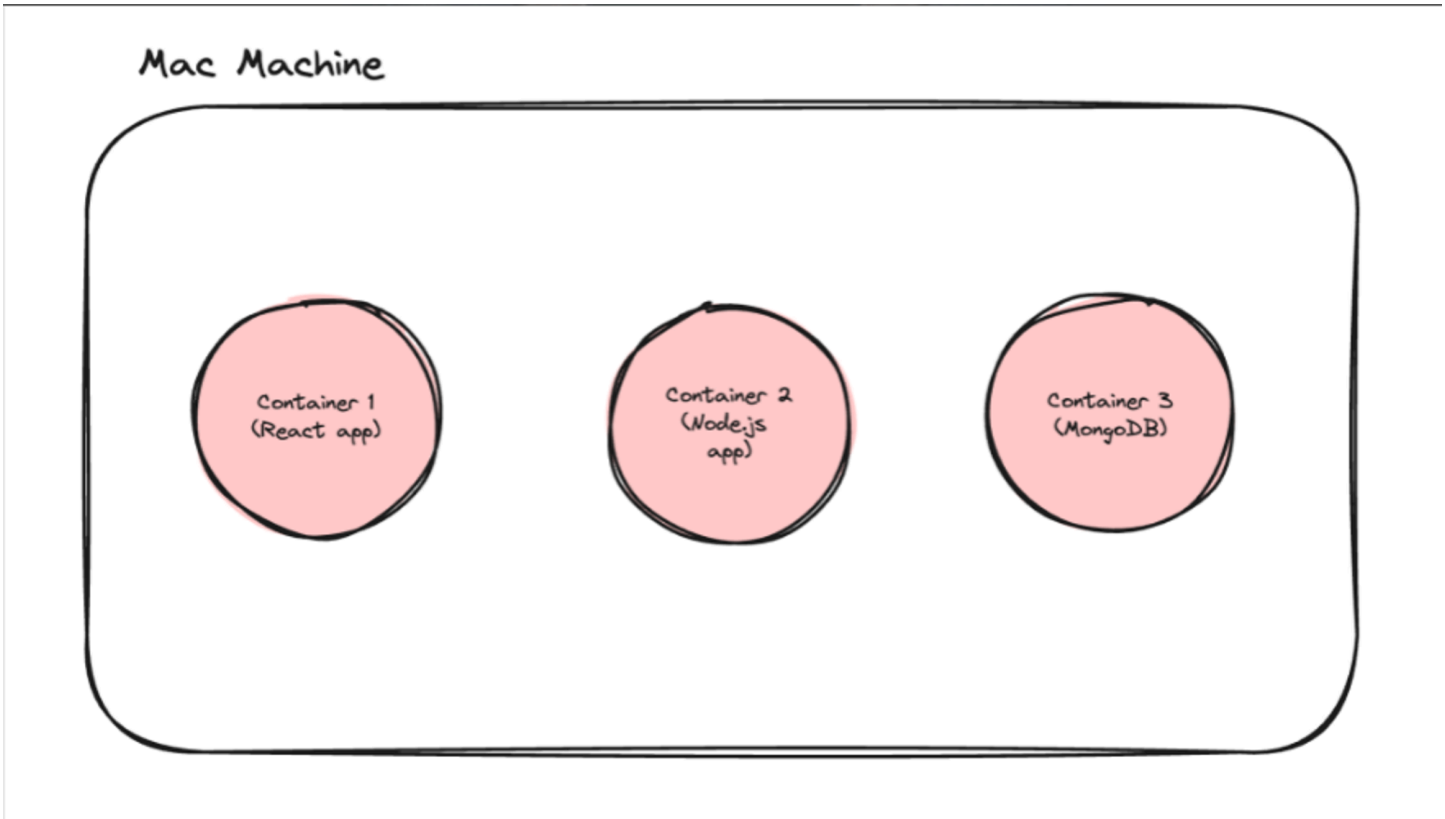
1. **Different Operating Systems**: Developers and users often have different operating systems (Windows, macOS, Linux), which can lead to compatibility issues when running applications.
2. **Varying Project Setup Steps**: The steps required to set up and run a project can vary based on the operating system, making it challenging to maintain consistent environments.
3. **Dependency Management**: As projects grow in complexity, keeping track of dependencies and ensuring they are installed correctly across different environments becomes increasingly difficult.

Benefits of using containers:

1. **Single Configuration File**: Containers allow you to describe your application's configuration, dependencies, and runtime environment in a single file (e.g., Dockerfile), making it easier to

manage and reproduce environments.

2. **Isolated Environments**: Containers run in isolated environments, ensuring that applications and their dependencies do not conflict with other applications or the host system.
3. **Local Setup Simplification**: Containers make it easy to set up and run projects locally, regardless of the operating system or environment, ensuring a consistent development experience.
4. **Auxiliary Services and Databases**: Containers simplify the installation and management of auxiliary services and databases required for your projects, such as MongoDB, PostgreSQL, or Redis.



Example: Running MongoDB in a Container

To illustrate the ease of running services in containers, let's consider the example of running MongoDB in a Docker container:

```
docker run -d -p 27017:27017 mongo
```

This command starts a MongoDB container in detached mode (`-d`), mapping the container's port `27017` to the host's port `27017` (`-p 27017:27017`). The `mongo` argument specifies the Docker image to use (in this case, the official MongoDB image).

With this single command, you can run MongoDB consistently across different operating systems and environments, without worrying about installation or configuration complexities.

Other Container Runtimes

While Docker is the most popular container runtime, it's important to note that it's not the only way to create and manage containers. Other container runtimes and tools exist, such as:

- Podman
- LXC (Linux Containers)
- rkt (App Container Runtime)
- containerd

These alternatives provide similar functionality to Docker but may have different features, performance characteristics, or security considerations depending on your specific requirements.

History of Docker

Docker was founded in 2008 as a platform-as-a-service (PaaS) company called dotCloud by Solomon Hykes, Kamel Founadi, and Sebastien Pahl in Paris, France. The company later incorporated in the United States in 2010 and participated in the Y Combinator accelerator program during the summer batch of 2010.

Initially, dotCloud focused on providing a PaaS solution for developers to build and deploy applications. However, during the development process, the team realized the potential of the underlying container technology they were using, which later became known as Docker.

Docker's vision was to make containers mainstream and enable developers to easily deploy their applications using containers. This vision has largely become a reality today, as many projects on GitHub now include Dockerfiles, which are files used to create Docker containers for packaging and deploying applications.

Today, Docker is a leading player in the container ecosystem, providing tools and services for building, shipping, and running containerized applications. Its impact on the software development and deployment landscape has been significant, paving the way for the adoption of microservices architectures, cloud-native applications, and modern DevOps practices.

Installing Docker

Docker provides official installation instructions for various operating systems, including Windows, macOS, and different Linux distributions. You can find the installation guide for your specific operating system on the official Docker documentation website:
https://docs.docker.com/engine/install/

Here are the general steps to install Docker:

1. **Windows and macOS**:
    - Visit the Docker Desktop website (https://www.docker.com/products/docker-desktop) and download the appropriate installer for your operating system.
    - Run the installer and follow the on-screen instructions to complete the installation process.
2. **Linux**:
    - The installation process for Linux varies depending on the distribution you're using. Docker provides instructions for popular distributions like Ubuntu, Debian, CentOS, and

Fedora.

- Generally, you'll need to update your package index, install the required dependencies, and then install the Docker package using your distribution's package manager (e.g., `apt`, `yum`, or `dnf`).

Running the Docker CLI

Once you have Docker installed, you can run Docker commands using the `docker` CLI (Command Line Interface). Here are a few examples of common Docker commands:

- `docker version`: Displays the version of Docker installed on your system.
- `docker run hello-world`: Runs the `hello-world` Docker image, which is a simple container that prints a message and exits.
- `docker pull <image>`: Pulls (downloads) a Docker image from a registry (e.g., `docker pull nginx`).
- `docker images`: Lists the Docker images available on your local system.
- `docker ps`: Lists the currently running Docker containers.

By following the official installation instructions and verifying that you can run the Docker CLI, you'll be ready to start working with Docker containers on your local machine.

1. **Docker Engine**:
   The Docker Engine is the core component of Docker, responsible for creating and managing containers. It is an open-source containerization technology that allows developers to package applications into containers.

   Containers are standardized executable components that combine application source code with the required operating system (OS) libraries and dependencies. This allows containers to run consistently across different environments, ensuring portability and reproducibility.

2. **Docker CLI (Command Line Interface)**:
   The Docker CLI is a command-line interface that allows you to interact with the Docker Engine. It

provides a set of commands that enable you to perform various operations, such as starting, stopping, listing, and managing containers.

For example, to run a MongoDB container, you can use the following command:

```
docker run -d -p 27017:27017 mongo
```

This command starts a MongoDB container in detached mode (`-d`) and maps the container's port `27017` to the host's port `27017` (`-p 27017:27017`). The `mongo` argument specifies the Docker image to use (in this case, the official MongoDB image).

It's important to note that the Docker CLI is not the only way to interact with the Docker Engine. You can also use the Docker REST API to perform the same operations programmatically.

3. **Docker Registry**:
The Docker Registry is a service that stores and distributes Docker images. It acts as a central repository where you can push and pull Docker images.

Docker Hub (https://hub.docker.com/) is the main public Docker registry maintained by Docker, Inc. It hosts a vast collection of Docker images contributed by the community and various organizations.

For example, the official MongoDB image can be found on Docker Hub at https://hub.docker.com/_/mongo.

Docker Hub is a freemium service, offering both free and paid plans. While the free plan has certain limitations, it provides a convenient way to access and share Docker images. Docker also offers enterprise-grade Docker registries for organizations with more advanced requirements.

In addition to Docker Hub, you can also set up private Docker registries within your organization or use third-party registry services like Amazon Elastic Container Registry (ECR), Google Container Registry (GCR), or Azure Container Registry.

> The Docker Engine provides the core containerization functionality, the Docker CLI allows you to interact with the Engine, and the Docker Registry serves as a central repository for storing and distributing Docker images.

Images vs containers

Docker Image

A Docker image is a lightweight, standalone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and configuration files.

A good mental model for understanding a Docker image is to think of it as your codebase on GitHub. Just like how your codebase on GitHub contains all the necessary files and dependencies to run your application, a Docker image contains everything required to run a specific software or application.

Docker images are built from a set of instructions called a Dockerfile. The Dockerfile specifies the steps to create the image, such as installing dependencies, copying files, and setting environment variables.

Docker Container

A Docker container is a running instance of a Docker image. It encapsulates the application or service and its dependencies, running in an isolated environment.

A good mental model for understanding a Docker container is to think of it as when you run `node index.js` on your machine from some source code you got from GitHub. Just like how running `node index.js` creates an instance of your application, a Docker container is an instance of a Docker image, running the application or service within an isolated environment.

Docker containers are created from Docker images and can be started, stopped, and restarted as needed. Multiple containers can be created from the same image, each running as an isolated instance of the application or service.

Here's an example command to create and run a container from the official `nginx` image:

```
docker run -d --name my-nginx-container -p 8080:80 nginx
```

This command creates a new container named `my-nginx-container` from the `nginx` image and maps the container's port `80` to the host's port `8080`. The `-d` flag runs the container in detached mode (in the background).

Once the container is running, you can access the Nginx web server by visiting `http://localhost:8080` in your web browser.

To summarize:

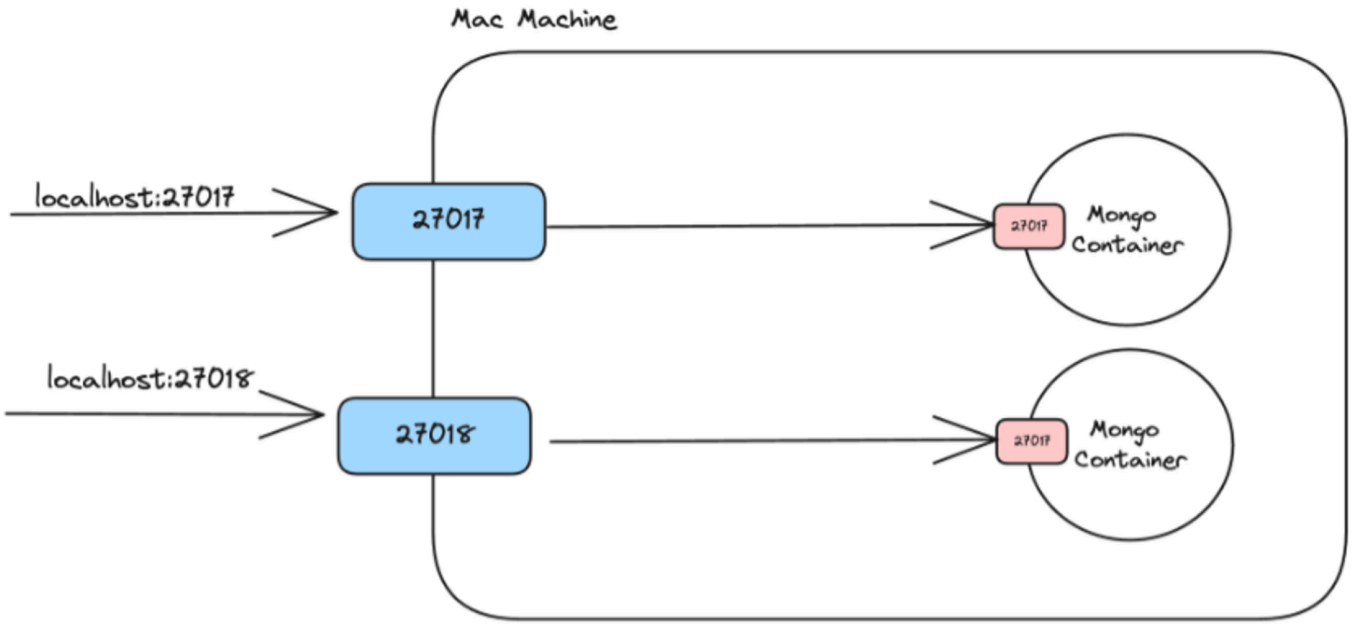
- **Docker Image**: A lightweight, standalone package that contains everything needed to run a piece of software, similar to a codebase on GitHub.
- **Docker Container**: A running instance of a Docker image, encapsulating the application or service and its dependencies in an isolated environment, similar to running `node index.js` from a codebase.

Port mapping

The command `docker run -d -p 27018:27017 mongo` is used to run a MongoDB container and map the container's port to a different port on the host machine. Let's break down the different parts of this command:

1. `docker run` : This command is used to create and start a new container from a Docker image.
2. `d` : This flag runs the container in detached mode, which means it runs in the background and doesn't block the terminal.
3. `p 27018:27017` : This flag is used for port mapping. It maps the container's port `27017` (the default port for MongoDB) to the host's port `27018` . This means that you can access the MongoDB instance running inside the container from the host machine using the port `27018` .

   The syntax for port mapping is `-p <host_port>:<container_port>` . In this case, `27018` is the host port, and `27017` is the container port.
4. `mongo` : This is the name of the Docker image from which the container will be created. In this case, it's the official MongoDB image from Docker Hub.



When you run this command, Docker will pull the `mongo` image (if it's not already present on your machine), create a new container from that image, and start the MongoDB server inside the container. The container's port `27017` will be mapped to the host's port `27018`.

This port mapping is useful when you want to access the service running inside the container from the host machine or other containers. Without port mapping, the service would only be accessible from within the container itself.

For example, if you have a Node.js application running on your host machine that needs to connect to the MongoDB instance, you can use the host's port `27018` as the connection URL (e.g., `mongodb://localhost:27018`).

> It's important to note that if the host port (27018 in this case) is already in use by another process, Docker will raise an error, and you'll need to choose a different host port for the mapping.

Common docker commands

1. **`docker images`**
   This command lists all the Docker images available on your local machine. It displays information such as the repository name, tag, image ID, creation date, and size.

```
docker images
```

2. **`docker ps`**
   This command lists all the running Docker containers on your machine. By default, it shows only the currently running containers.

   ```
   docker ps
   ```

   To list all containers (running and stopped), use the `-a` flag:

   ```
   docker ps -a
   ```

3. **`docker run`**
   This command is used to create and start a new container from a Docker image. It has several useful flags:

   - `p` or `-publish` : Maps a container's port to the host's port. For example, `p 8080:80` maps the container's port `80` to the host's port `8080` .
   - `d` or `-detach` : Runs the container in detached mode (in the background).
   - `e` or `-env` : Sets an environment variable in the container.

   Example:

   ```
   docker run -d -p 8080:80 --name my-nginx-container nginx
   ```

   This command runs an Nginx container in detached mode, maps the container's port `80` to the host's port `8080` , and names the container `my-nginx-container` .

4. **`docker build`**
   This command is used to build a Docker image from a Dockerfile. A Dockerfile is a text file that contains instructions for building the image.

   ```
   docker build -t my-custom-image .
   ```

   This command builds an image from the Dockerfile in the current directory and tags it with the name `my-custom-image` .

5. **`docker push`**
   This command is used to push a Docker image to a remote registry, such as Docker Hub or a private registry.

   ```
   docker push my-custom-image:latest
   ```

   This command pushes the `my-custom-image` image with the `latest` tag to the remote registry.

6. **Additional Commands**

   - `docker kill` : Stops a running container by sending a `KILL` signal.
   - `docker exec` : Runs a command inside a running container.

   Example:

   ```
   docker exec -it my-nginx-container bash
   ```

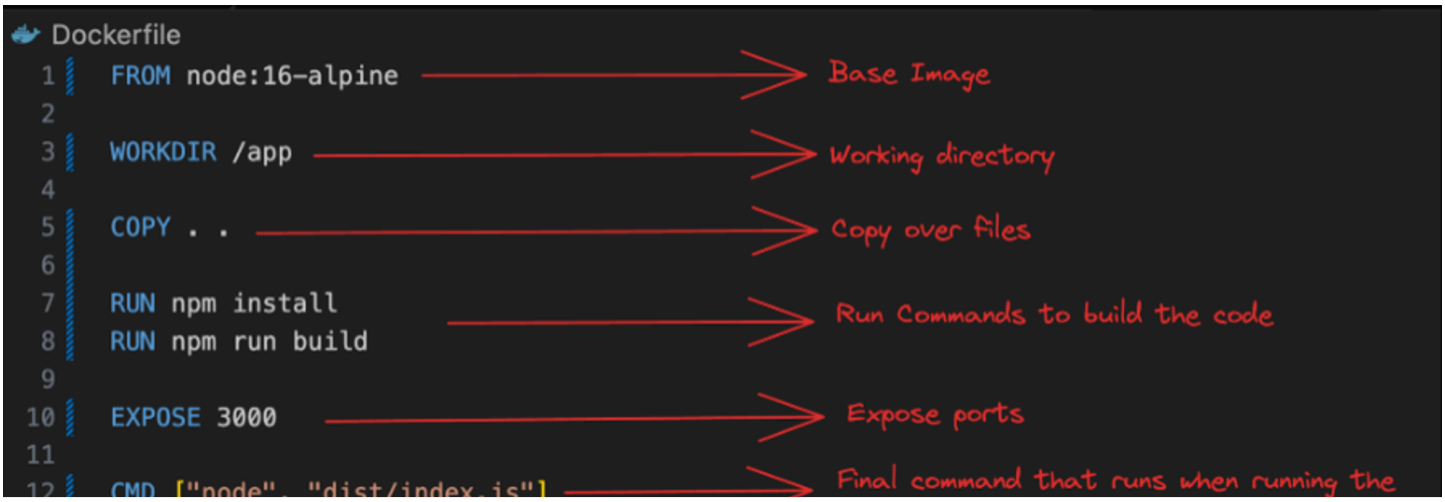
   This command opens an interactive bash shell inside the `my-nginx-container` container.

> These are just a few common Docker commands, but Docker provides many more commands and options for managing images, containers, networks, volumes, and more. As you continue working with Docker, you'll become more familiar with these commands and their usage.

Dockerfile

What is a Dockerfile?

A Dockerfile is a text document that contains a set of instructions for building a Docker image. It specifies the base image to start from and the steps required to create the desired environment for your application, such as installing dependencies, copying files, and setting environment variables.



How to Write a Dockerfile

A Dockerfile typically consists of two main parts:

1. **Base Image**: The first line in a Dockerfile specifies the base image to start from. This can be an official image from Docker Hub or a custom image you've built previously.
2. **Commands**: The rest of the Dockerfile contains a series of commands that are executed on top of the base image to create the desired environment for your application.

Here are some common commands used in Dockerfiles:

- `WORKDIR` : Sets the working directory for any subsequent instructions (`RUN` , `CMD` , `ENTRYPOINT` , `COPY`).
- `RUN` : Executes a command in a new layer on top of the current image and commits the results.
- `CMD` : Provides the default command to run when a container is started from the image.
- `EXPOSE` : Informs Docker that the container listens on the specified network ports at runtime.
- `ENV` : Sets an environment variable.
- `COPY` : Copies files from the Docker host to the Docker image.

Example Dockerfile

Let's create a Dockerfile for the backend application from the provided GitHub repository: https://github.com/100xdevs-cohort-2/week-15-live-1

```
# Use the official Node.js image as the base
FROM node:18

# Set the working directory
WORKDIR /app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application code
COPY . .

# Expose the port the app runs on
EXPOSE 3000

# Start the application
CMD ["npm", "start"]
```

Here's what each instruction does:

1. `FROM node:18` : Specifies the base image as the official Node.js 18 image from Docker Hub.
2. `WORKDIR /app` : Sets the working directory to `/app` inside the container.
3. `COPY package*.json ./` : Copies the `package.json` and `package-lock.json` files to the working directory.
4. `RUN npm install` : Installs the application's dependencies.
5. `COPY . .` : Copies the rest of the application code to the working directory.
6. `EXPOSE 3000` : Informs Docker that the container listens on port 3000 at runtime.
7. `CMD ["npm", "start"]` : Sets the default command to run when the container starts, which is `npm start` in this case.

Building and Running the Docker Image

To build the Docker image from the Dockerfile, navigate to the directory containing the Dockerfile and run the following command:

```
docker build -t my-backend-app .
```

This command builds the Docker image and tags it with the name `my-backend-app` .

Once the image is built, you can run a container from it using the following command:

```
docker run -p 3000:3000 my-backend-app
```

This command runs a container from the `my-backend-app` image and maps the container's port `3000` to the host's port `3000` using the `-p` flag.

> By creating a Dockerfile, you can easily package your application and its dependencies into a Docker image, ensuring consistent and reproducible environments across different systems.

Passing in env variables

Docker allows you to set environment variables when running a container using the `-e` or `--env` flag. This is particularly useful when you need to provide configuration values or sensitive data (like

database connection strings) to your application without hardcoding them in your codebase.

Here's an example of how you can pass an environment variable to a Docker container:

```
docker run -p 3000:3000 -e DATABASE_URL="postgres://avnadmin:AVNS_EeDiMIdW-dNT4Ox9l1n@pg-3533
```

Let's break down this command:

- `docker run` : This command is used to create and start a new container from a Docker image.
- `p 3000:3000` : This flag maps the container's port `3000` to the host's port `3000` . This is useful when your application listens on a specific port, and you want to access it from the host machine.
- `e DATABASE_URL="postgres://..."` : This flag sets an environment variable called `DATABASE_URL` inside the container. The value provided after the `=` sign is the actual value of the environment variable. In this case, it's a PostgreSQL connection string.
- `my-image-name` : This is the name of the Docker image from which the container will be created.

In the example command, we're setting the `DATABASE_URL` environment variable inside the container with a PostgreSQL connection string. This connection string can then be accessed and used by your application running inside the container.

To access environment variables in a Node.js application, you can use the `process.env` object. For example:

```
const databaseUrl = process.env.DATABASE_URL;
// Use the databaseUrl variable to connect to the database
```

> By passing environment variables to Docker containers, you can separate configuration values from your application code, making it easier to manage and deploy your application in different environments (e.g., development, staging, production) without modifying the codebase.

More commands

1. `docker kill`

The `docker kill` command is used to stop a running Docker container. It sends a `KILL` signal to the container, which forcibly terminates the container's main process.

Example:

```
docker kill my-container
```

This command will stop the container named `my-container`.

You can also use the container ID instead of the container name:

```
docker kill ab12cd34ef56
```

This command will stop the container with the ID `ab12cd34ef56`.

2. `docker exec`

The `docker exec` command is used to execute a command inside a running Docker container. It allows you to interact with the container's environment and perform various operations.

Example: List all contents of a container folder

```
docker exec my-container ls /path/to/directory
```

This command will list the contents of the `/path/to/directory` folder inside the `my-container` container.

Example: Running an Interactive Shell

```
docker exec -it my-container /bin/bash
```

The `-it` flags stand for "interactive" and "pseudo-tty". They allow you to run an interactive shell inside the container.

This command will open an interactive Bash shell inside the `my-container` container. You can then run commands and interact with the container's environment as if you were inside the container itself.

Here's a breakdown of the command:

- `docker exec`: The command to execute a command inside a running container.
- `it`: Flags to run the command in an interactive mode with a pseudo-tty.
- `my-container`: The name or ID of the container you want to interact with.
- `/bin/bash`: The command to run inside the container. In this case, it's the Bash shell.