Docker and Docker Compose

What is Docker?

Docker is an open-source platform that enables developers to manage applications within containers. Containers are lightweight, isolated environments that package an application with its dependencies, providing consistent and reproducible deployments across different computing environments that have Docker installed. Docker allows developers to build, ship, and run applications efficiently, regardless of the underlying infrastructure.

What is Docker Compose?

Docker Compose is a tool that allows you to define and run multi-container applications. With Docker Compose, you can create a YAML file that defines the services that make up your application, and then use a single command to start all of the services. This makes it easy to manage complex applications, and to share your application configuration with others.

Docker and Docker Compose: A Brief History

Firstly, Docker's History
Docker was initially released in 2013 by Solomon Hykes and his team. It quickly gained popularity due to its innovative approach to containerization. Docker's success was driven by its ability to provide a consistent runtime environment across different systems, simplifying application deployment and reducing the "it works on my machine" problem.
Next, Docker Compose
Docker Compose was introduced in 2014 as a separate project initially called "fig", later becoming an official Docker tool. It was created to address the need for managing complex applications with multiple interconnected containers. Docker Compose emerged as a solution for orchestrating container-based applications, allowing developers to define and manage their dependencies easily.

Docker Key Terminologies

Image:

A lightweight, standalone, executable package that contains everything needed to run an application, including the code, runtime, libraries, and system tools.

Container:

A container is a runnable instance of an image on the host machine. It is isolated from other containers and the host machine, and has its own file system, network, and process space.

Dockerfile:

A Dockerfile is a text file that contains the instructions for building a Docker image.

Registry:

A centralized repository that stores Docker images, such as Docker Hub or private registries.

Docker Hub:

Docker Hub is the official registry for Docker images.

Volume:

A persistent storage mechanism that allows data to be shared between containers or between a container and the host system.

Docker Compose Key Terminologies

Service:

A definition that describes how to run a specific container, including its image, configuration, dependencies, and networking requirements.

YAML Ain't Markup Language (YAML):

A human-readable data serialization format used by Docker Compose to define the application's structure and configuration.

Volume:

 A mechanism to persist data across container restarts or between containers.

Network:

A virtual network that allows containers within the same Docker Compose project to communicate with each other.
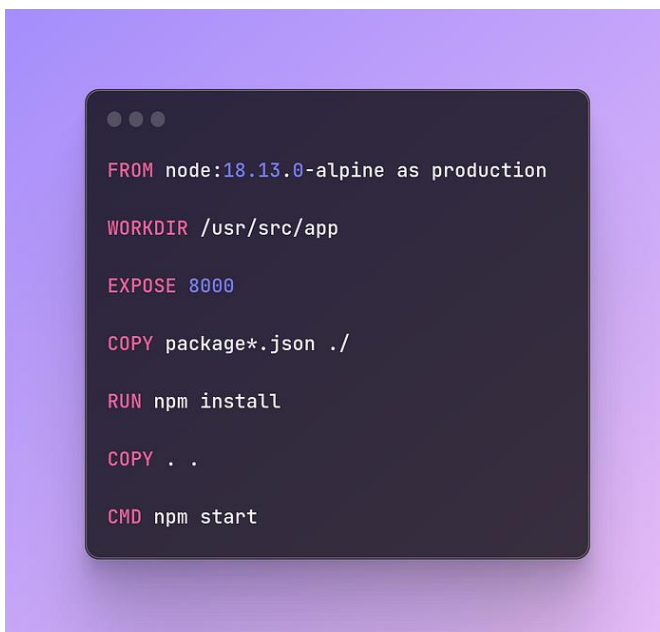
Docker Hub

Docker Hub is the official registry for Docker images. It hosts millions of images, including images for popular applications like Nginx, MySQL, and WordPress. Docker Hub simplifies the process of finding and using pre-built images, reducing the need for manual image creation and configuration. We will discuss more on working with Docker hub in my next article. Docker hub also provides a number of features for managing your images, such as:

1. Image search
2. Image versioning
3. Image tagging

Docker Compose: Local build and Using images from Docker hub

1. Docker file

A docker file as defined earlier, is a set of text commands which contains the instructions for building a docker image


```
FROM node:18.13.0-alpine as production

WORKDIR /usr/src/app

EXPOSE 8000

COPY package*.json ./

RUN npm install

COPY . .

CMD npm start
```

From the image above, we can see 7 commands defined, let's briefly run through them.

1. **From** command: This line tells Docker to use the **node:18.13.0-alpine** image as the base image for the production environment.

2. **Workdir** command: This line tells Docker to set the working directory to /usr/src/app. This is the directory where your application code will be located once a container is started with the built image.

3. **Expose** command: This line tells Docker to expose the port supplied, in this instance, the port is 8000. This is the port that your application will listen to.

4. **Copy** command: This line tells Docker to copy the files supplied, in this instance, the package.json file into the working directory. This file contains information about our Node Js application's dependencies.

5. **Run** Command: This line tells Docker to run the supplied command, in this instance, the command supplied is to run "npm install" which installs the applications dependencies.

6. The next command on the dockerfile is the **Copy** command again, this time, we are telling docker to copy all the files (excluding files supplied in .dockerignore file) from our local into the working directory i.e. all other aspects of our applications files.

7. Lastly, **Cmd** command, this line tells Docker to run the supplied command i.e. "npm start" command when the container starts. This will start your application.
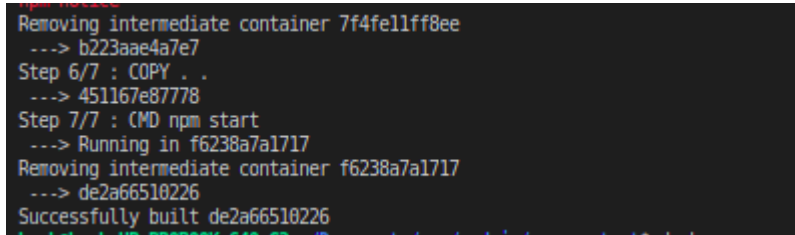
2. Docker build Image Locally

Now, let's build our Docker image. Firstly, let me show my project structure
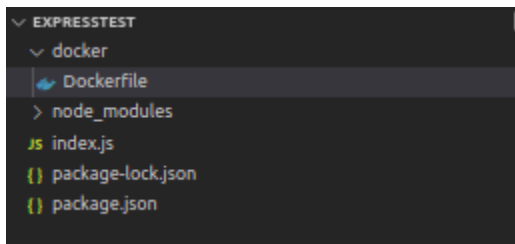
We can see the **Docker file** from above exists in a directory "Docker". Now in my terminal (on VS code), I can run this command, "**docker build. -f ./docker/Dockerfile**" which tells the Docker engine to build an image using the Docker file specified with the -f tag "/docker/Docker file", it should build a Docker image for me locally. The dot (.) specified after the "Docker build" informs Docker of the root location of my project i.e.

where my src and other applications files exists. So the command can be interpreted to be "Hey Docker, build an image for me, here is my root location (.) and the Docker file exists here (./docker/Docker file)".

Once Docker is done, a response similar to the image below is shown.

```
npm notice
Removing intermediate container 7f4fe11ff8ee
 ---> b223aae4a7e7
Step 6/7 : COPY . .
 ---> 451167e87778
Step 7/7 : CMD npm start
 ---> Running in f6238a7a1717
Removing intermediate container f6238a7a1717
 ---> de2a66510226
Successfully built de2a66510226
```
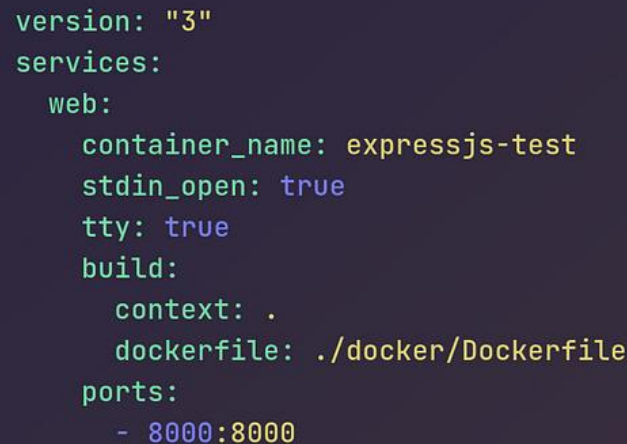
3. Docker run Container

From the image above, we can see, "Successfully built de2a66510226", with the image id being **de2a66510226.** Now we can use the image id to create a container by simply running this command "**Docker run -p 8000:8000 de2a66510226**" and now our application is running on the port specified.

Now, let me break down the last command we ran i.e. Docker run -p 8000:8000 de2a66510226

1. The **Docker run** command: This command tells Docker to run a container.

2. The **-p** flag: This flag tells Docker to expose port 8000 on the container and map it to port 8000 on the host machine. This means that you will be able to access your application at localhost:8000.

3. Lastly, we specified the image id from above when we built the Docker image, telling Docker to use that image in creating this container.

4. Docker Compose build locally from Docker file

Next, we would approach starting a container, however, this time, we would make use of the Docker compose tool instead. Personally, I use this more, as it's a lot more straightforward.

```yaml
version: "3"
services:
  web:
    container_name: expressjs-test
    stdin_open: true
    tty: true
    build:
      context: .
      dockerfile: ./docker/Dockerfile
    ports:
      - 8000:8000
```

From the image above, we can see 8 major YAML keys defined, let's briefly run through them.

Firstly, a Docker compose version is supplied, with a value of 3 and then a service titled "web" is defined next, which contains the commands to create a container

1. **container_name:** The name of the container that will be created for this service. This name must be unique within the Docker environment.

2. **stdin_open:** This is a Boolean value that indicates whether the standard input of the container should be open. This allows you to interact with the container using standard input, such as by running commands or sending data. The default value for this property is false.

3. **tty:** This is a Boolean value that indicates whether the container should have a pseudo-tty attached. If this property is set to true, then the container will have a pseudo-tty attached. This allows you to interact with the container using a terminal emulator. The default value for this property is false.

4. **build:** The build configuration for the container. This includes the context and Docker file that will be used to build the image. The **context** is the directory that contains the

files that will be used to build the image. The **Docker file** is the file that contains the instructions for building the image.

5. **ports:** A list of ports that will be exposed by the container. In this case, the container will expose port 8000 on the host machine.

Pros and Cons of Using Docker

First, the pros

Docker can help you to:

1. **Portability:** Docker allows you to package your application and its dependencies into a container, making it highly portable. Containers can be run on any machine that has Docker installed, regardless of the underlying operating system.

2. **Scalability:** Docker provides a lightweight and efficient way to scale your application. With Docker, you can easily spin up multiple containers to handle increased workload or replicate your application across multiple servers.

3. **Isolation:** Each Docker container operates in isolation, providing a secure and reproducible environment for your application. Containers have their own file systems, networking, and resource allocation, ensuring that changes or issues in one container do not affect others.

Now, the cons:

1. **Learning curve**: Docker has a learning curve, especially for those new to containerization. Understanding concepts like images, containers, and container orchestration can take some time and effort.

2. **Persistence:** By default, Docker containers are ephemeral, meaning any changes made within a container are lost once the container is stopped or removed. Persisting data and managing stateful applications require additional configuration.

3. **Image size:** Docker images can be large, especially if they include a full operating system. This can lead to increased storage requirements and longer image pull times, especially in bandwidth-constrained environments.

Pros and Cons of Using Docker Compose

As always, the pros come first,

1. **Easy multi-container orchestration:** Docker Compose simplifies the orchestration of multi-container applications by allowing you to define and manage all related services in a single configuration file. It provides a declarative approach to managing complex environments.

2. **Service dependency management:** Docker Compose allows you to define dependencies between services, ensuring that dependent services are started and stopped in the correct order. This makes it easier to handle complex application setups.

3. **Reproducible environments:** With Docker Compose, you can define the exact configuration and setup of your application environment, including networks, volumes, and environment variables. This makes it easy to recreate the same environment across different machines or deployments.

Next is the cons,

1. **Limited scalability:** While Docker Compose is excellent for managing small-to-medium-sized applications, it may not be the ideal choice for highly scalable and distributed systems. In such cases, more advanced container orchestration tools like Kubernetes, Docker Swarm are often preferred.

2. **Learning curve:** Like Docker, Docker Compose has its own learning curve, particularly for those new to container orchestration. Understanding the YAML syntax and various configuration options can take some time to master.

3. **Lack of advanced features:** Docker Compose provides basic orchestration features but may lack some advanced capabilities needed for complex production setups. For advanced features like auto-scaling, load balancing, and rolling updates, a more robust orchestration tool like Kubernetes is recommended.

Real-Life Examples

Real life examples of company out of the thousands using Docker (and probably Docker compose) would include the following:

1.Netflix

2.Airbnb

3. Circle CI

Conclusion

Docker and Docker Compose have revolutionized the way applications are deployed, managed, and scaled. By isolated containers and powerful orchestration tools, developers can create portable, scalable, and reproducible application environments. It takes time to learn and resources to consider, but the benefits of Docker and Docker Compose make them essential tools for modern software development. However, it is important to consider the specific needs and complexity of your application when deciding which tool to use.