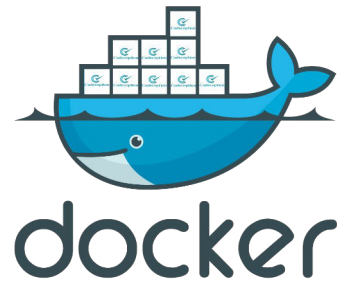
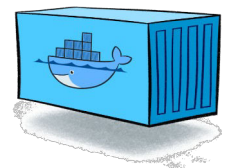


Docker Containers



What is Docker?

Docker is a tool that makes it easier to create, deploy, and run applications by using containers. Containers are a way to package software in a format that can run consistently on any machine, regardless of the environment and dependencies. With Docker, you can build a container image with all the necessary components of your application, and then run that image on any machine with Docker installed. This makes it easier to develop and test applications, and to deploy them to production. The containers isolate the application from the underlying system, so you can be sure that the application will work the same way, no matter where it is run.



What is Container?

A container is a lightweight, stand-alone, and executable package of a piece of software that includes everything needed to run it. Think of a container as a "mini-virtual machine" that runs on a host machine and provides a consistent environment for the software inside it. This allows the software to run the same way on any system that has a container runtime, without being affected by differences in the host system's environment or dependencies. Containers are a way to package, distribute, and run software in a consistent and efficient manner.



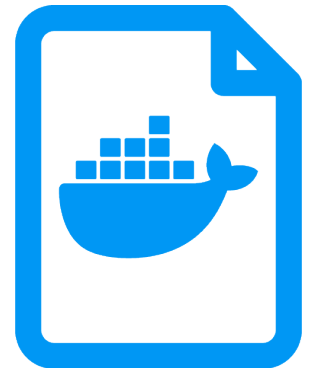
What is Docker Container?

A Docker container is a type of container that is created using the Docker platform and technology. It's a package of software that includes everything needed to run the application, and it runs in a self-contained and isolated environment on any system that has Docker installed. The container runs the same way on any machine, making it easy to develop, test, and deploy applications in a consistent and predictable manner.



What is Dockerfile?

A Dockerfile is a set of instructions for building a Docker image, which is a package of software that includes everything needed to run the application. It's like a recipe for creating a specific environment for your application to run in. The Dockerfile specifies the base image to use, sets environment variables, installs required software and dependencies, and configures the application to run. Once the Dockerfile is written, you can use it to build a Docker image that can be run on any system with Docker installed.



Sample Dockerfile for Node.js

```
● ● ●  
  
# Use the official Node.js image as the base image  
FROM node:14  
  
# Set the working directory to /app  
WORKDIR /app  
  
# Copy the package.json and package-lock.json files to the container  
COPY package*.json ./  
  
# Install the application's dependencies  
RUN npm install  
  
# Copy the remaining files to the container  
COPY . .  
  
# Specify the command to run the application  
CMD [ "npm", "start" ]
```

This Dockerfile uses the official Node.js image as the base, sets the working directory to /app, copies the package.json and package-lock.json files, installs the application's dependencies, copies the remaining files to the container, and runs the application using the command npm start.

To build a Docker image from this Dockerfile, you would run the following command in the same directory as the Dockerfile:

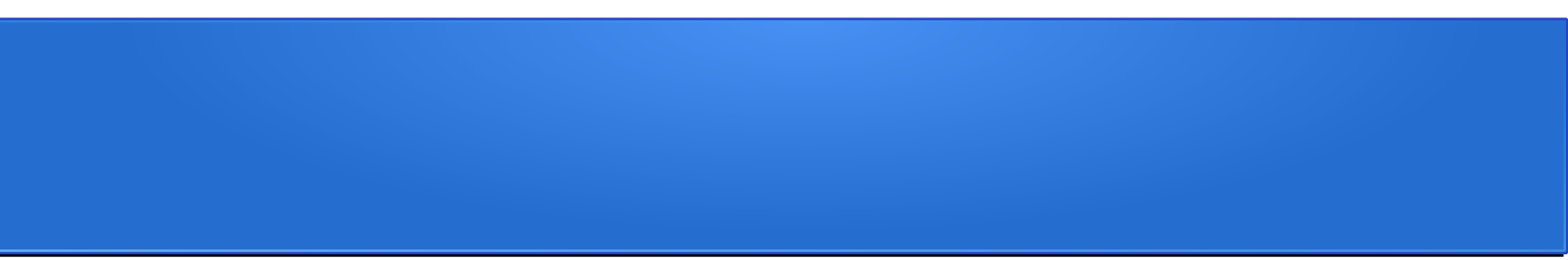
```
● ● ●  
  
docker build -t mynodeapp .
```

This will create a Docker image named mynodeapp based on the instructions in the Dockerfile.

How to create Docker Image

1. Write a Dockerfile: A Dockerfile is a script that contains instructions for building a Docker image. It specifies the base image to use, sets environment variables, installs required software and dependencies, and configures the application to run.
2. Build the Docker image: Once you have a Dockerfile, you can use the docker build command to build a Docker image. You'll run this command in the same directory as the Dockerfile, and specify the name and tag for the image. For example, `docker build -t mynodeapp:1.0 .`





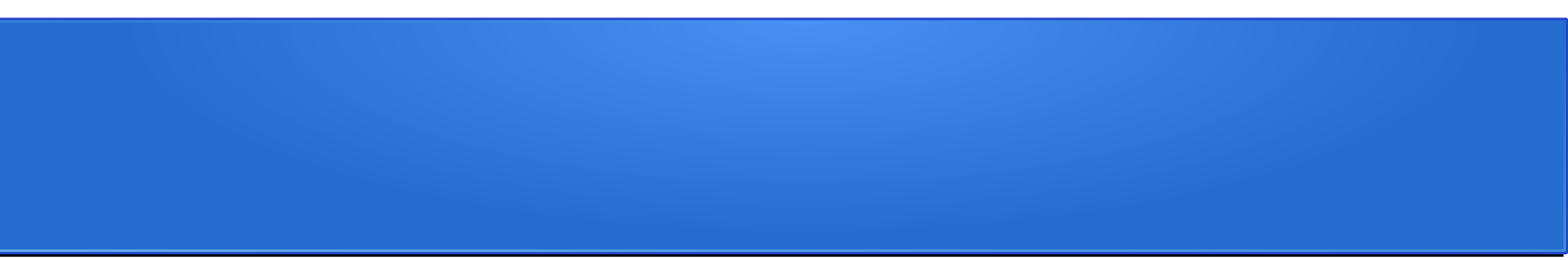
3. Test the Docker image: Once the Docker image is built, you can run it using the `docker run` command. You can test the image by running the application and making sure it works as expected.

4. Push the Docker image to a registry: Finally, you can push the Docker image to a Docker registry, such as Docker Hub, so that it can be easily distributed and used by others. To push the image, you'll need to log in to the registry and then use the `docker push` command.

These steps should get you started in creating a Docker image. The process can be more complex depending on the size and complexity of your application, but following these steps will give you a basic understanding of how to create a Docker image.

How to run the container with created Docker image?

1. Pull the Docker image from a registry: Before you can run the container, you need to have the Docker image on your local machine. You can do this by using the `docker pull` command to pull the image from a Docker registry, such as Docker Hub.
2. Run the Docker container: Once you have the Docker image, you can use the `docker run` command to run a container based on that image. For example, `docker run mynodeapp:1.0`. This will start a new container with the application inside, and you'll be able to access it just as you would if it were running on your local machine.



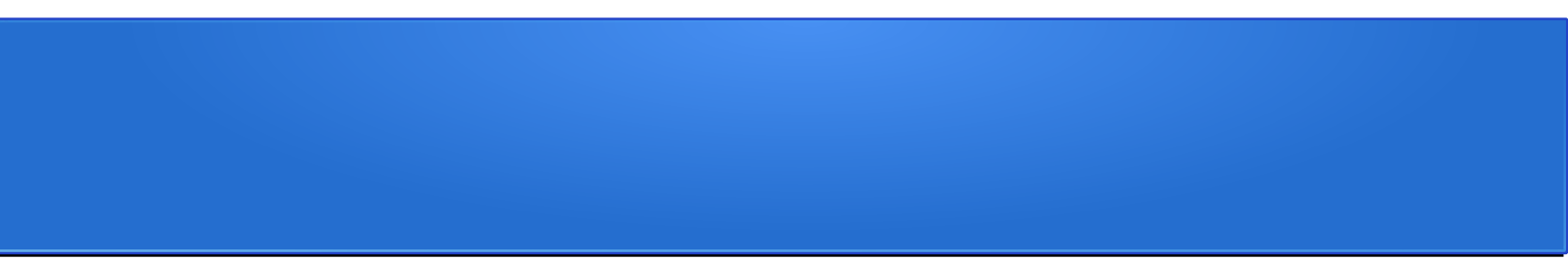
3. Access the application inside the container: The exact way to access the application inside the container will depend on the application, but you may be able to access it using a web browser, for example, by accessing `http://localhost:8080`.

4. Stop the container: When you're done using the container, you can use the `docker stop` command to stop the container. For example, `docker stop mynodeapp`.

These are the basic steps for running a container with a created Docker image. You can run multiple containers from the same image, and you can run containers in the background using the `-d` option with the `docker run` command.

Why Docker deployment is often considered to be better than local deployment?

1. Portability: Docker containers can run on any machine that has the Docker engine installed, regardless of the underlying infrastructure. This makes it easy to move your application from your local development environment to a test environment, a staging environment, or even to production.
2. Consistency: Docker containers run the same way in any environment, which means that you can be sure that your application will work the same way in production as it does in your local environment. This reduces the risk of environment-specific issues causing problems in production.
3. Isolation: Docker containers run in isolated environments, which means that each container has its own file system, network, and resources. This makes it easy to run multiple applications on the same host without them interfering with each other.
4. Scalability: Docker containers can be easily scaled to handle increased traffic or load. You can add more containers to your application as needed, and Docker will manage the distribution of traffic between the containers.



5. Versioning: Docker images can be versioned and stored in a registry, which makes it easy to roll back to a previous version if necessary. You can also easily manage multiple versions of your application, making it easy to test and deploy new features.

6.Ease of deployment: Docker makes it easy to automate the deployment of your application. You can write scripts to build and deploy your application, and you can use tools like Docker Compose and Docker Swarm to manage the deployment of your application to multiple nodes.

Overall, Docker deployment provides many benefits over local deployment, making it a popular choice for many organizations. It offers portability, consistency, isolation, scalability, versioning, and ease of deployment, which can help simplify the deployment and management of your applications.

Some additional information about Docker

1. **Portability:** Docker containers can run on any machine that has the Docker engine installed, regardless of the underlying infrastructure. This makes it easy to move your application from your local development environment to a test environment, a staging environment, or even to production.
2. **Consistency:** Docker containers run the same way in any environment, which means that you can be sure that your application will work the same way in production as it does in your local environment. This reduces the risk of environment-specific issues causing problems in production.
3. **Scalability:** Docker containers can be easily scaled to handle increased traffic or load. You can add more containers to your application as needed, and Docker will manage the distribution of traffic between the containers.
4. **Entrypoints:** The entrypoint is the command that is run when a container is started. You can specify an entrypoint in a Dockerfile, or when you run a container, to ensure that your application is started in the correct way.

5. Versioning: Docker images can be versioned and stored in a registry, which makes it easy to roll back to a previous version if necessary. You can also easily manage multiple versions of your application, making it easy to test and deploy new features.

6. Ease of deployment: Docker makes it easy to automate the deployment of your application. You can write scripts to build and deploy your application, and you can use tools like Docker Compose and Docker Swarm to manage the deployment of your application to multiple nodes.

7. Environment Variables: Environment variables are a way to store configuration data for your application. You can set environment variables in a Dockerfile, or when you run a container, to configure your application and make it easier to manage.



How to create Docker Image?

We have a simple Node application which will print Hello World on Port 3000

```
root@ip-172-31-69-137:~/my-app# ls
Dockerfile  app.js  node_modules  package-lock.json  package.json
```

```
root@ip-172-31-69-137:~/my-app# node app.js
Example app listening on port 3000!
```



Hello, World!

We have also created a sample Dockerfile

```
FROM node:16
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD node app.js
```


This command will help us to create docker image,

```
root@ip-172-31-69-137:~/my-app# docker build -t nodehelloworld1 .  
[+] Building 0.2s (10/10) FINISHED  
=> [internal] load build definition from Dockerfile  
=> => transferring dockerfile: 139B  
=> [internal] load .dockerignore  
=> => transferring context: 2B  
=> [internal] load metadata for docker.io/library/node:16  
=> [1/5] FROM docker.io/library/node:16@sha256:0dcce56017e82b86e92adcee01f0cd1382966a5199c36fbc8320004d800b35e7  
=> [internal] load build context  
=> => transferring context: 27.58kB  
=> CACHED [2/5] WORKDIR /app  
=> CACHED [3/5] COPY package*.json ./  
=> CACHED [4/5] RUN npm install  
=> CACHED [5/5] COPY . .  
=> exporting to image  
=> => exporting layers  
=> => writing image sha256:847a06553d386127a90a7d992236ab952f68727db2bd6a18019a86f8e02ced34  
=> => naming to docker.io/library/nodehelloworld1  
root@ip-172-31-69-137:~/my-app#
```

Using the -t flag with docker build will allow you to tag the image with a name and last dot means we are copying all the files in the folder into image

Once it is complete, check your images by typing this

```
root@ip-172-31-69-137:~/my-app# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nodehelloworld1	latest	847a06553d38	2 hours ago	917MB
<none>	<none>	0559483d30a1	2 hours ago	917MB
helloworld	latest	645d06a6a5	16 months ago	13.2kB



Run the following command to build the container

```
root@ip-172-31-69-137:~/my-app# docker run --name HelloWorld -p 80:3000 -d nodehelloworld128aac351340b42beffae40df34434ce1a721315f0e40c0b2a89a0e0088439ca9
```

-p: This publishes the port on the container and maps it to a port on our host

-d: This runs the container in the background.

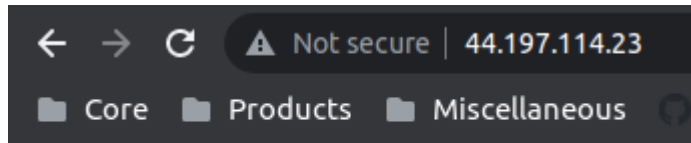
--name: This allows us to give the container a memorable name

Once your container is up and running, you can inspect a list of your running containers with

```
root@ip-172-31-69-137:~/my-app# docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
28aac351340b	nodehelloworld1	"docker-entrypoint.s..."	2 minutes ago	Up 2 minutes	0.0.0.0:80->3000/tcp, :::80->3000/tcp	HelloWorld

With your container running, you can now visit your application by navigating your browser to your server IP without the port:



Hello, World!

Downloaded from <https://www.cambridge.org/core>. University of Cambridge, on 01 Jun 2020 at 10:00:00, subject to the Cambridge Core terms of use, available at <https://www.cambridge.org/core/terms>. <https://doi.org/10.1017/S0022278X20000509>

First run `docker ps` and copy the Container ID
Then run below command

```
root@ip-172-31-69-137:~/my-app# docker ps
CONTAINER ID   IMAGE             COMMAND                  CREATED        STATUS        PORTS                               NAMES
28aac351340b   nodehelloworld1   "docker-entrypoint.s..." 7 minutes ago  Up 7 minutes  0.0.0.0:80->3000/tcp, :::80->3000/tcp  HelloWorld
root@ip-172-31-69-137:~/my-app# docker stop 28aac351340b
28aac351340b
root@ip-172-31-69-137:~/my-app#
```

And then

```
root@ip-172-31-69-137:~/my-app# docker system prune -a
WARNING! This will remove:
- all stopped containers
- all networks not used by at least one container
- all images without at least one container associated to them
- all build cache

Are you sure you want to continue? [y/N]
```

```
root@ip-172-31-69-137:~/my-app# docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
root@ip-172-31-69-137:~/my-app#						