# DevOps Certification Training

## Study Material - Docker

## Docker overview

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

## The Docker platform

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allows you to run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host. You can easily share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.

Docker provides tooling and a platform to manage the lifecycle of your containers:

- Develop your application and its supporting components using containers.
- The container becomes the unit for distributing and testing your application.
- When you're ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

## What can I use Docker for?

Fast, consistent delivery of your applications**.** Docker streamlines the development lifecycle by allowing developers to work in standardized environments using local containers which provide your applications and services. Containers are great for continuous integration and continuous delivery (CI/CD) workflows.

Consider the following example scenario:

- Your developers write code locally and share their work with their colleagues using Docker containers.
- They use Docker to push their applications into a test environment and execute automated and manual tests.
- When developers find bugs, they can fix them in the development environment and redeploy them to the test environment for testing and validation.

- When testing is complete, getting the fix to the customer is as simple as pushing the updated image to the production environment.

### Responsive deployment and scaling

Docker's container-based platform allows for highly portable workloads. Docker containers can run on a developer's local laptop, on physical or virtual machines in a data center, on cloud providers, or in a mixture of environments.

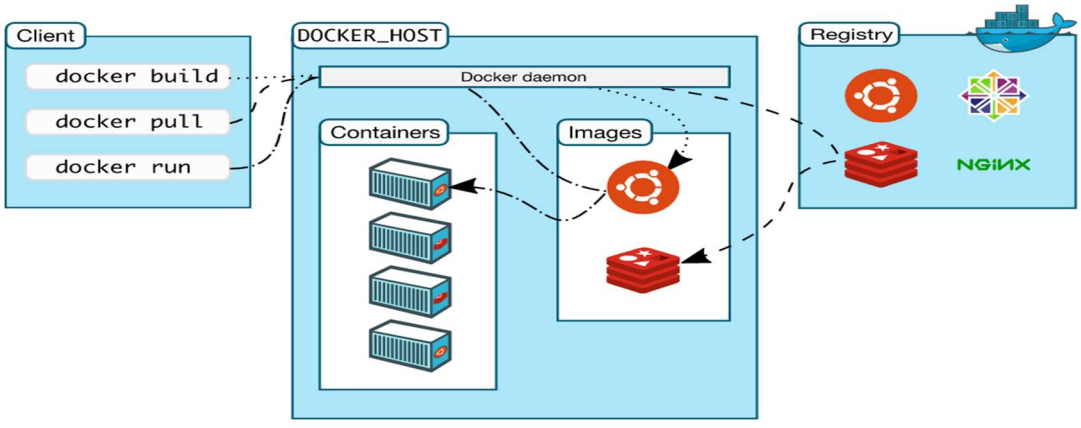
Docker's portability and lightweight nature also make it easy to dynamically manage workloads, scaling up or tearing down applications and services as business needs dictate, in near real time.

### Running more workloads on the same hardware

Docker is lightweight and fast. It provides a viable, cost-effective alternative to hypervisor-based virtual machines, so you can use more of your server capacity to achieve your business goals. Docker is perfect for high density environments and for small and medium deployments where you need to do more with fewer resources.

### Docker architecture

Docker uses a client-server architecture. The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.



### The Docker daemon

The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

### The Docker client

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

### Docker Desktop

Docker Desktop is an easy-to-install application for your Mac, Windows or Linux environment that enables you to build and share containerized applications and microservices. Docker Desktop includes the Docker daemon (dockerd), the Docker client (docker), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper. For more information, see Docker Desktop.

### Docker registries

A Docker *registry* stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

### Images

An *image* is a read-only template with instructions for creating a Docker container. Often, an image is *based on* another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.
You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a *Dockerfile* with a simple

syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

### Containers

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

Simply put, a container is a sandboxed process on your machine that is isolated from all other processes on the host machine. That isolation leverages kernel namespaces and cgroups, features that have been in Linux for a long time. Docker has worked to make these capabilities approachable and easy to use. To summarize, a container:

- is a runnable instance of an image. You can create, start, stop, move, or delete a container using the DockerAPI or CLI.
- can be run on local machines, virtual machines or deployed to the cloud.
- is portable (can be run on any OS).
- is isolated from other containers and runs its own software, binaries, and configurations.

### Containerize an application

For the rest of this guide, you will be working with a simple todo list manager that's running in Node.js. If you're not familiar with Node.js, don't worry. This guide doesn't require JavaScript experience.

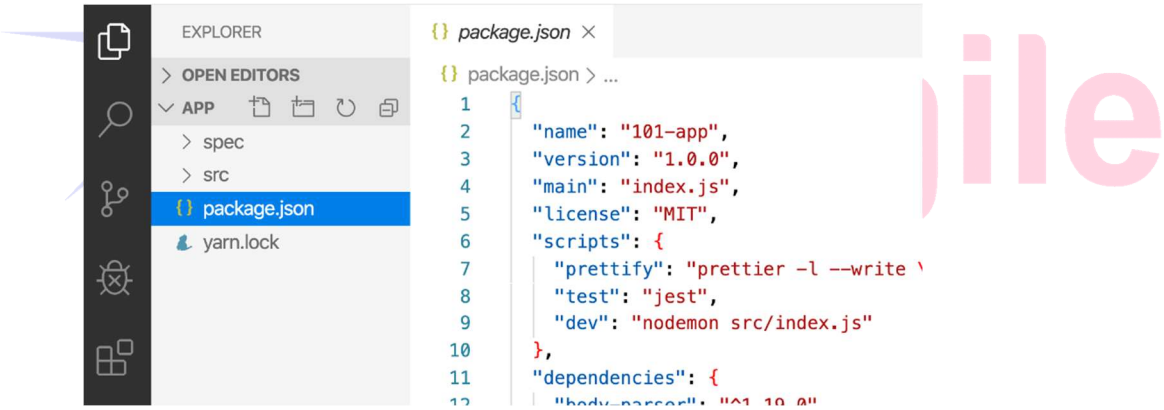
To complete this guide, you'll need the following:

- Docker running locally. Follow the instructions to download and install Docker.
- A Git client.
- An IDE or a text editor to edit files. Docker recommends using Visual Studio Code.
- A conceptual understanding of containers and images.

### Get the app

Before you can run the application, you need to get the application source code onto your machine.

1. Clone the getting-started repository using the following command:
2. `$ git clone https://github.com/docker/getting-started.git`
3. View the contents of the cloned repository. Inside the `getting-started/app` directory you should see `package.json` and two subdirectories (`src` and `spec`).




### Build the app's container image

In order to build the container image, you'll need to use a Dockerfile. A Dockerfile is simply a text-based file with no file extension. A Dockerfile contains a script of instructions that Docker uses to create a container image.

1. In the app directory, the same location as the package.json file, create a file named Dockerfile. You can use the following commands below to create a Dockerfile based on your operating system.

2. In the Windows Command Prompt, run the following commands listed below. Change directory to the app directory. Replace \path\to\app with the path to your getting-started\app directory.

```
$ cd \path\to\app
```

3. Create an empty file named Dockerfile.

```
$ type nul > Dockerfile
```

Using a text editor or code editor, add the following contents to the Dockerfile:

```
# syntax=docker/dockerfile:1

FROM node:18-alpine
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

4. Build the container image.

```
$ docker build -t getting-started
```

The docker build command uses the Dockerfile to build a new container image. You might have noticed that Docker downloaded a lot of “layers”. This is because you instructed the builder that you wanted to start from the node:18-alpine image. But, since you didn’t have that on your machine, Docker needed to download the image.

After Docker downloaded the image, the instructions from the Dockerfile copied in your application and used yarn to install your application’s dependencies. The CMD directive specifies the default command to run when starting a container from this image.

Finally, the -t flag tags your image. Think of this simply as a human-readable name for the final image. Since you named the image getting-started, you can refer to that image when you run a container.

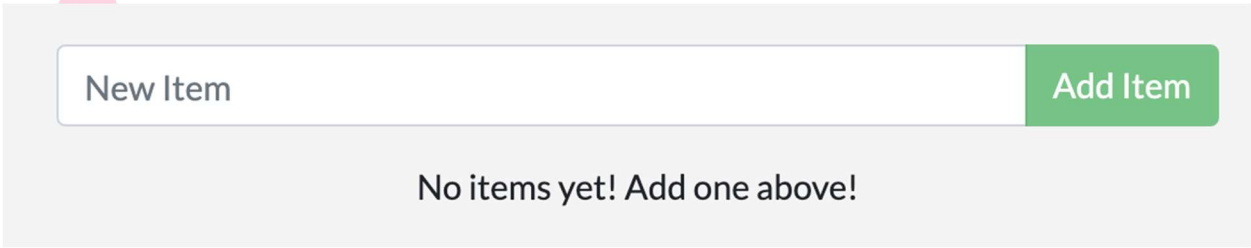
The . at the end of the docker build command tells Docker that it should look for the Dockerfile in the current directory.

**Start an app container**

Now that you have an image, you can run the application in a container. To do so, you will use the docker run command.

1. Start your container using the docker run command and specify the name of the image you just created:
2. $ docker run -dp 3000:3000 getting-started

   You use the -d flag to run the new container in "detached" mode (in the background). You also use the -p flag to create a mapping between the host's port 3000 to the container's port 3000. Without the port mapping, you wouldn't be able to access the application.
3. After a few seconds, open your web browser to http://localhost:3000. You should see your app.




4. Go ahead and add an item or two and see that it works as you expect. You can mark items as complete and remove items. Your frontend is successfully storing items in the backend.

At this point, you should have a running todo list manager with a few items, all built by you. If you take a quick look at your Docker Dashboard, you should see at least one container running that is using the getting-started image and on port 3000.

**Docker Compose :**

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

**Basic Docker Commands :**

1. Docker ps
2. Docker ps -al
3. Docker rm <container Id>
4. Docker rmi <image Id>
5. Docker images
6. Docker pull <fully qualified image name>
7. Docker push <fully qualified image name>
8. Docker start <container id>
9. Docker stop <container id>
10. Docker restart <container id>
11. Docker volume create <volume-name>
12. Docker inspect
13. Docker build -t <reponame>/<Imagename>:<tagName> -f <Dockerfile location>
14. Docker run -itd -p <systemport>:<containerport> <fully qualified image name>
15. Docker exec -it <container id> bash
16. Docker login
17. Docker logout
18. Docker push <fully qualified image name>
19. Docker-compose up -d -f <location of docker-compose.yml file>
20. Docker-compose down