In-depth Docker Tutorial

Like I mentioned earlier on in this section, this course is not a Docker course!

You can access the official Docker tutorial (which is free and great) by running the tutorial image:

```
docker run -dp 80:80 docker/getting-started
```

Then you can access http://127.0.0.1/tutorial to launch the official tutorial.

I recommend going through this (although it uses NodeJS as an example 😖), as it teaches you quite a few important commands and concepts, such as working with volumes and layers.

When I went through the official tutorial I took some notes, which you can see below. Remember these may differ from the official tutorial as the Docker team updates the tutorial regularly.

I hope the notes are helpful as a bit of a cheatsheet, but it doesn't beat going through the tutorial yourself and taking your own notes!

How to write a simple Dockerfile for a Node app

```
FROM node:12-alpine
# Adding build tools to make yarn install work on Apple silicon / arm64 machines
RUN apk add --no-cache python2 g++ make
```

```
WORKDIR /app
COPY . .
RUN npm install
CMD ["node", "src/index.js"]
```

Then build the image into a new container (the `.` below refers to the current directory, where Docker should find the `Dockerfile`). Optionally tag it:

```
docker build -t docker-image-tag .
```

How to run Docker as a daemon (background)

This prints out the container ID and runs it in the background.

```
docker run -d docker-image-tag
```

How to map ports from host machine to Docker container

This binds port 5000 of the Docker image to port 3000 of the host machine. This way you when you access `127.0.0.1:3000` with your browser, you'll access whatever the Docker image is serving in port `5000`.

Docs: https://docs.docker.com/engine/reference/commandline/run/#publish-or-expose-port--p---expose

```
docker run -p 127.0.0.1:3000:5000 docker-image-tag
```

Working with Docker volumes

In a Docker volume, the Docker container can store data in the Docker container's filesystem, and it is actually stored in the volume (which is a location in the host machine).

This is in contrast to a Bind Mount, which is another type of volume where the Docker container reads files (i.e. is provided files to read) from the host machine. The Docker container cannot modify those files when using Bind Mounts.

Feature	Named Volumes	Bind Mounts
Host Location	Docker chooses	You control
Mount Example (using `v`)	`my-volume:/usr/local/data`	`/path/to/data:/usr/local/data`
Populates new volume with container contents	Yes	No
Supports Volume Drivers	Yes	No

How to map a Named Volume from host to Docker container

```
docker run -v volume-name:/path/in/docker/image container-tag
```

For example for an app that needs port mapping and a volume:

```
docker run -dp 3000:3000 -v todo-db:/etc/todos getting-started
```

And to see *where* in the host machine the data is actually stored:

```
docker volume inspect volume-name
```

> While running in Docker Desktop, the Docker commands are actually running inside a small VM on your machine. If you wanted to look at the actual contents of the Mountpoint directory, you would need to first get inside of the VM.

How to use a Bind Mount to provide your app code to a Docker container

```
docker run -dp 3000:3000 \
    -w /app -v "$(pwd):/app" \
    node:12-alpine \
    sh -c "apk add --no-cache python2 g++ make && yarn install && yarn run dev"
```

- `-dp 3000:3000` - same as before. Run in detached (background) mode and create a port mapping
- `-w /app` - sets the container's present working directory where the command will run from
- `-v "$(pwd):/app"` - bind mount (link) the host's present `getting-started/app` directory to the container's `/app` directory. Note: Docker requires absolute paths for binding mounts, so in this example we use `pwd` for printing the absolute path of the working directory, i.e. the `app` directory, instead of typing it manually
- `node:12-alpine` - the image to use. Note that this is the base image for our app from the Dockerfile
- `sh -c "yarn install && yarn run dev"` - the command. We're starting a shell using `sh` (alpine doesn't have `bash`) and running `yarn install` to install *all* dependencies and then running `yarn run dev`. If we look in the `package.json`, we'll see that the `dev` script is starting `nodemon`.

Note that most of this is identical to the `Dockerfile` that you would create for your project. The only difference is the `-v "$(pwd):/app"` flag.

How to pass environment variables to a container

Use the `-e ENV_NAME=env_value` flag with `docker run`.

> **SECRETS IN ENVIRONMENT VARIABLES**
>
> Passing secrets like database connection strings or API keys to Docker containers can be done with environment variables, but it isn't the most secure way (the official Docker tutorial will tell you more).
>
> Instead a better option is to use your orchestration framework's secrets management system (that's a mouthful). The two major options are Kubernetes and Swarm, and each have their own secrets management system. More info on this later on!

Networking between two containers

First create a network with:

```
docker network create network-name
```

Then pass the `--network network-name` flag to `docker run`.

You can also pass `--network-alias` to `docker run` to give the container you are running a DNS name within the network.

Then create your containers and pass the network to them. For example, this starts up a MySQL image on `linux/amd64`. It also creates a volume and passes in two environment variables which the image uses for configuring MySQL:

```
docker run -d \
    --network network-name --network-alias mysql --platform linux/amd64 \
```

```
    -v todo-mysql-data:/var/lib/mysql \
    -e MYSQL_ROOT_PASSWORD_FILE=/run/secrets/mysql_root_password \
    -e MYSQL_DATABASE=todos \
    mysql:5.7
```

Then you could run another container on the same network:

```
docker run -dp 3000:3000 \
  -w /app -v "$(pwd):/app" \
  --network network-name \
  -e MYSQL_HOST=mysql \
  -e MYSQL_USER=root \
  -e MYSQL_PASSWORD_FILE=/run/secrets/mysql_password \
  -e MYSQL_DB=todos \
  node:12-alpine \
  sh -c "npm install && npm run dev"
```

CAUTION

In these I'm not passing the MySQL password directly as an environment variable. Instead, I'm passing the path to a file that contains the password.

That file is created by your Docker orchestration framework's secrets management system. That's a mouthful to say: you define the secret in your orchestration framework, and the framework creates a file which contains the password. That way, the password isn't stored in the environment which is a bit unsafe.

Your application (or, in this case, MySQL), would have to read the contents of the image to find the password.

More info on this when we learn about deploying our app in production!

How to run multiple containers using Docker Compose

1. Create a `docker-compose.yml` file in the root of your project.
2. Turn each of the `docker run` commands into a `service` in the `docker-compose.yml` file.
3. This is re-creating the flags passed to the `docker run` command, but in YAML format.

Example of the two `docker run`s above:

```
services:
  app:
    image: node:12-alpine
    command: sh -c "npm install && npm run dev"
    ports:
      - 3000:3000
    working_dir: /app
    volumes:
      - ./:/app
    environment:
      MYSQL_HOST: mysql
      MYSQL_USER: root
      MYSQL_PASSWORD_FILE: /run/secrets/mysql_password
      MYSQL_DB: todos
  mysql:
    image: mysql:5.7
    platform: linux/amd64
    volumes:
      - todo-mysql-data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD_FILE: /run/secrets/mysql_root_password
      MYSQL_DATABASE: todos
```

```
volumes:
  todo-mysql-data:
```

Then, just run `docker compose up -d` and it will start in the background!

You can see it in Docker desktop.

Tear it down and remove the containers (but not the volumes) with `docker compose down`.

Caching in Dockerfile layers

Each layer (i.e. each line of text) in a Dockerfile uses caching.

That means that if Docker doesn't detect that a layer has changed, it won't re-run it. It'll use the last value / files that were generated for the last build.

However, it also means that if one layer changes and has to be re-built, Docker will re-build all subsequent layers.

Therefore it's best to set up your Dockerfile so that you can maximise the amount of caching and reduce the chances of a cache bust.

For example, instead of this:

```
FROM node:12-alpine
WORKDIR /app
COPY . .
RUN npm install --production
CMD ["node", "src/index.js"]
```

You might do this:

```
FROM node:12-alpine
WORKDIR /app
COPY package.json package.lock ./
RUN npm install --production
COPY . .
CMD ["node", "src/index.js"]
```

That way if the `package.json` and `package.lock` files don't change, you won't re-run `npm install`.

In the first example, if *any* code files changed, `npm install` would run. Even if it was not needed because the requirements file didn't change.

Ignore certain files and folders with `.dockerignore`

Some files and folders can be safely ignored when copying over to the Docker container. For example, `node_modules` or the Python virtual environment.

Create a `.dockerignore` file in the root directory of your project (where `docker-compose.yml` lives), and add this (more examples of what to add for a Python project here):

```
node_modules
.venv
.env
*.pyc
__pycache__
```

SECRETS IN DOCKER IMAGES

> Don't include any secrets (like database connection strings or API keys) in your code. For local development you can use a `.env` file, but don't include the `.env` file in your Docker image!
>
> One of the benefits of Docker images is you can share them with others easily, but that's why you have to be very careful with what you include in them.