This document outlines the step-by-step instructions that I used to show mastery of the Docker workshop as provided on the docker website. In addition to the codes and commands executed, outcomes are shown by way of screenshots.

**Part 1: Containerize an application**

The purpose of this part is to run a simple Todo app in a docker container

To get started, I cloned the getting-started repository by running the following command:

***git clone*** [***https://github.com/docker/getting-started-app.git***](https://github.com/docker/getting-started-app.git)

A directory is created on the present working directory and this could be observed by listing content of the current directory.

Next, I needed to build the app’s image from a dockerfile. In the getting-started app directory, I created a file and pasted the following into it:

***# syntax=docker/dockerfile:1***

***FROM node:lts-alpine***

***WORKDIR /app***

***COPY . .***

***RUN yarn install --production***

***CMD ["node", "src/index.js"]***

***EXPOSE 3000***

To build the image from the Dockerfile, the following command was run in my terminal:

***docker build -t getting-started .***

Now that an image for the app was ready to use, we have to start a container (a running instance of an image). To do that we use the command below:

***docker run -d -p 127.0.0.1:3000:3000 getting-started***

At this point, a running container using the image created is available and seen when we run ***docker ps -a***

To observe the app running on the frontend, we can open our browser to [http://localhost:3000](http://localhost:3000/)

My output is as follows.

A screenshot of a computer

Description automatically generated

**Part 2: Update the application**

The purpose of this part is to make changes to the application shown in Part 1.

In the src/static/js/app.js file, update line 56 to use the new empty text.

*- <p className="text-center">No items yet! Add one above!</p>*

*+ <p className="text-center">You have no todo items yet! Add one above!</p>*

In other words, replace the first line with the second.

After making these changes, it is necessary to rebuild the image and run a container within it in order to see the reflected change in the app.

***docker build -t getting-started .***

***docker run -dp 127.0.0.1:3000:3000 getting-started***

It is important that the previous running container is stopped and removed before running the command above.

My output showing the change can be seen below:

A screenshot of a computer

Description automatically generated

Part 3: Share the Application

This part outlines steps involved in sharing the built image to a Docker hub repository and ultimately opening it up to the general public for use.

First, sign into your Docker hub account and create a repository making sure that visibility is set to **public**.

Push the image to your repository by running the command:

***docker tag getting-started YOUR-USER-NAME/getting-started***

(Remember to insert your docker hub username in the command above)

At this point it can be observed that your new image is available on docker hub.

A black screen with white text

Description automatically generated

A screenshot of a computer

Description automatically generated

**Part 4: Persist the DB**

In case you didn't notice, your todo list is empty every single time you launch the container. Why is this? In this part, you'll dive into how the container is working.

In this part, we mount the container to a created volume in order to persist the DB. Any newly run container will pick information from this volume to update the application.

Create a volume by using the command:

***docker volume create todo-db***

Ensure that there is no running container at this point and Start the todo app container, but add the --mount option to specify a volume mount. Give the volume a name, and mount it to /etc/todos in the container, which captures all files created at the path.

***docker run -dp 127.0.0.1:3000:3000 --mount type=volume,src=todo-db,target=/etc/todos getting-started***

To verify that the DB is persisted, we need to add a few to do items to the application and run an a new container to see if the added items would show on the new container.

A screenshot of my newly running container is shown below:

A screenshot of a computer

Description automatically generated

To know where Docker stores the data, we inspect the volume by running the command

***docker volume inspect todo-db***

Output:

A black screen with a black background

Description automatically generated with medium confidence

**Part 5: Use bind mounts**

A bind mount is another type of mount, which lets you share a directory from the host's filesystem into the container. When working on an application, you can use a bind mount to mount source code into the container. The container sees the changes you make to the code immediately, as soon as you save a file. This means that you can run processes in the container that watch for filesystem changes and respond to them.

In this chapter, you'll see how you can use bind mounts and a tool called [nodemon](https://npmjs.com/package/nodemon)to watch for file changes, and then restart the application automatically. There are equivalent tools in most other languages and frameworks.

In the getting-started app directory, Run the following command to start bash in an ubuntu container with a bind mount.

***docker run -it --mount type=bind,src="$(pwd)",target=/src ubuntu bash***

change directory to ‘src’ and create a new file named ‘myfile.txt’

Open the getting-started-app directory on the host and observe that the myfile.txt file is in the directory.

A screenshot of a computer

Description automatically generated

Stop the interactive container session with Ctrl + D.

A screenshot of a computer

Description automatically generated

**Development containers**

We have been able to use bind mounts to show filesystem changes. We can also use bind mounts for local development setups where changes to the app reflect immediately without the need to rebuild and run a new container.

With no running containers, Run the following command from the getting-started-app directory.

***docker run -dp 127.0.0.1:3000:3000 \***

***-w /app --mount type=bind,src="$(pwd)",target=/app \***

***node:18-alpine \***

***sh -c "yarn install && yarn run dev"***

In the src/static/js/app.js file, on line 109, change the "Add Item" button to simply say "Add":

Refresh the page in your web browser, and you should see the change reflected almost immediately because of the bind mount. Nodemon detects the change and restarts the server. It might take a few seconds for the Node server to restart. If you get an error, try refreshing after a few seconds.

Build a new image using

***docker build -t getting-started .***

**Part 6: Multi-container apps**

In this part we run the db as a separate container and have the two containers interact by placing a network between them.

Create a network by running ***docker network create todo-app***

Start a MySQL container and attach it to the network. You're also going to define a few environment variables that the database will use to initialize the database.

***docker run -d \***

***--network todo-app --network-alias mysql \***

***-v todo-mysql-data:/var/lib/mysql \***

***-e MYSQL\_ROOT\_PASSWORD=secret \***

***-e MYSQL\_DATABASE=todos \***

***mysql:8.0***

To confirm you have the database up and running, connect to the database and verify that it connects.

***docker exec -it <mysql-container-id> mysql -u root -p***

mysql> SHOW DATABASES

Exit the MySQL shell to return to the shell on your machine.

mysql> exit

To start a dev-ready container, run the following command:

***docker run -dp 127.0.0.1:3000:3000 \***

***-w /app -v "$(pwd):/app" \***

***--network todo-app \***

***-e MYSQL\_HOST=mysql \***

***-e MYSQL\_USER=root \***

***-e MYSQL\_PASSWORD=secret \***

***-e MYSQL\_DB=todos \***

***node:18-alpine \***

***sh -c "yarn install && yarn run dev"***

Open the app in your browser and add a few items to your todo list.

Connect to the mysql database and prove that the items are being written to the database. Remember, the password is secret.

***docker exec -it <mysql-container-id> mysql -p todos***

mysql> select \* from todo\_items

My output is shown below:

A screenshot of a computer

Description automatically generated

**Part 7: Use Docker Compose**

[Docker Compose](https://docs.docker.com/compose/) is a tool that helps you define and share multi-container applications. With Compose, you can create a YAML file to define the services and with a single command, you can spin everything up or tear it all down.

The big advantage of using Compose is you can define your application stack in a file, keep it at the root of your project repository (it's now version controlled), and easily enable someone else to contribute to your project. Someone would only need to clone your repository and start the app using Compose.

In the getting-started-app directory, create a file named compose.yaml

Paste the following into the compose.yaml file:

***services:***

***app:***

***image: node:18-alpine***

***command: sh -c "yarn install && yarn run dev"***

***ports:***

***- 127.0.0.1:3000:3000***

***working\_dir: /app***

***volumes:***

***- ./:/app***

***environment:***

***MYSQL\_HOST: mysql***

***MYSQL\_USER: root***

***MYSQL\_PASSWORD: secret***

***MYSQL\_DB: todos***

***mysql:***

***image: mysql:8.0***

***volumes:***

***- todo-mysql-data:/var/lib/mysql***

***environment:***

***MYSQL\_ROOT\_PASSWORD: secret***

***MYSQL\_DATABASE: todos***

***volumes:***

***todo-mysql-data:***

With no running containers, Start up the application stack using the docker compose up command. Add the -d flag to run everything in the background.

The app stack can be seen on my Docker desktop at this point.

A screenshot of a computer

Description automatically generated

***Docker compose down*** can be used to tear it all down!