Docker Training

Release 0.0.1

Mar 02, 2020

Training Overview:

1 Docker Training **3**
- 1.1 Overview . . . 3
- 1.2 The landing page . . . 3
- 1.3 Contribute . . . 3
- 1.4 Support . . . 4

2 Getting Started with Docker **5**
- 2.1 What is Docker? . . . 5
- 2.2 What is containerization? . . . 5
- 2.3 How is this different from virtual machines? . . . 5
- 2.4 What is a Dockerfile? . . . 5
- 2.5 What is a Docker Image? . . . 6
- 2.6 What is a Docker Container? . . . 6
- 2.7 What are Docker Volumes? . . . 6
- 2.8 What is Docker Compose? . . . 6

3 Why use Docker? **7**

4 Installing Docker **9**
- 4.1 Installing Docker on Linux . . . 9
- 4.2 Installing Docker on MacOS . . . 9
- 4.3 Installing Docker on Windows . . . 9

5 Container Driven Development **11**
- 5.1 Setting Up the App Dependencies . . . 11
- 5.2 Setting Up the Containerized Environment . . . 12
- 5.3 Creating a New Food Order . . . 13
- 5.4 Getting All Food Orders . . . 14
- 5.5 Updating a Food Order . . . 14
- 5.6 Deleting a Food Order . . . 15

6 Dockerizing Django DRF Real World Application **17**
- 6.1 Why use docker with the Django DRF app that has so few requirements? . . . 17
- 6.2 Steps to Dockerizing the Django DRF application: . . . 17
- 6.3 Running your dockerized application . . . 18
- 6.4 Running your dockerized application using docker-compose . . . 19

7 Docker NodeJS Training **23**
7.1 Dockerizing a Angular, Node.js, Express and MongoDB Application . . . 23
7.2 Dockerizing the React Frontend Real World Applications . . . 29

8 Training **35**

9 Indices and tables **37**

10 Robot Frameowrk Docker **39**
10.1 Why use Docker with the Robot Frameowrk . . . 39
10.2 Getting Started with Docker Robot Frameowrk . . . 39

11 Docker Shortcuts **41**
11.1 Install Docker . . . 41
11.2 Login to docker . . . 41
11.3 Checking if there's any container running . . . 42
11.4 Running a container . . . 42
11.5 Get logs for a container . . . 42
11.6 Stopping a docker container . . . 42
11.7 Starting a docker container . . . 42
11.8 Removing a container . . . 43
11.9 List docker images . . . 43
11.10 Moving Images from one host to another . . . 43
11.11 Remove a docker image . . . 43

12 About **45**

13 Contributing **47**
13.1 Questions . . . 47
13.2 Bugs . . . 47

14 References **49**

15 Useful Articles **51**

16 Training **53**

17 Indices and tables **55**

A collection of training developed and created by the Docker Kingston Group.

CHAPTER 1

Docker Training

Training is a collection of different trainings, developed and created by the Docker Community in Jamaica.

For a HTML version, please browse to our Training Website.

1.1 Overview

Different Plone Trainings:

- Getting Started with Docker.
- How to install docker.
- How to dockerize a Django application.

1.2 The landing page

The code for the landing page at https://dockertraining.readthedocs.io is in another repository: https://github.com/JamaicanDevelopers/docker.training.

1.3 Contribute

- Issue Tracker
- Source Code
- Website

1.4 Support

If you are having issues, please let us know by opening an issue in our Issue Tracker or asking a question on our Community Space.

CHAPTER 2

Getting Started with Docker

2.1 What is Docker?

Simply put, Docker is a containerization technology that enables us to cleanly abstract an environment configuration and dependencies to a file (or set of files), and run it in a protected, isolated environment on a host. Docker is similar to but more performant than, a virtual machine.

2.2 What is containerization?

Containerization involves bundling an application together with all of its related configuration files, libraries and dependencies required for it to run in an efficient and bug-free way across different computing environments.

2.3 How is this different from virtual machines?

Virtual machines (VM) uses virtualization technology, whereby a hypervisor is installed on the host which simulates another physical machine that can run another operating system on top of the host machine. Virtualization, like Docker, offers isolation and flexibility of execution of software within a particular environment. However, virtualization has a relatively high cost of resource utilization and CPU usage. Also, each VM has its own copy of an Operating System, applications and their related files, libraries and dependencies.

Docker uses less space and memory than running software within different VMs since it does not require a separate copy of an Operating System to run on the host machine. Docker uses the kernel of the host machine to create, build and run applications inside of a container.

2.4 What is a Dockerfile?

A Dockerfile is a file that contains a set of instructions that describe an environment configuration. For example, a React web app requires a NodeJS environment that should be accessible to anyone on the internet, the Dockerfile

would look something like the following:

```
FROM node:latest
RUN apt-get install -y vim
EXPOSE 80
```

2.5 What is a Docker Image?

A Docker image is a pre-built Dockerfile. It's ready to run and can be run on any host that has Docker installed. The concept of images is where Docker gets its portability. In our previous example, the latest version of the node Docker image is being used in our Dockerfile.

2.6 What is a Docker Container?

A Docker container is an instance of a Docker image. Containers can be started, running, restarted, and stopped. We're able to create as many containers from a single image as we need. This concept helps to scale up a service easier.

The capabilities of building, managing and sharing container images have helped make Docker the de-facto standard for deployment of scalable applications in the cloud. It is supported by many cloud providers and CI frameworks.

2.7 What are Docker Volumes?

Volumes are stored in a part of the host filesystem which is managed by Docker.

2.7.1 Uses of Volumes:

- decouple the container from the storage so that even when the container is not running or is deleted, files and folders that are in the volume is accessible via the host filesystem.
- Sharing data between containers

2.8 What is Docker Compose?

Docker Compose is a Docker tool for defining, connecting, managing and running multi-container Docker application. It mainly used to build microservices, clusters and layers.

CHAPTER 3

Why use Docker?

Docker enables you to rapidly deploy server environments in“containers.”You might question why to use Docker rather than VMware or Oracle’s VirtualBox?

On the Linux kernel, Docker utilizes the virtualization technology, but it does not create virtual machines. However, depending on your version of Windows and MacOS, you’ll have to use Docker with a virtual machine.

CHAPTER 4

Installing Docker

4.1 Installing Docker on Linux

Visit the Docker official setup guide for Linux at https://docs.docker.com/install/linux/docker-ce/ubuntu/

4.2 Installing Docker on MacOS

Visit the Docker official setup guide for MacOS at https://docs.docker.com/docker-for-mac/install/

4.3 Installing Docker on Windows

Visit the Docker official setup guide for Windows at https://docs.docker.com/docker-for-windows/install/

CHAPTER 5

Container Driven Development

Container driven dedopment is a means by which application is developed, tested and executed inside a containerized environment from the very beginning of the application development cycle.

In this training, we will focus on developing, running and testing a simple food ordering API built using node inside of a containerized environment. You'll also understand the HTTP methods (get, post, patch and delete) involved in creating a standard API.

To follow this article, a basic understanding of Docker, Docker Compose and JavaScript is required. Please ensure that you have Docker and Docker Compose installed before you begin. We'll be using NodeJS and the Express library to build out our application.

5.1 Setting Up the App Dependencies

To get started, we need to create a new folder and create a skeleton package.json file. The package.json file is what bootstraps our Node app and manages the dependencies of our project.

```
{
  "name": "order-web-api",
  "version": "1.0.0",
  "description": "Node.js on Docker",
  "author": "Oshane Bailey <b4.oshany@gmail.com>",
  "main": "app.js",
  "scripts": {
    "start": "nodemon app.js"
  },
  "dependencies": {
    "body-parser": "^1.19.0",
    "express": "^4.17.1",
    "nodemon": "^1.19.4"
  }
}
```

Next, we'll create a basic NodeJs server configuration. Let's create an app.js file and add the following code as its content:

```
const express = require("express");

const app = express();
const bodyparser = require("body-parser");

const port = process.env.PORT || 3200;

// middleware

app.use(bodyparser.json());
app.use(bodyparser.urlencoded({ extended: false }));

app.listen(port, () => {
  console.log(`running at port ${port}`);
});
```

We're done setting up the app.

5.2 Setting Up the Containerized Environment

Create the docker file that uses the node image and set up the application inside the container.

```
FROM node:10

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
# A wildcard is used to ensure both package.json AND package-lock.json are copied
# where available (npm@5+)
COPY package*.json ./

RUN npm install
# If you are building your code for production
# RUN npm ci --only=production

# Bundle app source
COPY . .

EXPOSE 8080
CMD [ "node", "start" ]
```

Next, create the docker-compose file that will manage our application and additional microservices.

```
version: '3.3'

services:
  web:
    build: .
    image: node:10
    ports:
      - '8080:8080'
    # restart: always
```

(continues on next page)

(continued from previous page)

```
    volumes:
      - ./:/usr/src/app
    networks:
      - restweb
networks:
  restweb:
```

However, if you run the docker-compose up -d command, the container will not run. If you run docker-compose logs, you'll notice that it can't find the express npm package. Since mount the current directory into the container at /usr/src/app, it overrides the directory and removes the node_modules folder; therefore, the express package no longer exists. To counter this, we will create an entry point file that runs the npm install command, which creates the node_modules folder and other related files. Lets called file entrypoint.sh.

```
cd /usr/src/app
npm install
npm start
```

Afterwards, edit the Dockerfile to use the entrypoint.sh as the to run upon running a container.

```
FROM node:10

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
# A wildcard is used to ensure both package.json AND package-lock.json are copied
# where available (npm@5+)
COPY package*.json ./

RUN npm install
# If you are building your code for production
# RUN npm ci --only=production

# Bundle app source
COPY . .
RUN mv entrypoint.sh /appstart.sh
RUN chmod 744 /appstart.sh

EXPOSE 8080
CMD [ "/appstart.sh" ]
```

5.3 Creating a New Food Order

Creating a new food order can be likened to sending a post request to an API, the http post method allows you to send data from the client to the API.

First we need a variable to hold all the orders. Normally, this would be a database (SQL, MongoDb), But we are focusing on the developing the API application inside of the container so we'll skip the database layer.

Let's create a variable to hold all the orders in our app.js file, like so:

```
const orders = [];
```

This will hold all food orders that come into our API from the client. To create a new food order, we'll collect the following:

- Food Name
- Customer Name
- Food Quantity

To create a new order, input to the following code just before the app.listen, like so:

```
app.post("/new_order", (req, res) => {
  const order = req.body;

  if (order.food_name && order.customer_name && order.food_qty) {
    orders.push({
      ...order,

      id: `${orders.length + 1}`,

      date: Date.now().toString()
    });

    res.status(200).json({
      message: "Order created successfully"
    });
  } else {
    res.status(401).json({
      message: "Invalid Order creation"
    });
  }
});
```

We created a new route, /new_order, with the post method to accept the incoming food order data, and we check if any of the required data needed to created a new order is valid, then we push a new order object to the the orders array we created earlier and we add an id and date key to the order. To test restart the server.

Now, lets test the creation endpoint with the command below:

```
curl -X "POST" -H "Content-Type: application/json" -d '{"food_name": "chicken",
↪"customer_name": "oshane", "food_qty": "2"}' "localhost:8080/new_order"
```

5.4 Getting All Food Orders

To retrieve all the food orders, simply input the code below, like so:

```
app.get("/get_orders", (req, res) => {
  res.status(200).send(orders);
});
```

This creates a get request on the /get_orders route and sends the orders as an array to the client.

5.5 Updating a Food Order

To update a food order, we use the patch method, like so:

```
app.patch("/order/:id", (req, res) => {
  const order_id = req.params.id;

  const order_update = req.body;

  for (let order of orders) {
    if (order.id == order_id) {
      if (order_update.food_name != null || undefined)
        order.food_name = order_update.food_name;

      if (order_update.food_qty != null || undefined)
        order.food_qty = order_update.food_qty;

      if (order_update.customer_name != null || undefined)
        order.customer_name = order_update.customer_name;

      return res
        .status(200)
        .json({ message: "Updated Succesfully", data: order });
    }
  }

  res.status(404).json({ message: "Invalid Order Id" });
});
```

This accepts the order id and the updates related to the order id and updates it by looping through the orders array to get the id that matches with the one in the orders array and updates it. If it doesn't find an order with the id passed in to the route it returns a 404 http code and a message of Invalid Order Id.

5.6 Deleting a Food Order

To delete a food order we create a route that accepts the id of the particular order that needs to be deleted, like so:

```
app.delete("/order/:id", (req, res) => {
  const order_id = req.params.id;

  for (let order of orders) {
    if (order.id == order_id) {
      orders.splice(orders.indexOf(order), 1);

      return res.status(200).json({
        message: "Deleted Successfully"
      });
    }
  }

  res.status(404).json({ message: "Invalid Order Id" });
});
```

This accepts the order id and finds the order in the orders array by looping current and getting the corresponding order with the id provided. Using the splice method we can remove the order from the orders array.

With the tools and methods covered in this tutorial, you should now be able to create simple REST APIs in Node.js using Express in a containerized environment. This is only an example of what can be done and you can extend this by connecting to a database, to make the data persistent.

CHAPTER 6

Dockerizing Django DRF Real World Application

The Django DRF app is an example of a Production-Ready Django JSON API, which contains real-world examples (CRUD, auth, advanced patterns, etc) that adheres to the RealWorld API spec.‘

See the project code at: https://github.com/DockerJamaica/Docker-Real-World-Examples.git

Normally, the requirements for running the Django DRF application are: * Python 3.7 * Python Virtualenv or PyEnv * SQLite

6.1 Why use docker with the Django DRF app that has so few requirements?

- Isolating dependencies and source code:
- Remove the need of having multiple version of python installed on the host machine.
- Remove version conflicts of python packages

6.2 Steps to Dockerizing the Django DRF application:

6.2.1 Create a Docker file

In the root folder of your project, create a file named *Dockerfile*. Next, add the following code to your docker file.

```
# The first instruction is what image we want to base our container on
# We Use an official Python runtime as a parent image
FROM python:3.5.7

# The enviroment variable ensures that the python output is set straight
# to the terminal with out buffering it first
ENV PYTHONUNBUFFERED 1
```

(continues on next page)

(continued from previous page)

```
# Get the Real World example app
RUN git clone https://github.com/gothinkster/django-realworld-example-app.git /drf_src

# Set the working directory to /drf
# NOTE: all the directives that follow in the Dockerfile will be executed in
# that directory.
WORKDIR /drf_src

RUN ls .

# Install any needed packages specified in requirements.txt
RUN pip install -r requirements.txt

VOLUME /drf_src

EXPOSE 8080

CMD python manage.py makemigrations && python manage.py migrate && python manage.py␣
↪runserver 0.0.0.0:8000
# CMD ["%%CMD%%"]
```

The first directive is based on the requirements for running the Django DRF application, we will need to use a python 3.5 environment. As a result, we will use a Python 3.5 image. For other Python images, see https://hub.docker.com/_/python

The next directive *ENV PYTHONUNBUFFERED 1* is an environment variable, which instructs Docker not to buffer the output from Python in the standard output buffer, but simply send it straight to the terminal.

Afterwards, the RUN command executes the git clone command on Real-World Django repository and set the folder name to django_drf. The django_drf folder is then set as the working directory and all the directives that follow in the Dockerfile will be executed in that directory.

Lastly, the requirements for the application is installed via pip.

6.2.2 Build our custom docker image

With that, we have our Dockerfile for the container image. You can now build an image based on your application requirements by running the following command:

```
docker build -t django_drf .
```

If you run the following command you will see the Python:3.5.7 image along with your image, *django_drf*:

```
docker images
```

6.3 Running your dockerized application

Execute the following command to create a container for the django_drf image, which your application will run in:

```
docker run -d -p 8080:8000  -v src:/drf_src --name django_drf_app django_drf
```

If *docker run* is successful, open your browser and access your API or application. You should be able to access it at *127.0.0.1:8080/admin*. If you can see the login page, then your container is up and running.

However, there aren't any users for our Django DRF application. In order to create admin user, we need to access our container to run the *createuser* command. The following command is used access your container bash interface:

```
docker exec -it django_drf_app /bin/bash
```

If you got an error stating *Error response from daemon: Container XXX is not running*, then run *docker start django_drf_app*, then run the *docerk exec* command above again.

If your current path changes to */drf_src*, then you have successfully access your Django DRF container. Run the *ls -al .* command to list the files in your project folder. Also, if you run *python –version*, you'll notice that it is Python 3.5.7.

To add an admin user, you need to run the following code in your container:

```
python manage.py createsuperuser
```

Once you set up your user account, you can log in and check out the application.

Even though Docker makes like easier and solves countless problems, like any other Tech, it has introduced few issues of its own. For instance, managing the process of multiple containers. However, managing Docker containers, processes and resources can easily manage by the ***Docker Compose*** tool. The *docker-compose* cli can be used to manage a multi-container for one or more applications. Also, It simplifies many of Docker options you would enter on the docker run cli. Docker compose makes it easier for Docker images and workflow to be reused.

6.4 Running your dockerized application using docker-compose

Unlike our Django DRF example that uses SQLite, a typical API deployment uses a full RDMS like PostgreSQL or MySQL, it is best to use more than one containers for production. For instance,you will need a separate container for the web server and a separate container for the database server. Docker compose will assist you in specifying how you want the containers to be built and connected, using a single command. In our case, we can use Docker compose to tell how our monolithic app should be built.

For us to use Docker Compose, we need to create *docker-compose.yml* file in the same location as the Dockerfile. Afterwards, add the following lines to your *docker-compose.yml* file:

```
# Specifies which syntax version of Docker compose
version: '3'

# Build a multiservice arhitecture.
services:
  # Create a service called web
  web:
    # Build an image from the files in the project root directory (Dockerfile)
    build: .
    # Assigns a name for the container. If no name is specified,
    # Docker will assign the container a random name
    container_name: drf_app
    # Mount the container `/drf` folder to the a `src` folder in the location
    # of the Dockerfile on the host machine.
    volumes:
      - ./src:/drf
    # Map port 8000 to port 9090 so that we can access the application on
    # our host machine by visiting 127.0.0.1:9090
    ports:
      - "9090:8000"
```

The first line in the *docker-compose.yml* file specifies which syntax version of Docker compose you want to use.

Next, we define a service called *web* . The *build* directive tells Docker compose to build an image from the files in the project root directory. The *command* directive is the default command that will be executed when Docker runs the container image.

The *container_name* directive assigns a name for the container. If no name is specified, Docker will assign the container a random name. The *volume* directive mounts the *src* folder in the project root directory to the container *drf* folder. In essence what this does is; It makes sure that when I edit any file in the project folder, the container folder is updated immediately.

Lastly, we expose the port we want to access the container on using the ports directive. Please note, the format of the ports directive is *<host-port>:<container-port>*.

With that done, you can now build and run the container with the command:

```
docker-compose up -d
```

If your build is successful, open your browser and access your API or application. You should be able to access it at *127.0.0.1:8080/admin*, *0.0.0.0:8080/admin* or *localhost:8080/admin*. If you can see the login page, then your container is up and running.

Please note, there's no root URL of the Django DRF application. Therefore, if you go to 127.0.0.1:8080, you will get a Page Not Found error. Only the Django admin panel along with API controls are defined in the application. See the url.py file to see the list of endpoints or paths that is accessible for the application.

6.4.1 Glossary

SQLite, a light-weight filesystem RDMS, which doesn't have any system requirements and slower than featured rich RDMS like PostgreSQL and MySQL.

Sources:

- *Building a Production Ready Django JSON API by Derek Howard <https://thinkster.io/tutorials/django-json-api>*

Known Issues and Fixes

The code for the landing page at https://dockertraining.readthedocs.io is in another repository: https://github.com/JamaicanDevelopers/docker.training.

If you are having issues, please let us know by opening an issue in our Issue Tracker or asking a question on our Community Space.

Docker Toolbox issues - Windows 10 Home

Docker Toolbox runs Docker engine on top of boot2docker VM image running in Virtualbox hypervisor. We can run Linux containers on top of the Docker engine. We can run Docker Toolbox on any Windows variants.

localhost:8080 is not working

The localhost address does not work when accessing the Django app from browser when using Docker Toolbox on Windows.

Solution: Unlike Docker Desktop, Dooker Toolbox uses Virtual Machine to run its Docker engine. As a result, the Docker Toolbox VM maybe behind a NAT address. Please check your Virtual Machine for Docker and use the IP

address for the VM. In my case, 192.168.99.100 worked for me as stated in an issue found [here](https://forums.docker.com/t/cant-connect-to-container-on-localhost-with-port-mapping/52716/13),

Credits to Jhamali for this fix.

Docker Desktop issues - Windows 10 Pro

Docker for Windows requires that you're running Windows Pro, Enterprise, or Education edition. Docker for Windows does not requires a Virtual Machine to run Docker containers. All containers are executed on the host kernel.

docker exec -it django_drf_app /bin/bash" on Win 10 Pro not working

Executing the following command *docker exec -it django_drf_app /bin/bash* results in an error using Docker Desktop on Windows 10 Pro.

Solution: Run the following command instead: *winpty docker exec -it django_drf_app bash*

Credits to Don-1

CHAPTER 7

Docker NodeJS Training

A collection of training developed and created by the Docker Kingston Group and Jamaican Developers Group.

7.1 Dockerizing a Angular, Node.js, Express and MongoDB Application

In this training, we will focus on dockerizing a node-based Angular + Express + application that is using MongoDB as it's database backend.

The application is a Real Estate Listing website. The codebase can be found at https://github.com/sjfortin/real-estate-listings.git.

Normally, the requirements for setting up and running this application are:

- Node
- NPM
- MongoDB

This requires manual installation of Node, NPM and MongoDB on each machine, which may be tedious and troublesome especially if there are some system requirements or another version of Node or MongoDB are installed.

7.1.1 Why use docker with the Angular + Express + MongoDB app that has so few requirements?

- Reduce cost of installing Node, NPM, MongoDB and setting up the application working environment on each machine.
- Isolating dependencies and source code:
- Remove the need of having multiple version of Node installed on the host machine.
- Remove version conflicts of Node packages

- Makes cloud migration easy. Docker runs on all the major cloud providers and operating systems, so apps containerized with Docker are portable across data centres and clouds.
- It works on everyone's machine. Docker eliminates the "but it worked on my laptop" problem.

7.1.2 Steps to Dockerizing the Real Estate Application:

Clone the codebase

```
git clone https://github.com/sjfortin/real-estate-listings.git
```

Create a Dockerfile

In the root folder of the project, create a file named `Dockerfile`. Next, add the following code to your docker file. **The ''Dockerfile'' will be used to generate the Docker image for the application.**

```
# The first instruction is what image we want to base our container on
# We Use an official Node version 10 image as a parent image
FROM node:8.10

# Create an environment variable for MongoDB URI
ENV MONGODB_URI='mongodb://localhost:27017/realestate'

# Set working directory for the project to /usr/src/app
# NOTE: all the directives that follow in the Dockerfile will be executed
# in working directory.
WORKDIR /usr/src/app

# Copy the contents of the project folder into the working directory of
# docker image
COPY . /usr/src/app/

# Install any needed packages specified in package.json
RUN npm install

EXPOSE 5000

# Periodically check if the application is running. If not, shutdown the
# container.
HEALTHCHECK --interval=2m --timeout=5s --start-period=2m \
  CMD nc -z -w5 127.0.0.1 5080 || exit 1

# Wait 5 seconds for the MongoDB connection
CMD echo "Warming up" && sleep 5 && npm start
```

> Note: The reason why we choose to use Node version 8.10 image is because some of the dependencies for the Angular app are depreciated in newer version of node. see https://hub.docker.com/_/node

Create a .dockerignore

In the root folder of the project, create a file named `.dockerignore`. Add the following file and folder names to the `.dockerignore` file

```
.vscode/
.git/
dump/
node_modules/
```

Any file or folder found in the `.dockerignore` file will not be added to the Docker image when the `COPY` directive is executed in the Dockerfile.

Building Angular Docker Image

Run the `docker build` command in the current directory with `-t` to name the Docker image as seen in the command below:

```
docker build -t realestate-angular ./
```

Setting up the Database and Other Requirements

Before we can run the application, we need to setup the MongoDB database and import the dummy data. Instead of manually installing MongoDB on our machines, we will be using the MongoDB Official Docker Image. However, there are two ways in which we can spin up a new instance of MongoDB. We can use the `docker run` command or `docker-compose`. In this tutorial, we will be using the `docker-compose` method.

Setting up MongoDB Database

In the root folder of the project, create a file named `docker-compose.yml`. In the `docker-compose.yml` file, we specify the version of `docker-compose` as `version 3` and create a database service for MongoDB.

In our database service, we will set a default username, password and database name for our MongoDB backend. In addition, we will expose the port for our database service for internal usage.

In the `server/data` folder, there are two JavaScript files that are used to populate the Mongo database. In addition, there two bson files located in the `dump/realestate` folder, which could be used to populate the database . However, we will not be using any of the sample data, instead, we are going to setup a new database in mongodb.

Copy the following code to your `docker-compose.yml` file.

```
version: '3'
services:
  database:
    image: mongo
    restart: always
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: example
      # Create a new database. Please note, the
      # /docker-entrypoint-initdb.d/init.js has to be executed
      # in order for the database to be created
      MONGO_INITDB_DATABASE: realestate
    volumes:
      # Add the db-init.js file to the Mongo DB container
      - ./db-init.js:/docker-entrypoint-initdb.d/init.js:ro
    ports:
      - '27017-27019:27017-27019'
```

> Note: The MongoDB Official Docker Image has a list of environmental variables that are used to configure MongoDB.

Next, create the `db-init.js` file in the root of the project as seen in the docker-compose file. Afterwards, add the following code to the file:

```
db.createUser({
  user: "user",
  pwd: "secretPassword",
  roles: [ { role: "dbOwner", db: "realestate" } ]
})

db.users.insert({
  name: "user"
})
```

The code above will create a new MongoDB user with the role of database owner.

Now that the database service has been defined, execute the following command to spin the MongoDB container.

```
docker-compose up -d
```

> Note: The `docker-compose up` command creates and runs the container for each service that is defined in the `docker-compose.yml` file and the `-d` option runs the container as a daemon (background process)

Afterwards, execute the following command to check if the Mongo DB container is running.

```
docker-compose ps
```

You should see something similar to the following output.

```
             Name                           Command              State                 ␣
↪                    Ports
-------------------------------------------------------------------------------------
↪-------------------------------------------------------------
real-estate-listings_database_1   docker-entrypoint.sh mongod   Up      0.0.0.0:27017-
↪>27017/tcp, 0.0.0.0:27018->27018/tcp, 0.0.0.0:27019->27019/tcp
```

Add Mongo Express Service to Manage MongoDB

Now that our MongoDB container is running and we can access Mongo databse. We can add support for Mongo Express. Mongo Express is a lightweight web-based administrative interface deployed to manage MongoDB databases interactively.

Add the following lines to your `docker-compose.yml` file:

```
mongo-express:
  image: mongo-express
  restart: always
  ports:
    - 8081:8081
  environment:
    ME_CONFIG_MONGODB_ADMINUSERNAME: root
    ME_CONFIG_MONGODB_ADMINPASSWORD: example
    ME_CONFIG_MONGODB_SERVER: database
  depends_on:
    - database
```

Note, the `mongo-express` service is using the MongoDB user's credentials that was set the database service and the database service name.

7.1.3 Running the application in the Docker Container

At this point, we can run our dockerized application by using the `docker run` command, however, for sustanability and simplicity of our software arhitecture and dependencies, we will be using `docker-compose` to run our dockerized application.

Before we can run our dockerized application using `docker-compose`, we need to create another service in our `docker-compose.yml` file to manage our application. Add the following lines to your `docker-compose.yml file`:

```
web:
  build: .
  image: realestate-angular
  environment:
    # Use the username and password found in the db-init.js file instead
    # of the root username.
    MONGODB_URI: mongodb://user:secretPassword@database/realestate
  depends_on:
    - database
  ports:
    - 8082:5000
```

Note: The `MONGODB_URI` environmental variable uses the username (root) and password (example) in the MongoDB URI that was defined in the database service for MongoDB. Also, it uses the MongoDB service name (database) as the MongoDB host, followed by the database name (realestate).

At this point, your `docker-compose` file should look like:

```
version: '3'
services:
  database:
    image: mongo
    restart: always
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: example
      # Create a new database. Please note, the
      # /docker-entrypoint-initdb.d/init.js has to be executed
      # in order for the database to be created
      MONGO_INITDB_DATABASE: realestate
    volumes:
      # Add the db-init.js file to the Mongo DB container
      - ./db-init.js:/docker-entrypoint-initdb.d/init.js:ro
    ports:
      - '27017-27019:27017-27019'

  mongo-express:
    image: mongo-express
    restart: always
    ports:
      - 8081:8081
    environment:
      ME_CONFIG_MONGODB_ADMINUSERNAME: root
      ME_CONFIG_MONGODB_ADMINPASSWORD: example
```

(continues on next page)

(continued from previous page)

```
      ME_CONFIG_MONGODB_SERVER: database
    depends_on:
      - database

  web:
    build: .
    image: realestate-angular
    environment:
      # Use the username and password found in the db-init.js file instead
      # of the root username.
      MONGODB_URI: mongodb://user:secretPassword@database/realestate
    depends_on:
      - database
    ports:
      - 8082:5000
```

Execute the following command to run the dockerized application along with the MongoDB Service:

```
docker-compose up -d --build
```

Afterwards, execute the following command to check if the application and Mongo DB container are running.

```
docker-compose ps
```

You should see something similar to the following output:

```
             Name                           Command              State                 ␣
↪                    Ports
-------------------------------------------------------------------------------------------
↪-------------------------------------------------------------------
real-estate-listings_database_1   docker-entrypoint.sh mongod   Up      0.0.0.0:27017-
↪>27017/tcp, 0.0.0.0:27018->27018/tcp, 0.0.0.0:27019->27019/tcp
real-estate-listings_web_1        /bin/sh -c npm start          Up      3000/tcp, 0.0.
↪0.0:8080->5000/tcp
```

If you wish to see the logs and output for the application and/or MongoDB, run the following command:

```
# See logs for all services
docker-compose logs -f

# See logs for only the application service
docker-compose logs -f web

# See logs for only the MongoDB service
docker-compose logs -f database
```

Finally

You can visit http://localhost:8080 to see the application in action.

You can find the finish source code for the project on DockerJamaica Github page

For more information on Docker and Docker Compose, please visit the following links:

- Docker
- Docker Compose

For list of available Docker and Docker Compose commands:

- Docker Commands
- Docker Compose Commands

If you find a bug in the project source code or documentation, you can help us by submitting an issue or submit a Pull Request with the fix to our Github repositories.

- Real Estate App repository
- Docker Training repository.

Looking for Docker events, conferences and meetups in Jamaica or somewhere to discuss Docker related topics? Whether you're a local, new in town, or just passing through, you'll be sure to find something on our Docker Community page.

- Docker Jamaica Community Page
- Docker Jamaica Events Page
- Docker Jamaica Meetup Page
- Docker Jamaica Slack Group. Please note, after joining the Slack Group, you will need to add yourself to the Docker slack channel

7.2 Dockerizing the React Frontend Real World Applications

In this training, we will focus on dockerizing the React + Mobx and React + Redux frontend codebases.

The React front-end codebases containing real world examples (CRUD, auth, advanced patterns, etc) that adheres to the RealWorld spec and API.

Let us start with dockerizing the React + Redux RealWorld example found at: **'https://github.com/gothinkster/react-redux-realworld-example-app'_**

Normally, the requirements for setting up and running the React + Redux application is: * Node * NPM

7.2.1 Why use docker with the React + Redux app that has so few requirements?

- Isolating dependencies and source code:
- Remove the need of having multiple version of Node installed on the host machine.
- Remove version conflicts of Node packages
- Makes cloud migration easy. Docker runs on all the major cloud providers and operating systems, so apps containterized with Docker are portable across datacenters and clouds.
- It works on everyone's machine. Docker eliminates the "but it worked on my laptop" problem.

7.2.2 Steps to Dockerizing the React + Redux application:

Clone the codebase

```
git clone https://github.com/gothinkster/react-redux-realworld-example-app.git react-
↪redux
```

Create a Dockerfile

In the root folder of your project, create a file named `Dockerfile`. Next, add the following code to your docker file.

```
# The first instruction is what image we want to base our container on
# We Use an official Node version 10 image as a parent image
FROM node:10

# Create app directory for Real World React example app
# NOTE: all the directives that follow in the Dockerfile will be executed in
# that directory.
WORKDIR /usr/src/app

# Copy the package.json file into our app directory
COPY . /usr/src/app/

# Install any needed packages specified in package.json
RUN npm install

RUN ls /usr/src/app
RUN ls /usr/src/app/public

EXPOSE 3000

CMD npm start
```

> Note: The reason why we choose to use Node version 10 image is because some of the dependencies for the React + Redux app are depreciated in newer version of node. see https://hub.docker.com/_/node

Add a .dockerignore

Add the `node_modules` in the .dockerignore file

```
echo 'node_modules' > .dockerignore
```

Any file or folder found in the `.dockerignore` file will not be added to the Docker image when the `COPY` directive is executed in the Dockerfile.

Building the React + Redux Docker Image

Build and tag the Docker image:

```
docker build -t react-redux .
```

If you run the following command you will see the node:10 image along with your image, `react-redux`:

```
docker images
```

Running the React + Redux Docker in a Container

Execute the following command to create a container from the react-redux image, which your application will run in:

```
docker run -d -p 8092:4100   --name react-redux-app react-redux
```

If `docker run` is successful, open your browser and access your API or application. You should be able to access it at `127.0.0.1:8092`. If you can see the ***conduit homepage***, then your container is up and running.

Setup local backend API via Docker Compose

By default and for convenience, the application is using the live API server running at **'https://conduit.productionready.io/api'_** that is provided by the RealWorld project organisers. You can view the API spec here which contains all routes & responses for the server.

However, we want to setup a local backend server for our React + Redux app. The source code for the backend server (available for Node, Rails and Django) can be found in the main RealWorld repo, but we don't have the time to manually setup the backend server. ***This is where Docker shines.* *We can use the Docker image for the Django backend server that we built in one of our previous trainings ('https://dockertraining.readthedocs.io/en/latest/django/index.html'_)***

If you've already created the Docker image for the Django DRF backend server, GREAT!!!

If you haven't, you can use the public Django DRF image on Docker Hub

```
docker pull realworldio/django-drf
```

Docker's containerization tool helps speed up the development and deployment processes especially when working with microservices. Docker makes it much easier to link together small, independent services.

We could use `docker run -d -p 8099:8000 --name django_drf_app realworldio/django-drf` command to run the Django DRF app in a container and update the API url in the React app to match the published localhost port for the Django DRF app.

However, by using Docker Compose, it will setup the linking between the two containers and ease the pain of manually running various Docker commands.

7.2.3 Add a docker-compose.yml file

```
version: '3'
# Build a multiservice arhitecture.
services:
   # Setup local instance of the Backend Server
   backend:
      # Use the public image for the Django Backend Server
      image: realworldio/django-drf:latest
      # Set the network for the two service so that they can communicate with each␣
↪other
      networks:
         - reactdrf
      volumes:
         - drf-backend:/drf_src
      # Map port 8000 to port 8199 so that we can access the application on
      # our host machine by visiting 127.0.0.1:8199
      ports:
         - "8199:8000"
   # Create a service called web for the React + Redux app
   web:
      # Build an image from the files in the project root directory (Dockerfile)
      build: .
      depends_on:
         - backend
```

(continues on next page)

(continued from previous page)

```
      # Mount the container `/drf` folder to the a `src` folder in the location
      # of the Dockerfile on the host machine.
      volumes:
         - drf-react-react:/usr/src/app/
      restart: always
      # Map port 3000 to port 8081 so that we can access the application on
      # our host machine by visiting 127.0.0.1:8081
      ports:
         - "8081:3000"
      networks:
         - reactdrf
networks:
   reactdrf:
volumes:
   drf-backend:
   drf-react-react:
```

7.2.4 Update the API URL for the React + Redux app

In the `src/agent.js`, change `API_ROOT` to the local server's URL (i.e. http://localhost:8199/api)

7.2.5 Build and run the Django + React microservices

```
docker-compose up -d
```

Run the following command to verify that the backend and frontend services are running:

```
docker ps
```

You should see something similar to:

```
CONTAINER ID        IMAGE                              COMMAND                  CREATED␣
↪            STATUS              PORTS                    NAMES
3764de81cd3d        react-mobx_web                     "docker-entrypoint.s..."   1␣
↪minutes ago       Up 1 minutes        0.0.0.0:8081->3000/tcp   react-mobx_web_1
88a299a68f59        realworldio/django-drf:latest      "/bin/sh -c 'python ..."   1␣
↪minutes ago       Up 1 minutes        0.0.0.0:8199->8000/tcp   react-mobx_backend_1
```

If yes, then you should be able to access both site via http://localhost:8199 and http://localhost:8081

Please note: by default, the Django backend app doesn't have any users. therefore, we need to add a user to make changes to the frontend React app.

7.2.6 Adding a backend Admin User

In order to create admin user, we need to access our container to run the `createuser` command. The following command is used access your container bash interface:

```
docker exec -it django_drf_app /bin/bash
```

To add an admin user, you need to run the following code in your container then enter the promoted credentials:

```
python manage.py createsuperuser
```

Afterwards, exit the Docker container running the exit command. You should be able to access the backend setting by visiting http://localhost:8199/admin

Codebase

- https://github.com/JamaicanDevelopers/django-realworld-example-app
- https://github.com/JamaicanDevelopers/react-mobx-realworld-example-app
- https://github.com/JamaicanDevelopers/react-redux-realworld-example-app

Sources:

- *Dockerizing a React App by Michael Herman <https://mherman.org/blog/dockerizing-a-react-app/>*
- *Run a React App in a Docker Container by Peter Jausovec <https://mherman.org/blog/dockerizing-a-react-app/>*

CHAPTER 8

Training

Dockerizing a Angular, Node.js, Express and MongoDB Application This training is intended for people who are using AngularJS + MongoDB and are new to Docker or want to learn about the best practices of AngularJS + MongoDB + Docker development.

Dockerizing the React Frontend Real World Applications This training is intended for people who are new to Docker or want to learn about the best practices of React + Docker development.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

CHAPTER 10

Robot Frameowrk Docker

The Robot Framework is a generic test automation framework for acceptance testing and acceptance test-driven development. It is a keyword-driven testing framework that uses tabular test data syntax.

Normally, the requirements for running the Django DRF application are: * Python 2.7 or greater * Python Virtualenv or PyEnv

10.1 Why use Docker with the Robot Frameowrk

- Rapidly get started with automated testing without additional setup
- Allows testers to focus on test case and not the executing environment
- Isolating dependencies and source code:
- Remove the need of having multiple version of python installed on the host machine.
- Remove version conflicts of python packages
- It works on everyone's machine. Docker eliminates the "but it worked on my laptop" problem.

10.2 Getting Started with Docker Robot Frameowrk

Clone this repository

```
git clone git@github.com:ypasmk/robot-framework-docker.git
```

Pull the image.

```
docker pull ypasmk/robot-framework
```

Run the tests example tests

```
cd robot-framework-docker && ./run_tests.sh
```

10.2.1 Contents

This image contains the following to facilitate robot testing

10.2.2 Xvfb

You can use it to start a visual display and fire up a browser for UI testing.

Example (suites/virtual_display.robot):

```
Start Virtual Display    1920    1080
```

10.2.3 Selenium2Library

More details here http://robotframework.org/Selenium2Library/Selenium2Library.html

Also have a look at **suites/virtual_display.robot**

10.2.4 HttpLibrary.HTTP

More details here https://github.com/peritus/robotframework-httplibrary

Example:

```
Create Http Context     api.some-end-point.com
GET                     /some/service/that/supports/get
Verify Status           200
${response}=            Get Response Body
[return]                ${response}
```

10.2.5 robotframework-sshlibrary

More details here http://robotframework.org/SSHLibrary/latest/SSHLibrary.html

10.2.6 robotframework-excellibrary

More details here http://navinet.github.io/robotframework-excellibrary/ExcelLibrary-KeywordDocumentation.html

ENJOY

For any requests or changes please open issues or create pull requests :)

10.2.7 Sources:

- *Robot Framework With Docker in less than 10 minutes by Ipatios Asmanidis <https://medium.com/@ypasmk/robot-framework-with-docker-in-less-than-10-minutes-7b86df769c22>*
- *robot-framework-docker <https://github.com/ypasmk/robot-framework-docker/>*

CHAPTER 11

Docker Shortcuts

11.1 Install Docker

11.1.1 Installing Docker on Debian based distro

```
sudo apt-get install docker
```

Allow non-root users to run docker

```
sudo usermod -aG docker $USER
```

Installing Docker on Windows

Go to https://hub.docker.com/editions/community/docker-ce-desktop-windows, download and install it. Once Docker is installed, use Docker Interactive Window or Powershell to use the following commands.

11.2 Login to docker

To use Docker's public images, you must have a Docker ID or account, go to https://hub.docker.com/ and register your account if you don't have one as yet. Otherwise, run the following command:

```
docker login
```

11.3 Checking if there's any container running

```
docker ps
```

List all containers, event if they aren't running

```
docker ps -a
```

11.4 Running a container

Docker run creates a new container from the docker image and starts the container docker run tutum/hello-world

```
docker run --name web1 tutum/hello-world
```

11.4.1 Running a docker container on a particular port

```
# docker run --name web2 -p exposePORTt:localPort tutum/hello-world
docker run --name web2 -p 8080:80 tutum/hello-world
```

11.4.2 Running docker containers in the background using daemon

Use *-d* or *–detach* to run a container in the background.

```
docker run -d --name web3 -p 8080:80 tutum/hello-world
docker run -d --name web4 -p 8081:80 tutum/hello-world
```

11.5 Get logs for a container

```
docker logs web3
```

```
for dlog in `docker ps -a -q`; do docker logs $dlog; done
```

11.6 Stopping a docker container

```
docker stop web3
```

```
docker stop $(docker ps -a -q)
```

11.7 Starting a docker container

```
docker start web3
```

11.8 Removing a container

```
# docker rm <container-name>
docker rm web3
```

```
docker rm $(docker ps -a -q)
```

Stop and remove all containers

```
docker rm $(docker ps -a -q) & docker stop $(docker ps -a -q)
```

11.9 List docker images

```
docker images
```

11.10 Moving Images from one host to another

You will need to save the Docker image as a tar file:

```
docker save -o <path for generated tar file> <image name>
```

Then copy your image to a new system with regular file transfer tools such as cp, scp or rsync(preferred for big files). After that you will have to load the image into Docker

```
docker load -i <path to image tar file>
```

11.11 Remove a docker image

```
docker rmi wordpress
```

if you do docker images and see *<none>:<none>* images in the list, these are dangling images and needs to be pruned, else they will unnecessary allocate your disks space. Run the following comand to remove dangling images

```
docker rmi $(docker images -f "dangling=true" -q)
```

```
docker rmi $(docker images -q)
```

```
docker system prune -a
```

It will output:

```
WARNING! This will remove:
     - all stopped containers
     - all volumes not used by at least one container
     - all networks not used by at least one container
     - all images without at least one container associated to them
Are you sure you want to continue? [y/N] y
```

CHAPTER 12

About

Docker Training is a collection of different trainings, developed and created by the Docker Community in Jamaica and the Jamaican Developers Group.

Looking for Docker events, conferences and meetups in Jamaica or somewhere to discuss Docker related topics? Whether you're a local, new in town, or just passing through, you'll be sure to find something on our Docker Community page.

- Docker Jamaica Community Page
- Docker Jamaica Events Page
- Docker Jamaica Meetup Page
- Docker Jamaica Slack Group. Please note, after joining the Slack Group, you will need to add yourself to the Docker slack channel

CHAPTER 13

Contributing

::: tip Key Point :bulb: Information on how to contribute. ::

We would love for you to contribute to this guide and help make it even better than it is today!

13.1 Questions

Do not open issues for general support questions as we want to keep GitHub issues for bug reports and feature requests.

You've got much better chances of getting your question answered on our forum ("Link to Docker Jamaica") where the questions should be tagged with `docs`.

13.2 Bugs

If you find a bug in the source code or documentation, you can help us by submitting an issue to our Github Repository.

Even better, you can submit a Pull Request with a fix.

- Oshane Bailey

CHAPTER 14

References

- Quick reStructuredText
- Should You Install Docker with the Docker Toolbox or Docker for Mac / Windows?
- What are Docker <none>:<none> images?

CHAPTER 15

Useful Articles

- Cleaning up docker to reclaim disk space

CHAPTER 16

Training

Dockerizing Django DRF Real World Application This training is intended for people who are new to Docker or want to learn about the best practices of Django + Docker development.

Docker NodeJS Training This training is intended for people who want to learn about the best practices of NodeJS + Docker development.

Robot Frameowrk Docker This training is intended for people who are new to Docker or want to learn about the best practices of Robot Frameowrk + Docker development.

CHAPTER 17

Indices and tables

- genindex
- modindex
- search

Looking for Docker events, conferences and meetups in Jamaica or somewhere to discuss Docker related topics? Whether you're a local, new in town, or just passing through, you'll be sure to find something on our Docker Community page.

- Docker Jamaica Community Page
- Docker Jamaica Events Page
- Docker Jamaica Meetup Page
- Docker Jamaica Slack Group. Please note, after joining the Slack Group, you will need to add yourself to the Docker slack channel