Hands On Exercises

Setting Up Docker Containers for Lab

Lab 1 Setting Up Docker Images

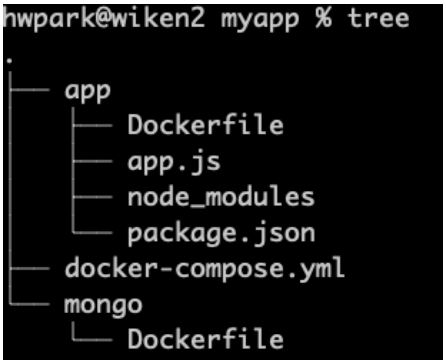
We will be using 3 docker containers to setup our Lab environment. The 3 containers are:

- MongoDB container
  - Container with MongoDB Server
  - mongo shell installed as part of our build
- Redis container
  - Container with Redis Server
- Application container – Container to run our Node.js applications
  - Container with Node.js
  - Node.js Package Manager (NPM) is installed
  - Mongoose library is installed
  - Express installed for quick web access
  - Redis client library installed

Docker compose allows us to quickly setup these 3 containers with shared network and durable storage.

1. Create working directory structure

   1.1 Use the following directory structure. We will create the files in them in the next section. You do not need to create the `node_modules` directory. That is automatically created by NPM, our node.js package manager. This directory keeps track of the NPM libraries installed.



2. Setup files for Docker Compose

   2.1 Create `docker-compose.yml` in the `myapp` directory with the following content

YAML files are sensitive to indentation. You should use an editor that can help you with YAML style content

```yaml
version: '3'  # Docker Compose file version

services:  # Defines all the services (applications) that make up your project

  app:  # The first service, your Node.js application
    build:  # Instructions for Docker to build your app's Docker image
      context: ./app  # The build context directory, Docker will look for a Dockerfile in this directory
    volumes:  # Specifies the directories that your app container will share with your host
      - ./app:/app  # Mounts the ./app directory on the host to /app in the container
      - /app/node_modules  # Creates an anonymous volume for /app/node_modules in the container
    ports:  # Exposes ports to the host machine
      - "3000:3000"  # Maps port 3000 of the container to port 3000 of the host
    environment:  # Sets environment variables in the container
      - MONGO_URL=mongodb://admin:password@mongo:27017/product-service?authSource=admin  # Connection string for MongoDB
      - REDIS_HOST=redis  # Hostname of the Redis service
    depends_on:  # Declares dependencies on other services
      - mongo  # The app service depends on the mongo service
      - redis  # The app service depends on the redis service

  mongo:  # The second service, your MongoDB database
    build: ./mongo  # Instructions for Docker to build your MongoDB's Docker image
    environment:  # Sets environment variables in the container
      - MONGO_INITDB_ROOT_USERNAME=admin  # The username for the MongoDB root account
      - MONGO_INITDB_ROOT_PASSWORD=password  # The password for the MongoDB root account
    ports:
      - "27017:27017"  # Exposes MongoDB's port to the host machine
    volumes:
      - mongo-data:/data/db  # Mounts the named volume "mongo-data" to /data/db in the container

  redis:  # The third service, your Redis database
    image: redis  # The Docker image to use for the Redis service
    ports:
      - "6379:6379"  # Exposes Redis's port to the host machine

volumes:
  mongo-data:  # Declares a named volume that persists data between container restarts
```

2.2 Create `Dockerfile` under `mongo` directory. Dockerfiles are used to instruct docker on how to build an image. The first line indicates the base image to start with. For example, in this build, the latest mongo image is used through the `From mongo` statement. The next set of instructions installs the mongodb shell.

```
FROM mongo

RUN apt-get update \
    && apt-get install -y mongodb-org-shell \
    && rm -rf /var/lib/apt/lists/*
```

2.3 Create `Dockerfile` under app directory. The container is built with Node.js version 18, which is the current LTS (Long Term Support) version. Dependency libraries are installed by inspecting package.json file. All files in the app directory is copied to the container and then npm is started.

```
# Use the official lightweight Node.js 18 image.
# https://hub.docker.com/_/node
FROM node:18

# Create and change to the app directory.
WORKDIR /app

# Copy application dependency manifests to the container image.
# A wildcard is used to ensure both package.json AND package-lock.json are copied.
# Copying this separately prevents re-running npm install on every code change.
COPY package*.json ./

# Install production dependencies.  Read from package.json for list of dependencies.
RUN npm install

# Copy local code to the container image.
COPY . .

# Start the application. This is the same as running npm start from the command line.
CMD [ "npm", "start" ]
```

2.4 Create `package.json` file under `app` directory. The ^ symbol before each version number means that npm will install the newest minor/patch version following the specified major version. In shared development environment, this may not be a good idea. It is usually better to synchronize and lock into the same version.

```
{
  "name": "myapp",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "^4.17.1",
    "mongoose": "^5.12.13",
    "redis": "^3.1.2"
  }
}
```

2.5 Create the initial app.js Node.js program. This simple program will create connections to MongoDB and Redis. It will also create an instance of express in order to provide a RESTful API access on port 3000. We will create 3 simple routes (REST endpoints). The root route at `/` will simply greet you with a welcome message. The `mongodb` route will test the connection to MongoDB by providing a list of the databases in the MongoDB server. Finally, the `redis` route simply provides a counter of number of times you

have accessed this page. It does so by setting a `counter` key:value that gets incremented on each access.

Setup the required modules.

- express - The Express library is a popular web application framework for Node.js. It provides a robust set of features and tools for building web applications and APIs. Express is designed to be minimalistic, flexible, and unopinionated, allowing developers to have full control over the architecture and design of their applications.
- mongoose - The Mongoose library is an Object-Document Mapping (ODM) library for Node.js and MongoDB. It provides a convenient way to interact with MongoDB by providing a schema-based solution for modeling and mapping JavaScript objects to MongoDB documents.
- redis -The Redis library for Node.js is a client library that allows you to interact with Redis. The library allow access to various data structures such as strings, hashes, lists, sets, sorted sets, and more, which can be manipulated and stored in memory.

```
// Import necessary modules
const express = require('express');
const mongoose = require('mongoose');
const redis = require('redis');

// Instantiate the Express application
const app = express();
```

Instantiate a new redis client object. In node.js, it is convention to use constants when saving instantiated classes, to ensure that there is no conflict or possibility of overwriting. We will use the `on` method to check if we successfully connect or there was an error. We will send to the console log, an appropriate message.

```
// Create and configure the Redis client
const client = redis.createClient({
    host: process.env.REDIS_HOST
});

// Log successful Redis connection
client.on('connect', () => {
    console.log('Connected to Redis');
});

// Log Redis errors
client.on('error', err => {
    console.log('Redis error: ', err);
});
```

Use the mongoose library object to connect to MongoDB. Most MongoDB clients use connection pooling. This allow clients to take advantage of connection pooling. When we create the new connection, the connection pool will be polled and reused when available.

```
// Connect to MongoDB
mongoose.connect(process.env.MONGO_URL, { useNewUrlParser: true, useUnifiedTopology: true })
    .then(() => console.log('MongoDB connected...'))
    .catch(err => console.error(err));
```

A route in Express applications is a REST endpoint. As such, the GET, POST, PUT, and DELETE operations are most often used to support database CRUD (create, read, update, delete) operations. Inside the route, we define the service logic for the endpoint. The route uses a callback function to define how to handle the request and response to the REST endpoint.

In JavaScript, a callback function is a function that is passed as an argument to another function and is invoked later at a specific point in the execution flow. By using callbacks, Express follows a style of asynchronous programming, where you define the behavior of your application by providing functions that will be called later when certain events occur (such as an incoming request). This allows for non-blocking and event-driven behavior, where the server can handle multiple requests simultaneously. As a callback function for another function, they often receive arguments as a result of the asynchronous operation from the main function.

For example, the `app.get()` will provide `req` (short for request) and `res` (short for response), representing the incoming request and the response that will be sent back to the client. These objects are passed to the callback function as parameters. The `(req,res) => {}` notation is called a Arrow function notation. Within this callback function, we define the logic to handle the request and generate the response.

Define the root route using the Express application object – `app`, that we instantiated earlier. The second parameter to the `app.get` method is a callback function. The root route will simply return a welcome message using the `res` class.

```
// Define the root route
app.get('/', (req, res) => {
    res.send("Welcome to MongoDB Redis class.");
});
```

Next, define a route for /redis. Here, we use the the redis client's `set` and `get` method to save and recall and key:value pair from the Redis in-memory database server.

In this example, a regular `function(){}` notation is used instead of the Arrow function notation. They both provide the same functionality with some slight differences. The Arrow function notation was introduced as part of ES6 (ECMAScript 2015) and provide a more concise syntax compared to regular functions. However, they are unable to use `this.<attribute>` syntax. Because we do not require this capability, either notation would work here.

```
// Define the /redis route
app.get('/redis', (req, res) => {
    // Get the counter value from Redis
    client.get('counter', function(err, reply) {
        let counter;
        // If the counter exists, use its value. Otherwise, start at 1.
        if(reply) {
            counter = reply;
        } else {
            counter = 1;
        }
        // Increment and save the counter back to Redis
        client.set('counter', ++counter);
        // Send the counter value as response
        res.json({counter: counter});
    });
});
```

Finally, start the Express server and listen on port 3000.

```
// Define the application's port and start the server
const port = process.env.PORT || 3000;
app.listen(port, () => console.log(`Server running on port ${port}`));
```

3. Build the Docker Containers

    3.1 Navigate to the `myapp` directory from a terminal.

    3.2 Execute the following command. Before you execute this command, make sure that you are running Docker Desktop.

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

4. Test the application

    4.1 From a browser navigate to the following addresses:

```
http://localhost:3000/
http://localhost:3000/mongodb
http://localhost:3000/redis
```

    4.2 On the `/redis` endpoint, try refreshing the browser and observe the counter increase.