**P1: Docker Installation**

This project was started by simply reviewing what images were readily available within DockerHub to use as base images for the final installation product. I tried many configurations, where some services would work, and some wouldn’t. This was ultimately an exercise in trial by fire. We do cover some basic IT operations within this program, but setting up a LAMP stack with Docker proved to be a pretty difficult task since I was unfamiliar with a lot of what would be needed to get Dockerfiles or docker-compose files to successfully run with the desired configuration

I went through many iterations of LAMP stacks that functioned properly but didn’t check all the boxes for the project (localhost 16 + hours). In the process of getting all this up and running I got to read A LOT of documentation, stack overflow threads, and many other information outlets that familiarized me a little more with containerized applications.

**Step 1: Downloading Docker for your Machine**

First head to <https://www.docker.com/> and follow the appropriate steps to download and install docker on your local machine. The bread and butter of docker is mostly run from within the terminal (command line), but you can use the docker desktop GUI interface to see images on your local machine and running containers.

A screenshot of a computer

Description automatically generated

Also, if you’re using VSCode, be sure to get the docker extension from the extensions market place. It’ll give you linting tools and intellisense for creating / modifying dockerfiles and docker-compose files.

**Step 2: Finding Examples / Dockerfiles / Repositories**

I continually had a service that would fail when I tried to construct the docker compose files from my base knowledge. I also followed along with many tutorials that would \*almost\* get me to the finish line, but something would inevitably not work (localhost). I’ve used MAMP for all my web development projects in the past, and I really like how all server information is presented “up front” when booting a MAMP server.

I tried to find resources for setting up an index page for the apache-php container, so it would be possible to see database connections, links to project directories, and easy access to phpMyAdmin without having to enter the port number after localhost each time.

In searching for how to emulate a MAMP landing page I found this GitHub repository: <https://github.com/sprintcube/docker-compose-lamp> . By the time I had found this repository I had already worked through 8 or 10 tutorials for setting up LAMP stacks in Docker, and I hadn’t even touched the node.js image yet. When I came across this repo and saw the screen shot of the landing page, I knew this was the kind of Docker installation I was going for.

Graphical user interface, website

Description automatically generated

After reading the README.md file to get a broad understanding of how this installation was intended to work, I cloned the repository and started making changes as necessary. To streamline the step by step, I’ll describe what was done by working down the directory structure.

**Step 3: Removing Fluff / Keeping Relevance / Understanding What it Does**

**Binaries directory (bin):**

This directory contains subdirectories for the major services required for this LAMP stack. It contains MySQL, node.js, and php7.4 – Apache 2 subdirectories. Each subdirectory contains a Dockerfile which is used to build the base image when we use docker-compose in a later step.

**PHP – Apache:**

To reduce size, I only used the latest **php7.x Dockerfile** located in the **bin** directory > **php74** and omitted all others. The Dockerfile contained a lot of add-ons, some of which I removed, but given my experience with the UPS API last semester and CURL problems, needing composer, etc., I saw this as an excellent “catch all” that could eliminate problems with API calls down the line when we start developing react applications. It’s possible that issues may not arise without these extra packages, but I’d like to keep using this for future development after this class as well. It also includes extensions for using PHP Data Objects (PDO) as well as MySQLi, which I thought may be worth looking into down the road.

**MySQL**

The MySQL Dockerfile is pretty bare bones. We just need the name of the image to build from. Specifically, MySQL 8 and a command to place default authentication into the mysql config file of the MySQL container.

**Node**

The node.js Dockerfile came straight from nodejs GitHub repository. I was actually directed here from DockerHub: <https://github.com/nodejs/docker-node>. Since the build file for php/ apache is built on a DebianBuster Linux distribution, I decided to use the same with the Node Dockerfile for the sake of consistency. I know it probably doesn’t matter, but buster sounds cool, so why not?

I did not include the docker-entrypoint.sh file and just exposed port 3000 of the node container for access with the LAMP stack.

**Configuration Directory (config)**

This directory stores subdirectories for php configuration and the apache web server configuration for our coveted “virtual host”, or where we can access all of our fancy links to completed projects and phpMyAdmin.

**Php**

Subdirectory houses the changes made to the container’s php.ini file so we can get our sweet short tags on. The author’s the repository included changes to the memory limit, **post\_max\_size**, which I assume is likely for WP installations, and **upload\_max\_filesize**. I did see any harm in leaving these changes but included **short\_open\_tags** for the php info page output.

**Vhosts**

Stores the apache config file for where our server root is. This was essentially left untouched, but it turns out it was the missing link in why I couldn’t get my “MAMP” emulating root pages to work. I was pleased when I found this.

**(We’ll Come Back to Data and Logs Directories - \*Volumes\*)**

**Src**

Stores all of our nodejs application information. I worked through installing node package manager locally, and “bootstrapping” a simple react app so I could grab all the associated files and (more importantly) the package.json file that is generated after running the node command to create a react app.

Node Steps:

1. Install Nodejs for access to npm (node package manager) @ <https://nodejs.org/en/>
2. Once nodejs is installed you essentially have access to npm, but npm is updated frequently, so use: npm install npm@latest -g in the terminal (Mac) or Powershell (windows) to update.
3. Install react capability: npm install react react-dom
4. Navigate to where you’d like to create your project.
5. Create a project directory with mkdir {projects}
6. Use the global flag -g to install the create react app feature to “bootstrap” a simple single page react application: npm install -g create-react-app
7. Change directories to your project directory: cd {projects }
8. Create your simple react app with: create-react-app {project\_name } and you’ll see the cool little node status bar in your terminal window while it installs all your dependencies.

Text

Description automatically generated

The files that are included within the {project\_name} directory that was just created are what we need to include in our **src** directory for Docker to successfully run a node image. If a relevant react package.json file is not present in a mounted volume directory, Node will build each time the docker-compose commands are run and will immediately exit. If there are no node projects from which to work, node refuses to run.

Simply copy all of the react files generated and move them to the src directory. We’ll talk about how to mount these files in a moment.

**Server Root Directory (www)**

This place is what I was so thrilled about. It contains multiple php test pages and server information like:

* Phpinfo.php <- Required for project
* An index page for linking server information such as php, MySQL, and Apache version. (this came from the cloned repo.) Quick links section for a link to the required phpinfo page, the phpMyAdmin installation, mysqli test connection, PDO test connection, and the mysqli connection required for the project. A link to test the “bootstrapped” react project has been added so we don’t have to continually enter localhost:3000 into the browser search bar.
* I also added another section to the index page which will link to completed projects as they are added in a “Projects” directory.

Text

Description automatically generated

As project directories are added to the docker stack, they’ll automatically populate with the appropriate link and index page. Projects directories P1 and P2 were added for testing.The directory also contains an assets folder for some pretty slick css for banner design. I simply changed the HTML content of the banner to suit the needs of this class.

Graphical user interface, website

Description automatically generated

The last two files in the main directory are where all the magic happen. We have our docker-compose.yml file, and an .env (environment variables) file. The docker-compose file relies heavily on the .env file, so we’ll cover the variables first and why they were chosen.

**Environment Variables File (.env)**

Of course, you don’t need an .env file, and all the iterations I went through before getting a successful build never used one, but after seeing how much easier it makes changing values in one central location, I actually implemented one with my wordpress docker environment as well.

A simple glance at this file tells a lot. We can denote which version of php we want to use, where the document root directory should be in our directory structure, the path to the php.ini file, an ALL IMPORTANT DATABASE\_HOST\_NAME. Pretty much all the values we need to set for customizing our docker build can be found here. Ports to connect from the host machine to the docker containers are also listed here as well as the project information for MySQL username, password, and required database. I chose the ports for each docker container service to correspond with what is commonly used for local server installations.

**Step 4: Building Docker-compose.yml (Bread and Butter)**

This file is essentially where all the magic happens and will tie together the directory structure we have. It uses the environment variables set in the .env file and gives path references to container directories so we can “mount volumes” in order to define the desired settings for our installation.

We’ll start from the top and work our way down.

* This docker-compose file is of version ‘3’. Version 3 is the latest major version format which enables cross-compatibility between Compose and Docker’s swarm mode. We’re not using swarm, but we may want to take a look at it in the future with this stack.
* Networks have been explicitly defined here, but the definitions don’t need to overindulge in detail. I simply borrowed from Docker’s documentation on [Specifying Custom Networks](https://docs.docker.com/compose/networking/#specify-custom-networks), and defined a “frontend” and “backend” network.
* The next major tag in the docker compose file is for services: Which is where most of the work is done. We can think about all of our containers we are running as “services” we need for the stack, right?
* Service: webserver – This service is our php – apache image, or as it’s appropriately named, our “webserver”.

Text

Description automatically generated with medium confidence

* As you can see from the image above; “services” is the parent tag, “webserver” is the first services listed, “build” is a tag used to tell docker-compose \*how\* we want to build the service container, and “context” tells docker-compose where to find the details to build from. In this case, we want to build from the robust PHP-Apache2-Buster image that has all those fancy PHP add-ons. That dockerfile is located in the bin directory, and the php version environment variable needs to match the version of php – apache that we want.
  + This can be cloned with multiple versions of php and database clients (EX: php8 and MariaDB)
* The next major tag in docker-compose is for “container\_name:”. It pretty much does what it says…. Names the container which you are building for this service. This enables us to easily reference the container if we need to SSH into it for changing configurations, shutting it down, etc. I recently learned that you can also use the container name to make reference from one container service to another in scripts. I placed names for each service in the .env file to use in the compose file for quick changes.
* Next is to set ports for each service. We reference the port numbers we set in the .env file and tie them to the corresponding container port for our “virtual machine”. For instance, the unsecure (80) and secure (443) HTTP ports on your machine can be tied to the same ports for the webserver service.

Text

Description automatically generated

* I explicitly set networks for containers to communicate with one another. I had previously done this using the “links” tag, and that’s actually what was specified in the original cloned repo. However, you can read [here](https://docs.docker.com/network/links/) about how Docker has deprecated the links feature and may eventually not work with future docker engine versions.
* The volumes tag is one of the most important for our LAMP installation. As the previous image illustrates, mounting volumes for these services is actually how we get the correct PHP settings we needed for this project, along with mounting the files we want to display in the Document Root when loading the LAMP Stack landing page. We also have mounted volumes the apache configuration and we’ve mounted a directory on the host machine to log errors from the containerized apache web server.
* The last tag for the webserver service is for environment variables. Here we’re just specifying the port for phpMyAdmin along with the MySQL root password. The full service description with comprehensive tagging is listed below:

webserver:

build:

context: ./bin/${PHPVERSION}

container\_name: '${PHPVERSION}\_apache2'

ports:

- "${HOST\_MACHINE\_UNSECURE\_HOST\_PORT}:80"

- "${HOST\_MACHINE\_SECURE\_HOST\_PORT}:443"

networks:

- frontend

- backend

volumes:

- ${DOCUMENT\_ROOT-./www}:/var/www/html

- ${PHP\_INI-./config/php/php.ini}:/usr/local/etc/php/php.ini

- ${VHOSTS\_DIR-./config/vhosts}:/etc/apache2/sites-enabled

- ${LOG\_DIR-./logs/apache2}:/var/log/apache2

environment:

PMA\_PORT: ${HOST\_MACHINE\_PMA\_PORT}

MYSQL\_ROOT\_PASSWORD: ${MYSQL\_ROOT\_PASSWORD}

**Docker-compose.yml – Node Service:**

Setting up the node service was done similarly to the webserver. A location to specify the dockerfile to build from, naming the container, specifying ports, etc. Volumes were also required to be mounted so we can read files from the host machine while we’re developing. What sets the build process apart from the other containers is that it requires a command to get up and running; two actually.

When spinning up the image for Node we have to make sure we provide node package manager, similar to how we did for building our local react project. Then we have to start npm. The command must be specified by language and specific command.

node:

build:

context: "./bin/${NODE}"

container\_name: "${NODE\_JS\_NAME}"

ports:

- "${HOST\_MACHINE\_NODE\_PORT}:3000"

networks:

- frontend

- backend

working\_dir: /home/node/app

environment:

PMA\_PORT: ${HOST\_MACHINE\_PMA\_PORT}

MYSQL\_ROOT\_PASSWORD: ${MYSQL\_ROOT\_PASSWORD}

volumes:

- ./src/test416:/home/node/app

command: bash -c "npm install && npm start"

**Docker-compose.yml – Database Service**

Similar to the previous two services, this section also outlines a location to build from and a container name. It’s networked only the “backend” network since this is the primary backend service. The default port for MariaDB and MySQL is 3306, so we have the host 3306 port mapped to the container 3306 port. If we were using PostgreSQL, we’d do 5432, etc.

The volumes for this service are responsible for persisting our database information (IE: databases we’ve created, tables, data, etc.). Upon spinning up the image, a ‘data’ directory will be created to keep all the information related to this containerized MySQL installation. Similar to the webserver logs, a database logs directory is mounted locally so we can inspect database faults when we, without doubt, encounter them.

The environment variables for the MySQL database service are related to the root user password, the database, user, and password we’re required to create for this project. Values for these fields are found in the .env file.

database:

build:

context: "./bin/${DATABASE}"

container\_name: '${DATABASE\_HOST\_NAME}'

networks:

- backend

ports:

- "127.0.0.1:${HOST\_MACHINE\_MYSQL\_PORT}:3306"

volumes:

- ${MYSQL\_DATA\_DIR-./data/mysql}:/var/lib/mysql

- ${MYSQL\_LOG\_DIR-./logs/mysql}:/var/log/mysql

environment:

MYSQL\_ROOT\_PASSWORD: ${MYSQL\_ROOT\_PASSWORD}

MYSQL\_DATABASE: ${MYSQL\_DATABASE}

MYSQL\_USER: ${MYSQL\_USER}

MYSQL\_PASSWORD: ${MYSQL\_PASSWORD}

**Docker-compose.yml – phpMyAdmin (PMA) Service**

Unlike all the other services built with this compose file, this is the only one that is referenced directly from DockerHub without a Dockerfile. It’s constructed similarly to the database service and includes information for the PMA service to be accessed.

phpmyadmin:

image: phpmyadmin/phpmyadmin

container\_name: 'phpmyadmin'

networks:

- backend

environment:

PMA\_HOST: '${DATABASE\_HOST\_NAME}'

PMA\_PORT: 3306

PMA\_USER: root

PMA\_PASSWORD: ${MYSQL\_ROOT\_PASSWORD}

MYSQL\_ROOT\_PASSWORD: ${MYSQL\_ROOT\_PASSWORD}

MYSQL\_USER: ${MYSQL\_USER}

MYSQL\_PASSWORD: ${MYSQL\_PASSWORD}

ports:

- '${HOST\_MACHINE\_PMA\_PORT}:80'

volumes:

- /sessions

- ${PHP\_INI-./config/php/php.ini}:/usr/local/etc/php/conf.d/php-phpmyadmin.ini

**Step 5: Spinning up the Stack**

After all of the information needed to build the stack is in the compose file and environment variables are set, we’re ready to enter docker commands into the terminal to start our containers and get to work. You can use the MacOS terminal, Windows powershell, or VSCode terminal, if that’s your preference. Just navigate to the directory holding the docker-compose.yml file you want to build and enter:

docker-compose up -d

The ‘-d’ flag denotes that you want your containers to run in “detached” mode, so docker won’t highjack our terminal and we can run the containers in the background. If -d is not specified, you’ll get container information as it’s happening from the docker engine.

Text

Description automatically generated

Note: The first time you build this stack, IT IS GOING TO TAKE SOME TIME. Docker has to do its work and go grab all of the dependencies to build the images if they’re not already on your machine. Specifically, the apache-php and node images take quite a chunk of time to build. Once, you’ve used this command and all the images have been created, you can access the stack by opening a browser and visiting: localhost That’s it... that’s all you need. Use the appropriate links to check your database connections and test the “bootstrapped” react project.

**Caution !**

I assume it’s from mounting volumes, but give the images some time to spin up before going crazy trying to access services. It can take around a minute for everything to go where it needs to be and start running correctly. The db service can take anywhere from 30 seconds to a minute, and I’ve seen the test react project take up to two minutes to spin up.

**Closing:**

I’m pretty happy with the results of this development stack. After I learn a little more about node.js operations, I’d like to add the functionality to display the node version in the Environments column at localhost.