Running WordPress with Docker

There are many ways to run WordPress locally on your development machine.

- You can install MAMP/WAMP/XAMP or any other bundled LAMP stack, configure a database and install WordPress
- You can install Local by Flywheel, start it up and get a WordPress instance

There is one more way to do it: You can use Docker.

Docker provides a way to run container-based application on your local system. All it requires is that you have Docker Engine (for Linux) or Docker Desktop (for macOS and Windows) installed.

As a quick introduction: Our WordPress installation using Docker requires two images:

- A WordPress image containing both Apache properly configured and the WordPress code ready to install
- A MySQL or MariaDB image to store the site's data

You can optionally add a phpMyAdmin image to manage the database via a GUI

The database and the code for WordPress are passed through to the local directory where we ran `docker-compose` from. That way the changes will persist when we shutdown the container.

Yes, I know that this has already been done multiple times by developers far more experienced in Docker than I am. This is still something I would use in my projects so I'm ok with learning using this combination.

Getting Started: Looking at the pieces

We'll break the `docker-compose.yaml` file into its component services. This will help explain how they interact with each other.

We first specify the version of Docker Compose that we're using. 3.9 is the latest version.

```
version: "3.9"
```

The services section presents the three services we will use for our WordPress installation: db, `wordpress` and the optional `phpmyadmin`.

The db service configures the MySQL database we will use for the project.

We use the latest version of the Docker image for MySQL container, specified by the image attribute pointing to the name and version of the application we want, in this case: `image: mysql:latest`.

If we need to specify a version, we can replace latest` with a specific version.

The volumes section specified the external volume name and the associated directory inside the image.

The environment section specified environment variables that will be passed to the container.

Because this is an internal service, we don't expect people to access the database directly, only via phpMyAdmin or WordPress, we don't specify a container to host port translation.

```
services:
  db:
    image: mysql:latest
    volumes:
      - ./mysql:/var/lib/mysql
    restart: unless-stopped
    environment:
      MYSQL_ROOT_PASSWORD: somewordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
```

The WordPress service is the core of the application. This container will host

WordPress and interact with the database. Because of it this is where we do most of the work.

depends_on explicitly tells Docker Compose that `wordpress` depends on the db service. This will cause the following to happen:

- `docker-compose up` starts services in dependency order. db will start before `wordpress`
- If you start a specific service using `docker-compose up SERVICE`, docker automatically includes SERVICE's dependencies. In our case, docker-compose up wordpress also creates and starts db
- `docker-compose stop` stops services in dependency order. `wordpress` will stop before db

using the volumes directive, we create a volume associating a directory on the host with the directory inside the container hosting the WordPress application. This will make the code available even if the container is not running.

The ports directive maps a port on the container to a port on the host. Wordpress maps port 80 in the container to port 8000 on the host. To access WordPress, just pont your browser to `http://localhost:8000`.

restart indicates the restart policy for the container. The possible values are:

- **always**: The container always restarts
- **on-failure**: Restart a container if the exit code indicates an on-failure error
- **unless-stopped** Restarts a container, unless the container is stopped (manually or otherwise)

The environment directive holds environment variables that will be passed to the container.

```
wordpress:
  depends_on:
    - db
  image: wordpress:latest
  volumes:
    - ./wordpress_data:/var/www/html
  ports:
    - "8000:80"
```

```
    restart: unless-stopped
    environment:
      WORDPRESS_DB_HOST: db
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
      WORDPRESS_DB_NAME: wordpress
```

The phpMyAdmin container is similar to the `wordpress` container but it's not as complicated.

> The only differences are:

- The container name (`phpmyadmin`)
- The image that we use (`phpmyadmin:latest`)
- The port we expose (phpmyadmin will use port 8181)
- The values under `environment`

```
phpmyadmin:
  container_name: phpmyadmin
  image: phpmyadmin:latest
  depends_on:
    - db
  restart: unless-stopped
  ports:
    - 8181:80
  environment:
    PMA_HOST: db
    MYSQL_ROOT_PASSWORD: wordpress
```

The final image we'll work with is one for the WordPress CLI. Rather than installing it in a custom image, I found out that there are prebuilt images we can use.

```
wpcli:
  image: wordpress:cli
  depends_on:
    - db
    - wordpress
```

```
    restart: unless-stopped
    user: xfs
    volumes:
      - wordpress_data:/var/www/html
```

The top level `volumes` directive makes the specified volumes available to other containers. We don't need the functionality but it still good to have in case our project grows.

```
volumes:
  db_data:
  wordpress_data:
```

Part 2: Customizing individual images

The code works fine as is using the container defaults but there may be times when you want to customize the images.

What we'll do is to create a separate Dockerfile for each of the containers we want to customize and then reference them from the `docker-compose` file.

Creating a Dockerfile

Customizing an image usually means creating a Dockerfile with your custom configuration and then building the image or referencing the image from a `docker-compose` file.

For this example we'll create a `Dockerfile` to modify the `wordpress` image.

The Dockerfile will do the following tasks:

1. Use the latest version of the WordPress image, indicated by `FROM wordpress:beta-6.0-beta2-php8.1-apache`. 6.0 Beta 2 is the latest version of WordPress as of this writing
2. Update all the packages in the image and install Vim and wget
3. Replaces `php.ini` with the one in the local directory

```
# 1
FROM wordpress:beta-6.0-beta2-php8.1-apache

# 2
RUN apt update && \
    apt upgrade -y && \
    apt autoremove

# 3
COPY php.ini /usr/local/etc/php
```

We then modify the `docker-compose` file to reference the custom `wordpress` image.

Assuming that the files for the custom image are in a subdirectory of the current current directory, we can modify the `docker-compose.yml` file to reference the custom image:

```
  wordpress:
    build: ./custom/wordpress
    depends_on:
      - db
    volumes:
      - ./wordpress_data:/var/www/html
    ports:
      - "8000:80"
    restart: unless-stopped
    environment:
      WORDPRESS_DB_HOST: db
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
      WORDPRESS_DB_NAME: wordpress
```

`docker` and `docker-compose` commands

Here are some commands that will be necessary to build and use the project.

Building and starting the containers

The first thing to do is to build the containers. `docker-compose up` will build the containers and start them.

- **docker-compose**: the command to run
- **up**: Creates the networks and containers specified in the docker-compose.yml file. It then runs the application
- **-d**: Run in detached mode. This means the containers will spin up and `docker-compose` will return you to the shell. Otherwise the terminal will wait until you kill the `docker-compose` process with `ctrl + c`

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

We only need to do this once, after building the images, we can run `docker-compose start` to start the containers without building them. This assumes you've ran `docker-compose up` once before.

Stopping and shuting down the containers

The inverse command of `docker-compose start` is `docker-compose stop`. This will stop the containers but keep them in place so you can start them up later without having to rebuild them.

`docker-compose down` is the inverse command to `docker-compose up`. It will stop and remove all the containers associated with the `docker-compose.yml` file.

Building the custom image

When working with `docker-compose` refering to the build system is enough to pick up the right version. But it will not be enough if you want to share the custom

image or if you want to use it in other projects.

For that the solution is to build the image using `docker build`. This will build the image based on the dockerfile we create and using the `-t` flag create a name and a tag so we can distinguish our image from others we may have installed.

```
docker build -t wordpress_local:wp5_custom_1.0 .
```

Logging in to the containers

The final command to run is `docker exec`. This will run the specified command in the specified container.

The command uses two flags:

- **-t**: Attach to STDIN, STDOUT and STDERR of specified container
- **-i**: Run in interactive mode

You can combine the two flags into one, `-ti`.

```
docker exec -ti wordpress bash
```

For our project, this is important, otherwise we wouldn't be able to run the WordPress CLI we installed in the custom image.

The example command will run the Bash shell inside the `wordpress` container.

Conclusion

In this post we've talked about building a WordPress installation using Docker.

What I particularly like is that I have access to the container so I can add and remove themes, plugins and even run the WordPress CLI without having to rebuild the image or have compilers and developer tools installed.

It is just a beginning exercise to explore how we can use integrate multiple tools and applications into a single Docker-based project.

More Reading

I found Use Docker to create a local WordPress development environment particularly useful when writing this post.