Docker compose

1. *Introduction*

Docker is a great tool for automating the deployment of Linux applications inside software containers, but to take full advantage of its potential each component of an application should run in its own individual container. For complex applications with a lot of components, orchestrating all the containers to start up, communicate, and shut down together can quickly become unwieldy.

The Docker community came up with a popular solution called Fig, which allowed you to use a single YAML file to orchestrate all your Docker containers and configurations. This became so popular that the Docker team decided to make Docker Compose based on the Fig source, which is now deprecated. Docker Compose makes it easier for users to orchestrate the processes of Docker containers, including starting up, shutting down, and setting up intra-container linking and volumes.

In this tutorial, we'll show you how to install the latest version of Docker Compose to help you manage multi-container applications.

2. *Installing Docker Compose*

Although we can install Docker Compose from the official Ubuntu repositories, it is several minor version behind the latest release, so we'll install Docker Compose from the Docker's GitHub repository. The command below is slightly different than the one you'll find on the Releases page. By using the -o flag to specify the output file first rather than redirecting the output, this syntax avoids running into a permission denied error caused when using sudo.

We'll check the current release and if necessary, update it in the command below:

```
sudo curl -L https://github.com/docker/compose/releases/download/1.21.2/docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose
```

Next we'll set the permissions:

```
sudo chmod +x /usr/local/bin/docker-compose
```

Then we'll verify that the installation was successful by checking the version:

```
docker-compose --version
```

This will print out the version we installed:

```
Output
docker-compose version 1.21.2, build a133471
```

3. Running a Container with Docker Compose

The public Docker registry, Docker Hub, includes a Hello World image for demonstration and testing. It illustrates the minimal configuration required to run a container using Docker Compose: a YAML file that calls a single image:

First, we'll create a directory for the YAML file and move into it:

```
mkdir hello-world
cd hello-world
```

Then, we'll create the YAML file:

```
nano docker-compose.yml
```

Put the following contents into the file, save the file, and exit the text editor:

```
my-test:
 image: hello-world
```

The first line in the YAML file is used as part of the container name. The second line specifies which image to use to create the container. When we run the command docker-compose up it will look for a local image by the name we specified, hello-world. With this in place, we'll save and exit the file.

Now, while still in the ~/hello-world directory, we'll execute the following command:

```
docker-compose up
```

The first time we run the command, if there's no local image named hello-world, Docker Compose will pull it from the Docker Hub public repository:

4. *Wordpress, PhpMyAdmin and MySQL Example for Docker Compose:*

This article provides a real-world example of using Docker Compose to install an application, in this case WordPress with PHPMyAdmin as an extra. WordPress normally runs on a LAMP stack, which means Linux, Apache, MySQL/MariaDB, and PHP. The official WordPress Docker image includes Apache and PHP for us, so the only part we have to worry about is MariaDB.

1. *Installing WordPress*

We'll be using the official WordPress and MariaDB Docker images. If you're curious, there's lots more info about these images and their configuration options on their respective GitHub and Docker Hub pages.

Let's start by making a folder where our data will live and creating a minimal docker-compose.yml file to run our WordPress container:

```
mkdir ~/wordpress && cd $_
```

Then create a ~/wordpress/docker-compose.yml with your favorite text editor (nano is easy if you don't have a preference):

```
nano ~/wordpress/docker-compose.yml
```

and paste in the following:

```
wordpress:
image: wordpress
```

This just tells Docker Compose to start a new container called wordpress and download the wordpress image from the Docker Hub.

2. *Installing* MariaDB and PhpMyAdmin

To add the MariaDB and PhpMyAdmin image to the group, Change docker-compose.yml to match the below (be careful with the indentation, YAML files are white-space sensitive)

```
wordpress:
image: wordpress
links:
- wordpress_db:mysql
ports:
- 8080:80
wordpress_db:
image: mariadb
environment:
MYSQL_ROOT_PASSWORD: examplepass
phpmyadmin:
image: corbinu/docker-phpmyadmin
links:
- wordpress_db:mysql
ports:
- 8181:80
environment:
MYSQL_USERNAME: root
MYSQL_ROOT_PASSWORD: examplepass
```

What we've done here is define a new container called wordpress_db and told it to use the mariadb image from the Docker Hub. We also told the our wordpress container to link our wordpress_db container into the wordpress container and call it mysql (inside the wordpress container the hostname mysql will be forwarded to our wordpress_db container).

Then we set an environment variable inside the wordpress_db container called MYSQL_ROOT_PASSWORD with your desired password. The MariaDB Docker image is configured to check for this environment variable when it starts up and will take care of setting up the DB with a root account with the password defined as MYSQL_ROOT_PASSWORD.

Then setting up a port forward so that we can connect to our WordPress install once it actually loads up. The first port number is the port number on the host, and the second port number is the port inside the container. So, this configuration forwards requests on port 8080 of the host to the default web server port 80 inside the container.

So far we've only been using official images, which the Docker team takes great pains to ensure are accurate. You may have noticed that we didn't have to give the WordPress container any environment variables to configure it. As soon as we linked it up to a properly configured MariaDB container everything just worked.

This is because there's a script inside the WordPress Docker container that actually grabs the **MYSQL_ROOT_PASSWORD** variable from our **wordpress_db** container and uses that to connect to WordPress.

The last part of the configuration is PhpMyAdmin image which grabs **docker-phpmyadmin** by community member **corbinu**, links it to our **wordpress_db** container with the name **mysql** (meaning from inside the **phpmyadmin** container references to the hostname **mysql** will be forwarded to our **wordpress_db** container), exposes its port 80 on port 8181 of the host system, and finally sets a couple of environment variables with our MariaDB username and password. This image does not automatically grab the **MYSQL_ROOT_PASSWORD** environment variable from the **wordpress_db** container's environment the way the **wordpress** image does. We actually have to copy the **MYSQL_ROOT_PASSWORD:examplepass** line from the **wordpress_db** container, and set the username to **root**.

With this configuration we can actually go ahead and fire up WordPress. This time, let's run it with the **-d** option, which will tell **docker-compose** to run the containers in the background so that you can keep using your terminal:

sudo docker-compose up -d

Then we can ***browse http://localhost:8080 and http://localhost:8181*** and see the worpress and PhpMyAdmin pages.

Go ahead and login using username **root** and password you set in the YAML file, and you'll be able to browse your database. You'll notice that the server includes a **wordpress** database, which contains all the data from your WordPress install.

You can add as many containers as you like this way and link them all up in any way you please. As you can see, the approach is quite powerful —instead of dealing with the configuration and prerequisites for each individual components and setting them all up on the same server, you get to plug the pieces together like Lego blocks and add components piecemeal. Using tools like Docker Swarm you can even transparently run these containers over multiple servers! That's a bit outside the scope of this tutorial though. Docker provides some [documentation] ((https://docs.docker.com/swarm/install-w-machine/)) on it if you are interested.

3. Creating the WordPress Site

Since all the files for your new WordPress site are stored inside your Docker container, what happens to your files when you stop the container and start it again?

By default, the document root for the WordPress container is persistent. This is because the WordPress image from the Docker Hub is configured this way. If you make a change to your WordPress site, stop the application group, and start it again, your website will still have the changes you made.

Let's try it.

Go to your WordPress from a web browser (e.g. http://localhost:8080). Edit the **Hello World!** Post that already exists. Then, stop all the Docker containers with the following command:

```
docker-compose stop
```

Try loading the WordPress site again. You will see that the website is down. Start the Docker containers again:

```
docker-compose up –d
```

Again, load the WordPress site. You should see your blog site and the change you made earlier. This shows that the changes you make are saved even when the containers are stopped.

4. Storing the Document Root on the Host Filesystem

It is possible to store the document root for WordPress on the host filesystem using a Docker data volume to share files between the host and the container.

Let's give it a try. Open up your docker-compose.yml file one more time:

```
nano ~/wordpress/docker-compose.yml
```

in the wordpress: section add the following lines:

```
wordpress:
...
  volumes:
    - ~/wordpress/wp_html:/var/www/html
    ...
```

Stop your currently running docker-compose session:

```
docker-compose stop
```

Remove the existing container so we can map the volume to the host filesystem:

```
docker-compose rm wordpress
```

Start WordPress again:

```
docker-compose -d
```

Once the prompt returns, WordPress should be up and running again — this time using the host filesystem to store the document root.

If you look in your ~/wordpress directory, you'll see that there is now a wp_html directory in it:

```
ls ~/wordpress
```

All of the WordPress source files are inside it. Changes you make will be picked up by the WordPress container in real time.

This experience was a little smoother than it normally would be — the WordPress Docker container is configured to check if /var/www/html is empty or not when it starts and copies files there appropriately. Usually you will have to do this step yourself.

5. *Installing Gitlab project using Docker-Compose*

- ***Prerequisites***

Your docker host needs to have 1GB or more of available RAM to run GitLab. Please refer to the GitLab hardware requirements documentation for additional information.

- ***Quick Start***

The quickest way to get started is using docker-compose.

```
wget https://raw.githubusercontent.com/sameersbn/docker-gitlab/master/docker-compose.yml
```

Generate random strings that are at least 64 characters long for each of GITLAB_SECRETS_OTP_KEY_BASE, GITLAB_SECRETS_DB_KEY_BASE, and GITLAB_SECRETS_SECRET_KEY_BASE. These values are used for the following:

- GITLAB_SECRETS_OTP_KEY_BASE is used to encrypt 2FA secrets in the database. If you lose or rotate this secret, none of your users will be able to log in using 2FA.
- GITLAB_SECRETS_DB_KEY_BASE is used to encrypt CI secret variables, as well as import credentials, in the database. If you lose or rotate this secret, you will not be able to use existing CI secrets.
- GITLAB_SECRETS_SECRET_KEY_BASE is used for password reset links, and other 'standard' auth features. If you lose or rotate this secret, password reset tokens in emails will reset.

Tip: You can generate a random string using pwgen -Bsv1 64 and assign it as the value of GITLAB_SECRETS_DB_KEY_BASE.

Start GitLab using:

```
docker-compose up
```

NOTE: Please allow a couple of minutes for the GitLab application to start.

Point your browser to ***http://localhost:10080*** and set a password for the root user account.
You should now have the GitLab application up and ready for testing.

6. *References*

https://www.digitalocean.com/community/tutorials/how-to-install-docker-compose-on-ubuntu-18-04

https://www.digitalocean.com/community/tutorials/how-to-install-wordpress-and-phpmyadmin-with-docker-compose-on-ubuntu-14-04

https://github.com/sameersbn/docker-gitlab