Dockerizing a Flask-MySQL app with docker-compose

Updated: April 23, 2018

In this tutorial we will go through an example of taking an existing simple web app based on Flask and MySQL and making it run with Docker and docker-compose.

It is considered a best practice for a container to have only one responsibility and one process, so for our app we will need at least two containers — one for running the app itself, and one for running the database. How do we coordinate these containers? This is where docker-compose comes in. From the official docs:

> *Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.*

Let's start!

All code used in this tutorial is available here:

stavshamir / **docker-tutorial**

Code for a creating a docker app with Flask and MySQL tutorial

63 54

If you don't have them installed yet, install Docker and docker-compose. We begin with the following project layout:

```
dockerize/
├── app
│   └── app.py
└── db
    └── init.sql
```

- *app.py* — contains the Flask app which connects to the database and exposes one REST API endpoint
- *init.sql* — an SQL script to initialize the database before the first time the app runs

Creating a Docker image for our app

We want to create a Docker image for our app, so we need to create a Dockerfile in the app directory. A Dockerfile contains a set of instructions describing our desired image and allow its automatic build.

```
# Use an official Python runtime as an image
FROM python:3.6

# The EXPOSE instruction indicates the ports on which a container
# will listen for connections
# Since Flask apps listen to port 5000  by default, we expose it
EXPOSE 5000

# Sets the working directory for following COPY and CMD instructions
# Notice we haven’t created a directory by this name - this instruction
# creates a directory with this name if it doesn’t exist
WORKDIR /app

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

# Run app.py when the container launches
COPY app.py /app
CMD python app.py
```

What this does is simply as described in the file—base the image on a Python 3.6 image, expose port 5000 (for Flask), create a working directory to which requirements.txt and app.py will be copied, install the needed packages and run the app.

We need our dependencies (Flask and mysql-connector) to be installed and delivered with the image, so we need to create the aforementioned requirements.txt file:

```
Flask
mysql-connector
```

Now we can create a docker image for our app, but we still can’t use it, since it depends on MySQL, which, as good practice commens, will reside in a different container. We will use docker-compose to facilitate the orchestration of the two independant containers into one working app.

Creating a docker-compose.yml

So let’s create a new file, docker-compose.yml, in our project’s root directory:

```
version: "2"
services:
  app:
    build: ./app
    links:
      - db
    ports:
      - "5000:5000"
```

We are using two services, one is a container which exposes the REST API (app), and one contains the database (db).

- *build:* specifies the directory which contains the Dockerfile containing the instructions for building this service
- *links:* links this service to another container. This will also allow us to use the name of the service instead of having to find the ip of the database container, and express a dependency which will determine the order of start up of the container
- *ports:* mapping of <Host>:<Container> ports.

```
db:
  image: mysql:5.7
  ports:
    - "32000:3306"
  environment:
    MYSQL_ROOT_PASSWORD: root
  volumes:
    - ./db:/docker-entrypoint-initdb.d/:ro
```

- *image:* Like the FROM instruction from the Dockerfile. Instead of writing a new Dockerfile, we are using an existing image from a repository. It's important to specify the version—if your installed mysql client is not of the same version problems may occur.
- *environment:* add environment variables. The specified variable is required for this image, and as its name suggests, configures the password for the root user of MySQL in this container. More variables are specified here.
- *ports:* Since I already have a running mysql instance on my host using this port, I am mapping it to a different one. Notice that the mapping is only from host to container, so our app service container will still use port 3306 to connect to the database.
- *volumes:* since we want the container to be initialized with our schema, we wire the directory containing our init.sql script to the entry point for this container, which by the image's specification runs all .sql scripts in the given directory.

We are now ready to start the dockerized app! But before we do that, let's peek at the code connecting to the database (app.py):

```
config = {
        'user': 'root',
        'password': 'root',
        'host': 'db',
        'port': '3306',
        'database': 'knights'
    }
connection = mysql.connector.connect(**config)
```

We are connecting as root with the password configured in the docker-compose file. Notice that we explicitly define the host (which is localhost by default) since the SQL service is actually in a different container than the one running this code. We can (and should) use the name 'db' since this is the name of the service we defined and linked to earlier, and the port is 3306 and not 32000 since this code is not running on the host.

Running the app

In order to run the our dockerized app, we will execute the following command from the terminal:

```
$ docker-compose up
```

You can see the image being built, the packages installed according to the requirements.txt, etc. If everything went right, you will see the following line:

```
app_1  |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

We can find out that everything is running as expected by typing this url in a browser or using curl, and receiving the following response:

```
{"favorite_colors": [{"Lancelot": "blue"}, {"Galahad": "yellow"}]}
```

You can access the database directly using the mysql client and following command (make sure your client is the same version of MySQL specified in the docker-compose.yml):

```
$ mysql --host=127.0.0.1 --port=32000 -u root -p
```

It is a must to specify the host IP, since when the default ‘localhost’ is used MySQL ignores the port parameter.