Build an Instagram clone using microservices - from 0 to production

Image by Pixelgeezer

Table of contents

Introduction

In this book we will be creating, deploying, and scaling an instagram clone using microservices. Microservices will be written using either Python or Javasrcript. This is a full stack approach which means we will be writing both the frontend, the backend, and will be dealing with scalability.

We start from the very basics of web applications and microservices and build from that to an instagram like application.

A few words about the book

This is a very practical book. My aim is that this book stays short and to the point. I couldn't find any books the way I wanted them to be so I decided to write this. This book is for people who want to move fast.

Who are you

I assume that you know programming to some extend and have an understanding of the very basics of web app development and docker. Although we will go through the basics of docker, it won't be in depth and you might want to consult other resources as well.

About the author

I am a professional software engineer working for Google with many years of experience on building web applications.

Intro to Docker & Microservices

Creating a simple python web app - `simplehello`

We will use Flask to create an example web app that we will name `simplehello` that when you visit in the browser will show a message "hello world!". Using Flask this is a very simple task: we only need to create one file named `app.py`. To keep things organised we will create a folder called `simplehello` where the code of this web app will reside. Then within the `simplehello` folder we can create the `app.py` file.

app.py

```
1  from flask import Flask
2  app = Flask(__name__)
3
4  @app.route('/')
5  def index():
6      return 'hello world!'
7
8  if __name__ == "__main__":
9      app.run(debug=True, host='0.0.0.0', port=5000)
```

Lines 1,2 import flask and create the flask `app`. Line 4 defines an API endpoint. The function defined on line 5 returns the content that either users will see when opening the index page of the app or services will receive as a reply when calling this API endpoint. Finally, line 9 runs the app on port 5000 in debug mode.

To run this app we need to follow 3 steps:

1. **Create a new virtual environment.** Virtual environments are isolated environments so that dependencies between projects are not messed up. The reason that we need them is because without virtual environments Python dependencies are installed globally. Therefore, if we globally install a specific version of Flask x we won't be able to run another project using Flask version y, unless we unistall version x and install version y. This goes for all packages, not just Flask. The solution the Python community came up with to address this problem are virtual environments. Within each virtual environment we can install different versions of each package, without one overlapping the other.

   To create a virtual environment we run the following commands in the same folder

```
# Create a new virtual environment
$ virtualenv venv

# Activate the virtual environment. Once you execute this command you will be inside the virtual environment
$ source venv/bin/activate

# You can tell you are in a virtual environment by the parentheses at the front
(venv)$
```

Later on we will need to deactivate (escape) the virtual environment. We can do so using the `deactivate` command:

```
# Escape the virtual environment by typing the following:
(venv)$ deactivate
```

2. **Install dependencies**

```
# Installing flask version 1.0.3 in the virtual environment
(venv)$ pip install flask
```

3. **Run the app**

```
# Run the app, if everything goes well after running the command by visiting
# http://localhost:5000/ you will see a "hello world!" message
(venv)$ python app.py
...
Running on http://0.0.0.0:5000/
...
```

Visit http://localhost:5000/ to see the "hello world!" message. Press CTRL + C to stop the server.

Now that you have seen how we can create an very simple web app we can go a step further and add a new API in `app.py` where we input our name in the URL so that it can greet us.

```
@app.route('/hello/<name>')
def hello(name):
    return 'hello there ' + name
```

Now if you run `$ python app.py` and visit http://localhost:5000/hello/instagram you will see the message “hello there instagram”.

Before moving on with Docker make sure that you escape the virtual environment:

```
(venv)$ deactivate
```

Docker

Introduction to Docker

The idea of Docker is quite simple: imagine buying a few new computer or creating a few new virtual machine (VM) with the operating system (OS) being the only thing installed. Once you boot the computers or open the VMs for the first time we can assume that all of them will be in the same state. By running the same commands on each of them all of them will be in the same state. This is because they start from the same state and then you make the same changes to the state. It's like programming, e.g. assume the following: State 0 -> `int x = 0;` State 1 -> `x = x + 15;` State 2 -> `x = x - 5;` At the end of "State 2" you expect `x` to equal 10, no matter how many times you run the program or where you run it.

In the same fashion in Docker, you start with a base image which is usually provided by the community and is the same for everyone. Like an Ubuntu image, or a Python image, etc... Then you specify a list of commands to be run in a specific order starting from top to bottom. No matter how many times you run the commands or where you run them the state after each command is the same. That's the gist of Docker. The file that specifies which commands to run is called the `Dockerfile`.

Dockerizing the python web app

Let's dockerize the above application. To do that we need to create a file in that directory called `Dockerfile`.

Dockerfile

```
# Use the official python 3.7 base image
FROM python:3.7

# Install Flask
RUN pip install Flask

# Set the working directory in the container to be /app
WORKDIR /app

# Copy the file "app.py" from the file system to the container
# and place it in the /app directory we've just created
COPY app.py /app

# Exposes port 5000 for publishing/linking
EXPOSE 5000

# Run the python app
CMD ["python", "app.py"]
```

We can build & run this dockerfile by running the following commands:

```
# This command builds a docker image which will be used to spawn container instances
$ docker build -t first_example .

# In this command we spawn a container from the first_example image and map container's port 5000 to localhost port 8080
$ docker run -d -p 8080:5000 first_example
```

Now visit http://localhost:8080/hello/instagram to see the message you got before.

At this point we should note that the server we are using at the moment is not a production grade server and should only be used in development. We can use a server like Gunicorn as our production server. Also, instead of having to manually install all the requirements by hand in the dockerfile (like `RUN pip install Flask`), we can write them in a `requirements.txt` file and copy it from the filesystem to the container.

requirements.txt

```
Flask
Gunicorn
```

By making these 2 changes (adding Gunicorn and `requirements.txt`), the final `Dockerfile` looks like this:

```
# Use the official python 3.7 base image
FROM python:3.7

# Set the working directory in the container to be /app
WORKDIR /app

# Copy the requirements.txt file from the filesystem to the container
ADD requirements.txt /app/requirements.txt

# Install dependencies listed in the requirements.txt file
RUN pip install -r requirements.txt

# Copy the file "app.py" from the file system to the container
# and place it in the /app directory we've just created
COPY app.py /app

# Exposes port 5000 for publishing/linking
EXPOSE 5000

# Run the python app
CMD ["gunicorn", "app:app", "-b", "0.0.0.0:5000"]
```

To run the updated `Dockerfile` we must execute the `docker build` and `docker run` commands again.

Creating and dockerizing a node.js app - `simplemath`

So far we have seen how to write and dockerize a Python web app. To make things more interesting let's try to create a second service which is going to do some basic math calculations and will later be called by the `simplehello` service. We can name this service `simplemath` and write it using javascript with node.js.

Create a new folder named `simplemath` in the same level as the `simplehello` directory and create 2 new empty files inside of `simplemath` named `Dockerfile` and `app.js`.

The final folder structure should look like this:

```
.
├── simplehello
|   ├── Dockerfile
|   ├── app.py
|   └── requirements.txt
└── simplemath
    ├── Dockerfile
    └── app.js
```

The contents of `app.js`:

```
const express = require('express')
const app = express()
const port = 3000

// / API responds with a hello world message
app.get('/', (req, res) => res.send('Hello world from node!'))

// /simplemath API takes two numbers `a` and `b` and multiplies them.
app.get('/simplemath', (req, res) => {
    // if no number is provided, then set default values
    let a = req.query.a || 5;
    let b = req.query.b || 10;
    res.send('mul: ' + a*b);
});

app.listen(port, () => console.log(`Node app listening on port ${port}!`))
```

Once you have done this we need to setup the environment for node.js:

```
$ cd simplemath/

# Initialises npm package
$ npm init
(click enter to all of them)

# Install express.js - a web app framework
$ npm install express

# Run the javascript web application
$ node app.js
```

If everyhing went well you should see “Hello world from node!” when visiting http://localhost:3000. Whereas by visiting http://localhost:3000/simplemath?a=2&b=3 you should see the result of multiplying `a` and `b`: “mul: 6”. You can change the numbers in the url `a=2&b=3` to be any other number that you like.

Dockerise the node.js web app

Dockerfile

```
# Use the official node base image
FROM node:12.2-alpine

# Set the working directory in the container to be /app
WORKDIR /app

# Copy the package.json file from the host inside the container
COPY package.json /app

# Use the package.json to run npm install
RUN npm install

# Copy all the code files from the host inside the container in the /app path
COPY . /app

# Expose port 3000 for incoming connections
EXPOSE 3000

# Run the server
CMD ["node", "app.js"]
```

Build it:

```
# This command builds a docker image which will be used to spawn container instances
$ docker build -t simplemath .
```

Run it:

```
# In this command we spawn a container from the simplemath image and map container's port 3000 to localhost port 8081
$ docker run -d -p 8081:3000 simplemath
```

By visiting http://localhost:8081/simplemath?a=3&b=4 you should see the result of the multiplication to be 12.

Microservices with docker-compose

We could use the `docker` command to setup networks and then have the two containers communicate, but this is time consuming as we need to spend time investigating how docker networking works and getting it right all the time. Even if you do that the process won't be as fast and intuitive as it can be. docker-compose solves this very problem; it uses yaml to define the services of an application in a intuitive manner and provides intuitive commands for orchestrating services on a specific host machine.

docker-compose

So far we have built and run the 2 services independently. Now we want to orchestrate the 2 services and have the `helloworld` service call `simplemath`.

docker-compose.yml

```
version: "3"

services:
  helloworld.api:
    build: helloworld/
    ports:
      - "5000:5000"
    depends_on:
      - simplemath.api

  simplemath.api:
    build: simplemath/
```

Now that we have defined the 2 services we can use their names in the code. We change the `helloworld/app.py` file as follows:

```
import requests
...
...
@app.route('/hello/<name>')
def hello(name):
    payload = {'a': 6, 'b': 4}
    r = requests.get('http://simplemath.api:3000/simplemath',
params=payload)
    return 'hello there ' + name + ' ' + r.text
```

As you can see, we use `simplemath.api:3000`. This is the API defined in the `docker-compose.yml` file.

We also need to add the `requests` library in the `requirements.txt` file.

requirements.txt

```
Flask
Gunicorn
requests
```

Before running the microservices we need to build the `docker-compose.yml` specification file:

```
# Build the services
$ docker-compose build
```

Then to run:

```
# Run the services
$ docker-compose up
```

Visit http://localhost:8080/hello/instagram and you will see the following message: "hello there instagram mul: 24".

When we visit the above URL, we visit the `/hello/<name>` endpoint in the Python app (`simplehello` service), which, in its turn, calls the `simplemath` service asking for the result of the multiplication of 6*4 that is appended to the returned message.

To stop the microservices press CTRL + C and if you want to remove containers, networks, and other things that the docker-compose command created under the hood, run:

```
$ docker-compose down
```

So far we have seen how to create two micro-services in different programming languages, dockerize them, and connect them using docker-compose. Now that you know the basics of microservices we can move on to more advanced topics.