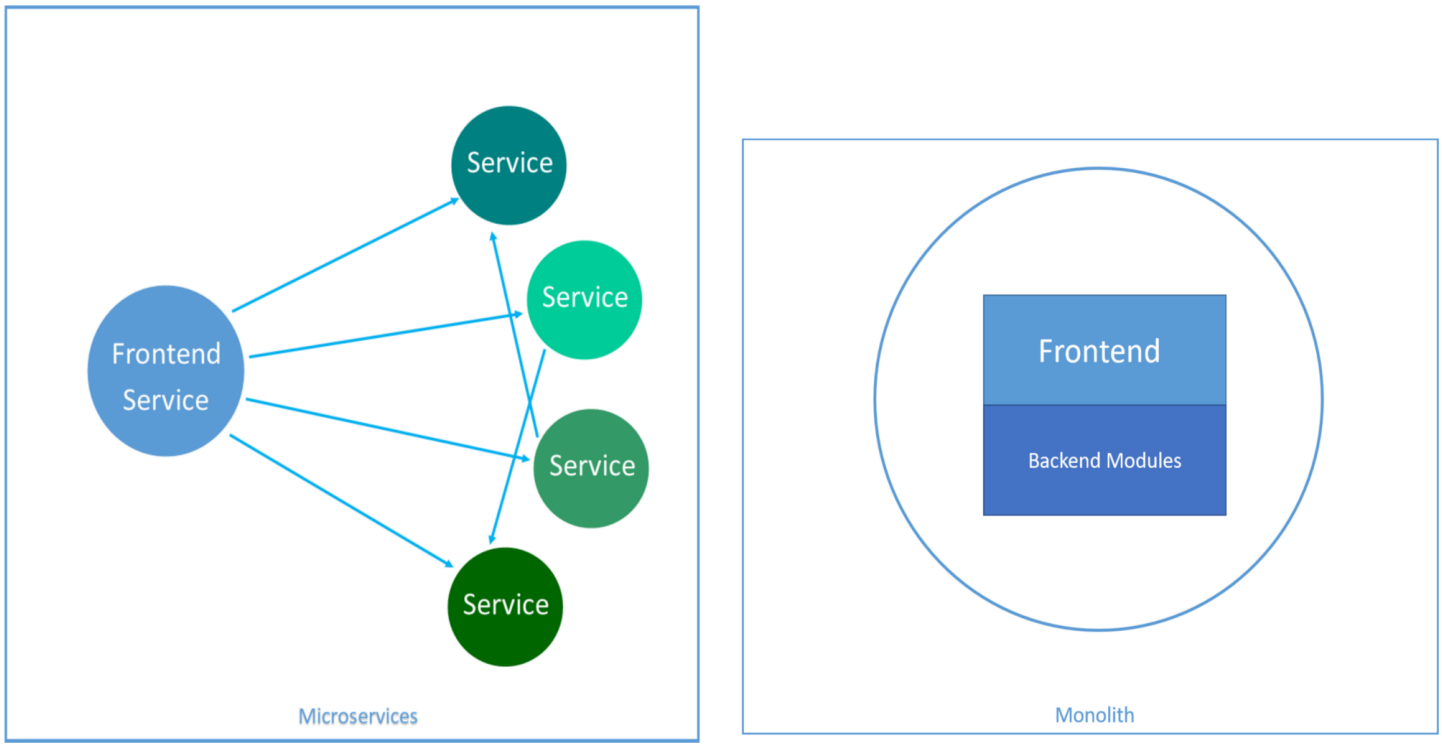
Deploying Microservices with Docker Compose

John Olafenwa
Oct 3, 2019 · 7 min read



Microservices and Monoliths: Credits: John Olafenwa

Introduction to Microservices

Web applications are often built as a single large code base often written in a single programming language. This so-called “Monolithic” applications are often organized into several modules and classes that performs different functions. However, they are still one single large code base running on the same software stack. The downside to this approach is that all components and all developers working on the same application has to use the same software stack, and in the case of very large complex applications managed by multiple teams, pushing updates can be very painful as separate components cannot be re-deployed independently of other components.

Microservices are the very opposite of this. In this new paradigm, every application is broken into several smaller services each of which is often run in its own separate container. Rather than having multiple modules and multiple classes, our application becomes a suite of multiple containers operating in unison. To understand this clearly. Lets consider we are building an enterprise application with functions for accounts management, employee management and assets tracking. In a basic microservice setup, we could have a service for the application frontend, a service for accounts management, a service for employee management and another for assets tracking. This is one single application split into 4 separate services. Each single one hosted in its own container, written in its own programming language of choice and completely independent of the way the other services are built. Consequentially, separate teams can work on separate services and re-deploy those services without affecting the work of the other teams.

All the services consume each others function via well defined APIs over REST or gRPC.

In fact, while in the past few years, it is common to talk about a website like Google search being written in Python and C++, today, that very statement is starting to become invalid. Applications are no longer viewed as a code base written in a particular language, rather they are a suite of independent services, with each service written in any language suited to its developers.

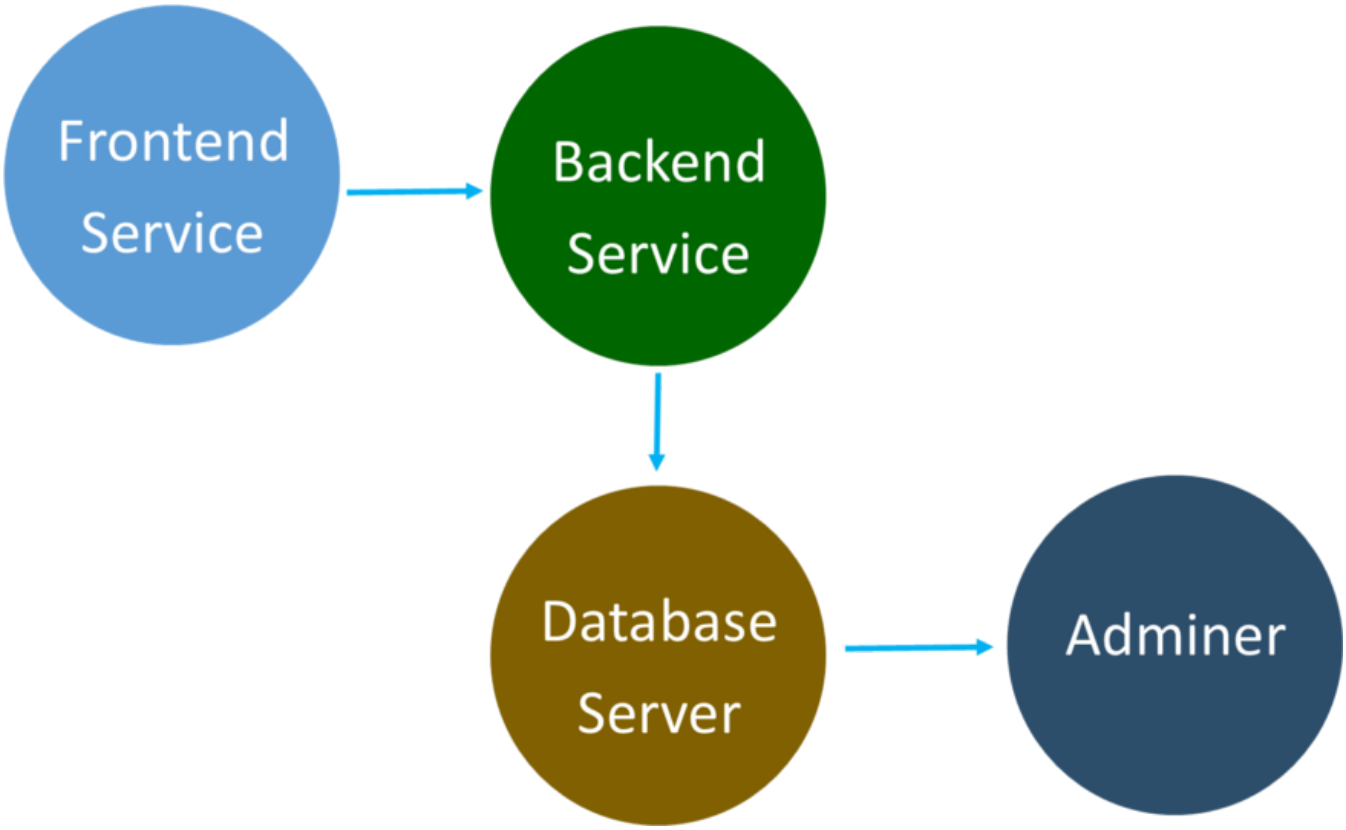
Building A Microservice Application

In this tutorial, we are going to build a review website for “Avengers Endgame” . This application would be divided into the following services.

- **Frontend Service**
- **Backend Service**
- **PostreSQL Database Service**
- **Adminer Database Admin Service**

Each of the above will be a separate docker container and we shall be leveraging on a number of concepts explained in the past three tutorials on docker to build this

microservice.



Structure of the Endgame Review Application: Credits: John Olafenwa

The Frontend Service

Our frontend service is to provide two pages, one for submitting reviews and the other for displaying all previously submitted reviews.

> *The full code for this project is available on https://github.com/johnolafenwa/DevPlanet*
>
> *clone the DevPlanet repo and you will find the frontend folder in*
>
> *devplanet/samples/microservice*

The two important files we shall explain here are the app.py and the Dockerfile for the frontend.

app.py

```
1  from flask import Flask,request,render_template,redirect
2  import requests
```

```
import os

class Review(object):
    def __init__(self,name,review):
        self.name = name
        self.review = review

app = Flask(__name__)

@app.route("/",methods=["POST","GET"])
def index():

    if request.method == "POST":
        name = request.form["name"]
        review = request.form["review"]

        res = requests.post("http://backend:5000/reviews/add",data={"name":name,"review":review

        if res.status_code == 200:
            return redirect("/reviews")

    return render_template("index.html")

@app.route("/reviews",methods=["GET"])
def reviews():

    res = requests.post("http://backend:5000/reviews/list").json()
    reviews = []

    for review in res["list"]:
        reviews.append(Review(review[0],review[1]))

    return render_template("reviews.html",reviews=reviews)

app.run(host="0.0.0.0",port=5000)
```

endgame-frontend-app.py hosted with ♡ by GitHub view raw

Here, in the landing page served by the **index** function. We retrieved form submission details and we attempt to create a new review entry by sending a post request to the **reviews/add** endpoint on the **backend** service. This part is shown again below.

```
res = requests.post("http://backend:5000/reviews/add",data=
{"name":name,"review":review})
```

If the response code is 200 indicating success, we redirect the user to the reviews list.

The backend service would be deployed on port 5000 in our microservice.

The reviews are displayed by the **reviews** function. This function calls the **/reviews/list** endpoint on the **backend** service to retrieve a list of all reviews previously submitted.

```
res = requests.post("http://backend:5000/reviews/list").json()
```

Dockerfile

```
FROM python:3.6-slim

WORKDIR /app

COPY . /app

RUN pip3 install -r requirements.txt

EXPOSE 5000

CMD ["python3","app.py"]
```

requirements.txt

```
flask

requests
```

Now, ensure you are in the **devplanet/samples/microservice/endgamefrontend** folder

Run

```
sudo docker build -t endgamefrontend:v1 .
```

This would build a docker image tagged **endgamefrontend:v1**

The Backend Service

Our backend service directly interacts with the database service. It's major function is to create new review entries in the database and to query the database for all submitted reviews.

> *Files for the backend service is located in the endgamebackend folder*

The core files here are app.py, dockerfile and requirements.txt

app.py

As you can observe from the code above, our backend service has two major endpoints, **/reviews/add** and **/reviews/list**, each of these performs operations on the database and returns a json response to the caller(the frontend service).

A key part of this file that needs explaining is

```
DB_USER = os.environ["DB_USER"]

DB_PASSWORD = os.environ["DB_PASS"]

DB = os.environ["DB"]

DB_HOST = os.environ["DB_HOST"]
```

Here, we are using environment variables to pass in the database credentials, also, we do not assume a fixed service name or endpoint for the **DB_HOST** as it could easily be a database hosted outside our application cluster.

Dockerfile

```
FROM python:3.6-slim

WORKDIR /app

COPY . /app

RUN pip3 install -r requirements.txt

EXPOSE 5000

CMD ["python3","app.py"]
```

requirements.txt

```
flask

psycopg2-binary
```

Ensure you are in the **devplanet/samples/microservice/endgamebackend** folder

Run

```
sudo docker build -t endgamebackend:v1 .
```

This would build a docker image tagged **endgamebackend:v1**

That completes building the docker images for our application. The postgre SQL database image and the adminer image are available on docker hub, so we are just going to pull those right from the hub.

Deploying A Microservice Application with Docker Compose

In previous chapters, we ran our application using **sudo docker run**. However, in this case, we need to run multiple docker images as a single deployment. Hence, we shall be using an orchestration tool. Here comes the realm of Docker Compose and Kubernetes. Here we shall use docker compose as it is simple enough and sufficient to run our application, while it is a powerful tool, it is not comparable to the breadth of functionalities of Kubernetes and the universe of tools built for kubernetes. The next series is dedicated to kubernetes.

Install Docker Compose

Docker compose comes pre-installed with docker on windows and macos however, if running on linux, please install compose as seen below.

```
sudo curl -L
"https://github.com/docker/compose/releases/download/1.24.1/docker-
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

Make it executable

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

Create Your Deployment File

> *You will find the file **deploy.yaml** in **devplanet/samples/microservice** folder*

As seen below, our deployment is simply a bunch of yaml instructions. By now, you should know that YAML is the unofficial programming language of the cloud native world.

Breakdown of The Deployment File

Every compose file is made of services, each service runs a specific docker image.

DB SERVICE

The **db** service runs the **Postgre SQL** image. The **environment variables;** here are used to initialize the database server.

```
POSTGRES_USER: john

POSTGRES_PASSWORD: example

POSTGRES_DB: endgame
```

These same credentials would be passed to the backend service.

The volume

```
- dbvolume:/var/lib/postgresql/data
```

Maps the **dbvolume** to the directory where **postgresql** stores its data. It is absolutely essential to map a volume to our database instance else we lose all data in case of any restart.

ADMINER SERVICE

Adminer is an administrative tool for a wide range of databases. We can use it just like PHPMyAdmin. Here we exposed it on port 9090, so we can access it from the browser and manage our database.

BACKEND SERVICE

Our backend service runs the **endgamebackend:v1** image. Key points to note here are the environment variables.

```
DB_USER: john

DB_PASS: example

DB: endgame

DB_HOST: db
```

We use them to provide the credentials to access the database.

Finally, there is the **links** section. We use this to allow the backend service access the db service.

```
- db
```

FRONTEND SERVICE

This runs the image **endgamefrontend:v1**

Since it is the frontend users will access, it is exposed on port 80. Also, since we would need to call the backend service from the frontend, we added a link to the backend.

VOLUMES

The last piece in our compose file is the section where we explicitly defined the volume **dbvolume**. Any volumes you use in compose must be defined here.b

> *Note that only two services;* ***frontend*** *and* ***backend*** *were exposed publicly using port mappings. The* ***backend*** *and* ***postre*** *services are only accessed internally.*

RUNNING THE DEPLOYMENT

Ensure you are in the directory **devplanet/samples/microservice** folder.

Now run

```
sudo docker-compose -f deploy.yaml up
```

This would spin up multiple containers, running all your services.

Wait about 30 seconds for all services to initialize and visit **localhost**

To manage the database, visit adminer on **localhost:9090**

The interface below will appear

When adminer comes up, MySQL is selected by default, remember to select Postgre SQL from the list

Stopping Your Deployment

To stop your application including the database and all running services, simply run.

```
sudo docker-compose -f deploy.yaml down
```

SUMMARY

So far we have created a microservice application and successfully deployed it alongside Postgre SQL using docker compose.

Docker compose offers a lot more features for scaling and managing our deployments. We shall go over some more of them in the next tutorial.

While compose can be used to orchestrate containers on clusters of servers. We shall limit our usage of compose to a single node machine. The goal of this series is to offer foundation knowledge on building containers and microservices.

More advance scaling across servers would be treated in the coming series on kubernetes. Stay tuned to DevPlanet.

For any questions or guidance, You can reach me on twitter @johnolafenwa

Email: johnolafenwa@gmail.com

Docker Microservices Docker Compose Cloud Native Web Development