Nish Gowda

About Follow

Containerizing Microservices in Docker Images

 Nish Gowda Oct 17 · 8 min read

Microservices and Docker. In the world of web development, the two are not really mutually exclusive. A microservice architecture has a lot of benefits when developing

them so they can communicate effectively with each other.

Let's start with a quick and easy example. We're going to be developing a simple note-taking app. Users can sign up, create notes, edit notes, and all that vanilla CRUD stuff. We're also going to be containerizing our front end as well (which will be a React app) just to leave no stones unturned.

Setup

Docker (duh!). Make sure you have docker installed and know the basics of it. This isn't a tutorial on how to use docker, so if you don't know that much about it, you can learn more here.

In this example, we're going to be using Node.js and React to develop our web app. So let's see how we're going to break down this into several services. To do this, we need to think about what we need:

1. A database. We need to persist data securely so users can view their data. In this example, I'm going to be using PostgreSQL, but the essential ideas should carry over to whatever database you want.
2. Some kind of API that interacts with our database to serve to our frontend.
3. Security and authentication. This is essential. We need our web app to be secure, so we need proper user authentication and security.
4. A frontend so that our users can interact with our web app.

Ok, so now that we know what we need, let's see how we can separate these features into separate services.

Each one of these features will run in their **own separate container** on their own port. Spinning up containers is simple, so all we really need to do is create Dockerfiles so we can create images of our services.

The Dockerfiles

The API:

```
FROM node:14.9.0

WORKDIR /app

COPY package.json /app

RUN npm install

COPY . .

EXPOSE 3000

ENV NODE_ENV production

CMD ["npm", "start"]
```

See, simple. This is a basic Dockerfile that will tell our container to do exactly what we need it to do. We specify the node version we're using on our current machine. We copy the package.json in our working directory and then install our dependencies. Then we expose the port this container will run on and set an environment variable so that our app knows it's in production when the container is running (this is a good practice to use).

Our authentication service:

```
FROM node:14.9.0

WORKDIR /app

COPY package.json /app

RUN npm install

COPY . .

EXPOSE 3004
```

Pretty much verbatim to our API service. The only difference is the port they're running on. Again, you can run your authentication service with an API service together as a combined server file, but we're splitting it up so we need them to run on separate ports.

The frontend

```
# stage1 - build react app first
FROM node:14.9.0 as build

WORKDIR /app

ENV PATH /app/node_modules/.bin:$PATH

COPY ./package.json /app/

RUN npm install --silent

COPY . /app

RUN npm run build

# stage 2 - build the final image and copy the react build files
FROM nginx:1.19.1

COPY --from=build /app/build /usr/share/nginx/html

# new

COPY ./nginx/nginx.conf /etc/nginx/conf.d/default.conf

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

So, this one is going to be a little different from our others. We're going to be serving our React app in a container. Since we're looking for a production build, react will build a static version of our app and store them as static files. So, we need Nginx to proxy our

Also, please note the locations of the Nginx files. This is going to depend on where you've placed your files so bear that in mind if you copy and paste this.

Building the Images

Building the images is now easier than making a pie.

Yes, a pie.

All we need to do is run the basic command of building our images out of our newly made containers. To do this, cd into the directory of the Dockerfile and run:

```
docker build -t <app_name>:<version>
```

This will build an image out of your source code from the Dockerfile. If it's the first time you run this command, you may notice that it takes a while. This is because of all the dependencies it has to install. The cool part about docker is that it caches these dependencies so that if you make a tweak to your source code and then build it again, you'll notice it will take half the time.

Finally, a good idea, if you're putting this all into production, is to push your images to Docker Hub or some docker image hosting provider. Docker Hub is simple to use, check this out if you haven't used it before.

Orchestrating our containers

Now that we've built all of our containers, we need a way so that our they can work and talk to each other. There are a plethora of ways to orchestrate containers so they can do precisely this, but perhaps the simplest way is to use docker-compose.

To quote the official Docker 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."

We're going to be writing a docker-compose.yml file that allows us to orchestrate our images together.

Here's an example:

```
version: '3.7'
services:
    front-main:
        image: nishgowda/front-main:1.0
        volumes:
            - ./nginx/nginx.conf:/etc/nginx/nginx.conf

        depends_on:
            - api
            - auth
        ports:
            - "80:80"

    db:
        image: postgres
        environment:
            POSTGRES_PASSWORD: pass
            POSTGRES_USER: root
            POSTGRES_DB: notes
        volumes:
            - ./pgdata:/var/lib/postgresql/data
        ports:
            - "5432:5432"

    api:
        image: nishgowda/notes-api:1.0
        depends_on:
            - db
        ports:
            - "3000:3000"

    auth:
        image: nishgowda/auth:1.0
        depends_on:
            - db
        ports:
            - "3004:3004"
```

Ok, so that's a lot. Let's break it down.

Here we just specify the version of docker-compose we're using and telling our compose file we're going to be defining our services below.

```
front-main:
        image: nishgowda/front-main:1.0
        volumes:
            - ./nginx/nginx.conf:/etc/nginx/nginx.conf

        depends_on:
            - api
            - auth
        ports:
            - "80:80"
```

Aha! Now we're getting to the juicy parts. So, here we define a service called 'front-main' that uses the docker image that serves our frontend. Notice, though, that we've specified volumes. This allows our container to work with specific files from our machine. Since our frontend image depends on an nginx.conf file, we have to declare it as a volume. Now, we also specify what this service depends on. Since our website is going to be served on nginx, we need to add our other services so they can communicate. Finally, the port mapping. We're going to use port 80 so that people can just visit the clean URL of our website and no one has to visit some random port.

```
db:
        image: postgres
        environment:
            POSTGRES_PASSWORD: pass
            POSTGRES_USER: root
            POSTGRES_DB: notes
        volumes:
            - ./pgdata:/var/lib/postgresql/data
        ports:
            - "5432:5432"
```

available, so we can just specify that image. Then we specify the environment this container is going to be using, this is just a way for us to tell postgres what the password, user, and database is going to be. Next, we add our volumes. Here, we specify a directory where postgres will write our data into. Finally, the port. Postgres usually runs on 5432, so that's just what I'm using here.

Aside:

If you have certain data on your local postgres server that you want to copy over onto your postgres container. Docker has a lot of cool ways to interact directly with containers, so if you need some help with that, I recommend checking out this video.

```
api:
        image: nishgowda/notes-api:1.0
        depends_on:
            - db
        ports:
            - "3000:3000"
```

Here's our API service. Nothing too fancy. We've specified the image to use and set the ports, but something different. If you look, you can see that we've noted the service depends on our database service. Since our API needs to connect to our database, we have to tell it to use the database specifications of our postgres container.

Now, in development, you probably set the configurations of your database in your API source code so it connects to the one running on your local machine. Since we're setting up a *production* environment, you're going to have to alter your source code so that the database client connects to the container. Just match the configurations to the ones you set for the container and **specify the host** as ***db*** (or whatever you named the database service).

```
auth:
        image: nishgowda/auth:1.0
        depends_on:
            - db
```

Finally, our auth service. The explanation of this is pretty much that same for our API service, all that we're changing is the image and the ports.

Another aside:

You might wonder how we're going to connect our auth service with our API. Well again, this isn't exactly a tutorial on how to code it up, but I will lie out a general idea of how you might do it.

First, I would set up the authentication service so that once the user logs in, we can issue some kind of cookie that has the user's credentials onto the user's browser. This way, on our API file, we just have to check if the cookie exists and then decode it to serialize the user.

You can do this by using JSON web tokens or even with session authentication, just make sure you follow some good practices when using cookies.

The long-awaited Nginx file

Ok, time for the grand reveal. Let's look at how our nginx.conf file is going to look like.

```
server{
        listen 80;
        server_name localhost;


        location / {
            root   /usr/share/nginx/html;
            index  index.html index.htm;
            try_files $uri $uri/ /index.html;
        }
        location /api/{
            proxy_pass "http://api:3000";
            proxy_set_header    X-Forwarded-For $remote_addr;
        }
        location /auth {
            proxy_pass "http://auth:3004";
            proxy_set_header    X-Forwarded-For $remote_addr;
        }
        #error_page  404              /404.html;
```

```
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   /usr/share/nginx/html;
        }
    }
```

So you might notice two things that look weird. One, the default location, and the error page.

```
location / {
            root   /usr/share/nginx/html;
            index  index.html index.htm;
            try_files $uri $uri/ /index.html;
        }

        ...
 error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   /usr/share/nginx/html;
        }
```

This will allow nginx to serve our static frontend files for the default location and error pages, respectively. Make sure you have this **before** you build your image for your frontend.

Second, our API and auth locations.

```
location /api/{
            proxy_pass "http://api:3000";
            proxy_set_header    X-Forwarded-For $remote_addr;
        }
        location /auth {
            proxy_pass "http://auth:3004";
            proxy_set_header    X-Forwarded-For $remote_addr;
        }
```

So, most of this will probably seem normal to you if you've used nginx before. The only difference that you might notice is that we've specified our service defined in our

The grand finale

Ok, so if you've followed all the steps so far you should be ready to spin up your web app with docker-compose. Just run:

```
docker-compose up
```

And it should all start! Note you can also add a '-d' flag after the up command to run it in detached mode.

So that's it! Check out my GitHub, Website, or LinkedIn and let me know if you have questions or want more explanations about something.

Docker Microservices Docker Compose Nginx Microservice Architecture