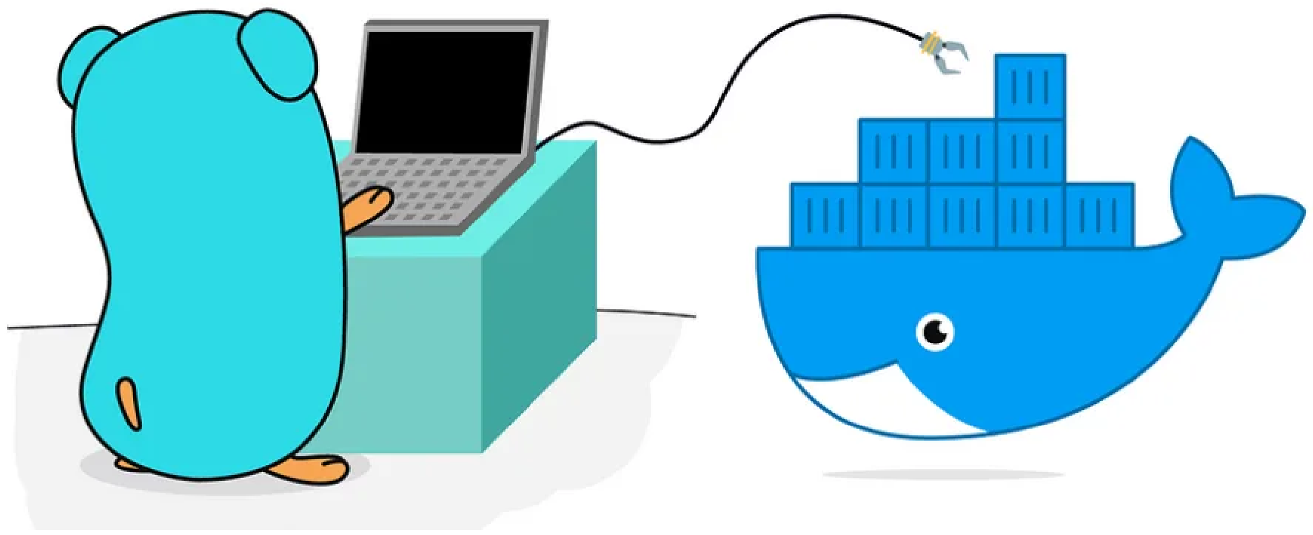


Alex Awesome
Posted on Mar 22 • Updated on Mar 24

❤️ 6

Run a Golang, Nginx, and React App in Docker

#tutorial #docker #go #react

Did you try to run your Golang, Nginx, and React apps locally and fight compatibility issues?
You need docker to save time. If you are very brave you can even use this scheme in production. But you will be judged by mature programmers with long beards.
In this article, we'll guide you through the process of running your Golang, Nginx, and React app in a Docker container, so you can avoid the headache of managing multiple environments.
I guess you know about docker if you can find this article.
Don't worry, you don't need to be a Docker expert or a rocket scientist to follow along.

Let's get started!

Setting Up the Environment:

Install the docker if you haven't done it yet.

We also will use Docker Compose. You have to have it. Don't be scared it's very easy to start using.

Next, create a new directory on your machine to hold your app files. Let's call it «docker-me-up-baby».

```
mkdir docker-me-up-baby
cd docker-me-up-baby
```

Now, create «docker-compose.yml». This file will define the services that make up your app and their configurations. Here's an example docker-compose.yml file for our simplest application:

```
version: '3'
services:
  web:
    build: ./nginx
    ports:
      - "80:80"
    depends_on:
      - api
  api:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8080:8080"
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    depends_on:
      - api
```

What the hell is going on here 👆?

This docker-compose.yml file does a few things:

- It defines four services: web (for Nginx), api (for Golang), db (for PostgreSQL), and frontend (for React).
- It specifies the build context for the api and frontend services, which tells Docker Compose where to find their Dockerfiles.
- It exposes port 80 on the web service to port 80 on the host machine.
- It exposes port 3000 on the frontend service to port 3000 on the host machine.

That's it for setting up the environment with Docker Compose! Now, you can build and run your app with a single command. Docker Compose takes care of building the images, creating the containers, and linking them together. All you need to do is run:

```
docker-compose up
```

I guess it won't work right now but believe me it's the right command.

Docker Compose will start all the services defined in the docker-compose.yml file, and you will access your app by visiting http://localhost in your browser.

Building and dockerizing the simplest Golang API:

Let's implement our rocket-science API

```go
// main.go
package main

import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "Hello, World!")
    })

    http.ListenAndServe(":8080", nil)
}
```

It's great, isn't it?!

Lets dockerize this one (I'm assuming that Golang part is in the root directory of our project):

```
# Official Golang image (You shouldn't use the `latest` version in production but I'm a b
FROM golang:latest

# Working directory
WORKDIR /app

# Copy everything at /app
COPY . /app

# Build the go app
RUN go build -o main .

# Expose port
EXPOSE 8080

# Define the command to run the app
CMD ["./main"]
```

You can understand what is happening in this file by reading the comments.
Let's repeat one more time:

- Dockerfile gets the official Golang image.
- Sets the working directory to /app.
- Copies the contents of the current directory into the /app directory in the container.
- Builds the go app using the `go build` command.
- Exposes port `8080`.
- Defines the command to run the app (in this case, `./main`).

To build the Docker image for this Golang API, we cat run the following command in the app's directory (but we won't):

```
docker-compose build api
```

Actually, we don't need to do it because we will build everything everywhere all at once later.

We will do the same with running our container. I just have to mention that you can use this command for running the container

```
docker-compose up api
```

But we don't need it right now.

If you are so impatient, you can run `build` and `run` commands above. You can be sure that everything is ok if you see the "Hello world!" console. You can also access the API by visiting http://localhost:8080.

That's it for building the Docker image for the simplest Golang API!

Let's go ahead.

Set up the Nginx server (in Docker of course)

Let's put Nginx files to a new directory named `nginx` inside the app's directory. Inside the `nginx` directory, create a new file named `nginx.conf` with the following configuration:

```
server {
    listen 80;
    server_name localhost;

    location / {
        proxy_pass http://api:8080;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

This configuration sets up an Nginx server to listen on port 80 and forward all requests to the Golang API running on port 8080. Why is "api" in `http://api:8080`? It's name of the Golang API service defined in the docker-compose.yml file.

To build the Nginx image in a Docker container, create a new file named "Dockerfile" inside the "nginx" directory:

```
# Official Nginx image (Yes, in this article I always use the `latest`. Kill me!)
FROM nginx:latest

# Copy Nginx configuration file to the container
COPY nginx.conf /etc/nginx/conf.d

# Expose port 80
EXPOSE 80
```

One more time:

- This Dockerfile gets the official Nginx runtime image as the parent image.
- Copies the Nginx configuration file to the container.
- Exposes port 80.

For build and run you can use the same commands as for the Golang API container but with name of service `web`.

```
docker-compose build web
docker-compose up web
```

If you run golang and nginx services by `docker-compose up API web` you can check the `http://localhost` in the browser.

Create the React app.

This article is not about building a React application. We can use something simple (From root project directory)

```
npx create-react-app frontend
cd frontend
```

Add the API integration:

Add the stupidest API integration in all over the world.
Create a new file named "api.js" in the "frontend/src" directory:

```
const api = {
  async getHello() {
    const response = await fetch('/api');
    const data = await response.json();
    return data.message;
  }
}

export default api;
```

For development, we will use the Docker.

`frontend/Dockerfile`:

```
# Use a Node.js image
FROM node:latest

# Set the working directory
WORKDIR /app

# Copy the package.json and package-lock.json files
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy everything to the container
COPY . .

# Expose port
EXPOSE 3000
```

This Dockerfile

- Gets the Node.js image.
- Sets the working directory to /app.
- Copies the package.json and package-lock.json files to the container.
- Installs dependencies

```
docker-compose build frontend
docker-compose up frontend
```

You can see something at `http://localhost:3000`

Let's simulate production build

It's just a demonstration. It's the worst way for real-life applications. Anyway, build the build

```
npm run build
```

Add the React build files to your Nginx configuration.

To include the React build files in your Nginx server, let's make `nginx.conf` file like this:

```
server {
    listen 80;
    server_name localhost;

    location / {
        root /app/frontend/build;
        try_files $uri /index.html;
    }

    location /api {
        proxy_pass http://api:8080;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

This configuration sets up an Nginx server to serve the React app from the `build` directory and forwards all API requests to the Golang API running on port 8080.

And finally, let's add a little correction to the docker-compose:

```
version: '3'
services:
  web:
    build: ./nginx
    ports:
      - "80:80"
    depends_on:
      - api
    volumes:
      - frontend/build:/app/frontend/build
  api:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8080:8080"
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    depends_on:
      - api
```

Got it? We added `volumes`. It will mount our `frontend/build` directory to the Nginx container.

All together

Now that you've built the Docker images for your Golang API, Nginx server, and React app, it's time to run the Docker container and test your app.

Start the Docker container:

```
docker-compose up
```

You can use

```
docker-compose up -d
```

to run it in the background

Test the app:

To test your app, visit http://localhost in your browser. You should see your React app being served by the Nginx server, with API requests being forwarded to the Golang API.

You can also test your Golang API by visiting http://localhost/api in your browser. You should see a JSON response with a "message" field containing the text "Hello, world!".

Stop it!

Stop the Docker container:
To stop the Docker container, press `Ctrl + C` in your terminal or run the following command if you use `-d` option:

```
docker-compose down
```

This will stop and remove all the Docker containers for your app.

That's it! You've successfully built and run a Docker container for your Golang, Nginx, and React app. Docker Compose makes it easy to manage all the services in your app and ensure they work together correctly.

--

If you want to learn more about Docker and app deployment, here are some additional resources you can check out:

- Docker's official documentation: Docker's documentation provides detailed information on how to use Docker and Docker Compose to build and run containers. It includes guides, tutorials, and reference materials for beginners and advanced users.
- Docker Hub: Docker Hub is a public registry of Docker images, where you can find images for various services and software. You can also use Docker Hub to store and share your own images.
- Docker Stack is a tool provided by Docker that allows you to deploy a multi-container Docker application on a Docker Swarm cluster. It enables you to define your application's services, networks, and volumes in a single file called a "Compose file" and deploy the entire stack to the swarm.
- Kubernetes: Kubernetes is an open-source container orchestration system that automates the deployment, scaling, and management of containerized applications. It can be used to deploy and manage Docker containers in a production environment.

I don't post links on purpose. Technology development is very fast. Find actual ones on Google.

-

Follow me on Telegram, and you can read all my articles https://t.me/awesomeprog.

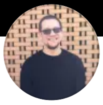
Top comments (0)

Code of Conduct • Report abuse

DEV Community

50 CLI Tools You Can't Live Without

>> Check out this classic DEV post <<



Alex Awesome

Hi! You are awesome.

LOCATION
Barcelona

JOINED
May 19, 2021

More from Alex Awesome

Telegram OAuth Authorization for Your Site

#tutorial #go #php #node

Golang Unit Testing with examples

#go #testing #programming

DEV Community

🌚 **Life is too short to browse without dark mode**