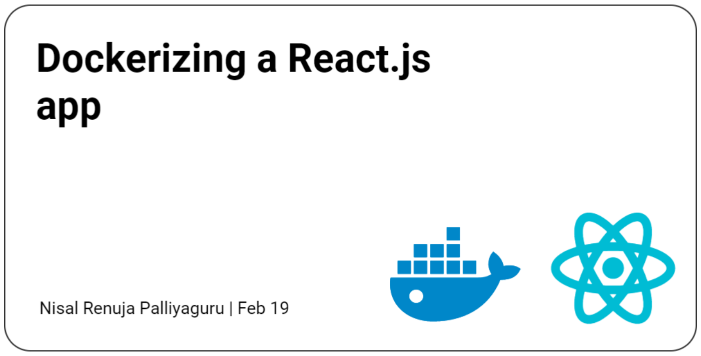
Published in SLIIT FOSS Community · Follow

Upgrade

Search

Nisal Renuja Palliyaguru
Feb 19 · 6 min read · Listen

Dockerizing a React.js app



Today, We will dockerize a React application. We will set up Docker with auto-reloading for development setup and optimized multistage docker build for production deployment. We can even dockerize Next.js or Gatsby Static builds with the same process.

There are many advantages of using Docker. Right now Docker is the defacto standard of containerizing applications. It is easy to build, package, share, and ship applications with Docker. As Docker images are portable it is easy to deploy the application to any modern cloud provider.

Initialize React application

Let's start by creating a React application. You can use your existing React project. For this blog post, I am creating a new React application with `create-react-app`.

```
$ npx create-react-app react-docker
```

<folder-name>

Sync With Github:1. Change directory to the proje

Here I created a new React app named `react-docker`. Let's verify the app by running the `npm start` command inside the project directory.

```
$ npm start
```

It will start the app and we can verify by visiting http://localhost:3000 in the browser. The application should be running.

Write Dockerfile.

Now let's create a Docker image for the React application. We need a Dockerfile to create Docker images. Let's create a file named `Dockerfile` in the root directory of the React application.

Dockerfile (-> Inside the Dockerfile file, paste the following lines)

Nisal Renuja Palliyaguru
7 Followers

Developer 💻 | Undergraduate 👨‍🎓 | Tech Enthusiast

Follow

Related

Using express-react-views for templating engine in express app.

How to fetch data from an API in ReactJS ? Application Programmi…

Live Reload with Docker and React
Getting live reload to wo…

Setting up a simple standard Knex/Express RESTful API with…

Help Status Writers Blog Careers Privacy Terms About Knowable

```
~~COPY package.json ./~~
~~COPY yarn.lock ./~~
~~RUN yarn install --frozen-lockfile~~
~~COPY . .~~
~~EXPOSE 3000~~
~~CMD ["npm", "start"]~~
```

FROM node:14-alpineWORKDIR /appCOPY package.json ./COPY

Here We are using node v14 alpine as the base image to build and run the application. We are running the `npm start` command default command which will run the React development server. We also need the `.dockerignore` file which will prevent `node_modules` other unwanted files to get copied in the Docker image.

.dockerignore

```
node_modules
npm-debug.log
Dockerfile
.dockerignore
```

Let's build the Docker image from the Dockerfile by running the `docker build` command. Here we are tagging it with the name `react-docker`.

Note: Installer the docker command m

```
$ docker build -t react-docker .
```

<folder-name>

Note2: With the Docker installed in the m

After building the docker images we can verify the image by running the `docker images` command. We can see an image with the name `react-docker` is created.

```
$ docker images
```

We can run the Docker image with the `docker run` command. Here we are mapping the system port 3000 to Docker container port 3000. We can verify if the application is running or not by visiting http://localhost:3000.

```
$ docker run -p 3000:3000 react-docker
```

<folder-name>

Add Docker Compose

The React application is working fine inside the docker container, but we need to build and run the docker container every time we make any changes in the source files as auto-reloading is not working with this setup. We need to mount the local `src` folder to the docker container `src` folder, so every time we make any change inside the `src` folder, it will auto-reload the page on any code changes.

We will add the `docker-compose.yml` file to the root of the project to mount the local `src` folder to the `/app/src` folder of the container.

docker-compose.yml

version: "3"services: app: build:

```
~~version: "3"~~
~~services:~~
  app:
```

```
  ~~app:~~
    ~~build:~~
      ~~context: .~~
      ~~dockerfile: Dockerfile~~
    ~~volumes:~~
      ~~- ./src:/app/src~~
    ~~ports:~~
      ~~- "3000:3000"~~
```

Run the `docker-compose up` command to start the container. The React development server will be running inside the container and will be watching the `src` folder.

```
$ docker-compose up
```

We can't ship this docker image to the production as it is not optimized and runs a development server inside. Let's rename the `Dockerfile` as `Dockerfile.dev` and update the `docker-compose.yaml` file to use the `Dockerfile.dev` file. We will use docker-compose and the `Dockerfile.dev` file only for development. We will create a new Dockerfile for the production build.

```
$ mv Dockerfile Dockerfile.dev
```

docker-compose.yml

```
version: "3"

services:
  app:
    build:
      context: .
      dockerfile: Dockerfile.dev
    volumes:
      - ./src:/app/src
    ports:
      - "8000:8000"
```

Add Production Dockerfile

Let's first verify the React application production config by running the `yarn build` command to build the app for production.

```
~~$ yarn build~~
```

$ npm run build

We can verify the production build by running it locally. I am using `serve` to serve the `build` folder files.

```
$ npx serve -s build
```

After verifying the server locally we can create a new Dockerfile for the production build. We will be using multi-stage builds to create the docker image. The first stage is to build the production files and the second stage is to serve them.

Dockerfile

FROM node:14-alpine AS builderWORKDIR /appCOPY package.

```
~~FROM node:14-alpine AS builder~~
~~WORKDIR /app~~
~~COPY package.json ./~~
```

```
~~COPY yarn.lock ./~~
~~RUN yarn install --frozen-lockfile~~
~~COPY . .~~
~~RUN yarn build~~

~~FROM nginx:1.19-alpine AS server~~
~~COPY --from=builder ./app/build /usr/share/nginx/html~~
```

The `builder` stage is nearly the same as the previous Dockerfile. Instead of running the `npm start` command here, we are running the `yarn build` command (npm run build) to build the production files.

We will use `Nginx` to serve the files. It will create a very lightweight image. From the builder stage, we need to copy the files of the `build` folder to the `/usr/share/nginx/html` folder. Nginx Docker image uses this folder to serve the contents. Nginx Docker image will use the port `80` to serve the files and auto expose that port.

Let's build the image again by running the `docker build` command and verify if the image is built or not by running the `docker images` command.

```
$ docker build -t react-docker .
                  <folder-name>

$ docker images
```

The size of the production docker image will be very less in comparison to the development one. Let's run the docker image with the `docker run` command. Here we are mapping the host `3000` port with the container's port `80`

Note: From ~400mb to ~20mb

```
docker run -p 3000:80 react-docker
```

The application should be running fine on http://localhost:3000. Now let's verify if the client-side routing is working fine or not. For that, we need to install the `react-router-dom` to the application.

```
~~$ yarn add react-router-dom~~
```

$ npm install react-router-dom

We also need to add a few routes and links to verify. I just copied the example from the react-router website to test.

```
import React from "react";
import { BrowserRouter as Router, Switch, Route, Link } from "react-
router-dom";                      Routes

export default function App() {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/about">About</Link>
            </li>
            <li>
              <Link to="/users">Users</Link>
            </li>
          </ul>
        </nav>
        <~~Switch~~>
          ~~<Route path="/about">~~
            ~~<About />~~

          ~~</Route>~~
          ~~<Route path="/users">~~
            ~~<Users />~~
          ~~</Route>~~
```

<Routes> <Route path="/about" element={<About />} /> <Ro

```
          ~~<Route path="/">~~
            ~~<Home />~~
          ~~</Route>~~
        <~~/Switch~~>
      </div>
    </Router>
  );
}

function Home() {
  return <h2>Home</h2>;
}

function About() {
  return <h2>About</h2>;
}

function Users() {
  return <h2>Users</h2>;
}
```

Let's verify the local setup by running the development server and **visiting the web page and clicking on every link and refreshing the pages.**

```
$ npm start
```

The application should be working fine on the local development server. Now try the same thing with docker-compose. First, we need to build the image again as auto-reload works only with the `src` folder as we only mount that. For changes outside the `src` folder, we need to build the image again with the `docker-compose build` command.

```
$ docker-compose build
$ docker-compose up
```

Now Let's try the same thing with the production docker build. First, we need to build the docker image and run the image again.

```
docker build -t react-docker .
docker run -p 3000:80 react-docker
```

Note: No errors.No Nginx config

Accessing the pages other than the index directly should throw a 404 error. The React application here is a single-page application. Thus the routing is happing on the client-side with JavaScript and when we hit any route it first hits the Nginx server and tries to find the file there and when it was unable to find the fine it throws the 404 error.

We need to pass a custom Nginx configuration to the docker image. We will create an `etc` folder in the project's root directory and create an `nginx.conf` file there.

etc/nginx.conf

```
server {
    listen   80;
    listen   [::]:80 default ipv6only=on;

    root /usr/share/nginx/html;
    index index.html;

    server_tokens  off;
    server_name _;

    gzip on;
    gzip_disable "msie6";


    gzip_vary on;
    gzip_proxied any;
    gzip_comp_level 6;
    gzip_buffers 16 8k;
    gzip_http_version 1.1;
    gzip_min_length 0;
    gzip_types text/plain application/javascript text/css
application/json application/x-javascript text/xml application/xml
application/xml+rss text/javascript application/vnd.ms-fontobject
application/x-font-ttf font/opentype;

    location / {
        try_files $uri /index.html;
    }
}
```

Here we are configuring Nginx to fall back to `/index.html` if it is unable to find the route. We also enable gzip compression for the contents.

We need to copy the custom Nginx configuration file to the `/etc/nginx/conf.d` folder. Ngnix will auto-read all the configurations from that folder.

```
FROM node:14-alpine AS builder
WORKDIR /app
COPY package.json ./
COPY yarn.lock ./
RUN yarn install --frozen-lockfile
COPY . .
RUN yarn build

FROM nginx:1.19-alpine AS server
COPY ./etc/nginx.conf /etc/nginx/conf.d/default.conf
COPY --from=builder ./app/build /usr/share/nginx/html
```

After copying the custom Nginx configuration file we need to build and run the docker image again.

```
$ docker build -t react-docker .
$ docker run -p 3000:80 react-docker
```

Visiting all the routes and refreshing the pages should work fine.

All the source code for this tutorial is available on GitHub.