Member-only story

How to Build PERN Stack for Production

A step by step guide with an example project

Bhargav Bachina · Follow

Published in Bachina Labs

10 min read · May 1, 2022

Listen More

There are so many ways we can build React apps and ship them for production. One way is to build the React app with NodeJS and PostgreSQL as a database. There are four things that make this stack popular and you can write everything in Javascript. The four things are PostgreSQL, React, Express, and NodeJS. This stack can be used for a lot of uses cases in web development.

Developing the application is one part and you need to package it based on your deployment needs once the development part is completed. There are so many ways we can package and ship PERN Stack to production: manual, with webpack, with Gulp, etc we will see all these approaches in detail.

Prerequisites

There are some prerequisites for this post. You need to have a NodeJS installed on your machine and some other tools that are required to complete this project.

- NodeJS
- Express Framework
- PGAdmin
- PostgreSQL

- Postgresapp
- node-postgres
- VSCode
- Postman
- nodemon
- dotenv
- Create React App
- Typescript
- react-bootstrap

NodeJS: As an asynchronous event-driven JavaScript runtime, Node.js is designed to build scalable network applications.

Express Framework: Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.

node-postgres: Non-blocking Postgresql Client for NodeJS

PGAdmin: pgAdmin is an Open Source administration and development platform for PostgreSQL

PostgreSQL: Open Source relational Database

VSCode: The editor we are using for the project. It's open-source and you can download it here.

Postman: Manual testing your APIs

nodemon: To speed up the development

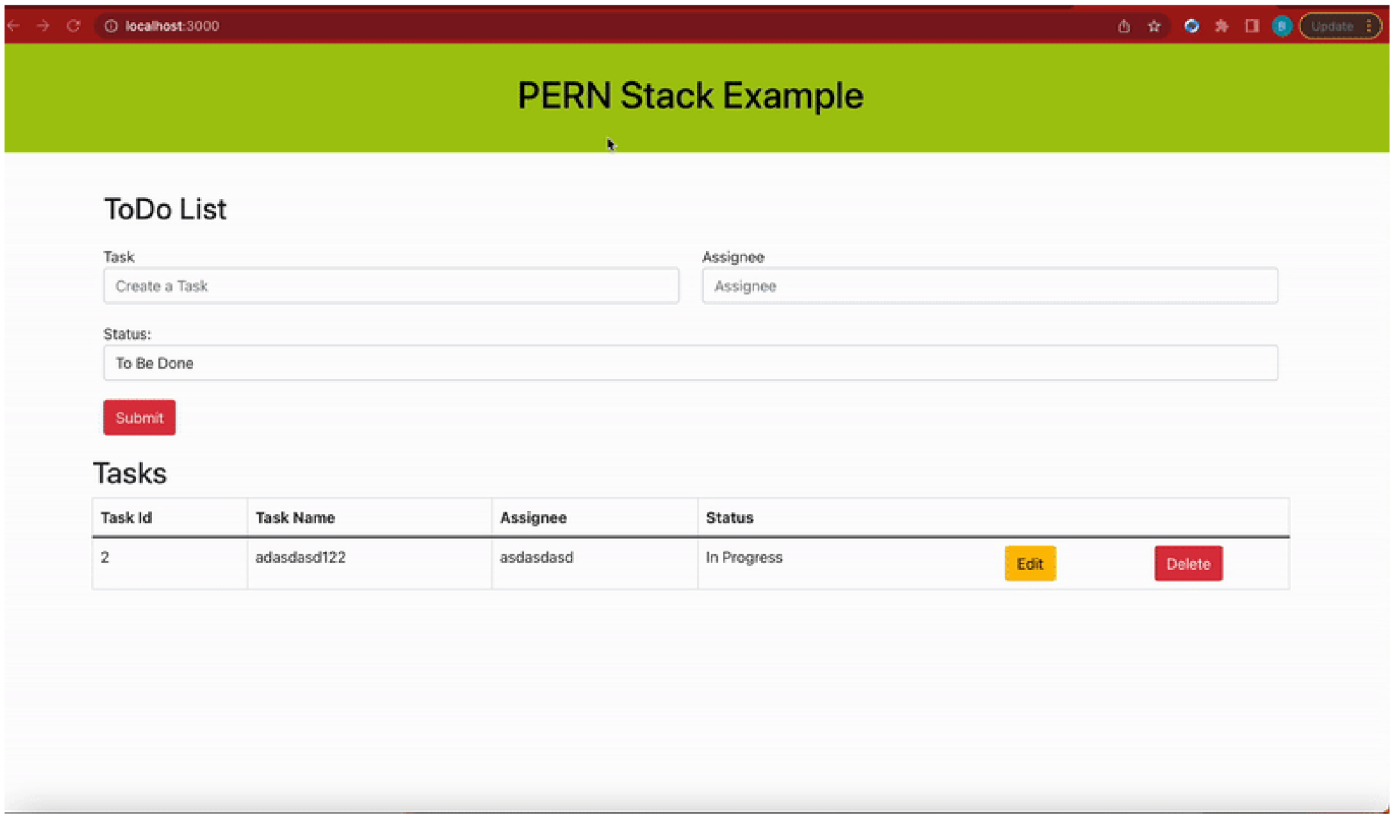
If you are a complete beginner and don't know how to build from scratch, I would recommend going through the below articles. We used these projects from this article

as a basis for this post.

> *How To Get Started With React*
>
> *How To Develop and Build React App With NodeJS*
>
> *How to Build NodeJS REST API with Express and PostgreSQL*
>
> *How to write production-ready Node.js Rest API — Javascript version*

Example Project

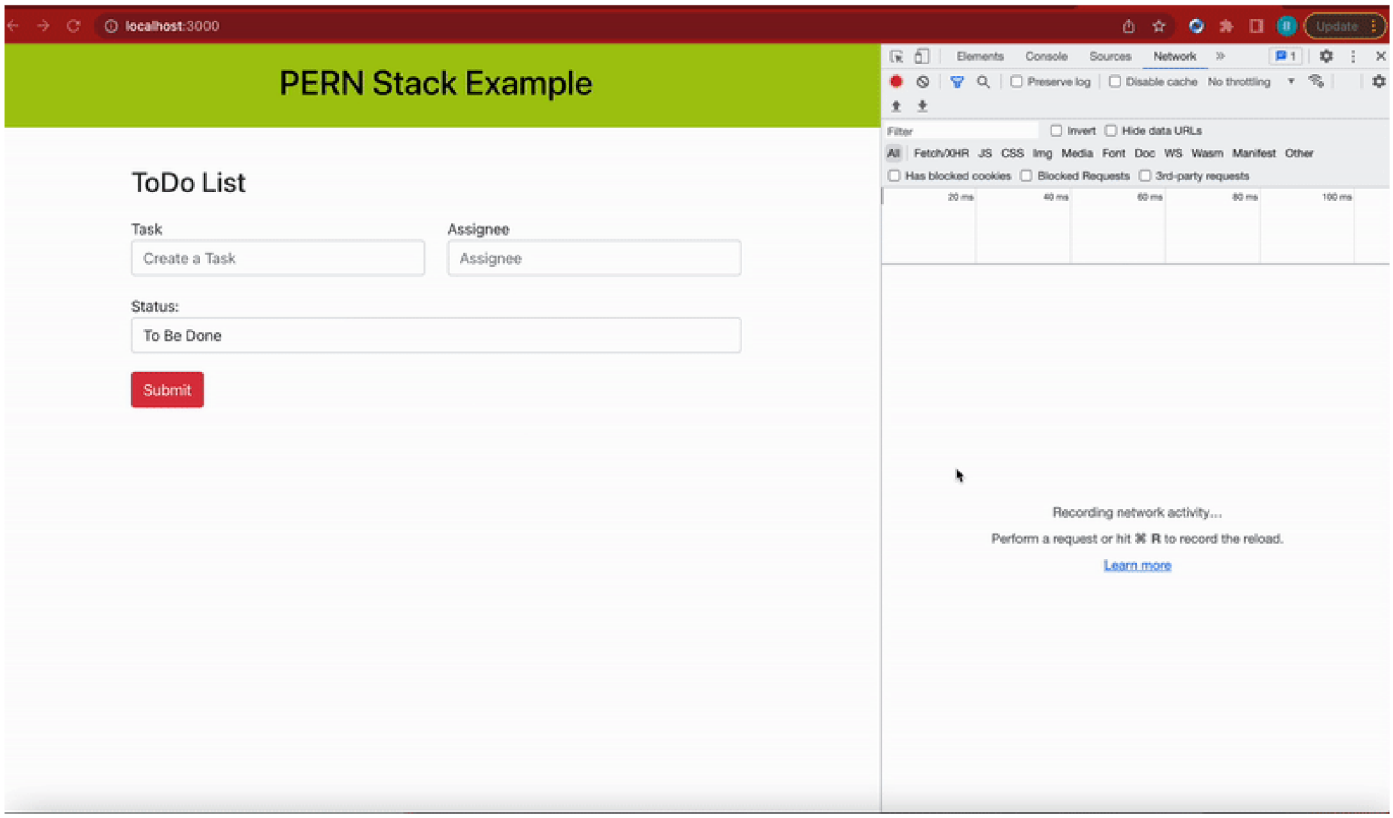
Here is an example of a simple tasks application that creates, retrieves, edits, and deletes tasks. We actually run the API on the NodeJS server and you can use PostgreSQL to save all these tasks.



Example Project

As you add users we are making an API call to the nodejs server to store them and get the same data from the server when we retrieve them. You can see network calls in the

following video.



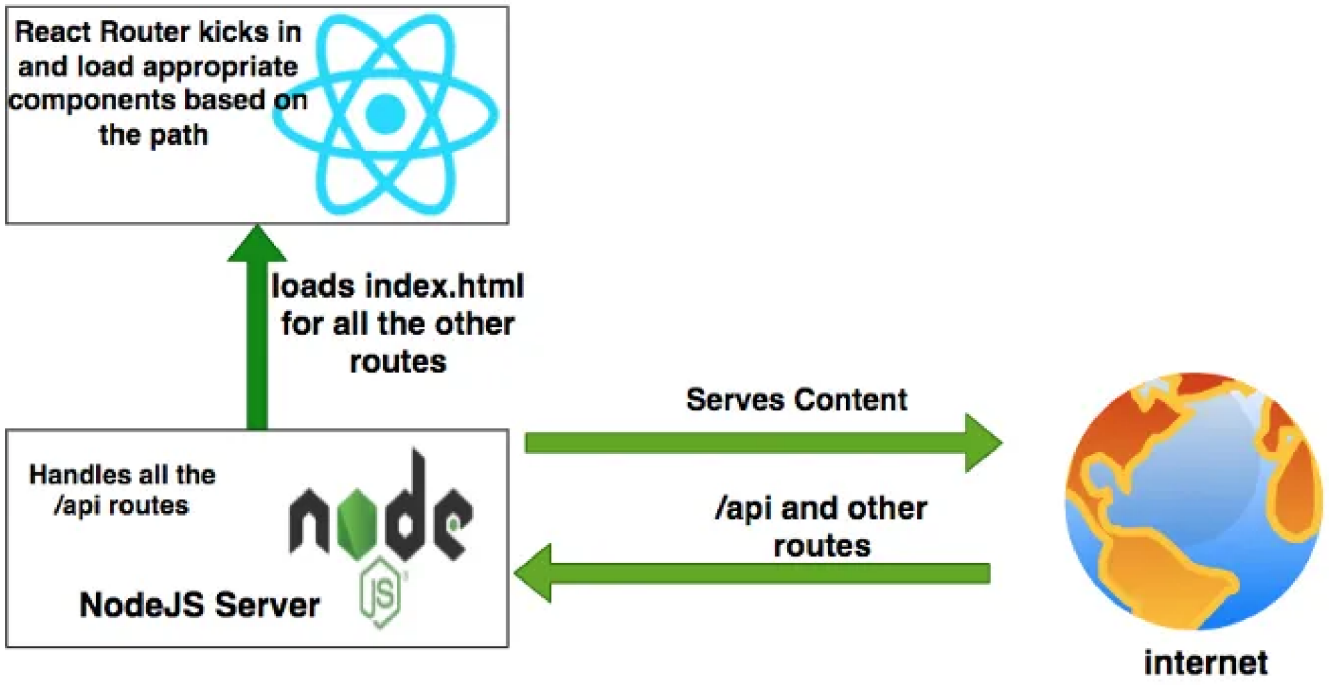
Network Calls

Here is a Github link to this project. You can clone it and run it on your machine.

```
// clone the project
git clone https://github.com/bbachi/pern-stack-example.git

// React Code
cd ui
npm install
npm start

// API code
cd api
npm install
npm run dev
```

PERN Stack Development

As we said earlier, PERN Stack uses four technologies such as PostgreSQL, Express, React, and NodeJS. React is a javascript library for building web apps and it doesn't load itself in the browser. We need some kind of mechanism that loads the **index.html** (single page) of React application with all the dependencies(CSS and js files) in the browser. In this case, we are using node as the webserver which loads React assets and accepts any API calls from the React UI app.



PERN Stack

If you look at the above diagram all the web requests without the **/api** will go to React routing and the React Router kicks in and loads components based on the path. All the paths that contain **/api** will be handled by the Node server itself.

Here is the complete detailed article about the development.

> *How To Develop and Build PERN Stack*

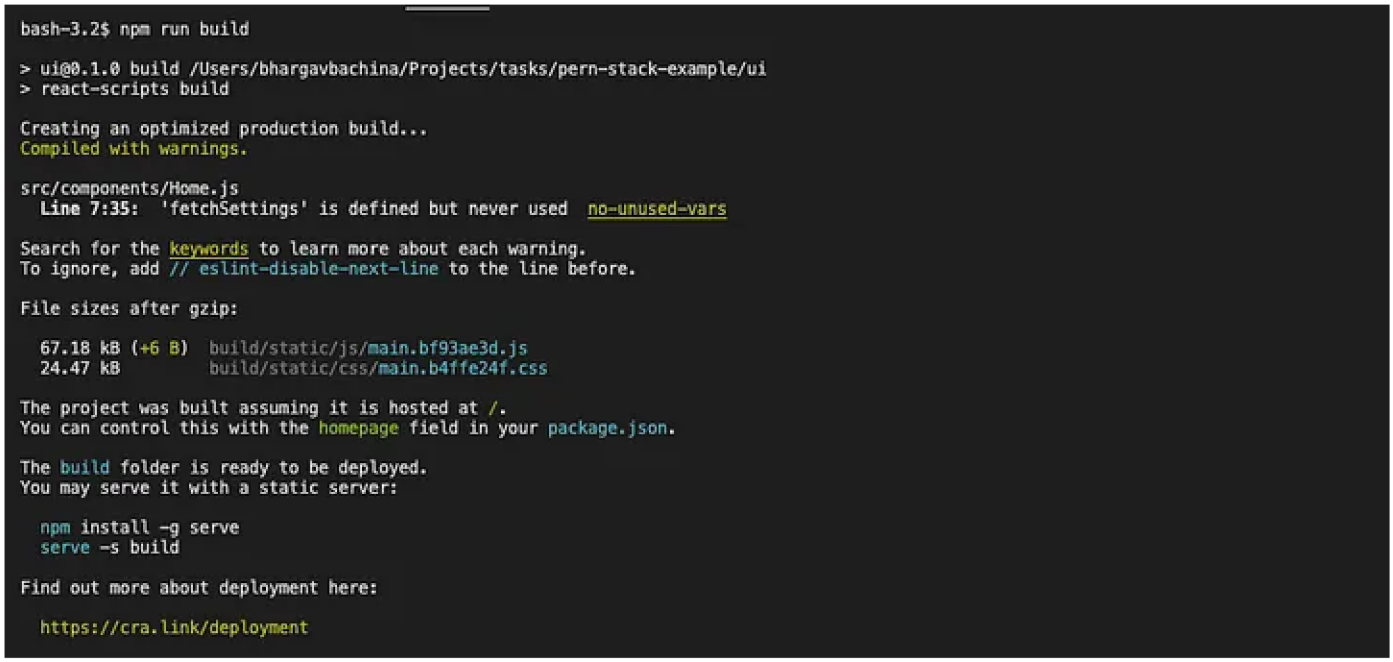
Manual Implementation

In this manual implementation, we build the React app and place the appropriate code into one folder and run or deploy the application. As a first step, we need to build the

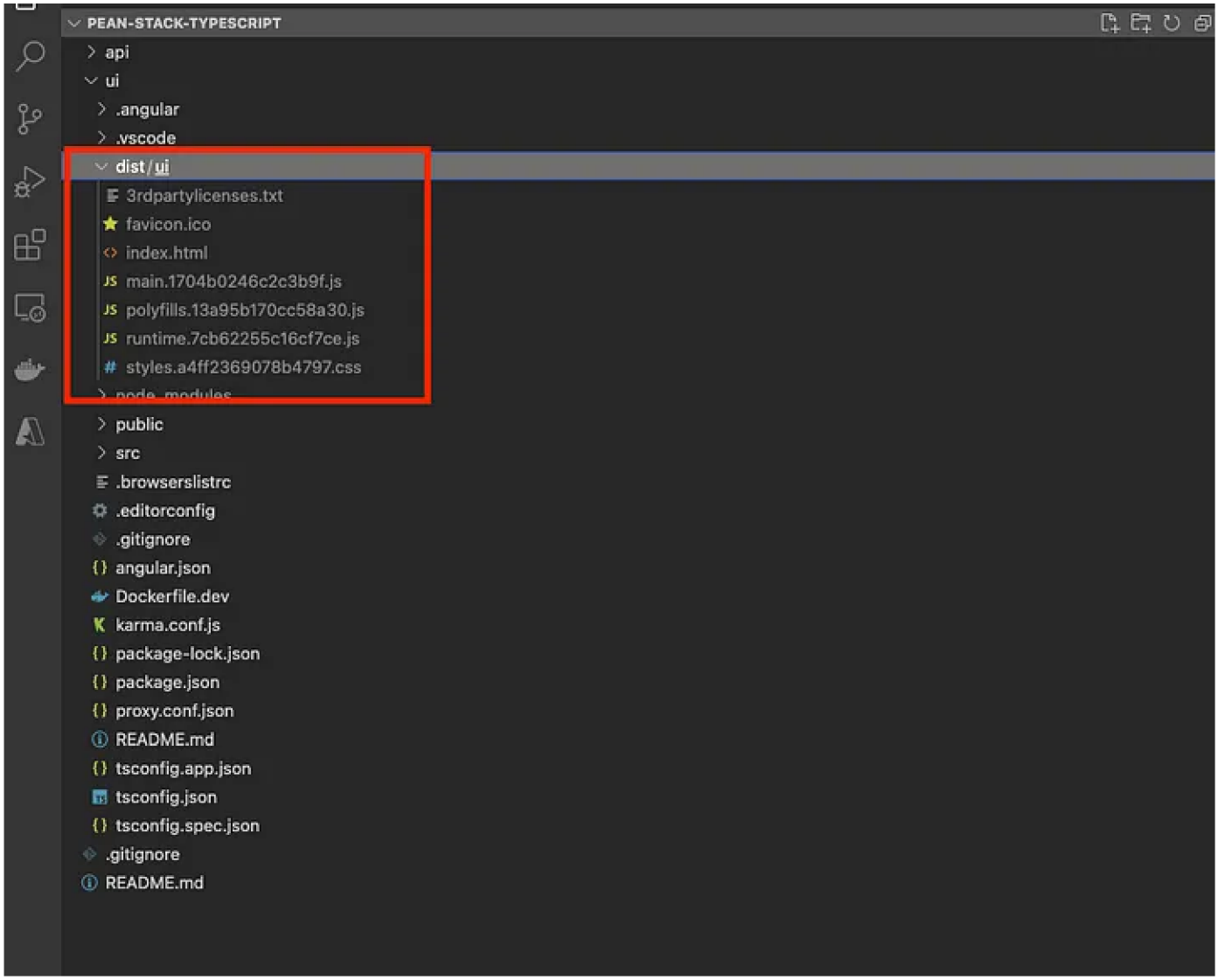
React app and all the static assets and built files are placed into the **build** folder.

```
// change to ui directory
cd ui

// build the app
npm run build
```

All the built and static assets are placed under this folder **build/**

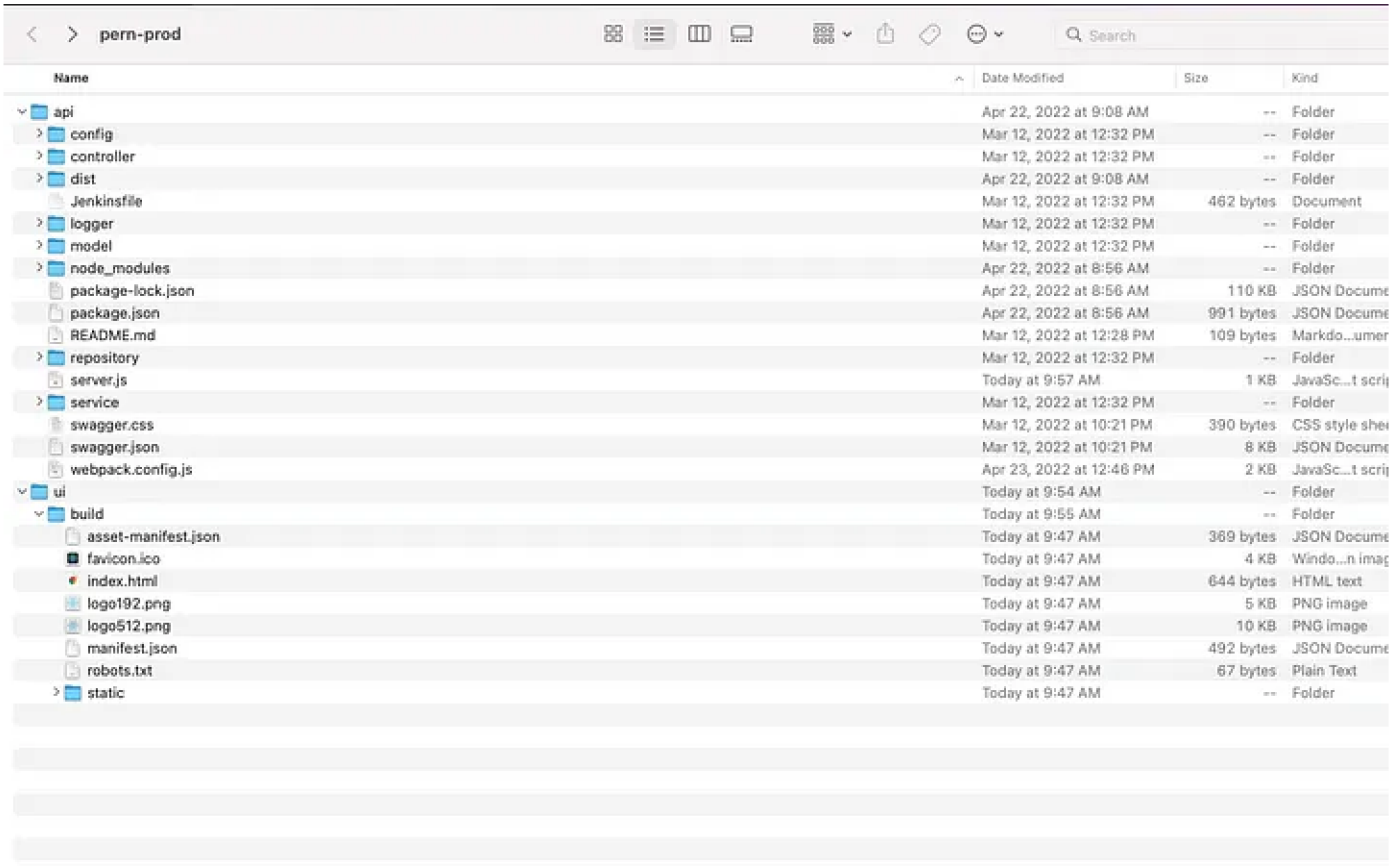


npm run build



React Build

Once you built the application, all you need to do is create a separate folder and place the nodejs-related stuff in that folder. Let's create a folder called **pern-prod** and put the api folder (node server code) and the folder build inside it.



Project Folder

Run the application

Let's run the app by importing the whole folder pern-prod into the VSCode editor and installing the dependencies for the server.

```
//change the directory
cd pern-prod/api

// install dependencies
npm install

// run the app
node server.js
```

You need to make sure two things before you start the application. First, you need to put any environment variables such as PostgreSQL Connection string, PORT, etc. There are some other options as well check the below-detailed article.

> *Reading Environment Variables In NodeJS REST API*

```
// exporting env variables
export PORT=3080
export ENVIRONMENT=production
export
PG_CONNECTION_STR=postgres://pgadmin@webappdemopostgre:Tester@123@weba
ppdemopostgre.postgres.database.azure.com:5432/tasks
```

The second one is that update the UI static assets in the sever.js file as below. Notice the line numbers **17** and **42**.

```
1   const express = require('express');
2   const bodyParser = require('body-parser');
3   require('dotenv').config()
4   const fs = require('fs')
5   const path = require("path");
6
7   const taskController = require('./controller/task.controller')
8
9   const swaggerUi = require('swagger-ui-express');
10  const swaggerDocument = require('./swagger.json');
11  const customCss = fs.readFileSync((process.cwd()+"/swagger.css"), 'utf8');
12
13
14  const app = express();
15  const port = process.env.PORT || 3000;
16
17  app.use(express.static(path.join(__dirname, '../ui/build/')));
18
19  // let express to use this
20  app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerDocument, {customCss}));
21
22  app.use(bodyParser.json());
23
24  app.get('/api/tasks', (req, res) => {
25      taskController.getTasks().then(data => res.json(data));
26  });
27
28  app.post('/api/task', (req, res) => {
29      console.log(req.body);
30      taskController.createTask(req.body.task).then(data => res.json(data));
31  });
32
```

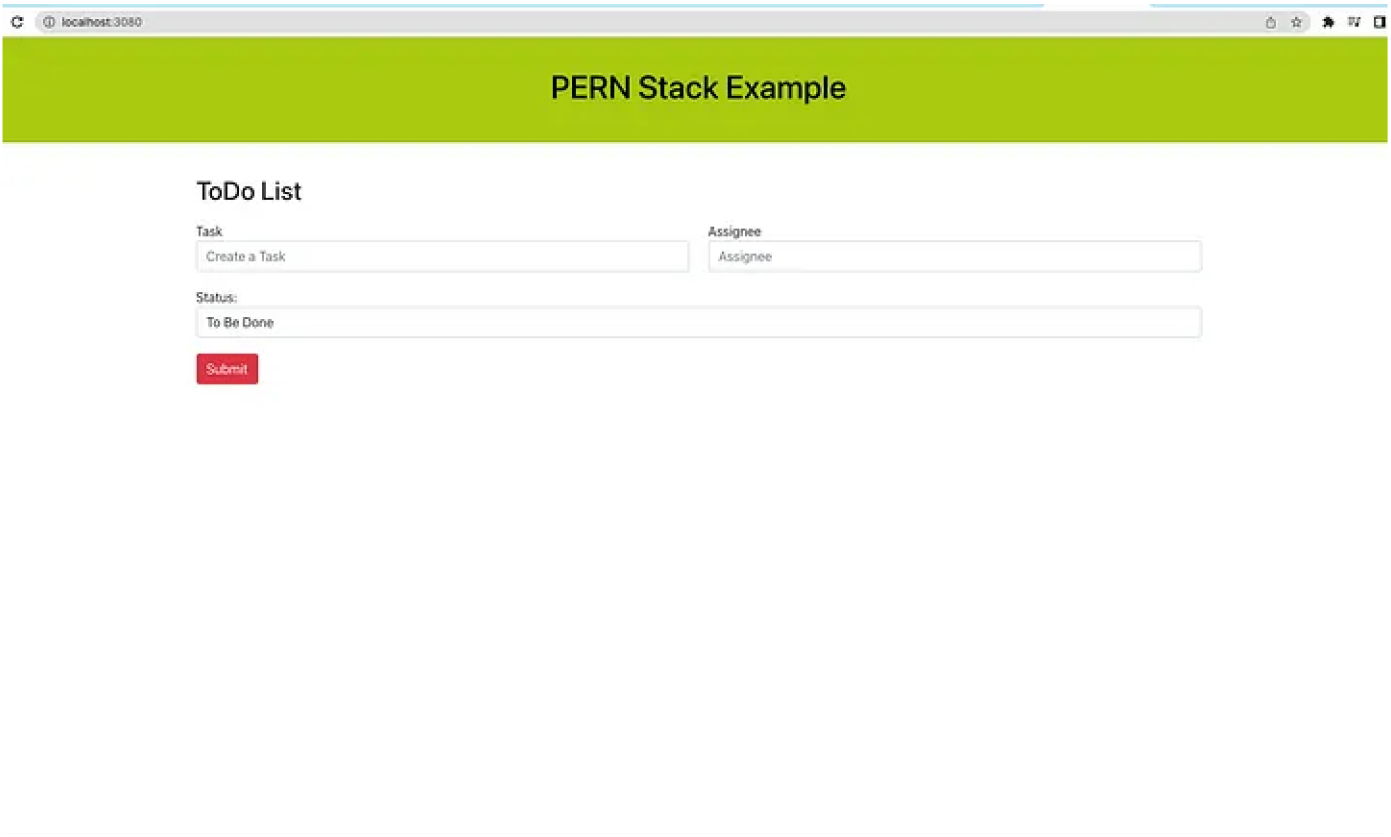
```
33  app.put('/api/task', (req, res) => {
34      taskController.updateTask(req.body.task).then(data => res.json(data));
35  });
36
37  app.delete('/api/task/:id', (req, res) => {
38      taskController.deleteTask(req.params.id).then(data => res.json(data));
39  });
40
41  app.get('/', (req, res) => {
42      res.sendFile(path.join(__dirname, '../ui/build/index.html'));
43  });
44
45
46  app.listen(port, () => {
47      console.log(`Server listening on the port  ${port}`);
48  })
```

server.js hosted with ❤ by GitHub view raw

server.js

server listening on port 3080

The app is running on the server port 3080.



PERN Stack running on port 3080

Disadvantages

All the below steps should be done manually and these are time-consuming tasks.

- All the below steps should be done manually and these are time-consuming tasks.
- We have to build React code manually
- We have to place all the built files of react and API files into a separate folder
- We need to install node dependencies before we run the app
- We have to export all the environment variables

With Webpack

In the above implementation, once we put everything in the folder we need to install dependencies for the nodejs server to run the app. This is an additional step we need to do before running the app.

We can skip this step with the webpack. When we build the React code, the React CLI uses a webpack internally to build and bundle the entire code into a few files. We can use the same for the nodejs server as well.

First, we need to install a webpack globally and in the project as well.

```
// install webpack
npm install -g webpack webpack-cli
npm install webpack webpack-cli --save
```

We need to have a **webpack.config.js** in the **/api** folder since a webpack looks for this file. Here is the file. We have an entry file and output file and it is placed in the root folder. You can define environment variables based on the environment so that you don't have to export all the variables you need to run the application.

```
1  const path = require('path');
2  const webpack = require('webpack');
3
4  const environment = process.env.ENVIRONMENT;
5
6  console.log('environment:::::', environment);
7
8  /*
9
10 HOST=localhost
11 USER=bhargavbachina
12 DB=bhargavbachina
13 DIALECT=postgres
14 PORT=3080
15
16 */
17
18 let ENVIRONMENT_VARIABLES = {
19   'process.env.HOST': JSON.stringify('localhost'),
20   'process.env.USER': JSON.stringify('bhargavbachina'),
21   'process.env.DB': JSON.stringify('bhargavbachina'),
22   'process.env.DIALECT': JSON.stringify('postgres'),
23   'process.env.PORT': JSON.stringify('3080'),
24   'process.env.PG_CONNECTION_STR': JSON.stringify("postgres://bhargavbachina:''@localhost:5432/bha
```

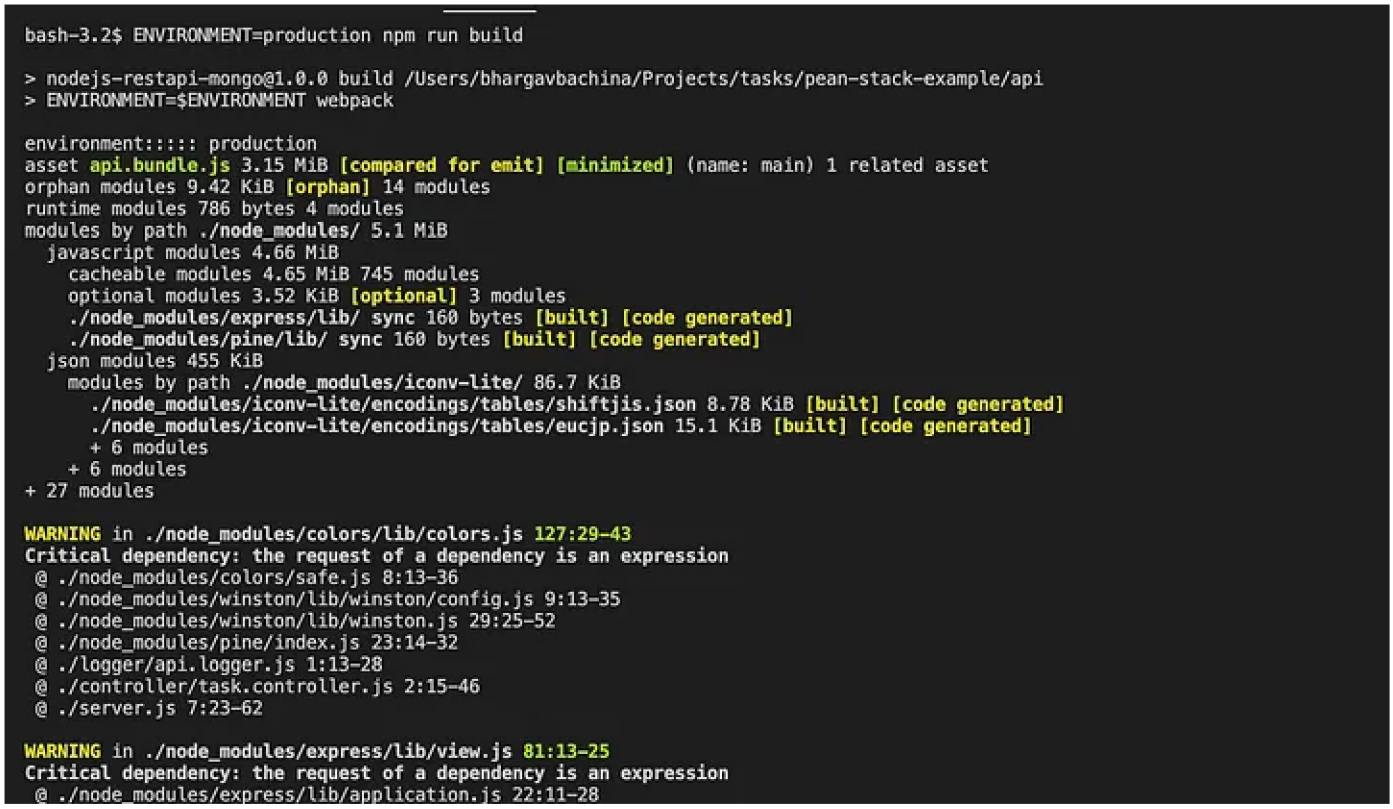
```
25  };
26
27  if (environment === 'test') {
28    ENVIRONMENT_VARIABLES = {
29      'process.env.HOST': JSON.stringify('localhost'),
30      'process.env.USER': JSON.stringify('bhargavbachina'),
31      'process.env.DB': JSON.stringify('bhargavbachina'),
32      'process.env.DIALECT': JSON.stringify('postgres'),
33      'process.env.PORT': JSON.stringify('3080'),
34      'process.env.PG_CONNECTION_STR': JSON.stringify("postgres://bhargavbachina:''@localhost:5432/b
35    };
36  } else if (environment === 'production') {
37    ENVIRONMENT_VARIABLES = {
38      'process.env.HOST': JSON.stringify('localhost'),
39      'process.env.USER': JSON.stringify('bhargavbachina'),
40      'process.env.DB': JSON.stringify('bhargavbachina'),
41      'process.env.DIALECT': JSON.stringify('postgres'),
42      'process.env.PORT': JSON.stringify('3080'),
43      'process.env.PG_CONNECTION_STR': JSON.stringify("postgres://pgadmin@webappdemopostgre:Tester@1
44    };
45  }
46
47  module.exports = {
48    entry: './server.js',
49    output: {
50      path: path.resolve(__dirname, 'dist'),
51      filename: 'api.bundle.js',
52      libraryTarget: 'commonjs'
53    },
54    target: 'node',
55    plugins: [
56      new webpack.DefinePlugin(ENVIRONMENT_VARIABLES),
57    ],
58    //externals: ['pg', 'pg-hstore']
59    externals: [
60      { pg: { commonjs: ['pg'] } },
61      { 'pg-hstore': { commonjs: ['pg-hstore'] } }
62    ],
63  };
```

Let's build and bundle it. All you need to do is to run this command `webpack` and the webpack looks for this file called **webpack.config.js** and builds the entire server code and put it into one file called **api.bundle.js.** Here is the modified **package.json** file.

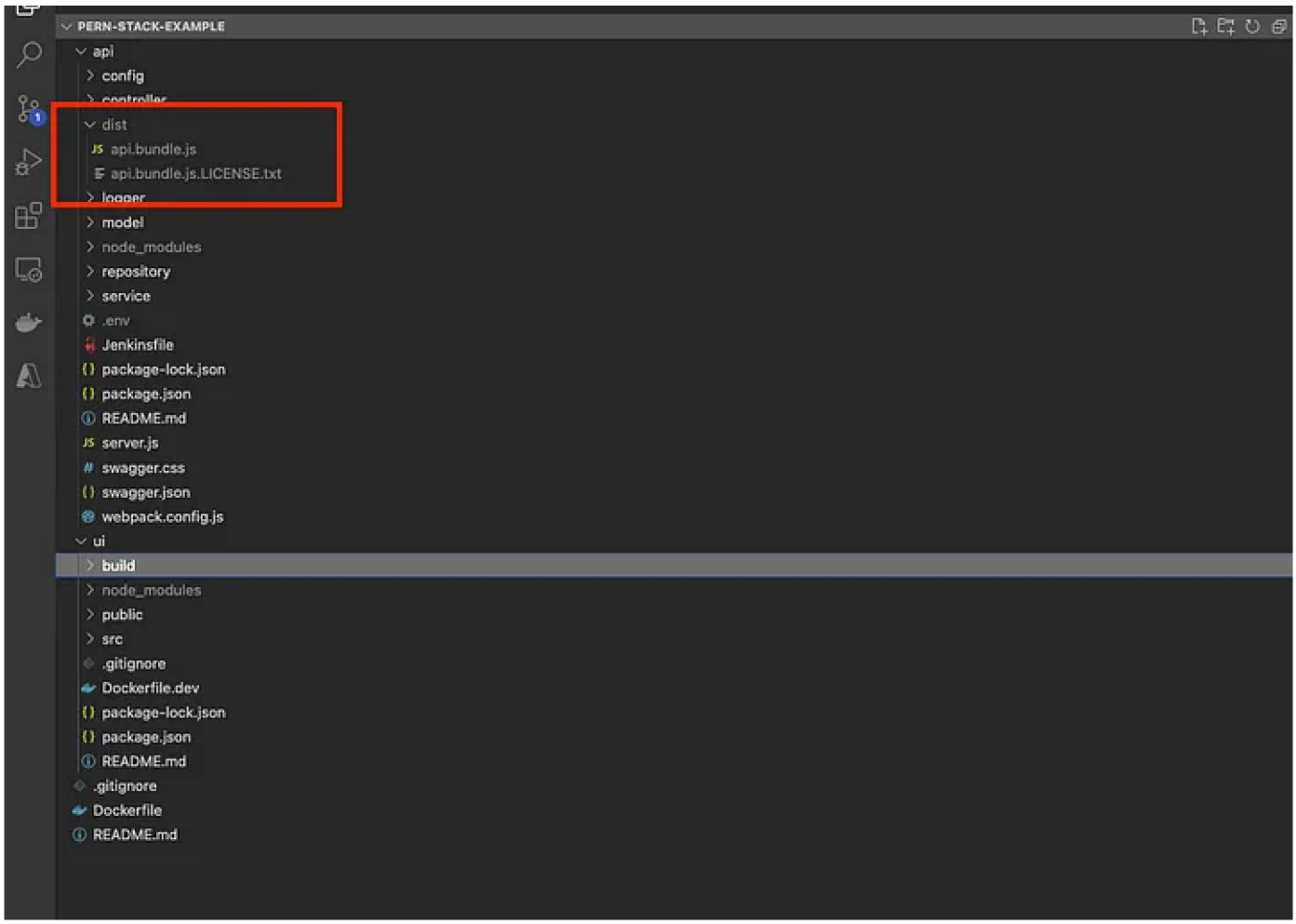
```
{
  "name": "nodejs-restapi-mongo",
  "version": "1.0.0",
  "description": "Example Project on how to build and develop REST API with NodeJS and MongoDB",
  "main": "index.js",
  "scripts": {
    "dev": "nodemon ./server.js localhost 3081",
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "ENVIRONMENT=$ENVIRONMENT webpack"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/bbachi/nodejs-restapi-mongo.git"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "bugs": {
    "url": "https://github.com/bbachi/nodejs-restapi-mongo/issues"
  },
  "homepage": "https://github.com/bbachi/nodejs-restapi-mongo#readme",
  "dependencies": {
    "dotenv": "^8.2.0",
    "express": "^4.17.1",
    "pg": "^8.7.3",
    "pine": "^1.1.1",
    "sequelize": "^6.17.0",
    "swagger-jsdoc": "^6.1.0",
    "swagger-ui-express": "^4.3.0"
  },
  "devDependencies": {
    "nodemon": "^2.0.7",
    "webpack": "^5.72.0",
    "webpack-cli": "^4.9.2"
  }
}
```

If the filename is different than webpack.config.js you need to pass that filename with the webpack command `webpack <filename>`. Once you build the server code all you need is **api.bundle.js** file. You don't even need any packages, package.json, etc. Run the following command to build the server code.

```
ENVIRONMENT=production npm run build
```

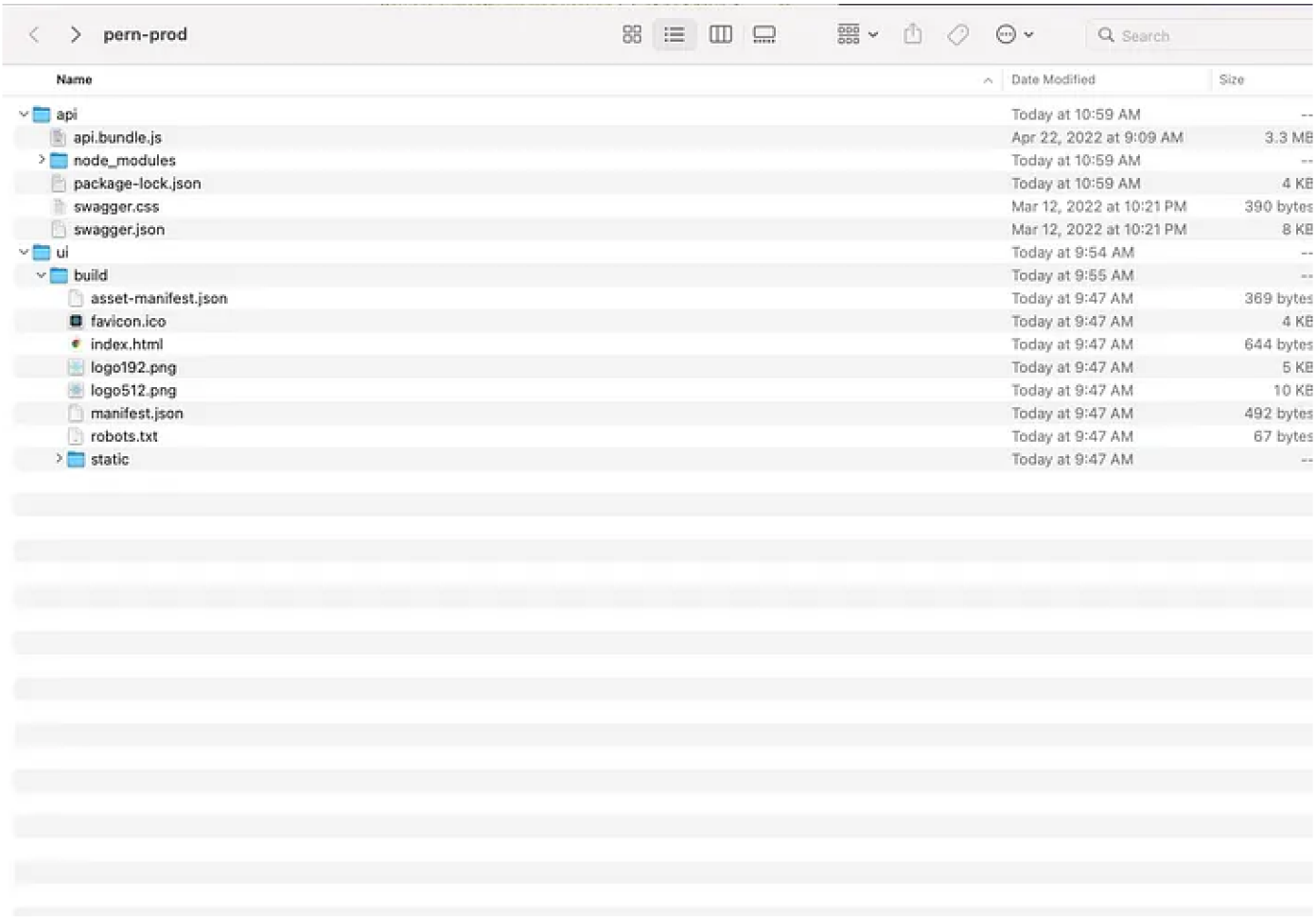


ENVIRONMENT=production npm run build



Server Bundle

You can copy just the bundle file into the API folder.



Build

With this, we can skip the step `npm install` (installing dependencies) and you can just run `node api.bundle.js` and you can see the app running on port **3080. Since we need to have pg installed on the server where this code is running you need to have that node module installed.**

```
+ pg@8.7.3
bash-3.2$ node api.bundle.js
process.env.PG_CONNECTION_STR  postgres://pgadmin@webappdemopostgre:Tester@123@webappdemopostgre.postgres.database.azure.com:5432/tas
Server listening on the port  3080
Executing (default): DROP TABLE IF EXISTS "tasks" CASCADE;
Executing (default): DROP TABLE IF EXISTS "tasks" CASCADE;
Executing (default): CREATE TABLE IF NOT EXISTS "tasks" ("id"   SERIAL , "task" VARCHAR(255) NOT NULL, "assignee" VARCHAR(255), "stat
ITH TIME ZONE, "updateddate" TIMESTAMP WITH TIME ZONE, "createdby" VARCHAR(255), "updatedby" VARCHAR(255), "createdAt" TIMESTAMP WITH
 WITH TIME ZONE NOT NULL, PRIMARY KEY ("id"));
Executing (default): SELECT i.relname AS name, ix.indisprimary AS primary, ix.indisunique AS unique, ix.indkey AS indkey, array_agg(a
tname) AS column_names, pg_get_indexdef(ix.indexrelid) AS definition FROM pg_class t, pg_class i, pg_index ix, pg_attribute a WHERE t
id AND a.attrelid = t.oid AND t.relkind = 'r' and t.relname = 'tasks' GROUP BY i.relname, ix.indexrelid, ix.indisprimary, ix.indisuni
Drop and re-sync db.
```

node api.bundle.js

Disadvantages

The only thing we have solved is to skipping node dependencies installation. There are still things we are doing here manually.

- We have to build React code manually
- We have to build API code manually
- We have to place all the built files into a separate folder

Packaging With Gulp

In the above sections, we have seen manual steps and these steps have to be eliminated. We can achieve complete automation with the gulp. All the following steps can be made automated with the gulp.

- Clean the directory if exists
- Create a directory if not exists to put all the production build
- Build React code with react-scripts
- Place the react code into the production directory
- Build the server code with the webpack
- Place the server code into the production directory
- Finally, zip all the code.

Let's install all the required gulp packages to accomplish the above points.

```
// install gulp globally
npm install gulp -g

// install as dev dependency
npm install gulp gulp-zip fancy-log del webpack-stream --save-dev

// gulp              - core library
// gulp-zip          - zipping the code
// fancy-log         - logging
// del               - deleting files/folders
// webpack-stream    - Build with webpack
```

when you run the command `gulp` it looks for the gulpfile.js file and executes all the tasks mentioned in that file. We can execute these tasks one by one or in parallel with the help of these modules `series, parallel`. Here is the file **gulpfile.js.**

```
const { src, dest, series, parallel } = require('gulp');
const del = require('del');
const fs   = require('fs');
const zip = require('gulp-zip');
const log = require('fancy-log');
const webpack_stream = require('webpack-stream');
const webpack_config = require('./webpack.config.js');
var exec = require('child_process').exec;

const paths = {
  prod_build: '../prod-build',
  server_file_name: 'api.bundle.js',
  react_src: '../ui/build/**/*',
  react_dist: '../prod-build/ui/build',
  zipped_file_name: 'pern-prod.zip'
};

function clean()  {
  log('removing the old files in the directory')
  return del('../prod-build/**', {force:true});
}

function createProdBuildFolder() {

  const dir = paths.prod_build;
  log(`Creating the folder if not exist  ${dir}`)
  if(!fs.existsSync(dir)) {
    fs.mkdirSync(dir);
    log('📁  folder created:', dir);
  }

  return Promise.resolve('the value is ignored');
}

function buildReactCodeTask(cb) {
    log('building React code into the directory')
    return exec('cd ../ui && npm run build', function (err, stdout, stderr) {
      log(stdout);
      log(stderr);
```

```
40        cb(err);
41      })
42    }
43
44    function copyReactCodeTask() {
45      log('copying React code into the directory')
46      return src(`${paths.react_src}`)
47            .pipe(dest(`${paths.react_dist}`));
48    }
49
50    function copyNodeJSCodeTask() {
51      log('building and copying server code into the directory')
52      return webpack_stream(webpack_config)
53            .pipe(dest(`${paths.prod_build}/api/`))
54    }
55
56    function zippingTask() {
57      log('zipping the code ')
58      return src(`${paths.prod_build}/**`)
59          .pipe(zip(`${paths.zipped_file_name}`))
60          .pipe(dest(`${paths.prod_build}`))
61    }
62
63  function installPG(cb) {
64      log('zipping the code ')
65      return exec('cd ../prod-build && npm install pg', function (err, stdout, stderr) {
66          log(stdout);
67          log(stderr);
68          cb(err);
69      })
70  }
71
72
73
74  exports.default = series(
75    clean,
76    createProdBuildFolder,
77    buildReactCodeTask,
78    parallel(copyReactCodeTask, copyNodeJSCodeTask),
79    installPG,
80    zippingTask
81  );
```

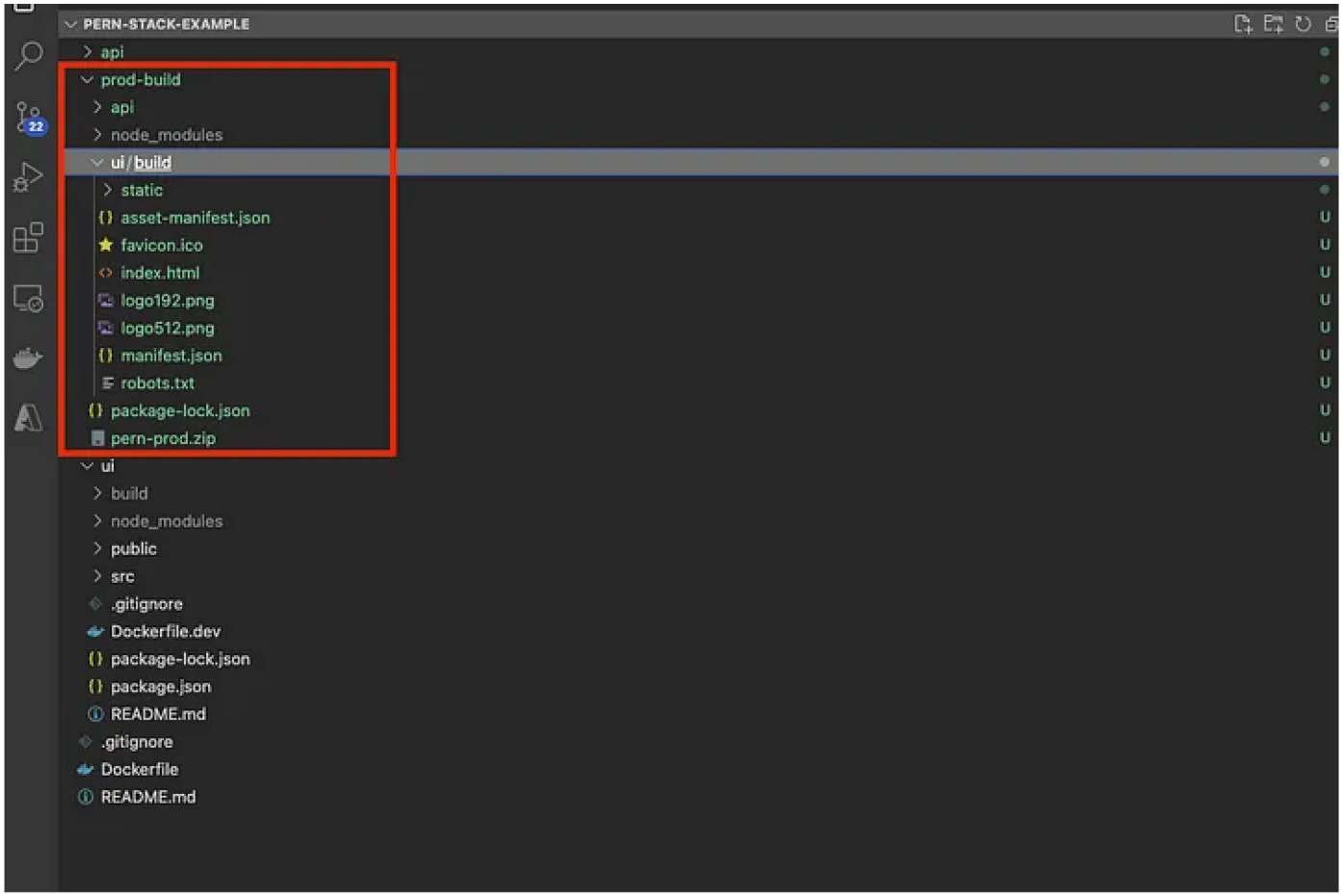
gulpfile.js

You can actually see some tasks are run one by one and others are in parallel. For example, copying React code and building server code (line 68) can be run in parallel because there is no dependency between these. With the gulpfile.js in place, all you need to do is issue this command `gulp` in the package.json file.

npm scripts

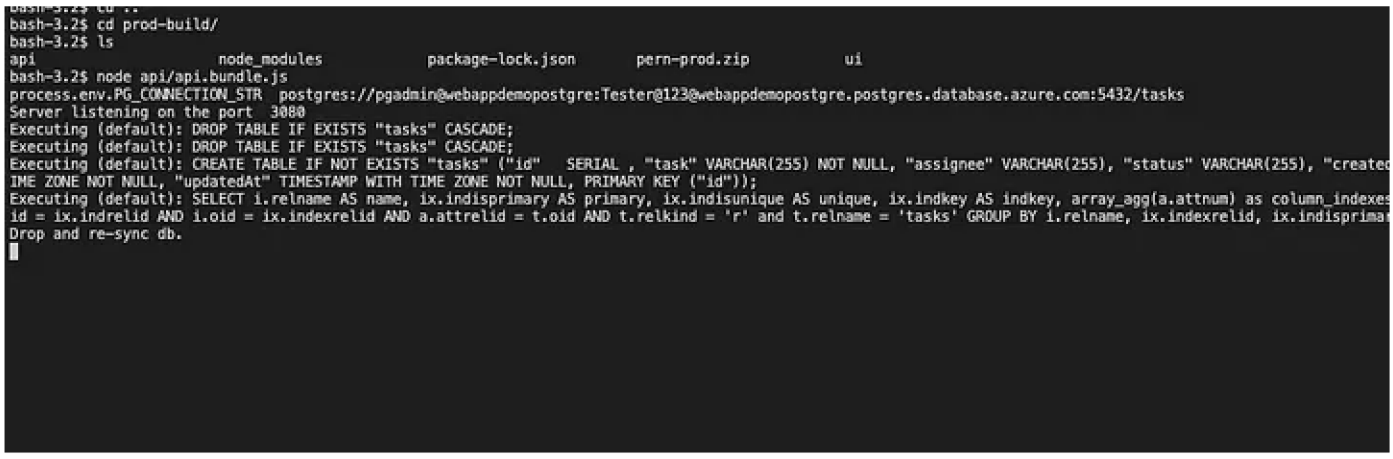
Building with gulp

Once the build is complete, you can see the packaged app under the folder prod-build.



production build

You can run the app with this command by going to the folder `prod-build`



App running on the port 3080

With Docker

We have seen different implementations so far. All these implementations include putting all the related files together and packaging them. We used different tools and bundlers to do that. But with Docker, we place all the files in the Docker file system and create a Docker image out of it. Here is the detailed article about dockerizing the **PERN** Stack application.

> *How To Dockerize PERN Stack*

Here is the Dockerfile to create an image for this application.

```
# Stage1: UI Build
FROM node:14-slim AS ui-build
WORKDIR /usr/src
COPY ui/ ./ui/
RUN cd ui && npm install && npm run build

# Stage2: API Build
FROM node:14-slim AS api-build
WORKDIR /usr/src
COPY api/ ./api/
RUN cd api && npm install && ENVIRONMENT=production npm run build
RUN ls

# Stage3: Packagign the app
FROM node:14-slim
WORKDIR /root/
RUN npm install pg
COPY --from=ui-build /usr/src/ui/build ./ui/build
COPY --from=api-build /usr/src/api/dist .
COPY api/swagger.css .
RUN ls

EXPOSE 3080

CMD ["node", "api.bundle.js"]
```

Dockerfile hosted with ❤ by GitHub view raw

Dockerfile

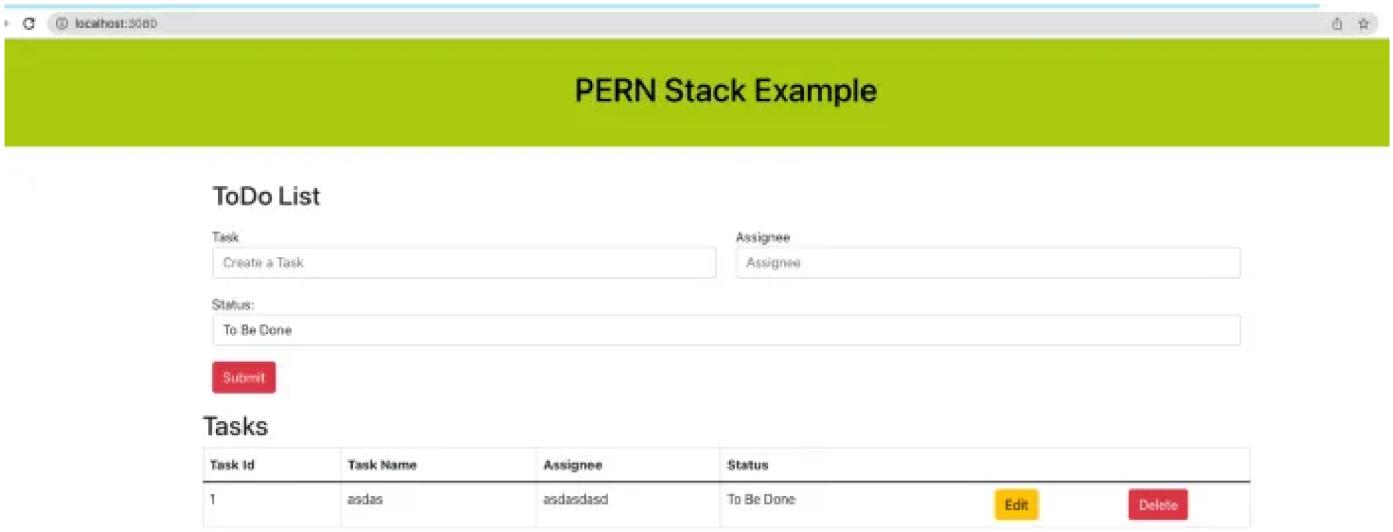
You can follow the below commands to create an image and run the container.

```
// build the image
docker build -t pern-image .

// check the images
docker images

// run the image
docker run -d -p 3080:3080 --name pern-stack pern-image

// check the container
docker ps
```

```
bash-3.2$ docker run -d -p 3080:3080 --name pern-stack pern-image
e34be38f4d78211d7fddc7ac9443b9e28565960003fa5e5954b521eaf9fdd82d9
bash-3.2$ docker ps
CONTAINER ID   IMAGE         COMMAND                  CREATED         STATUS         PORTS                    NAMES
e34be38f4d78   pern-image    "docker-entrypoint.s…"   6 seconds ago   Up 5 seconds   0.0.0.0:3080->3080/tcp   pern-stack
bash-3.2$
```

docker ps

You can access the application on the web at this address http://localhost:3080.



PERN Stack Running on port 3080

Summary

- In traditional architectures, there are so many ways we can package and ship the PERN app to production.
- If you are new to the React app with nodejs backend, please follow this link to get familiar with it.
- With the manual implementation, we have to build the React code, place the appropriate file, and zip the code manually.
- We can automate all these tasks with the help of gulp.
- Create-react-app uses a webpack internally to build the React code. We can use the same with the NodeJS code as well. In this way, we can skip installing all the dependencies.
- Docker is another way to package your application but you need to run those Docker images on some container platforms such as Docker, EKS, ECS, etc.
- Always use multi-stage builds while building your Docker images so that you can avoid unnecessary files packaged into your build.
- Always automate the tasks with some kind of tools such as gulp or grunt.

Conclusion

Always automate packaging your app. In that way, you can save lots of time and be more productive. In future posts, we can take this packaged app and deploy it in different environments.

Programming | Software Development | JavaScript | Software Engineering | Web Development

Follow

Written by Bhargav Bachina

17.8K Followers · Editor for Bachina Labs

Entrepreneur | 600+ Tech Articles | Subscribe to upcoming Videos
https://www.youtube.com/channel/UCWLSuUulkLlQvbMHRUfKM-g | https://www.linkedin.com/in/bachina

---

More from Bhargav Bachina and Bachina Labs

Bhargav Bachina in Bachina Labs

250 Practice Questions For Terraform Associate Certification

Read and Practice these questions before your exam

· 47 min read · Jul 13, 2020

Bhargav Bachina in Bachina Labs

Kubernetes—Learn Sidecar Container Pattern

Understanding Sidecar Container Pattern With an Example Project

· 6 min read · Sep 7, 2020

373 2

Bhargav Bachina in Bachina Labs

How To Implement Micro-Frontend Architecture With Angular

Everything you need to know about microservice oriented architecture for the frontend from beginner to advanced

· 8 min read · Jan 9, 2020

2K 12

Bhargav Bachina in Bachina Labs

Docker—A Beginner's guide to Dockerfile with a sample project

A step by step guide to understanding the Dockerfile

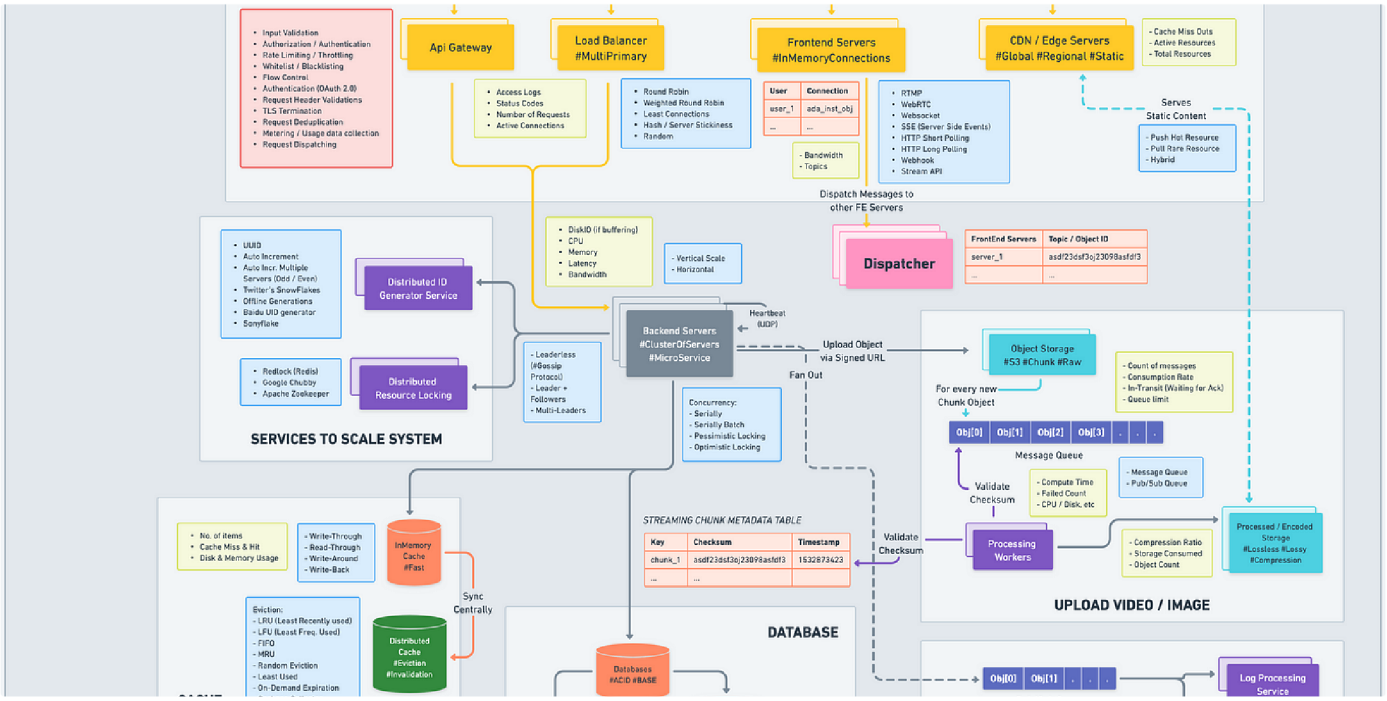
· 6 min read · Feb 24, 2019

Yasas Sandeepa in Level Up Coding

Microservices Explained with Node.js | Concepts and Hands on Experience

Build a server application in Node.js & Docker from scratch with microservice architecture.

· 23 min read · Feb 1



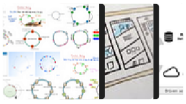
Love Sharma in ByteByteGo System Design Alliance

System Design Blueprint: The Ultimate Guide

Developing a robust, scalable, and efficient system can be daunting. However, understanding the key concepts and components can make the...

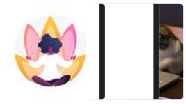
· 9 min read · Apr 20

Lists



General Coding Knowledge

20 stories · 135 saves



Stories to Help You Grow as a Software Developer

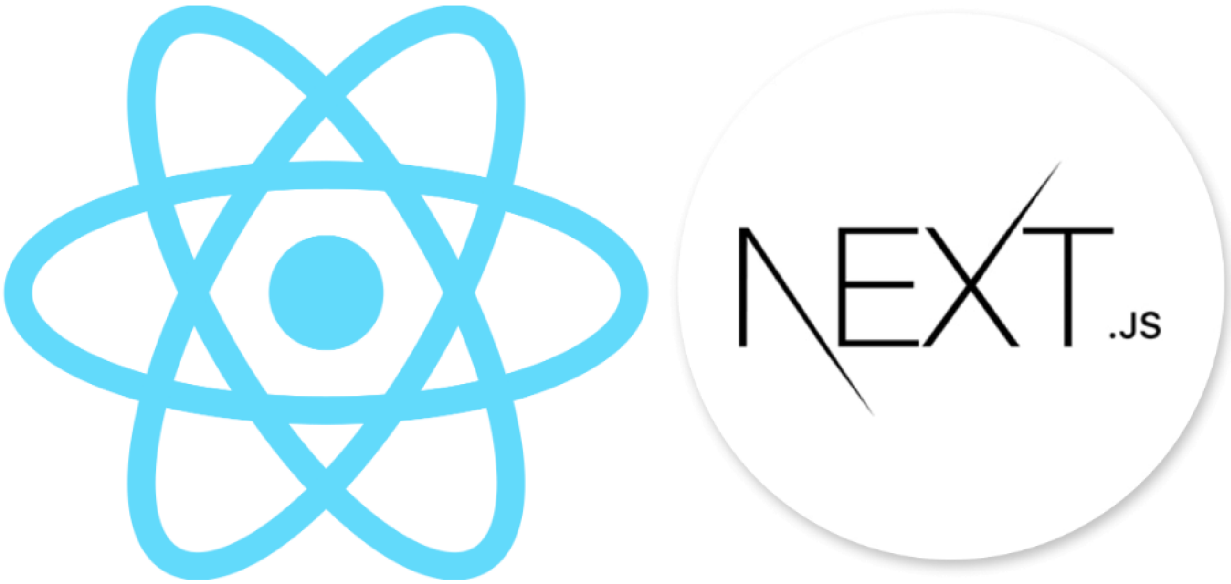
19 stories · 213 saves

It's never too late or early to start something

13 stories · 48 saves

Coding & Development

11 stories · 78 saves



Adhithi Ravichandran

Understanding Server Components in React 18 and Next.js 13

With the release of Next.js 13, they have a new /app directory that has newer approaches to data rendering, fetching, and also uses the...

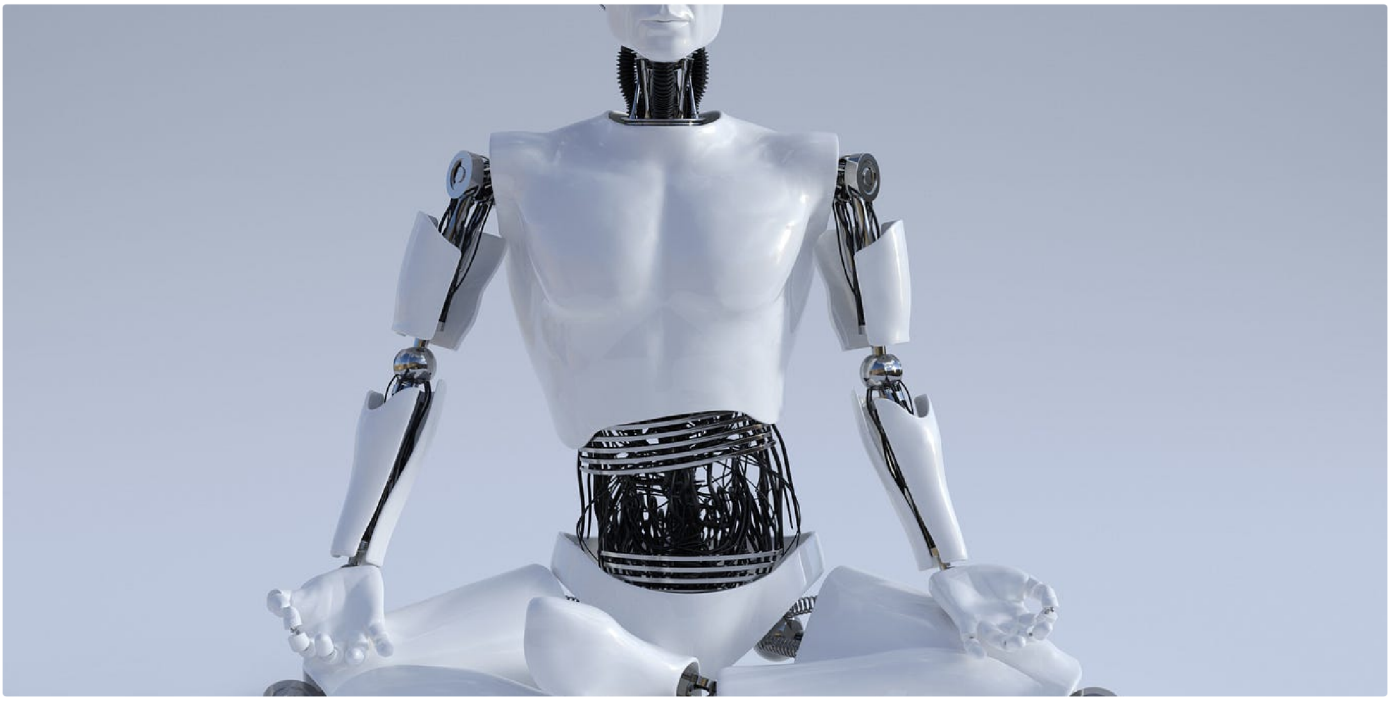
· 5 min read · Feb 15

Kristen Walters in Adventures In AI

5 Ways I'm Using AI to Make Money in 2023

These doubled my income last year

· 9 min read · Jul 19



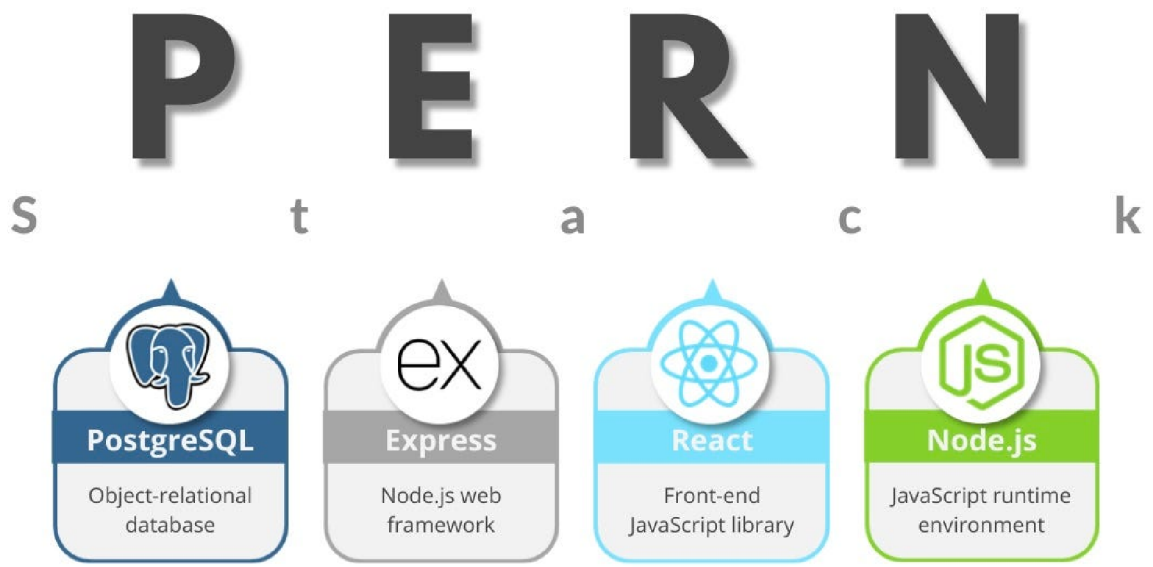
The PyCoach in Artificial Corner

You're Using ChatGPT Wrong! Here's How to Be Ahead of 99% of ChatGPT Users

Master ChatGPT by learning prompt engineering.

✦ · 7 min read · Mar 17

30K 530



Rita Alves

Get Started with the PERN Stack: An Introduction and Implementation Guide

A step-by-step example of how to implement the PERN Stack

9 min read · Mar 2

470

See more recommendations