

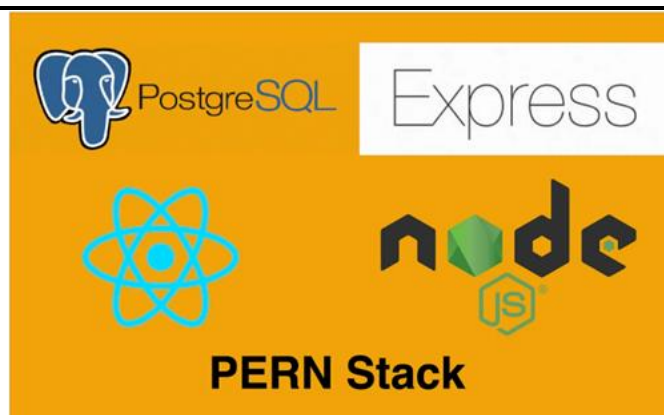


Comprehensive Development and Build Strategies for the PERN Stack

Bhargav Bachina

Abstract This technical paper presents an integrated approach to developing and deploying React applications using the PERN stack, comprising PostgreSQL, React, Express, and NodeJS. This combination is celebrated for its cohesiveness and the convenience of using JavaScript across the entire development stack, making it an ideal choice for various web development scenarios. The document explores the construction process of a React-based frontend, the implementation of PostgreSQL as the database layer, and the utilization of Express and NodeJS for the middleware, providing a step-by-step guide through the development of an example PERN stack project. Topics covered include project setup, database configuration, API development, and user interface construction, alongside modern development practices such as Docker containerization and environment management. By offering a concise yet comprehensive overview, this paper aims to equip developers with the necessary tools and knowledge to effectively employ the PERN stack in building robust full-stack web applications, highlighting the stack's relevance and efficiency in current web development practices.

Keywords Programming, Web Development, Software Development, Software Engineering, JavaScript



Numerous methodologies exist for constructing React applications and preparing them for production deployment. A prominent approach involves utilizing a combination of NodeJS and PostgreSQL for server-side operations and database management, respectively. This method is part of a broader, highly regarded stack known as the PERN stack, which stands out for its cohesive use of JavaScript across all layers of development. The components of this stack—PostgreSQL, React, Express, and NodeJS—offer a powerful, integrated environment that caters to a wide array of web development needs.

In this approach, the frontend is entirely developed using React, allowing for dynamic and responsive user interfaces. PostgreSQL serves as the backend database, chosen for its robustness and capability to handle complex data structures as a document store. Express and NodeJS form the application's middle layer, handling server-side logic and HTTP requests, thereby linking the frontend with the database efficiently.



This paper aims to delve into the intricacies and practical implementation of the PERN stack. We will provide a detailed examination, following a structured, step-by-step approach illustrated with a real-world example project. This will include setting up the development environment, integrating each component of the stack, and demonstrating how they work together to create a seamless, full-stack web application. Through this detailed guide, readers will gain insights into leveraging the PERN stack for efficient web development, showcasing its versatility and power in modern web application construction.

- Introduction
- Prerequisites
- Example Project
- Project Structure
- Install PostgreSQL on Local Machine
- Install PGAdmin Tool
- Create a Database Table
- Building API
- Configure PostgreSQL In API
- Externalize the Environment Variables
- Building UI
- Make API Calls From UI
- Development Environment Setup
- Running on Docker Compose
- Dockerize PERN Stack
- Linting
- Unit Testing API
- Unit Testing UI
- Integration Tests
- Build for production.
- Demo
- Summary
- Conclusion

1. Introduction

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.

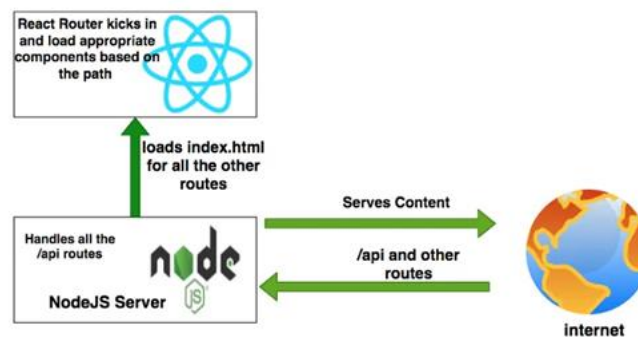


Figure 1: PERN Stack



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.

2. 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** (<https://nodejs.org/en/>)
- **Express Framework** (<https://expressjs.com/>)
- **PGAdmin** (<https://www.pgadmin.org/>)
- **PostgreSQL** (<https://www.postgresql.org/>)
- **Postgresapp** (<https://postgresapp.com/>)
- **Node-Postgres** (<https://www.npmjs.com/package/pg>)
- **VSCode** (<https://code.visualstudio.com/>)
- **Postman** (<https://www.postman.com/>)
- **Nodemon** (<https://nodemon.io/>)
- **Dotenv** (<https://www.npmjs.com/package/dotenv>)
- **Create-React-App** (<https://create-react-app.dev/>)
- **Typescript** (<https://www.typescriptlang.org/>)
- **React Bootstrap** (<https://react-bootstrap.github.io/>)

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.

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

PostgreSQL: Open-Source relational Database

Node-Postgres: Non-blocking Postgresql Client for NodeJS

VSCode: The editor we are using for the project. **It's open-source and you can download it here** (<https://code.visualstudio.com/>).

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** (<https://medium.com/bb-tutorials-and-thoughts/how-to-get-started-with-react-ba61895a8f0c>)
- **How To Develop and Build React App with NodeJS** (<https://medium.com/bb-tutorials-and-thoughts/how-to-develop-and-build-react-app-with-nodejs-bc06fa1c18f3>)
- **How to Build NodeJS REST API with Express and PostgreSQL** (<https://medium.com/bb-tutorials-and-thoughts/how-to-build-nodejs-rest-api-with-express-and-postgresql-674d96d5cb8f>)
- **How to write production-ready Node.js Rest API — Javascript version** (<https://medium.com/bb-tutorials-and-thoughts/how-to-write-production-ready-node-js-rest-api-javascript-version-db64d3941106>)

3. Example Project

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

https://miro.medium.com/v2/resize:fit:1400/1*BRzkjumicvmmidiD8Yn_DyQ.gif

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.

https://miro.medium.com/v2/resize:fit:720/1*Yw9JI54tMWx554iAIW8XUA.gif



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 Codecd uinpm install npm start
```

```
// API codecd apinpm install npm run dev
```

4. Project Structure

Let's understand the project structure for this project. We will have two package.json: one for the **React** and another for **nodejs API**. It's always best practice to have completely different node_modules for each one. In this way, you won't get merging issues or any other problems regarding web and server node modules collision. It's easier to convert your PERN Stack into any other stack later such as replacing the API code with microservices and serving your UI through the NGINX web server.



Figure 2: Project Structure

If you look at the above project structure, all the React app resides under the ui folder and nodejs API resides under the **api** folder.

5. Install PostgreSQL on Local Machine

There are so many ways to install PostgreSQL on your local machine from the below link. The Postgres.app is the easiest and fastest one.

<https://www.postgresql.org/download/macosx/>

You can click on the Postgres.app and download the app from that page.

Netel This installer is hosted by EDB and not on the PostgreSQL community servers. If you have issues with the website it's hosted on, please contact [webmaster](#). This installer includes the PostgreSQL server, pgAdmin, a graphical tool for managing and developing your databases, and StackBuilder, a package manager for the PostgreSQL tools and drivers. StackBuilder includes management, integration, migration, replication, geospatial, connectors and other tools. This installer can run in graphical, command line, or silent install modes. The installer is designed to be a straightforward, fast way to get up and running with PostgreSQL on macOS. Advanced users can also download a zip archive of the binaries, without the installer. This download is intended for users who wish to include PostgreSQL as part of their own software.

Platform support

The installers are tested by EDB on the following platforms. They will generally work on newer versions of macOS as well:

PostgreSQL Version	64-bit macOS Platforms
14	10.14 - 12.x (amd64), 12.x (arm64)
13	10.14 - 11.0
12	10.13 - 10.15
11	10.12 - 10.14
10	10.11 - 10.13
9.6	10.10 - 10.12

Postgres.app

Postgres.app is a simple, native macOS app that runs in the menu bar without the need of an installer. Open the app, and you have a PostgreSQL server ready a the server shuts down.

Homebrew

PostgreSQL can also be installed on macOS using Homebrew. Please see the Homebrew documentation for information on how to install packages. A list of PostgreSQL packages can be found using the Braumeister search tool.

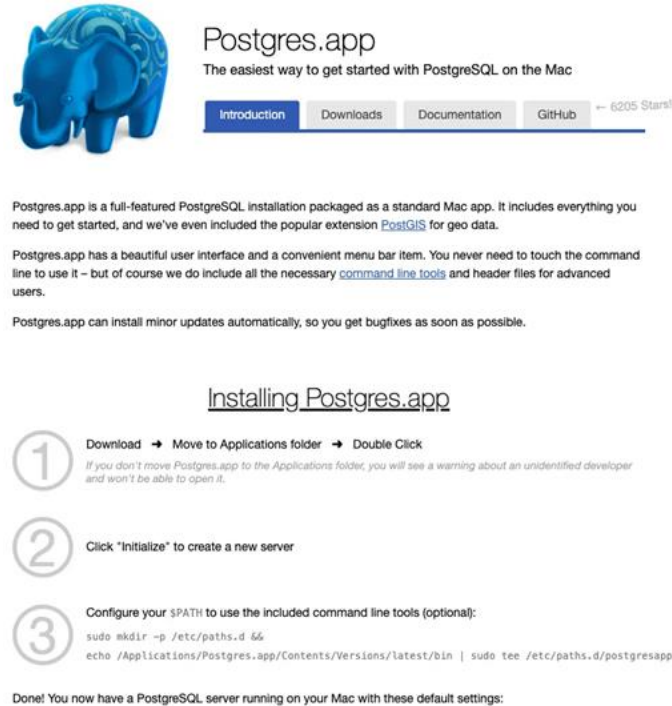
MacPorts

PostgreSQL packages are also available for macOS from the MacPorts Project. Please see the MacPorts documentation for information on how to install ports.

Figure 3: Postgres.app



You can go through the below installation steps and initialize the Database.



The screenshot shows the Postgres.app website. At the top left is a blue elephant logo. The main heading is "Postgres.app" with the subtitle "The easiest way to get started with PostgreSQL on the Mac". Below this are navigation tabs for "Introduction", "Downloads", "Documentation", and "GitHub", along with a "6205 Stars!" badge. The main content area contains three paragraphs of text describing the app's features and installation process. Below the text is a section titled "Installing Postgres.app" with three numbered steps: 1. Download → Move to Applications folder → Double Click; 2. Click "Initialize" to create a new server; 3. Configure your \$PATH to use the included command line tools (optional), with terminal commands: `sudo mkdir -p /etc/paths.d && echo /Applications/Postgres.app/Contents/Versions/latest/bin | sudo tee /etc/paths.d/postgresapp`. At the bottom, it says "Done! You now have a PostgreSQL server running on your Mac with these default settings:" followed by a table.

Host	localhost
Port	5432

Figure 4: Download and Install Instructions

If everything is successful, you can see the below screen with the database named after the username on the machine.

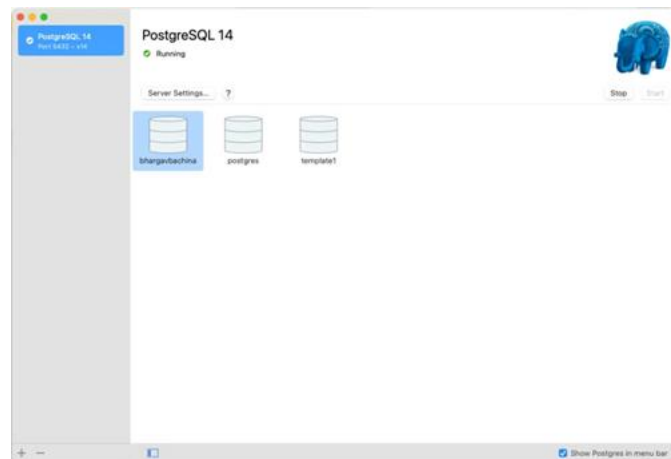


Figure 5: Postgres.app

6. Install PGAdmin Tool

The pgAdmin tool is the open-source administration and development platform for PostgreSQL. You can install this tool from the following location.

<https://www.pgadmin.org/>





Figure 6: pgAdmin Tool

Once installed, you can open that and connect to the PostgreSQL server with the following credentials. It changes based on your username folder.

// name of the servername: local (You can name anything)

// Hostnamehost name: localhost

// User Nameusername: <user name based on the above postgres.app>

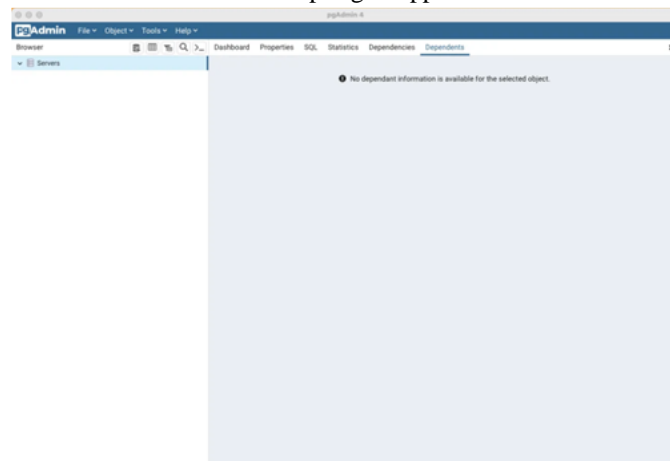


Figure 7: pgAdmin Tool

Let's connect to the server by clicking on the register below.

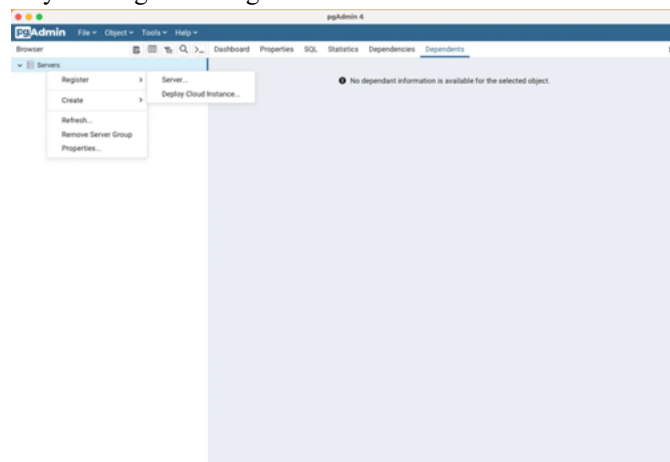


Figure 8: Register Server



The server name can be anything that you give for your server such as local, dev, test, etc.

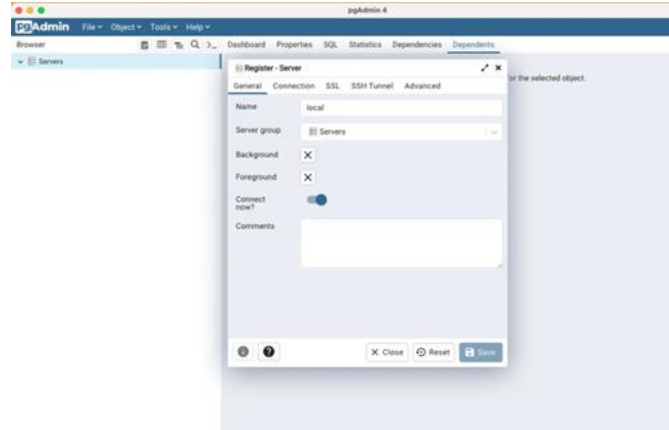


Figure 9: Server Name

Let's give all the details such as HostName, port, username, etc under the connection tab.

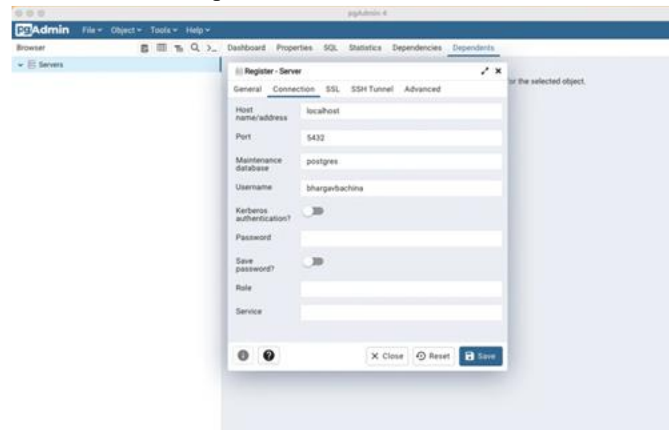


Figure 10: Connection Details

Once connected, you can see the details below.

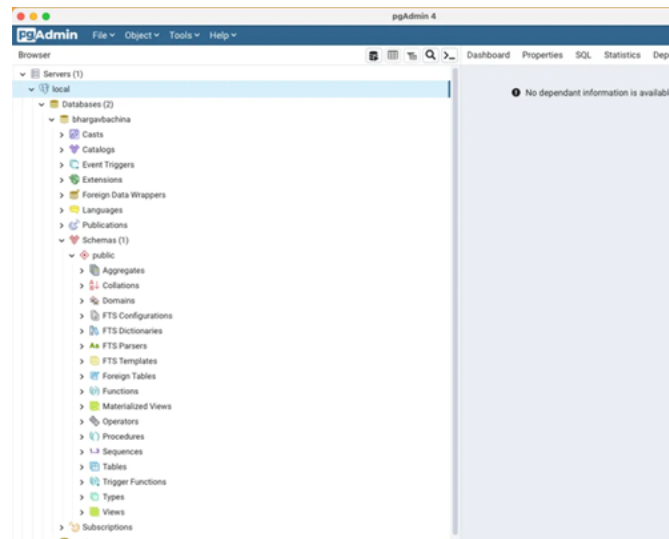


Figure 11: Connected

7. Create a Database Table

Let's create a table by clicking on the Query Tool as below.



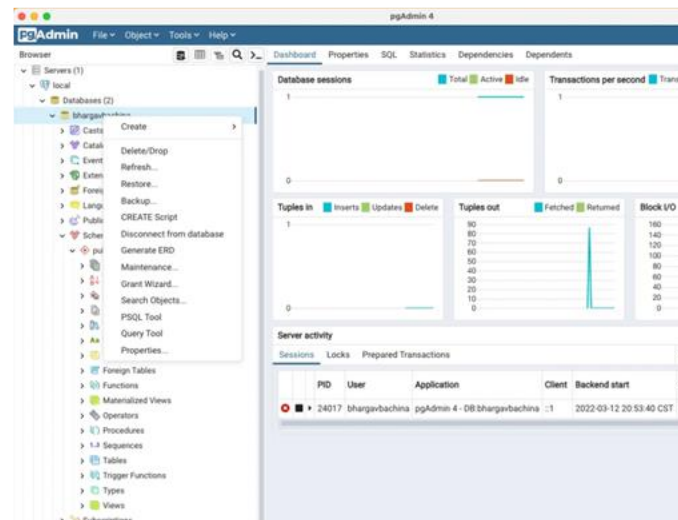


Figure 12: Query Tool

Let's run the following query to create the database table.

<https://gist.github.com/bbachi/9dbcc0530954f4a8320eda5ecd34b5c3#file-database-sql>

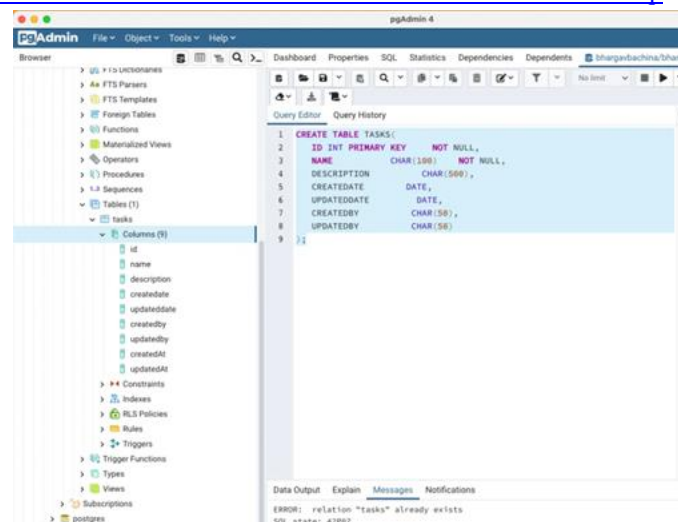


Figure 13: Table Created

8. Building API

We have configured PostgreSQL in the previous section, it's time to build the API. I would recommend you go through two articles posted in the prerequisites section. Let me put those here as well.

- **How to Build NodeJS REST API with Express and PostgreSQL** (<https://medium.com/bb-tutorials-and-thoughts/how-to-build-nodejs-rest-api-with-express-and-postgresql-674d96d5cb8f>)
- **How to write production-ready Node.js Rest API — Javascript version** (<https://medium.com/bb-tutorials-and-thoughts/how-to-write-production-ready-node-js-rest-api-javascript-version-db64d3941106>)

The starting point of the API is the *server.js* file in which we define all the routes and import the express. Here is the file where the nodejs server runs on port 3080 and starts listening for the incoming requests.

<https://gist.github.com/bbachi/ecbfc405a9add7964d5a154ec9e222cc#file-server-js>

We have defined 4 routes for CRUD operations. Notice that we are using four different HTTP methods for creating, updating, reading, and deleting operations. The request comes to these routes and each route calls the respective method in the controller class. You can read the body of the incoming requests in the req object



defined in each route. The result of these methods is a promise based so you need to use *then* method to read and send back to the client with the method `res.json()`.

Here is the controller class in which we are calling the service class with `async/await`. The `async/await` is the cleaner way of reading promises. You don't need `async/await` here since we are directly returning the result of the service class.

<https://gist.github.com/bbachi/cf3d9b47e600f3327715a80c624a8472#file-task-controller-js>

Let's look at the service class in which we call the repository to interact with the PostgreSQL data.

<https://gist.github.com/bbachi/4c0206ee864bf83a8916642f7e9b9ba5#file-task-service-js>

You need to know how to configure PostgreSQL Connection in the NodeJS before looking at the repository so that you can read the data from PostgreSQL. Let's find that out in the following section.

9. Configure PostgreSQL in API

Let's configure the `pg` Client from our application. The first thing we need to do is to get the connection string or connection details. You can get it from the properties below.

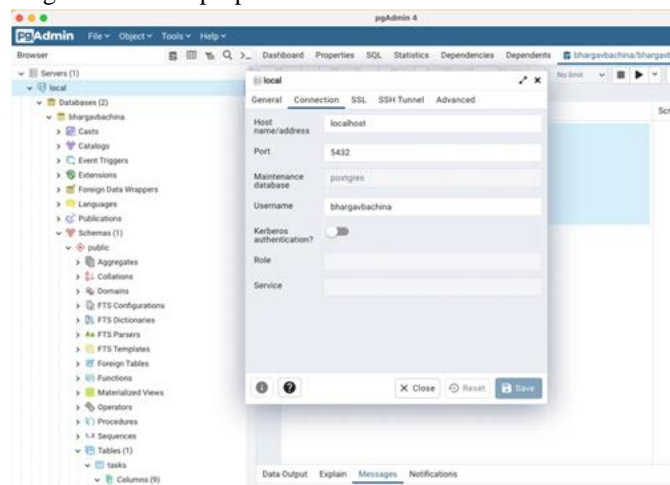


Figure 14: Connection Details

The next thing is to install the `pg` client with the following command.

```
// install client and sequelize
npm install pg npm install sequelize
```

```
// node-postgres home page https://node-postgres.com/
```

Let's place the connection string and database name in the application properties file below. You have to URL encode the password if you have any special characters in the password.

Here is the configuration file in which you connect to PostgreSQL with the help of the connection string. We are using `pg` and `sequelize` to connect with PostgreSQL for all the queries. These tools make it easy for you to interact with PostgreSQL.

<https://gist.github.com/bbachi/720d64b801d81ed69278de2548aba8c6#file-db-config-js>

The next thing we should define is the schema for the database model as below.

<https://gist.github.com/bbachi/d00f608146c63195178a8df31e094a8c#file-task-model-js>

Finally, we have a repository class as below using the above model for the CRUD operations.

<https://gist.github.com/bbachi/a3680ab2e51ddf0fd9ad82c03ce831a0#file-task-repository-js>

10. Externalize the Environment Variables

We have seen how to configure your PostgreSQL connection in the API. We need to store this kind of configuration outside of your app so that you can build once and deploy it in multiple environments with ease.

We need to use the `dotenv` library for environment-specific things. `Dotenv` is a zero-dependency module that loads environment variables from a `.env` file into `process.env`. Storing configuration in the environment separate from code is based on **The Twelve-Factor App** (<http://12factor.net/config>) methodology.

The first step is to install this library `npm install dotenv` and put the `.env` file at the root location of the project.



<https://gist.github.com/bbachi/38e674535f4f0851245b98bce36e8e94#file-env>

We just need to put this line `require('dotenv').config()` as early as possible in the application code as in the `server.js` file.

<https://gist.github.com/bbachi/1a29181bbce1029e85af3802504a778f#file-server-js>

Let's define the configuration class where it creates a connection with the connection details from the properties. We are using `pg` client to connect with PostgreSQL for all the queries. This client makes it easy for you to interact with PostgreSQL. We are fetching the connection details with the `dotenv` library and connecting it to PostgreSQL with `pg` client. We are exposing one function from this file `connect`.

Sequelize is a promise based NodeJS ORM tool for many relational databases such as Postgres, MYSQL, etc.

<https://gist.github.com/bbachi/d2a5ec615c9a8477b5b3369e07bddfb3#file-db-config-js>

11. Building UI

Once you create the separate folder for React code you need to start with the following command to scaffold the React structure with the help of React CLI. We will not build the entire app here instead we will go through important points here. You can clone the entire GitHub Repo and check the whole app.

`npm create-react-app ui`

Here is the `index.js` file for the app and App Component as the bootstrap component which means this is the first component that loads in the browser.

<https://gist.github.com/bbachi/a464fcd3f35291b8ff72bb25d5cea219#file-index-js>

Here is the starting point of the application in which we define the home component to load for the path `/`. You need to import the `react-router-dom` library for the routing part of the app. The Home page will be loaded when we start the application.

<https://gist.github.com/bbachi/ca05d3eb1c3dee19bc797bf983ef5da1#file-app-js>

Here is the home component. This is a simple application where you add, update, and delete tasks. You can go through the GitHub repo to check the rest of the files. We are using two hooks here one is for maintaining the local state and another is for fetching the data from the API.

<https://gist.github.com/bbachi/0f4dbda7b02c1db0c66605a743557eb1#file-home-js>

We have another two important components here one is for the `createTask` Form component, and another is for the `Tasks` table.

<https://gist.github.com/bbachi/b02b50bb1ae61e4f4ff84f6279a642f7#file-createtask-js>

Run the React code in local with the following command which runs on the port **3000** on localhost. Make sure you are in the root folder of React code which is `todo-app` here.

`cd uinpm start`



Figure 15: React Code running on port 3000.

12. Make API Calls From UI

Here is the service file which calls the API, in this case. We have four API operations to get, add, edit, and delete tasks with root path `/api`.

<https://gist.github.com/bbachi/a0118082cfcf01b85b20b4a3cd8b36ab#file-todoservice-js>

From the react components you can call this service to get the data using React Hooks. Here is an example.

<https://gist.github.com/bbachi/38cd2ff52039c62e553e0a48e0120f74#file-home-js>

You can look at the below article for a detailed post.

How To Make API calls in React Applications (<https://medium.com/bb-tutorials-and-thoughts/how-to-make-api-calls-in-react-applications-7758052bf69>)



13. Development Environment Setup

Usually, the way you develop and the way you build and run production are completely different.

In the development phase, we run the nodejs server and the React app on completely **different ports**. It's easier and faster to develop that way. If you look at the following diagram the React app is running on port **3000** with the help of a webpack dev server and the nodejs server is running on port **3080**.

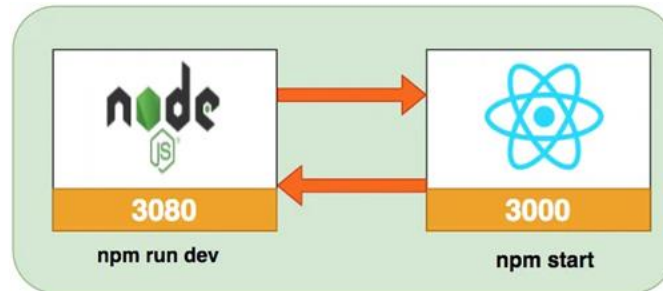


Figure 16: Development Environment

There should be some interaction between these two. We can proxy all the API calls to nodejs API. Create-react-app provides some inbuilt functionality and to tell the development server to proxy any unknown requests to your API server in development, add a `proxy` field to your `package.json` of the React. Have a look at the `package.json` below. Remember you need to put this in the React UI `package.json` file.

<https://gist.github.com/bbachi/9ae4dbdb179ca15a5507ee2a7296c262#file-package-json>

Now you can run both React UI and NodeJS API on different ports and the React Code interacts with the API.

// React Codecd uinpm installnpm start

// API codecd apinpm installnpm run dev

https://miro.medium.com/v2/resize:fit:720/1*j7EWjQuwx76WTOILWDvXHQ.gif

https://miro.medium.com/v2/resize:fit:1400/1*Yw9Jl54tMWx554iAlW8XUA.gif

14. Running on Docker Compose

Docker Compose is useful when we don't have the development environment setup on our local machine to run all parts of the application to test or we want to run all parts of the application with one command. For example, if you want to run NodeJS REST API and PostgreSQL database on different ports and need a single command to set up and run the whole thing. You can accomplish that with Docker Compose.

Coming Soon!!

15. Dockerize PERN Stack

Docker is an enterprise-ready container platform that enables organizations to seamlessly build, share, and run any application, anywhere. Almost every company is containerizing its applications for faster production workloads so that they can deploy anytime and sometimes several times a day. There are so many ways we can build a PERN Stack. One way is to dockerize it and create a docker image so that we can deploy that image any time or sometimes several times a day.

Coming Soon!!

16. Linting

We need to lint our project in that way it's easier to follow some standards in your project. We will see this in a separate article.

Coming Soon!!

17. Unit Testing API

There are so many tools out there to unit test your application such as Mocha, Chai, etc. We need a separate article for that to cover different libraries.

Coming Soon!!



18. Unit Testing UI

We will see how to unit test with UI with jest library.

Coming Soon!

19. Integration Tests

We will use cypress for the integration tests.

Coming Soon!

20. Build for production

We must build the project for production in a different way. We can't use the proxy object. Here is the detailed article on how to package your app for production.

Packaging Your React App with NodeJS Backend For Production (<https://medium.com/bb-tutorials-and-thoughts/packaging-your-react-app-with-nodejs-backend-for-production-7ddae2b84f1b>)

21. Demo

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

https://miro.medium.com/v2/resize:fit:720/1*BRzkjumicvmmdiD8Yn_DyQ.gif

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.

https://miro.medium.com/v2/resize:fit:720/1*Yw9Jl54tMWx554iAlW8XUA.gif

22. Summary

- 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.
- We will have two package.json: one for the React and another for nodejs API. It's always best practice to have completely different node_modules for each one.
- You can get the connection string and configure the NodeJS application to talk to PostgreSQL with pg client, etc.
- Node-Postgres: Non-blocking Postgresql Client for NodeJS
- PGAdmin: pgAdmin is an Open-Source administration and development platform for PostgreSQL.
- We need to use the dotenv library for environment-specific things. Dotenv is a zero-dependency module that loads environment variables from a .env file into process.env. Storing configuration in the environment separate from code is based on The Twelve-Factor App (<http://12factor.net/config>) methodology.
- In the development phase, we run the nodejs server and the React app on completely different ports. It's easier and faster to develop that way.
- We need to lint our project in that way it's easier to follow some standards in your project.
- There are so many tools out there to unit test the API such as Mocha, Chai, etc.
- We can unit test with UI with jest library.
- We will use cypress for the integration tests.
- We must build the project for production in a different way. We can't use the proxy object.

23. Conclusion

In conclusion, this paper has presented a comprehensive approach to building and deploying React applications using the PERN stack, consisting of PostgreSQL, React, Express, and NodeJS. This stack is celebrated for its unified JavaScript development environment, catering to various web development scenarios. We emphasized best practices such as maintaining separate package.json files for the React and NodeJS components to ensure



modularity and manage dependencies effectively. Further, the integration of PostgreSQL with NodeJS through the pg client, the utilization of PGAdmin for database management, and the implementation of dotenv for environment variable management were discussed, aligning with The Twelve-Factor App methodology. Development strategies included running the NodeJS server and React app on different ports for efficiency and adopting linting to maintain code standards.

Moreover, the paper highlighted the importance of unit testing using tools like Mocha, Chai, and Jest, and integration testing with Cypress, to ensure robust application functionality. Lastly, we underscored the distinct approaches required for building the project for production, diverging from development practices such as the use of a proxy object, to ensure optimal performance and security. This guide serves as a foundational resource for developers seeking to leverage the PERN stack for efficient and scalable web application development.

References

- [1]. JavaScript Documentation <https://developer.mozilla.org/en-US/docs/Web/JavaScript>
- [2]. PostgreSQL Documentation <https://www.postgresql.org/docs/>
- [3]. Express.js Documentation <https://expressjs.com/>
- [4]. NodeJS Documentation <https://nodejs.org/docs/latest/api/>

